Technische Universität Graz
Institut für Elektronik
Inffeldgasse 12/1
8010 Graz
Österreich

IFE
TU Graz

Masterarbeit Telematik

Bitserielles Kommunikationssystem zur Übertragung von Video-, Audio- und Steuerdaten in Wearable Computing Applikationen

ausgeführt am
Institut für Elektronik
der Technischen Universität Graz
Leiter: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Bösch

von:
Michael Hemetsberger, BSc. 0530630

in Zusammenarbeit mit der Firma Spintower KG

Betreuer:
Ass.Prof. Dipl.-Ing. Dr.techn. Gerhard Stöckler

Graz, 5. Oktober 2012

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ....

(Unterschrift)

Kurzfassung

Die vorliegende Arbeit beschäftigt sich mit dem Entwurf eines bitseriellen Kommunikationssystems zur Übertragung von Video-, Audio- und Steuerdaten in Wearable Computing Applikationen, basierend auf dem Serializer/Deserializer-Konzept.

Als zentrale Steuereinheit dient der Mikrocontroller i.MX31 von Freescale, der die Daten zur Video- und Audioausgabe an ein Peripheriegerät zur Verfügung stellt. Zusätzlich nimmt der i.MX31 die Videosignale eines CMOS Sensors, sowie das Audiosignal eines Mikrophons am Peripheriegerät entgegen, welches zusätzlich über eine Datenschnittstelle des i.MX31 gesteuert wird.

Die Übertragungsrichtung vom i.MX31 zur Peripherie wird als Downchannel definiert, während die Übertragungsrichtung von der Peripherie zum i.MX31 als Upchannel bezeichnet wird. Die physikalische Anbindung der Peripherie an den i.MX31 wird mit einem differentiellen Leitungspaar je Kommunikationsrichtung durchgeführt, wobei auch die Spannungsversorgung der Peripherie über die selben Leitungen zur Verfügung gestellt wird.

In dieser Arbeit wird ein Konzept zur Phantomspeisung von Peripheriegeräten mit Hilfe induktiver Einkopplung einer Gleichspannung auf die differentiellen Leitungspaare des Up- und Downchannels vorgestellt, sowie ein Konzept, welches die Übertragung aller geforderten Daten mittels den SerDes DS90UR241 bzw. DS90UR124 ermöglicht.

Außerdem beinhaltet diese Arbeit die Fertigungsunterlagen von zwei Prototypen, wobei das LVDS-Testboard zur Evaluierung der ausgewählten Hardware, sowie zur Prüfung des Phantomspannungskonzepts dient, während das RPI-Testboard zur Evaluierung des Gesamtkonzepts verwendet wird.

Stichwörter:
Serializer, Deserializer, i.MX31, Downchannel, Upchannel, Phantomspeisung, LVDS-Testboard, RPI-Testboard

Abstract

This thesis covers the design of a bit-serial communication system to transmit video, audio and control data in wearable computing applications based on existing serializer and deserializer chips.

The microcontroller i.MX31, developed by Freescale Semiconductor, Inc., has to be used as the central processing unit to handle the control of video and audio output data, post-processed on peripheral devices. Additionally, this microcontroller has to receive video input data of the CMOS image sensor OV2640 and digital audio input data of a microphone. The CMOS image sensor and the microphone are situated on the peripherals.

The transmission channel connecting the main control unit and the peripherals is called down-channel. The opposite transmission channel is called up-channel. The physical link between the i.MX31 and the peripherals is realized by two differential signaling interfaces. Moreover, the same cables have to be used to supply the peripherals with power.

Within this thesis, a concept for delivering phantom power through the electrical lines of the up- and down-channels by inductive coupling is presented. In addition, an implementation approach allowing transmission of all required data sources by using the serializer DS90UR241 and the deserializer DS90UR124 is developed.

As a result, this thesis includes the engineering data of two prototypes. First of all, a so-called LVDS test-board used to validate the concept of phantom powering. Finally, a so-called RPI test-board used for validation purposes of the bit-serial communication system.

Keywords:

serializer, deserializer, i.MX31, down-channel, up-channel, phantom powering, LVDS test-board, RPI test-board

Danksagung

Ich möchte mich an dieser Stelle besonders bei meinem Betreuer an der TU Graz, Herrn Prof. Dipl.-Ing. Dr.techn. Gerhard Stöckler, für die Vermittlung dieser Masterarbeit und die tatkräftige Unterstützung bei der Ausführung dieser Projektarbeit bedanken.

Des weiteren möchte ich mich beim Geschäftsführer der Firma Spintower KG, Herrn Dipl.-Ing. Mario Schwaiger, für die Projektidee und die Projektfinanzierung bedanken.

Ein herzliches Dankeschön geht auch an Herrn Eduard Dorner, der mich bei der Bestückung und Fertigung der Leiterplatten fachlich unterstützt hat.

Mein ganz besonderer Dank geht jedoch an meine Eltern, die mir durch ihre große finanzielle Unterstützung das Studium überhaupt erst ermöglicht haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Unter Wearable Computing versteht man die Verfügbarkeit von Rechenressourcen, die am Körper des Anwenders getragen werden können, bzw. die in die Kleidung des Anwenders eingearbeitet sind.

Im Allgemeinen wird bei diesen Systemen zwischen Anwendungen unterschieden, die ständig vom Benutzer getragen werden und immer aktiv sind – wie zum Beispiel Hörgeräte – und Applikationen, die den Anwender zur Unterstützung bei einer speziellen Aufgabe dienen und nur solange getragen werden und aktiv sind, solange der Anwender mit der Bewältigung dieser Aufgabe beschäftigt ist. Als Beispiel für ein solches Szenario könnte ein Museumsführer genannt werden, wobei der Besucher für die Dauer seines Aufenthaltes im Museum mit einem Gerät ausgestattet wird, welches ihm zu jedem Ausstellungsstück Zusatzinformationen über ein Head-Mounted Display, sowie über Kopfhörer zur Verfügung stellt.

(a) (b)

Abbildung 1.1: (a) zeigt ein Wearable Computing System Anfangs des 21.Jahrhunderts (vgl. [1], o.S.); (b) zeigt den Mobile Field Assistent der Firma Spintower, entwickelt 2010

Besonders durch die Miniaturisierung leistungsstarker Hardware wird die Forschungsarbeit im Bereich von Wearable Computing stark vorangetrieben. Während vor einigen Jahren die Rechenleistung bei Wearable Computing Systemen in Form eines Laptops – siehe Abbildung 1.1(a) – getragen werden musste, sind heute bereits Geräte möglich, welche die Funktionalität eines Desktop-PCs aufweisen, jedoch aufgrund ihres geringen Gewichtes vom Anwender am Körper getragen werden können und diesem somit das Arbeiten am PC an jedem beliebigen Ort ermöglichen (vgl. [2], S.13ff).

Ein wesentlicher Aspekt im Forschungsgebiet von Wearable Computing ist auch, dass das System den Benutzer in keiner Weise stören sollte. Das bedeutet, dass das Wearable Computing System nahezu autonom arbeiten sollte und der Anwender seine volle Aufmerksamkeit der gerade zu bewältigenden Aufgabe widmen kann. Die Waerable Computing Anwendung ist dazu mit verschiedensten Sensoren ausgestattet um seine Umwelt interpretieren zu können, sodass nur wenige Interaktionen mit dem Benutzer notwendig sind (vgl. [2], S.13ff).

Wearable Computing Systeme unterscheiden sich gegenüber konventionellen Computersystemen dadurch, dass der Anwender nicht mit der Bedienung des Computers beschäftigt ist, sondern dass das Computersystem den Anwender bei seiner Tätigkeit begleitet und so gut wie möglich unterstützt (vgl. [3], S.140).

1.1 Motivation

Das Unternehmen Spintower KG befasst sich mit der Entwicklung von qualitativ hochwertigen Wearable Computing Systemen und spezialisiert sich dabei auf Anwendungen, welche die Übertragung von Video-, Audio- und Nutzdaten in Echtzeit ermöglichen. Anwendungsbeispiele sollten dabei im Bereich von Sicherheitsdiensten, Rettungswesen und der mobilen Überwachung liegen.

Bereits im Juni 2010 konnte die Firma Spintower KG auf der RESEARCH AUSTRIA – eine Messe für Wirtschaft, Forschung und Innovation – eine erste Version eines Gerätes präsentieren, welches in der Lage ist, Video- und Audiodaten in Echtzeit über ein Mobilfunknetz zu übertragen (vgl. [4], o.S.). Als zentrale Recheneinheit wird bei diesem Gerät, welches unter dem Namen Mobile Field Assistent – siehe Abbildung 1.1(b) – entwickelt wird, der Mikrocontroller i.MX31 verwendet. Dieser Mikrocontroller wurde speziell für leistungsstarke Multimedia Anwendungen mit geringem Energieverbrauch vom Halbleiterhersteller Freescale entwickelt.

Die Signale zur Peripherie werden bei diesem ersten Prototypen jedoch analog übertragen. Dazu müssen zwei Leitungen für das Stereo-Audiosignal (`L`, `R`), zwei Leitungen für ein Mikrofon mit symmetrischer Signalführung (`MIC+`, `MIC-`), eine Leitung für das Videosignal (`CVBS`) und zwei weitere Leitungen zur Spannungsversorgung (`V+`, `GND`) zur Peripherie geführt werden. Geräte werden also mit einem sieben-poligen Kabel an den MFA angeschlossen, wobei dieses Kabel auch geschirmt sein muss, um Störungen am hochfrequenten Videosignal gering zu halten.

Die Videoausgabe erfolgt auf einem Display, welches im MFA eingebaut ist und somit die Videoausgabe nicht dezentral angezeigt werden kann. Die Übertragung von Steuer- oder Nutzdaten von der zentralen Steuereinheit zur Peripherie ist beim ersten Prototyp des MFAs noch nicht berücksichtigt.

Nachteile dieses Prototyps liegen darin, dass ein dickes und relativ starres Kabel zur Verbindung der Peripherie verwendet werden muss und dieses den Anwender des

MFAs bei seiner Bewegungsfreiheit einschränkt. Weiters verliert das Videosignal durch die Umwandlung von einem analogen Signal in ein digitales Signal – sodass es vom i.MX31 verarbeitet werden kann – an Qualität. Auch eine dezentrale Anzeige der Videoausgabe wäre in einem Wearable Computing System wünschenswert.

Die Motivation dieser Arbeit steckt darin, die digitalen Video-, Audio- und Datenschnittstellen der zentralen Recheneinheit an den Peripheriegeräten zur Verfügung zu stellen, sodass durch zusätzliche Digital/Analog-, Analog/Digital-Umwandlungen keine Qualitätsverluste entstehen. Das Problem bei dieser Idee liegt jedoch darin, da die Display- und Kameraschnittstellen des i.MX31 viele parallele Leitungen mit hochfrequenten Signalen aufweisen und somit sehr störanfällig sind.

1.2 Aufgabenstellung

Es soll ein bitserielles Kommunikationssystem unter dem Namen Remote Peripheral Interface Bus – kurz RPI-Bus – entwickelt werden, das in der Lage ist, gleichzeitig Video-, Audio- und Steuerdaten in Wearable Computing Applikationen zu übertragen. Die Übertragung dieser Daten soll dabei in einem Vollduplex-Modus erfolgen können und als zentrale Recheneinheit des Systems soll ein Entwicklungsboard der Firma Bluetechnix mit einem i.MX31 Coremodule dienen.

Die Anzahl der Busleitungen zwischen zentraler Recheneinheit und Peripherie soll sich dabei auf zwei Leitungen je Übertragungsrichtung beschränken. Somit kann die Flexibilität der physikalischen Busleitungen gewährleistet werden, welche bei einem breiteren Bus nicht gegeben wäre.

Zusätzlich soll die Spannungsversorgung der Peripheriegeräte bis zu einer gewissen Strombelastung über die Busleitungen erfolgen können. Besitzt ein Peripheriegerät eine höhere statische Stromaufnahme, so muss dieses Gerät extra über einen Akku versorgt werden.

Auch das An- bzw. Abstecken von Geräten am Bussystem soll während des Betriebs und ohne Benutzerinteraktion erfolgen und vom Coremodule verwaltet werden.

Das Bussystem soll außerdem den gleichzeitigen Betrieb von bis zu vier Geräten ermöglichen, wobei hier einige Vereinfachungen zu treffen sind. Aufgrund der Tatsache, dass das Coremodule nur ein Interface zum Anschluss eines Displays besitzt, wird das gleiche Videosignal an, allen am RPI-Bus angeschlossenen, Displays angezeigt. Sind am RPI-Bus mehrere CMOS-Kameras angeschlossen, so kann immer nur eine dieser Kameras aktiv sein, da das Coremodule auch nur ein CMOS-Sensor Interface aufweist. Außerdem ist nur jeweils eine Audioausgabe-, sowie eine Audioeingabequelle im gesamten System sinnvoll. Die Datenkommunikation zwischen Coremodule und Peripherie soll zwar gleichzeitig mit der Übertragung von Video- und Audiodaten erfolgen können, dennoch kann immer nur mit einem, der maximal vier Geräte, über diese Schnittstelle kommuniziert werden.

Zusammengefasst können für den RPI-Bus folgende Spezifikationen festgelegt werden.

1.3 Spezifikation des RPI-Busses

- zwei Busleitungen je Kommunikationsrichtung, um physikalische Flexibilität zu gewährleisten
- gleichzeitiger Betrieb von bis zu vier Peripheriegeräten
- Spannungsversorgung von Peripheriegeräten mit einer statischen Stromaufnahme von bis zu 500 mA über die Busleitungen
- Hot-Plugging der Peripheriegeräte
- eine Videoaus- und Videoeingabequelle, sowie eine Audioaus- und Audioeingabequelle im gesamten System
- Datenkommunikation immer nur mit einem der maximal vier Peripheriegeräten
- Ansteuerungen von Displays mit einer Auflösung von bis zu 800×600 px und einer Bildwiederholfrequenz von 60 fps

1.4 Entwicklungsumgebung

Als Entwicklungsumgebung für die Hardware des RPI-Busses soll ein Developementboard für den i.MX31 von der Firma Bluetechnix dienen. Bluetechnix bietet zu dem in Abbildung 1.2 dargestellten Entwicklungsboard auch Adapterplatinen zum Ansteuern von LCDs sowie von CMOS-Kameras an. Dazu werden alle zum Betrieb eines Displays oder einer Kamera notwendigen Signalleitungen zu den entsprechenden Steckverbindern geführt. Die Adapterplatinen werden an den gekennzeichneten Steckverbindern angeschlossen, wobei zusätzliche Steckverbinder die Kontaktierung eines jeden einzelnen Pins des i.MX31 ermöglichen.

Das Entwicklungsboard wird mit einer Gleichspannung von 12 V betrieben, aus denen zwei Spannungsdomänen (3.3 V und 5 V) für die verschiedenen Funktionsblöcke erzeugt werden. Während der Entwicklungsphase muss das Developementboard über Ethernet mit einem Host-PC verbunden werden. Am Host-PC wird einerseits das Kernel-Image für den i.MX31 erzeugt, welches schließlich über Ethernet geladen wird und andererseits befindet sich auch das Dateisystem des i.MX31 am Host-PC.

Bluetechnix bietet zusätzlich zu den Adapterplatinen auch Unit-Tests zum Testen der verschiedenen Funktionsblöcke des i.MX31 an.

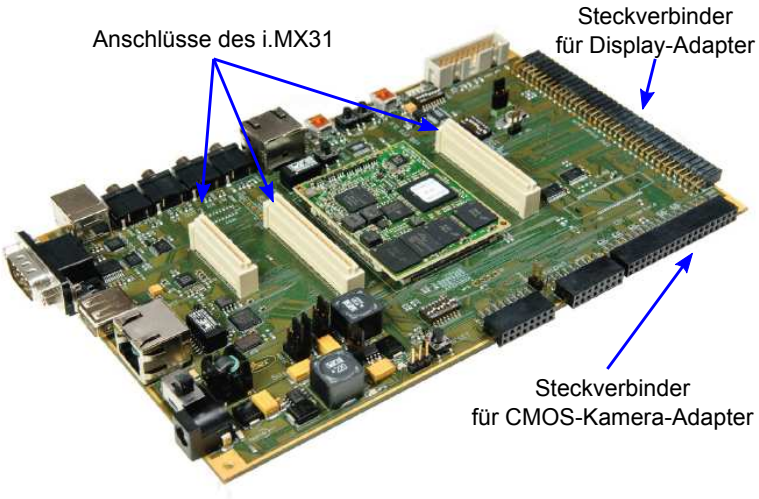


Abbildung 1.2: i.MX31 Entwicklungsboard mit gekennzeichneten Steckverbindern zum Anschluss der Adapterplatinen (vgl. [5], S.1)

Um diese Unit-Tests ausführen zu können, muss das Entwicklungsboard zusätzlich über eine USB Schnittstelle mit dem Host-PC verbunden werden. Der Host-PC erkennt diesen Anschluss als COM-Port und somit kann mit Hilfe eines Terminalprogramms und den korrekten Einstellungen für den seriellen Port auf die Konsole des i.MX31 zugegriffen werden. Über die Konsole können schließlich die besagten Unit-Tests gestartet werden.

2 Serializer - Deserializer Konzept

Die Motivation im Hinblick auf den RPI-Bus steckt darin, eine Vielzahl an parallel vorliegenden, digitalen Signalen über eine Zweidrahtleitung zu übertragen. Halbleiterhersteller wie Texas Instruments oder MAXIM bieten bereits seit Jahren Chiplösungen an, welche die Übertragung von mehreren parallelen Leitungen über ein differentielles Leitungspaar ermöglichen. Diese Systeme wurden vor allem für den automotiven Bereich zur Übertragung des Videosignals einer Rückfahrkamera zu einem Display im Cockpit entwickelt.

Prinzipiell bestehen diese Systeme immer aus einem Serializer/Deserializer Paar. Der Serializer besitzt mehrere parallele, digitale Eingänge, welche mit einem ausreichend hohen Taktsignal eingelesen werden und schließlich als serieller Bitstrom zum Deserializer übertragen werden, der aus dem eingelesenen Bitstrom wieder die ursprünglich parallel vorliegenden Daten rekonstruiert. Damit der Deserializer die Daten richtig ausgeben kann, muss das Taktsignal – mit dem der Serializer die Daten einliest – auch beim Deserializer vorliegen. Um dieses Taktsignal auch dem Deserializer zur Verfügung stellen zu können, existieren verschieden Lösungsansätze.

Unterscheidungen bei den verschiedenen SerDes-Konzepten können jedoch auch bei den Ein- und Ausgangskompatibilitäten getroffen werden. Die parallelen Eingänge eines Serializers bzw. die parallelen Ausgänge eines Deserializers sind meistens mit dem LVCMOS-Standard kompatibel. Der serielle Bitstrom wird schließlich über eine differentielle Schnittstelle übertragen, wobei hier vorwiegend die Standards LVDS, CML oder LVPECL verwendet werden.

2.1 Differentielle Schnittstellen bei SerDes

2.1.1 Funktionsweise und Eigenschaften differentieller Schnittstellen

Bei differentiellen Schnittstellen werden die Daten symmetrisch übertragen. Das bedeutet, dass die Signale im Vergleich zur unsymmetrischen Datenübertragung anstatt über eine Signalleitung, mittels eines differentiellen Leitungspaares übertragen werden. Das Signal auf einer Leitung ist stets komplementär zum Signal auf der

anderen Leitung des Leitungspaares und die Nutzinformation steckt somit in der Differenz der beiden Signale.

Der wesentliche Vorteil bei einer symmetrischen Übertragung liegt in der erhöhten Störfestigkeit im Vergleich zur unsymmetrischen Übertragung. Während bei der unsymmetrischen Übertragung eingekoppelte Störungen die Daten verfälschen können, bleibt das Signal bei einer symmetrischen Übertragung meist unverfälscht. Störungen wirken sich bei der symmetrischen Datenübertragung meist auf beide Signalleitungen gleich aus, wodurch sich das Differenzsignal einer gestörten Übertragung nicht von dem einer ungestörten Übertragung unterscheidet (vgl. [6], S.1-4ff).

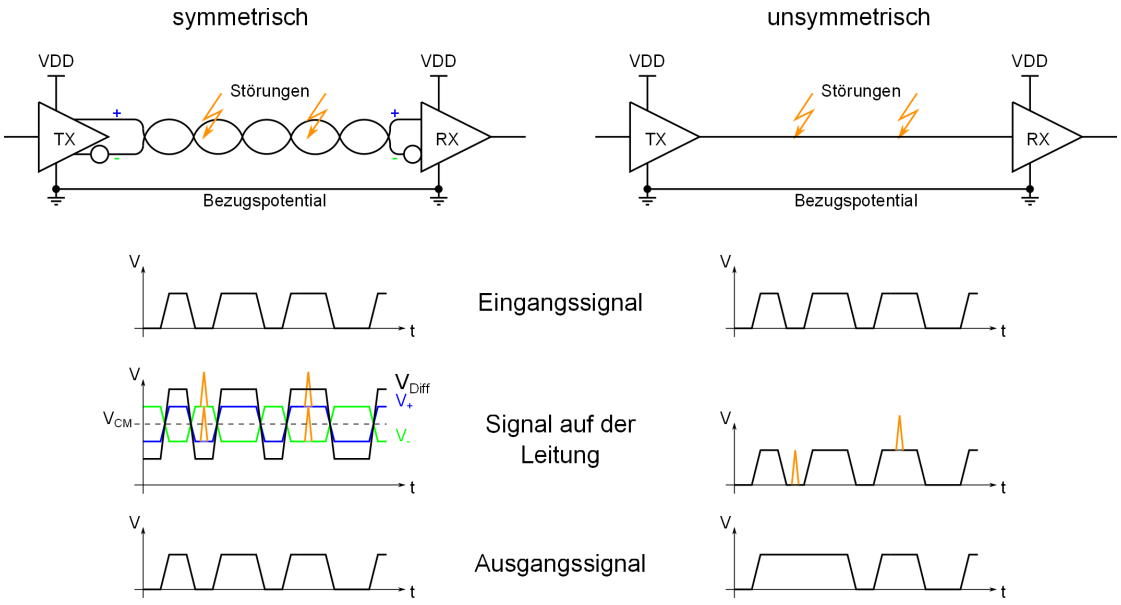


Abbildung 2.1: Einfluss von Störungen auf die Signalintegrität bei symmetrischer und unsymmetrischer Datenübertragung

Neben der höheren Störfestigkeit bieten differentielle Schnittstellen im Vergleich zu Single-Ended Schnittstellen auch noch den Vorteil der niedrigeren Störabstrahlung, da sich bei richtiger Signalführung die Gleichtaktstörungen auf dem differentiellen Leitungspaar nahezu auslöschen. Ein Grund dafür, dass die Störabstrahlung trotzdem ungleich Null ist, liegt darin, dass sich meist Signalanstiegs- und Signalabfallszeiten unterscheiden.

Da bei der symmetrischen Datenübertragung der Spannungspegel auf einer Leitung dem Spannungspegel mit entgegengesetztem Vorzeichen auf der anderen Leitung entspricht, müssen Sender und Empfänger sowohl mit einer positiven als auch mit einer negativen Betriebsspannung versorgt werden. Abhilfe dafür schafft das Auslenken des differentiellen Signals um eine definierte Gleichspannung, sodass durch die Differenzspannung niemals das Bezugspotential des Senders oder des Empfängers unterschritten werden kann. In Abbildung 2.1 ist diese Gleichspannung als V_{CM} beim Signalverlauf auf den Leitungen einer symmetrischen Datenübertragung eingezeichnet.

Problematisch wirkt sich nun nur noch die galvanische Kopplung des Senders und des Empfängers über das gemeinsame Bezugspotential aus. Störströme über die gemeinsame Masse verschieben das Bezugspotential von Sender und Empfänger zueinander und somit verschiebt sich auch die Gleichspannung V_{CM}, wodurch es zu einer Falschinterpretation der Daten beim Empfänger kommen kann. Wird jedoch das zu übertragende Signal so modifiziert, dass es beinahe keinen Gleichspannungsanteil aufweist, indem einerseits die Anzahl an High- und Low-Bits über die Zeit möglichst ausgeglichen gehalten wird und andererseits nur wenige Bits gleicher Art hintereinander auftreten, so können Sender und Empfänger über serielle Kapazitäten vom Übertragungsmedium entkoppelt werden. Man spricht hierbei von einer AC-Kopplung zwischen Sender und Empfänger (vgl. [7], S.32f).

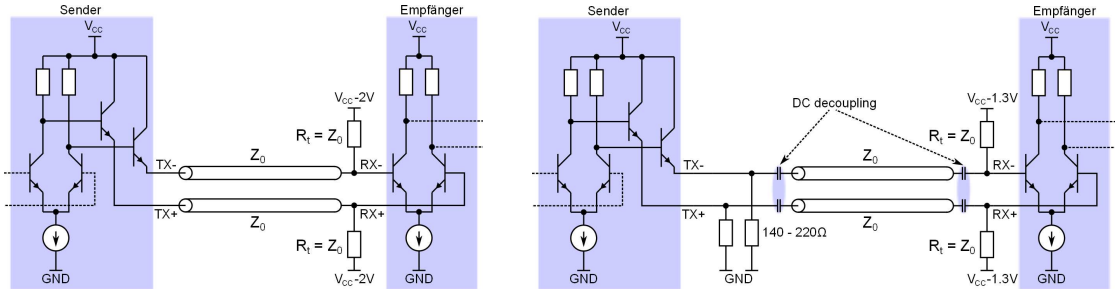
Die vorhin erwähnte Gleichspannung V_{CM}, um die das differentielle Signal ausgelenkt wird, wird über die seriellen Kapazitäten nicht zum Empfänger übertragen, sondern nur das hochfrequente Differenzsignal. Der Empfänger, der ebenfalls mit seriellen Kapazitäten von den Übertragungsleitungen entkoppelt ist, beaufschlagt seine Eingangstruktur mit der gleichen Gleichspannung V_{CM}, wodurch die empfangenen Daten wieder einen positiven Offset bekommen und somit auf eine negative Spannungsversorgung beim Empfänger verzichtet werden kann. Die beim Empfänger generierte Gleichspannung ist nun auch unabhängig von der am Sender, wodurch sich Störströme über das gemeinsame Bezugspotential nicht mehr auf die Übertragung auswirken.

Bei den in SerDes Applikationen verwendeten differentiellen Schnittstellen werden, wie bereits erwähnt, hauptsächlich die Standrads LVDS, CML und LVPECL verwendet, auf die nun näher eingegangen wird.

2.1.2 Low-Volatage Positive-Emitter-Coupled Logic – LVPELCL

Beim LVPECL Standard handelt es sich um eine Weiterentwicklung des ECL Standards, der aufgrund seines Spannungsversorgungsbereiches inkompatibel mit anderen Logikpegelstandards ist. Die LVPECL Interface-Struktur hingegen wird mit einer Gleichspannung von 3.3 V betrieben. Als Ausgangsstruktur bei einem LVPECL Sender wird eine Differenzverstärkerstufe aus Bipolartransistoren verwendet, deren Emitter über eine Stromsenke mit GND verbunden sind. Die Ausgänge der Differenzverstärkerstufe sind mit Emitterfolgern verbunden, welche schließlich den Strom für die Ausgangsstufe liefern (vgl. [8], S.1f).

Eine LVPECL Eingangsstruktur besteht wiederum aus einer Differenzverstärkerstufe, wobei die Emitter der BJTs wieder über eine gemeinsame Stromsenke mit GND verbunden sind. Die Eingänge der Differenzverstärkerstufe entsprechen den differentiellen Signalleitungen (vgl. [9], S.6).



(a) (b)

Abbildung 2.2: Ein- und Ausgangsstruktur von LVPECL Sender und Empfänger; (a) Leitungsterminierung von LVPECL Sender und Empfänger bei DC-Kopplung; (b) Leitungsterminierung von LVPECL Sender und Empfänger bei AC-Kopplung; (vgl. [9], S.5f)

In Abbildung 2.2 sind die Strukturen von LVPECL Sendern und Empfängern dargestellt, wobei bei der Terminierung des LVPECL Senders unterschieden wird, ob Sender und Empfänger über eine gemeinsame Gleichspannung V_{CM} gekoppelt sind. Abbildung 2.2(a) zeigt eine LVPECL Applikation, bei der Sender und Empfänger über eine gemeinsame Gleichspannung verkoppelt sind. Die Leitungsterminierung wird in diesem Fall durch zwei Widerstände durchgeführt, die den gleichen Werte wie der Wellenwiderstand des Übertragungsmediums aufweisen sollten und beim Empfänger platziert werden. Der Abschluss erfolgt gegen eine Gleichspannung ($V_{CC}-2\,V$) und reduziert Leitungsreflexionen, die aufgrund von Impedanzänderungen auf dem Übertragungsweg entstehen.

Ist eine gleichspannungsfreie Signalübertragung möglich, so werden LVPECL Sender und Empfänger wie in Abbildung 2.2(b) terminiert. Der LVPECL Sender wird über Widerstände im Bereich von $140\,\Omega$ - $220\,\Omega$ gegen GND abgeschlossen. Der LVPECL Empfänger wird hingegen mittels Widerständen in der Größenordnung des Wellenwiderstandes vom Übertragungsmedium gegen eine Gleichspannung ($V_{CC}-1.3\,V$) terminiert und das Differenzsignal wird somit beim Empfänger wieder um eine Gleichspannung V_{CM} ausgelenkt (vgl. [9], S.6).

Der wesentliche Vorteil bei der LVPECL Technologie liegt darin, dass Übertragungsgeschwindigkeiten von bis zu $10\,Gb/s$ möglich sind. Dies wird durch hohe Umladeströme gewährleistet, welche die Pegelwechsel auf der Übertragungsleitung veranlassen. Mit den höheren Strömen steigt jedoch auch die Leistungsaufnahme der Treiberstrukturen, was wiederum ein Nachteil bei dieser Technologie ist (vgl. [7], S.12).

2.1.3 Current-Mode Logic – CML

Bei CML handelt es sich um eine der einfachsten differentiellen Schnittstellen, da die Abschlussnetzwerke meist schon in die Sender und Empfänger integriert sind,

wodurch die Anzahl an benötigten, externen Bauteile und somit der Platzbedarf und die Kosten reduziert werden. CML Schnittstellen ermöglichen – ebenfalls wie LVPECL Schnittstellen – Übertragungsgeschwindigkeiten von bis zu 10 Gb/s, bei jedoch niedrigerer Leistungsaufnahme (vgl. [8], S.3).

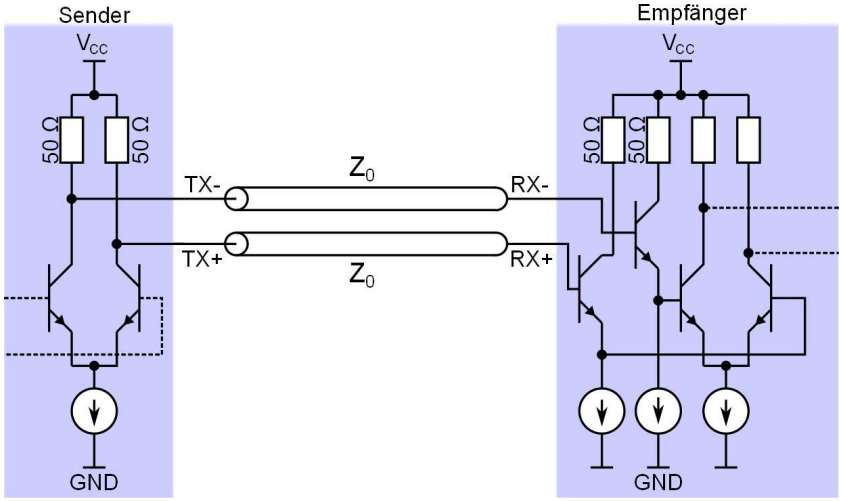


Abbildung 2.3: Ein- und Ausgangsstruktur von CML Sender und Empfänger (vgl. [8], S.3)

In Abbildung 2.3 sind die Strukturen eines CML Senders und Empfängers dargestellt. Die CML Ausgangsstufe bildet im wesentlichen wieder eine Differenzverstärkerstufe aus BJTs mit 50 Ω Kollektorwiderstände und einer Stromsenke bei den Emittern. Die Kollektorwiderstände bilden dabei die Ausgangsimpedanz des CML Senders.

Die Eingangsstufe des CML Senders besitzt ebenfalls eine Impedanz von 50 Ω und setzt sich aus Emitterfolgern und einer nachgeschalteten Differenzverstärkerstufe zusammen (vgl. [8], S.3).

2.1.4 Low-Voltage Differential Signaling – LVDS

Während die Ein- und Ausgangsstrukturen von LVPECL und CML Sendern und Empfängern sehr ähnlich sind, sind die Strukturen beim LVDS Standard gänzlich anders. Im Gegensatz zu LVPECL und CML werden die Ein- und Ausgangsstrukturen bei LVDS in CMOS-Technologie gefertigt und auch die Stromsenken besitzen eine wesentlich niedrigere Stromergiebigkeit. Somit weist der LVDS Standard die niedrigste Leistungsaufnahme, aber gleichzeitig auch die kleinste Datenübertragungsrate – von bis zu 3.125 Gb/s – auf.

Die Ein- und Ausgangsstruktur eines LVDS Senders und Empfängers ist in Abbildung 2.4 dargestellt. Die Gates der MOST `MP1` und `MN1`, sowie `MP2` und `MN2` sind jeweils miteinander verbunden und werden mit einem komplementären Signal angesteuert. Das bedeutet, dass immer nur einer der beiden PMOST und NMOST

leitend sein kann. Werden die Transistoren so angesteuert, dass `MP2` leitend ist, so sperrt der PMOST `MP1`. Gleichzeitig sperrt auch der NMOST `MN2`, der Transistor `MN1` befindet sich jedoch im leitenden Arbeitsbereich (vgl. [7], S.9ff).

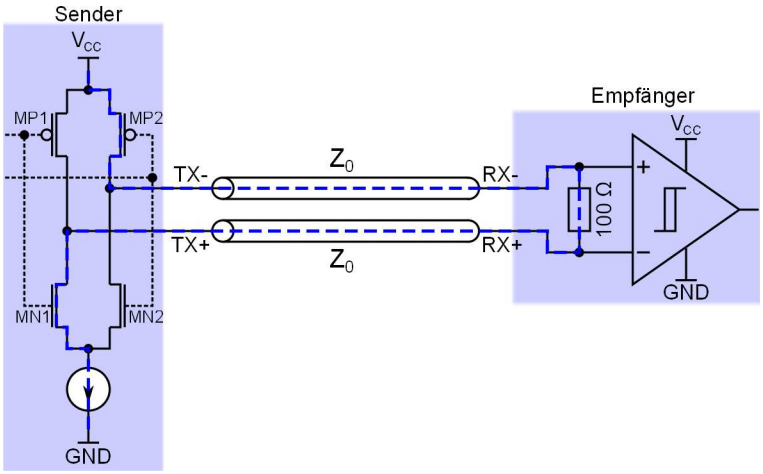


Abbildung 2.4: Ein- und Ausgangsstrukturen von LVDS Sender und Empfänger (vgl. [7], S.9)

Es kommt zu einem Stromfluss wie in Abbildung 2.4 blau strichliert eingezeichnet. Der Stromfluss verursacht am 100 Ω Terminierungswiderstand des Empfängers einen Spannungsabfall, der von einem Schmitt-Trigger je nach Stromrichtung in einen der logischen Zustände High oder Low umgewandelt wird.

2.1.5 Zusammenfassung

In den Abschnitten 2.1.2 bis 2.1.4 wurden kurz die Vor- und Nachteile der verschiedenen differentiellen Schnittstellen – wie sie bei SerDes-Applikationen verwendet werden – erläutert. Im Hinblick auf den RPI-Bus, welcher als Bussystem in Wearable Computig Applikationen Anwendung finden sollte, ist vor allem die Leistungsaufnahme der Schnittstellentreiber interessant, da als Energieversorgung nur ein Akku zur Verfügung steht. Eine geringe Leistungsaufnahme der Schnittstellentreiber geht jedoch immer mit einer niedrigeren Datenübertragungsrate und niedrigeren Differenzspannungen einher. Anhand der unter Punkt 1.3 getroffenen Spezifikationen kann die maximale Datenrate vom RPI-Bus mit 1 Gb/s abgeschätzt werden, wodurch sich zeigen lässt, dass der LVDS Standard im Hinblick auf die maximal mögliche Übertragungsgeschwindigkeit völlig ausreicht. Aufgrund der niedrigeren Differenzspannung bei diesem Standard, wird jedoch auch die maximal mögliche Leitungslänge des Bussystems beschränkt. Da das RPI-Bussystem am Körper getragen wird, sind nur geringe Leitungslängen notwendig und somit kann der Nachteil von geringeren Differenzspannungen beim LVDS Standard vernachlässigt werden. Es ist also ersichtlich, dass SerDes Anwendungen welche das differentielle Signal über ein LVDS Interface übertragen am besten für den RPI-Bus geeignet sind, da die Nachteile von LVDS noch nicht zum Tragen kommen, der große Vorteil der

geringsten Leistungsaufnahme jedoch schon. In Tabelle 2.1 sind die Eigenschaften der verschiedenen Standards nochmals gegenübergestellt.

Standard	Datenrate	Differenzsp.	Leistungsaufnahme
LVPECL	10 Gb/s	± 800 mV	mittel bis hoch
CML	10 Gb/s	± 800 mV	mittel
LVDS	3.125 Gb/s	± 350 mV	niedrig

Tabelle 2.1: Gegenüberstellung des LVPECL, CML und LVDS Standards im Hinblick auf maximale Datenrate, Differenzspannung und Leistungsaufnahme (vgl. [7], S.9)

2.2 SerDes Architekturen

Im Nachfolgenden werden die drei grundlegenden Architekturen von Serializer/Deserializer Paaren beschrieben und es werden deren Vor- und Nachteile angeführt. Abschließend wird eine Bewertung der verschiedenen Konzepte im Hinblick auf deren Verwendbarkeit beim RPI-Bus durchgeführt.

2.2.1 Parallel Clock SerDes

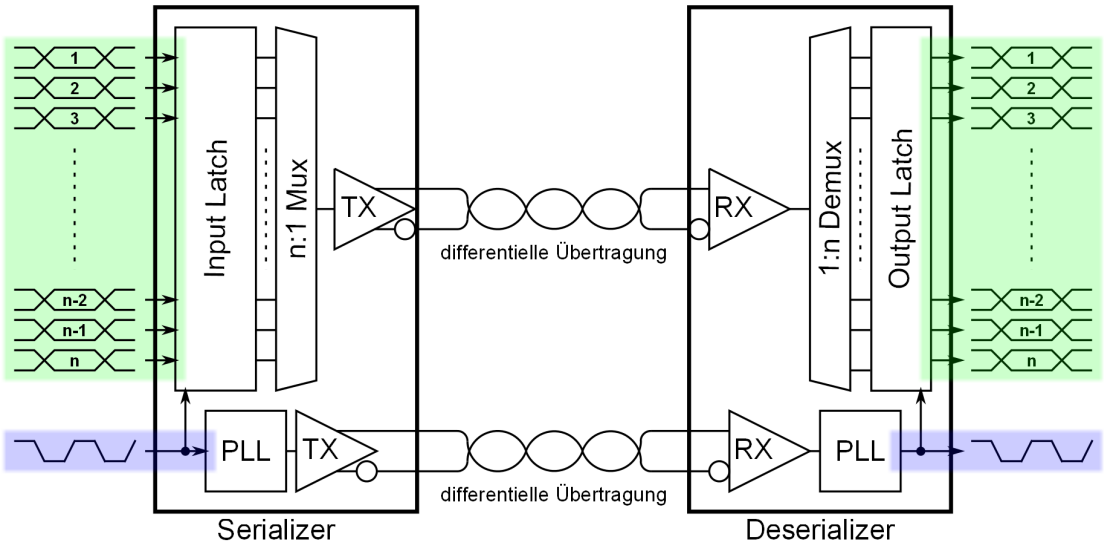


Abbildung 2.5: Parallel Clock SerDes Architektur (vgl. [7], S.19)

Bei dieser Serializer/Deserializer Architektur wird parallel zum seriellen Bitstrom auch das Taktsignal, mit welchem beim Serializer die Daten eingelesen werden, zum Deserializer übertragen. Sowohl der serialisierte Bitstrom, als auch das Taktsignal werden dabei über ein differentielles Leitungspaar übertragen. Der Nachteil bei dieser Architektur liegt also darin, dass der Deserializer über mindestens vier Leitungen mit dem Serializer verbunden werden muss. In Bezug auf den RPI-Bus kann diese SerDes Architektur ausgeschlossen werden, da laut den Spezifikationen 1.3 nur zwei Leitungen je Kommunikationsrichtung verwendet werden dürfen. In Abbildung 2.5 ist die prinzipielle Architektur eines Serializer/Deserializer Paares zu sehen, bei dem das Taktsignal parallel zum Datenstrom mitgeführt wird (vgl. [7], S.19).

2.2.2 8b/10b SerDes

Wie die Bezeichnung schon andeutet, übertragen diese Serializer den seriellen Datenstrom im 8b/10b Code. Bei diesem Kodierungsverfahren werden jeweils 8 Bit auf 10 Bit abgebildet. Die 10 Bit Symbole sind so gewählt, dass sich die Anzahl der Nullen um maximal zwei von der Anzahl der Einsen unterscheidet. Somit ergibt sich ein nahezu gleichspannungsfreier Bitstrom, da die Anzahl der Einsen gleich der Anzahl der Nullen gehalten wird. Da des Weiteren maximal fünf gleiche Bits, durch die Aneinanderreihung der 10 Bit Symbole, hintereinander auftreten können, wird es dem Deserializer mit einem zusätzlichen Referenztakt ermöglicht, das ursprüngliche Taktsignal aus dem seriellen Bitstrom abzuleiten (vgl. [7], S.21).

Nachteilig bei dieser Architektur wirkt sich einerseits die Notwendigkeit eines Referenztakts beim Deserializer aus und andererseits auch der Datenoverhead von 25 %, der durch das Umsetzen eines 8 Bit Wortes auf ein 10 Bit Wort entsteht. Nebenbei werden die 10 Bit Codes zwischen Steuerzeichen im seriellen Bitstrom eingebettet, sodass der Deserializer die einzelnen 10 Bit Wörter voneinander trennen kann.

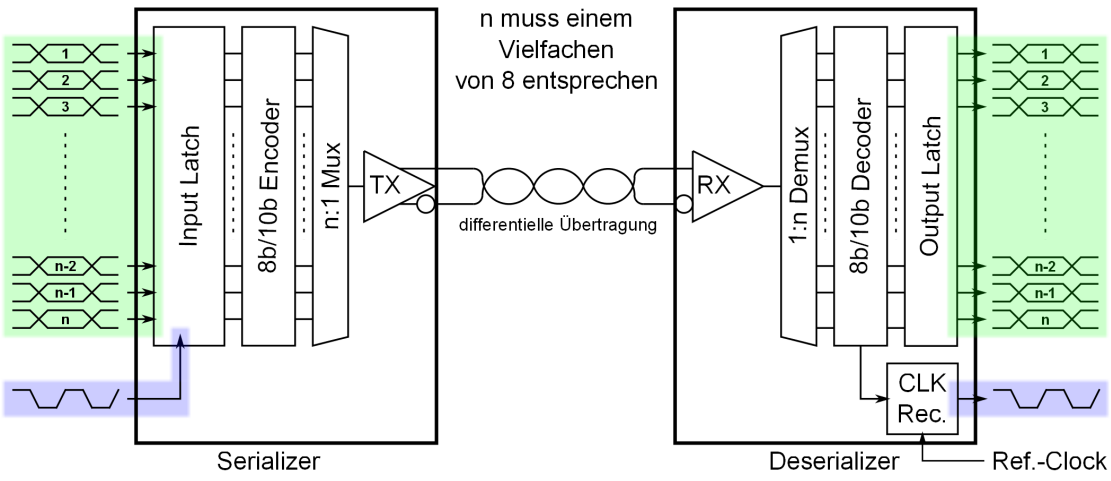


Abbildung 2.6: 8b/10b SerDes Architektur (vgl. [7], S.21)

2.2.3 Embedded Clock SerDes

Als dritte Variante sind noch Serializer/Deserializer Paare anzuführen, bei denen der Takt – mit dem die Daten beim Serializer eingelesen werden – in den seriellen Bitstrom eingebettet wird. Dazu wird jeweils zu Beginn des zu übertragenden Datenpakets ein High-Bit und am Ende ein Low-Bit eingefügt. Durch die Aneinanderreihung der Datenpakete ergeben sich somit periodische ⎍-Flanken, wodurch sich der Deserializer auf das Taktsignal des Serializers synchronisieren kann (vgl. [7], S.20).

Im Vergleich zur 8b/10b SerDes Architektur, benötigen Systeme mit eingebettetem Taktsignal keinen Referenztakt zur Taktrückgewinnung und auch der Anteil an Nutzdaten im seriellen Bitstrom ist höher als der bei 8b/10b SerDes.

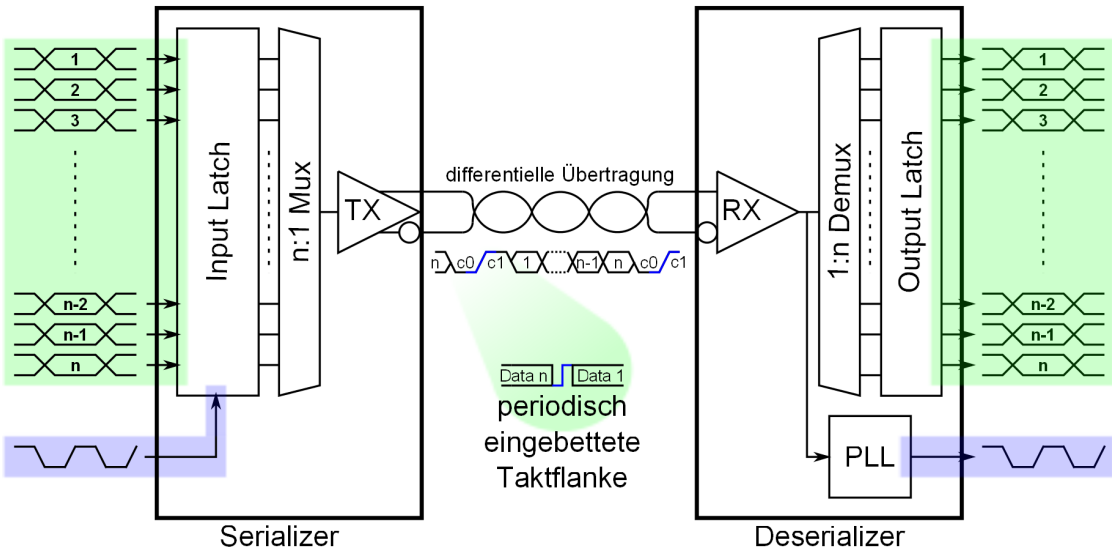


Abbildung 2.7: Embedded Clock SerDes Architektur (vgl. [7], S.20)

2.2.4 Zusammenfassung

Aus den Betrachtungen der Punkte 2.2.1 bis 2.2.3 ist ersichtlich, dass besonders die Embedded Clock SerDes Architektur im Hinblick auf die Problemlösung beim RPI-Bus geeignet ist. Der Serializer ist – im Gegensatz zur Parallel Clock SerDes Architektur – nur über ein differentielles Leitungspaar mit dem Deserializer verbunden, wodurch die Spezifikation mit je zwei Leitungen pro Übertragungsrichtung erfüllt werden kann.

Im Vergleich zur 8b/10b SerDes Architektur kann zusätzlich auf die Notwendigkeit eines Referenztaktes verzichtet werden, wodurch sich der Aufwand beim Schaltungsentwurf verringert und auch der Anteil an Nutzdaten im seriellen Bitstrom ist bei der unter Punkt 2.2.3 vorgestellten Variante höher als bei 8b/10b SerDes.

Somit können beim Embedded Clock Ansatz bei niedrigeren Taktfrequenzen gleich viele Nutzdaten übertragen werden, wie bei der 8b/10b SerDes Variante, wodurch in weiterer Folge auch mit einer geringeren Leistungsaufnahme des Serializers und des Deserializers gerechnet werden kann.

3 Konzeptentwurf

Das RPI-Bussystem muss die gleichzeitige Anbindung von Videoausgabe-, Videoeingabe, Audioausgabe-, sowie von Audioeingabegeräten ermöglichen. Zusätzlich soll über eine Datenschnittstelle die Steuerung aller Peripheriegeräte, sowie die Kommunikation mit verschiedensten Sensoren ermöglicht werden. Die Kommunikationsrichtung zur Übertragung der Video-, Audio- und Datenausgabe zu den Peripheriegeräten wird in weiterer Folge als Downchannel bezeichnet, während die entgegengesetzte Übertragungsrichtung als Upchannel definiert wird.

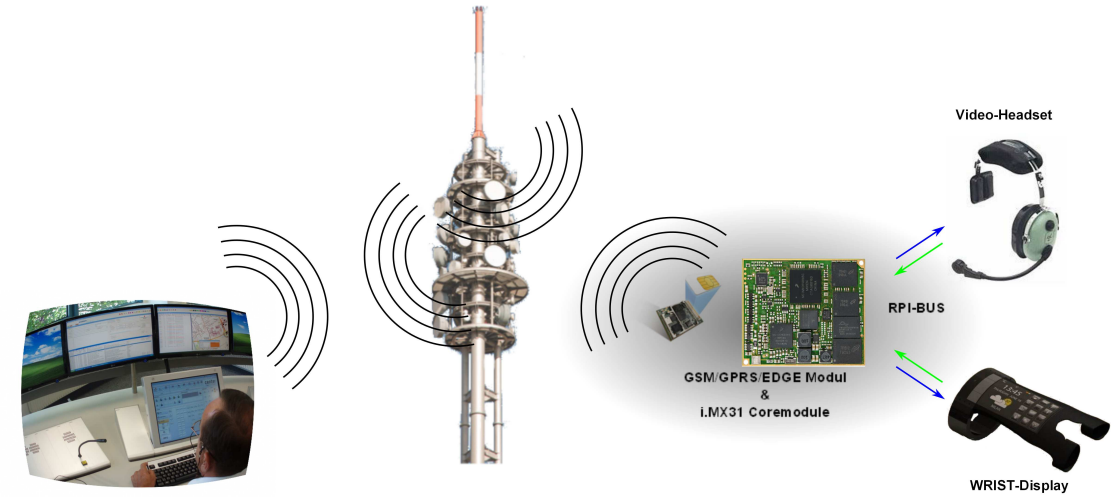


Abbildung 3.1: Typisches Szenario in dem das RPI-Buskonzept Anwendung finden kann; die Wearables wie beispielsweise ein WRIST-Display oder ein Video-Headset werden über den RPI-Bus mit einer zentralen Recheneinheit verbunden (vgl. [10], [11], [12], [13], [14])

In Abbildung 3.1 ist ein typisches Szenario für den RPI-Bus dargestellt. An ein mobiles Gerät – welches als zentrale Recheneinheit ein i.MX31 Coremodule besitzt – werden über den RPI-Bus ein Display, sowie ein Video-Headset angeschlossen.

Das Wearable Computing System verfügt außerdem über ein GSM/GPRS/EDGE-Modul, welches die Funkanbindung in ein Mobilfunknetz ermöglicht. Über diese Funkverbindung kann ein Experte, der sich weit entfernt in einem Kompetenzzentrum befindet, dem Anwender der Wearable Computing Applikation Hilfestellung über die Kopfhörer bzw. über das Display geben. Zusätzlich erhält der Experte mittels Audio- und Videoaufnahmen einen Eindruck von der Situation, in der sich

der Anwender des Wearable Computing Systems befindet, wodurch der Kommunikationsaufwand zwischen Experte und Anwender reduziert wird.

Der i.MX31 wird nun im Hinblick auf seine Funktionsblöcke untersucht und es wird gezeigt, welche Schnittstellen und Möglichkeiten zur Verfügung stehen, um die Übertragung der geforderten Daten zu realisieren. Dazu ist in Abbildung 3.2 ein stark vereinfachtes Blockschaltbild des i.MX31 mit allen wichtigen Funktionsblöcken ersichtlich, wobei die für den RPI-Bus benötigten Blöcke blau eingefärbt sind.

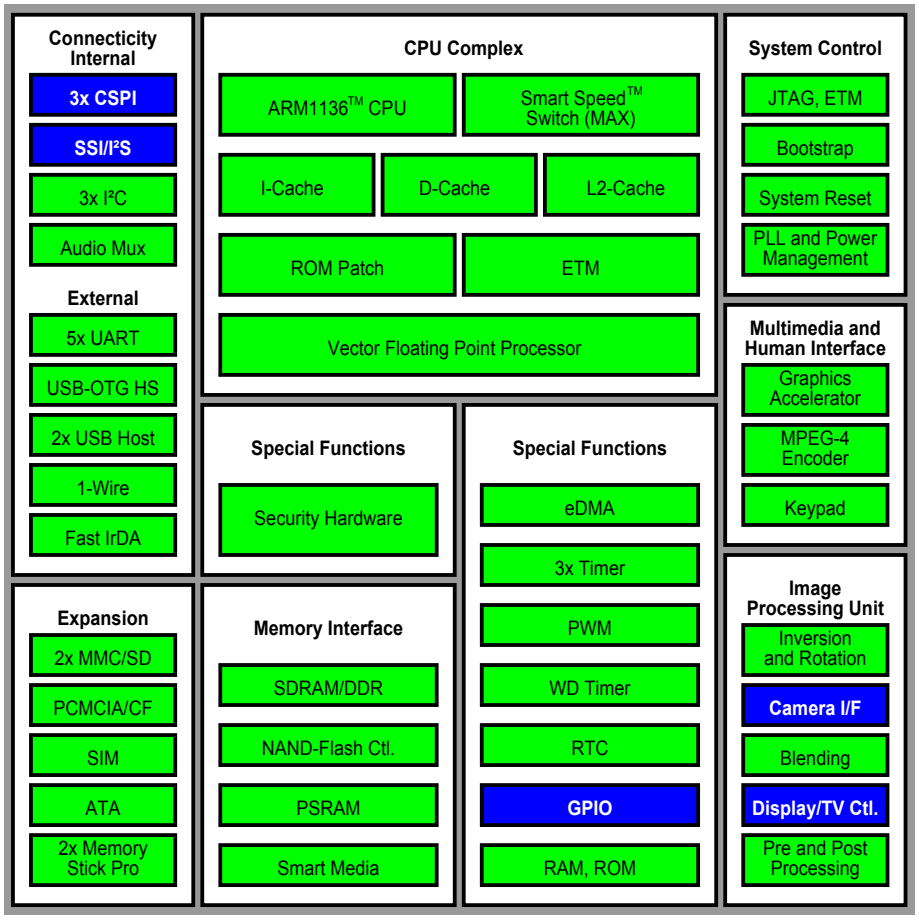


Abbildung 3.2: stark vereinfachtes Blockschaltbild des i.MX31 (vgl. [15], S.1-3)

3.1 Image Processing Unit – IPU

Die Image Processing Unit des i.MX31 ermöglicht die Bild- und Videoverarbeitung und bietet dazu eine Schnittstelle zur Ansteuerung einer CMOS-Kamera, sowie Schnittstellen zum Ansteuern von asynchronen oder auch synchronen Displays an.

3.1.1 Display Interface – Display/TV Ctl.

Die Videoausgabe kann beim i.MX31 mittels synchronen oder asynchronen Displays erfolgen. Asynchrone Displays weisen im Gegensatz zu synchronen Displays einen Speicher auf, in dem das aktuelle Anzeigebild abgelegt wird. Wird der angezeigte Bildinhalt nicht verändert, so müssen einem asynchronen Display keine Anzeigeinformationen übermittelt werden, da diese aus dem Speicher abgerufen werden können. Aufgrund der höheren Komplexität asynchroner Displays sind diese im Vergleich zu synchronen, oder sogenannten Dumb-Displays, wesentlich teurer. Daher werden in Endprodukten bevorzugt synchrone Displays verbaut. Der Nachteil bei Dumb-Displays liegt aber darin, dass kontinuierlich die Anzeigeinformation zur Verfügung gestellt werden muss (vgl. [15], S.44-4ff).

Für die weiteren Betrachtungen beim Konzeptentwurf des RPI-Busses wird davon ausgegangen, dass Peripheriegeräte mit synchronen Displays ausgestattet sind. Um diese Art von Displays ansteuern zu können, stellt die Image Processing Unit des i.MX31 die in Tabelle 3.1 angeführten Steuer- und Datenleitungen zur Verfügung.

Name	Typ	Funktion
DISPB_D3_DATA[17:0]	Daten	RGB Pixeldaten
DISPB_D3_VSYNC	Steuerung	signalisiert dem Display den Beginn eines neuen Frames
DISPB_D3_HSYNC	Steuerung	signalisiert dem Display den Beginn einer neuen Zeile
DISPB_D3_CLK	Takt	Pixeltakt
DISPB_D3_DRDY	Steuerung	signalisiert Gültigkeit der anliegenden Pixeldaten
DISPB_D3_SPL	Steuerung	nur für HR-TFT Displays erforderlich
DISPB_D3_CLS	Steuerung	nur für HR-TFT Displays erforderlich
DISPB_D3_REV	Steuerung	nur für HR-TFT Displays erforderlich

Tabelle 3.1: Takt-, Steuer- und Datensignale des Display Interfaces des i.MX31

Zusammengefasst kann also festgehalten werden, dass für den Downchannel zur Videoausgabe 22 parallele Signalleitungen notwendig sind.

In Abbildung 3.3 ist das Timing-Diagramm, der vom Synchronous Display Controller (SDC) generierten Signale, zum Ansteuern eines Dumb-Displays dargestellt.

Das Signal DISPB_D3_VSYNC signalisiert dem angesteuerten Display mit einer ⌝-Flanke den Beginn eines neuen Frames und die einzelnen Bildpunkte des Displays werden zeilenweise beschrieben. ⌝⌟-Pulse von DISPB_D3_HSYNC signalisieren dabei

einen Zeilenwechsel. Zwischen zwei ⎿-Übergängen von DISPB_D3_VSYNC müssen beispielsweise bei einer Display-Auflösung von 320×240 Pixel, 240 ⎿⎍-Pulse von DISPB_D3_HSYNC auftreten. Nach einem ⎿⎍-Puls von DISPB_D3_HSYNC wird mit einer ⎿-Flanke von DISPB_D3_DRDY die Gültigkeit des Pixeltakts signalisiert und es werden sequentiell die 18 parallelen Farbwerte jedes Pixels einer Zeile an den Signalleitungen DISPB_D3_DATA[17:0] angelegt, welche dann bei ⎿-Flanken von DISPB_D3_CLK vom Display übernommen werden. Durch entsprechende Konfiguration der IPU kann jedoch auch festgelegt werden, dass die Pixeldaten bei ⎍-Flanken gültig sind.

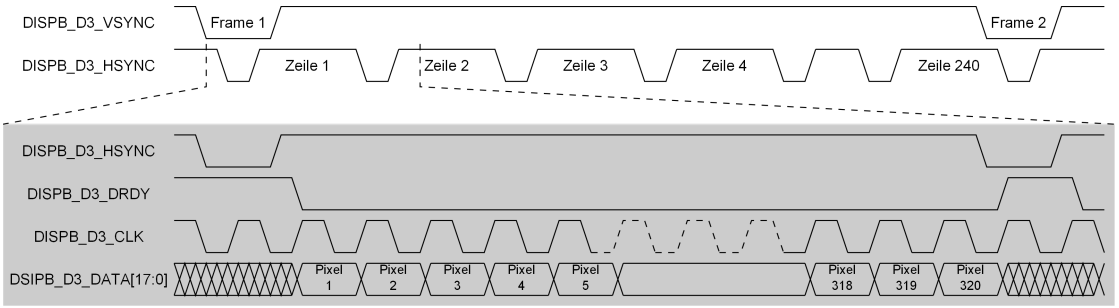


Abbildung 3.3: Timing-Diagramm des Display Interfaces des i.MX31 mit Detailansicht der Signalverläufe einer Displayzeile mit 320 Pixel (vgl. [16], S.61)

3.1.2 CMOS Sensor Inteface – Camera I/F

Zum Anschluss einer CMOS-Kamera weist die Image Processing Unit ein COMS Sensor Interface auf. Diese Schnittstelle kann in drei verschiedenen Modi betrieben werden (vgl. [16], S.57ff):

- Pseudo BT.656 Video Mode,
- Gated Clock Mode und
- Non-Gated Clock Mode

Da die zu verwendende CMOS-Kamera (OV2640 von Omnivison) im Gated Clock Mode arbeitet, wird nur diese Betriebsart beschrieben. In diesem Fall nimmt die CMOS Sensor Schnittstelle der Image Processing Unit die in Tabelle 3.2 angeführten Signale entgegen.

Name	Typ	Funktion
SENSB_DATA[9:0]	Daten	RGB Pixeldaten
SENSB_PIX_CLK	Takt	Pixeltakt

Fortsetzung auf nächster Seite

Tabelle 3.2 – *fortgesetzt von vorheriger Seite*

Name	**Typ**	**Funktion**
`SENSB_VSYNC`	Steuerung	signalisiert den Beginn eines neuen Frames
`SENSB_HSYNC`	Steuerung	signalisiert den Beginn einer neuen Zeile; Pixeltakt ist nur gültig solange dieses Signal High ist

Tabelle 3.2: Takt-, Steuer- und Datensignale des CMOS-Kamera Interfaces des i.MX31

In Abbildung 3.4 ist schließlich das Timing-Diagramm der CMOS Sensor Schnittstelle im Gated Clock Mode ersichtlich. Diesmal kennzeichnet ein ⎍-Puls von `SENSB_VSYNC` den Beginn eines neuen Frames. Mit einer ⎍-Flanke von `SENSB_HSYNC` wird der Beginn einer neuen Zeile signalisiert und die Gültigkeit von `SENSB_PIX_CLK` ist für die High-Dauer von `SENSB_HSYNC` gegeben. Die an den parallelen Datenleitungen `SENSB_DATA[9:0]` anliegenden Pixeldaten werden schließlich sequentiell für eine Zeile mit jeder ⎍-Flanke von `SENSB_PIX_CLK` eingelesen.

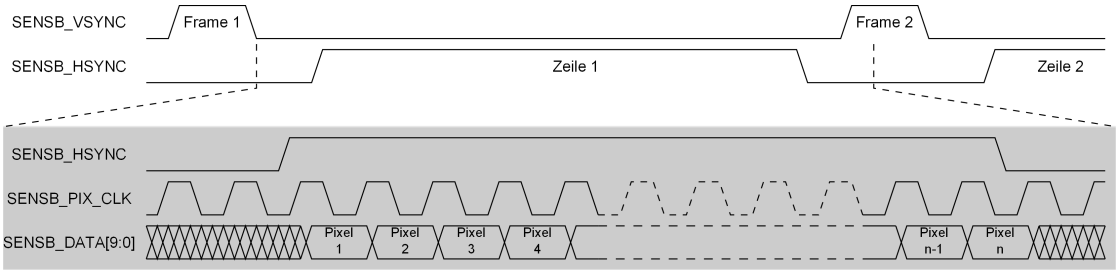


Abbildung 3.4: Timing-Diagramm des CMOS Sensor Interfaces des i.MX31 mit Detailansicht der Signalverläufe einer Bildzeile mit n Pixel (vgl. [16], S.58)

Anzumerken ist beim CMOS Sensor Interface noch, dass die parallel anliegenden Pixeldaten an `SENSB_DATA[9:0]` in verschiedenen Formaten codiert vorliegen können (vgl. [15], S.44-3).

- YUV 4:4:4 (10 Bit pro Pixel),
- 8 Bit Raw-RGB und
- 10 Bit Raw-RGB

Wie zu erkennen ist, kann bei den Pixeldaten ein Format gewählt werden, bei dem pro Pixel nur 8 Bit vorliegen und somit nur die Leitungen `SENSB_DATA[9:2]` zwingend erforderlich sind. Das bedeutet, dass zur Videoeingabe für den Upchannel zumindest 11 parallele Leitungen benötigt werden.

3.2 Synchronous Serial Interface – SSI/I²S

Das Synchronous Serial Interface des i.MX31 ist eine serielle vollduplex Schnittstelle, welche die Kommunikation mit einer Vielzahl an seriellen Geräten ermöglicht. Im Sinne des RPI-Busses soll dieses Interface zur Übertragung der Audiodaten verwendet werden, da es auch die Audiostandards Intel AC97, sowie den inter-IC sound bus (I²S) unterstützt (vgl. [15], S.45-1).

In weiterer Folge wird das SSI nur noch im Hinblick auf den I²S-Modus betrachtet, da dieses Protokoll zur Übertragung von digitalen Audioinformationen beim RPI-Bus Anwendung finden soll.

Das Synchronous Serial Interface kann sowohl als I²S Master als auch als I²S Slave Schnittstelle konfiguriert werden. Wird das Interface als Master konfiguriert, so generiert das SSI das für die I²S Übertragung erforderlich Takt- und Steuersignal. Im Slave-Modus wird jedoch das SSI fremd getaktet und auch das Steuersignal muss von der Peripherie generiert werden (vgl. [15], S.45-14ff).

In Tabelle 3.3 sind die erforderlichen Leitungen für das I²S Interface angeführt. Die grau eingefärbten Tabelleneinträge müssen im Falle einer Slave-Konfiguration von einem Peripheriegerät generiert werden. Die seriellen Daten `SD` werden beim Downchannel vom i.MX31 zur Verfügung gestellt, während beim Upchannel das Peripheriegerät die Audioinformation zum Mikrocontroller überträgt.

Name	Typ	Funktion
`SCK`	Takt	Taktsignal
`WS`	Steuerung	Links-/Rechts-Kanal Information
`SD`	Daten	serielle Audiodaten

Tabelle 3.3: Takt-, Steuer- und Datensignal der I²S Schnittstelle; grau hinterlegte Einträge sind vom I²S Master zu generieren

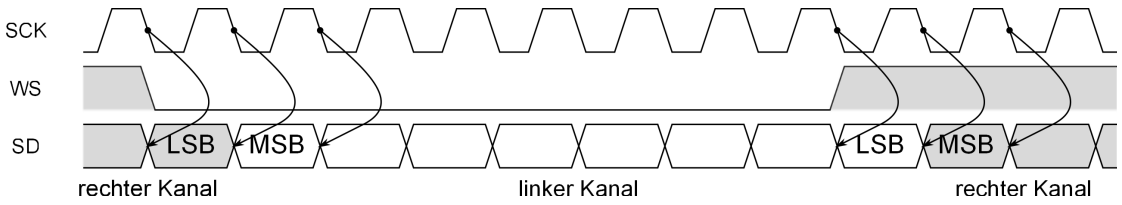


Abbildung 3.5: Timing Diagramm des I²S Protokolls (vgl. [15], S.45-14)

In Abbildung 3.5 ist das Timing-Diagramm einer I²S Übertragung dargestellt. Mit ⌋⌈-Flanken von `SCK` werden die Daten an `SD` vom I²S Empfänger übernommen. Die

Leitung WS kennzeichnet dabei, ob die seriellen Daten für den linken, oder rechten Audiokanal bestimmt sind. Ein Low-Pegel von WS entspricht einer Linkskanalinformation, ein High-Pegel hingegen einer Rechtskanalinformation. Das Signal WS weist für die gesamte Dauer eines Datenwortes den gleichen Pegel auf und somit ist dessen Frequenz gleich das Doppelte der Abtastfrequenz des Audiosignals. Zu beachten ist jedoch, dass erst mit der zweiten ⌝⌞-Flanke von SCK nach einem Pegelwechsel von WS die Übertragung eines neuen seriellen Datenwortes beginnt.

Tabelle 3.4 listet die Anzahl der benötigten Leitungen für die Audioübertragung über den RPI-Bus bei entsprechender Konfiguration der I²S Schnittstellen des i.MX31 auf.

Werden die I²S Schnittstelle des i.MX31 als Master konfiguriert, so müssen die Signale SCK und WS sowohl für die Audioaus- als auch für die Audioeingabe über den Downchannel übertragen werden. Im Falle einer Slave-Konfiguration müssen jedoch nur die seriellen Daten SD zur Audioausgabe über den Downchannel übertragen werden, da die Peripherie die Takt- und Steuersignale für die Audioaus- und Audioeingabe erzeugt.

I²S Konfiguration des i.MX31	**Downchannel**	**Upchannel**
Master	5	1
Slave	1	5

Tabelle 3.4: Benötigte Signalleitungen zur Audioübertragung für Down- und Upchannel bei Master- oder Slave-Konfiguration des i.MX31

3.3 Configurable Serial Peripheral Interface – CSPI

Das CSPI-Modul des i.MX31 erlaubt im Vergleich zu anderen seriellen Übertragungen eine schnelle Datenkommunikation mit wenigen Software-Interrupts. Zusätzlich ist bei SPI im Gegensatz zu I²C eine vollduplex Kommunikation gegeben. Durch die flexible Konfiguration des CSPI-Moduls können alle, die für SPI definierten Protokolle eingestellt werden, wobei auch die Konfiguration als SPI Master oder SPI Slave möglich ist. Des Weiteren weist dieses Interface vier Chip Select Signale auf, um eine Adressierung von Geräten ohne Softwareprotokoll durchführen zu können (vgl [15], S.24-1).

Wie bereits angedeutet, wird bei SPI grundsätzlich zwischen vier verschiedenen Protokollen bzw. Modi unterschieden. Der Verbindungsaufbau zwischen SPI Master

und SPI Slave wird mit einer ⌝-Flanke von $\overline{\texttt{SS}}$ eingeleitet. Die Datenübernahme an den Leitungen `MOSI` und `MISO` erfolgt je nach Modus bei unterschiedlichen Flanken von `SCLK`. Bei `PHA` gleich Low wird die Art der ersten Taktflanke zum Triggern der Daten verwendet, bei `PHA` gleich High die Art der zweiten Flanke. Der logische Zustand von `POL` entspricht dem Idle-Zustand von `SCLK`. Zu beachten ist jedoch, dass bei einer SPI-Konfiguration von `PHA` gleich Low, der SPI Slave bereits bei einer ⌝-Flanke von $\overline{\texttt{SS}}$ seine Daten an `MISO` anlegen muss, da bereits mit der ersten Flanke von `SCLK` die Daten vom Master eingelesen werden. In Abbildung 3.6 sind die Timings der verschieden Modi abgebildet.

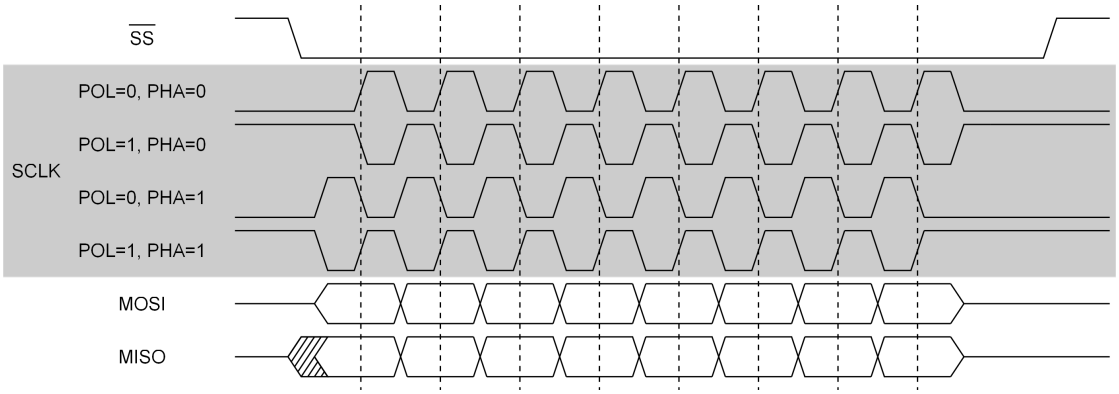


Abbildung 3.6: Timing-Diagramm der verschiedenen SPI Modi; je nach verwendetem Modus ist eine der vier Signalverläufe von `SCLK` gültig (vgl. [15], S.24-17)

Name	Typ	Funktion
$\overline{\texttt{SS}}$	Steuerung	Slave Select; zur Adressierung eines Slaves
`SCLK`	Takt	Taktsignal
`MOSI`	Daten	Master Out Slave In; Daten vom i.MX31 zur Peripherie
`MISO`	Daten	Master In Slave Out; Daten von der Peripherie zum i.MX31

Tabelle 3.5: Takt-, Steuer- und Datenleitungen der SPI Schnittstelle; grau hinterlegte Tabelleneinträge müssen vom Master generiert werden

Im Hinblick auf den RPI-Bus ist nur die Sinnhaftigkeit einer Master-Konfiguration des SPI am i.MX31 gegeben, da der Mikrocontroller für die Verwaltung aller angeschlossenen Geräte verantwortlich ist und er entscheidet, mit wem er kommunizieren möchte. Das bedeutet, dass die in Tabelle 3.5 grau eingefärbten Tabelleneinträge vom i.MX31 erzeugt werden müssen und demnach über den Downchannel übertragen werden. `MISO` entspricht hingegen der Datenrückleitung von der Peripherie zum i.MX31 und wird über den Upchannel gesendet.

3.4 Downchannel

Wie bereits erwähnt, wird im Hinblick auf den RPI-Bus die Kommunikationsrichtung von der zentralen Recheneinheit zur Peripherie als Downchannel bezeichnet. Über diesen Kanal sollen die Signale zur Video- und Audioausgabe, sowie die Signale für die Datenkommunikation übertragen werden. Aus den Betrachtungen von 3.1 bis 3.3 ist ersichtlich, dass eine große Anzahl an parallelen Leitungen benötigt wird, um die geforderten Signale zu den Weareables übertragen zu können. In Tabelle 3.6 sind nochmals alle für den Downchannel relevanten Signale angeführt.

Name	Typ	Anzahl der Signale
DISPB_D3_DATA[17:0]	Video	18
DISPB_D3_VSYNC	Video	1
DISPB_D3_HSYNC	Video	1
DISPB_D3_CLK	Video	1
DISPB_D3_DRDY	Video	1
SCKo	Audio	1
WSo	Audio	1
SDo	Audio	1
$\overline{\text{SS}}$	Daten	1
SCLK	Daten	1
MOSI	Daten	1
Signale Downchannel		28

Tabelle 3.6: Zusammenfassung der Signale, die über den Downchannel zu den Wearables übertragen werden

Wie in Abschnitt 3.2 gezeigt, können die SSI Schnittstellen zur Audioübertragung beim i.MX31 als Slaves konfiguriert werden, wodurch nur noch das Datensignal zur Audioausgabe an die Peripheriegeräte übertragen werden muss. Eine Zusammenfassung der Signale $\overline{\text{SS}}$ und MOSI durch eine entsprechende Signalverarbeitung reduziert die Anzahl an benötigten Signalleitungen für den Downchannel um eine weitere Leitung. Werden die parallel vorliegenden Signale nun mittels denen unter Punkt 2.2 beschriebenen Serializer übertragen, so kann der Pixeltakt DISPB_D3_CLK zum Einlesen der Daten verwendet werden und der zu verwendende Serializer muss somit 24 parallele Eingänge aufweisen. Da in der Aufgabenstellung 1.2 die Vereinfachungen getroffen wurden, dass an allen Peripheriegeräten das gleiche Video- und Audiosignal ausgegeben wird, kann der Downchannel zusätzlich als Broadcast-Kanal entworfen werden. Das bedeutet, dass der i.MX31 nur mit einem Serializer verbunden ist, wobei sich nach diesem Serializer noch ein 4:1 Repeater befinden muss, der das differentielle Signal für die vier Wearables zur Verfügung

stellt. Somit werden zusätzlich die Anzahl an benötigter Hardware und die dadurch entstehenden Kosten reduziert.

3.5 Upchannel

Beim Upchannel handelt es sich um die Kommunikationsrichtung von der Peripherie zur zentralen Recheneinheit. Über diesen Kanal werden die Signale zur Video-, Audio- und Dateneingabe übertragen. Zusätzlich werden auch die Steuersignale für die Audioausgabe über die Upchannels gesendet, sodass die benötigte Leitungsanzahl beim Downchannel reduziert werden kann. In Tabelle 3.7 sind nochmals alle Signale angeführt, welche über den Upchannel übertragen werden.

Name	Typ	Anzahl der Signale
SENSB_DATA[9:2]	Video	8
SENSB_VSYNC	Video	1
SENSB_HSYNC	Video	1
SENSB_PIX_CLK	Video	1
SCKo	Audio	1
WSo	Audio	1
SCKi	Audio	1
WSi	Audio	1
SDi	Audio	1
MISO	Daten	1
Signale Upchannel		17

Tabelle 3.7: Zusammenfassung der Signale, die über den Upchannel zu den Wearables übertragen werden

Das Taktsignal SENSB_PIX_CLK kann auch beim Upchannel wieder als Taktsignal für den Serializer verwendet werden wodurch der Serializer beim Upchannel nur 16 parallele Eingänge aufweisen muss. Wird jedoch der gleiche Serializer wie beim Upchannel verwendet, so können die restlichen acht Leitungen für redundante Datenübertragung verwendet werden.

3.6 Spannungsversorgung der Peripherie

Wie in den Spezifikationen unter Punkt 1.3 festgehalten wurde, sollte auch die Spannungsversorgung der Wearables über die Busleitungen des RPI-Busses erfolgen. In weiterer Folge wird die Spannungsversorgung der Peripherie als Phantomspeisung bezeichnet, da diese gleichzeitig mit den Daten auf den Busleitungen existiert, jedoch vom Empfänger nicht wahrgenommen werden sollte.

Um eine derartige Spannungsversorgung zu ermöglichen, sollte das differentielle Leitungspaar des Downchannels zur Übertragung einer Gleichspannung verwendet werden, während über das Leitungspaar des Upchannels das Bezugspotential hergestellt wird. Die differentiellen Leitungspaare des Bussystems werden also mit einer zusätzlichen Gleichspannung beaufschlagt.

Um die Strukturen der Schnittstellentreiber jedoch nicht zu zerstören, müssen die Ausgänge der Serializer bzw. die Eingänge der Deserializer kapazitiv von den Übertragungsleitungen entkoppelt werden, sodass die Gleichspannung nicht zu den Aus- und Eingangsstrukturen der Serializer und Deserializer übertragen wird. Damit jedoch das hochfrequente differentielle Signal auf den Übertragungsleitungen nicht gegen die Gleichspannung oder gegen das Bezugspotential kurzgeschlossen wird, muss die Spannungsversorgung induktiv auf die Übertragungsleitungen eingekoppelt werden.

Eine kapazitive Entkopplung der Serializer und Deserializer vom Übertragungsmedium erfordert jedoch eine gleichspannungsfreie Signalübertragung, wodurch die Auswahl der zu verwendenden SerDes erneut eingeschränkt wird. Auch an die Induktivitäten, die zur Einkopplung der Phantomspannung auf die Übertragungsleitungen notwendig sind, werden einige Anforderungen gestellt.

Die Induktivitäten dürfen im Gleichspannungsbereich nur eine sehr geringe Impedanz aufweisen, um unerwünschte Gleichspannungsverluste zu vermeiden. Zusätzlich müssen die Induktivitäten – zur Einkopplung der Phantomspeisung auf das differentielle Leitungspaar – über einen weiten Frequenzbereich eine hohe Impedanz besitzen, sodass die Flankensteilheit und der Pegel des Differenzsignals nicht wesentlich beeinflusst wird damit es vom Deserializer noch korrekt interpretiert werden kann.

3.7 Zusammenfassung

Es kann festgehalten werden, dass das RPI-Buskonzept zur Datenübertragung über den Downchannel einen Serializer und einen Repeater aufweisen muss. Um die Zusammenfassung der Signale $\overline{\texttt{SS}}$ und `MOSI` durchführen zu können, muss ein weiteres Schaltungselement miteinbezogen werden, wobei hier ein FPGA oder

CPLD Einsatz finden könnte. Für die Kommunikation mit der Peripherie über den Upchannel muss jeweils ein Deserializer vorgesehen werden.

Da beim RPI-Bus auch mehrere gleiche Peripheriegeräte angeschlossen werden können, aber die zentrale Recheneinheit nur ein Interface zum Einlesen von Video-, Audio- und Steuersignalen besitzt, muss ein Multiplexing der Eingangsschnittstellen vorgenommen werden, welches ebenfalls mittels FPGA oder CPLD durchgeführt werden kann.

Abbildung 3.7 zeigt ein erstes Blockschaltbild, wie das Konzept des RPI-Busses umgesetzt werden könnte, sodass alle getroffenen Spezifikationen erfüllt werden.

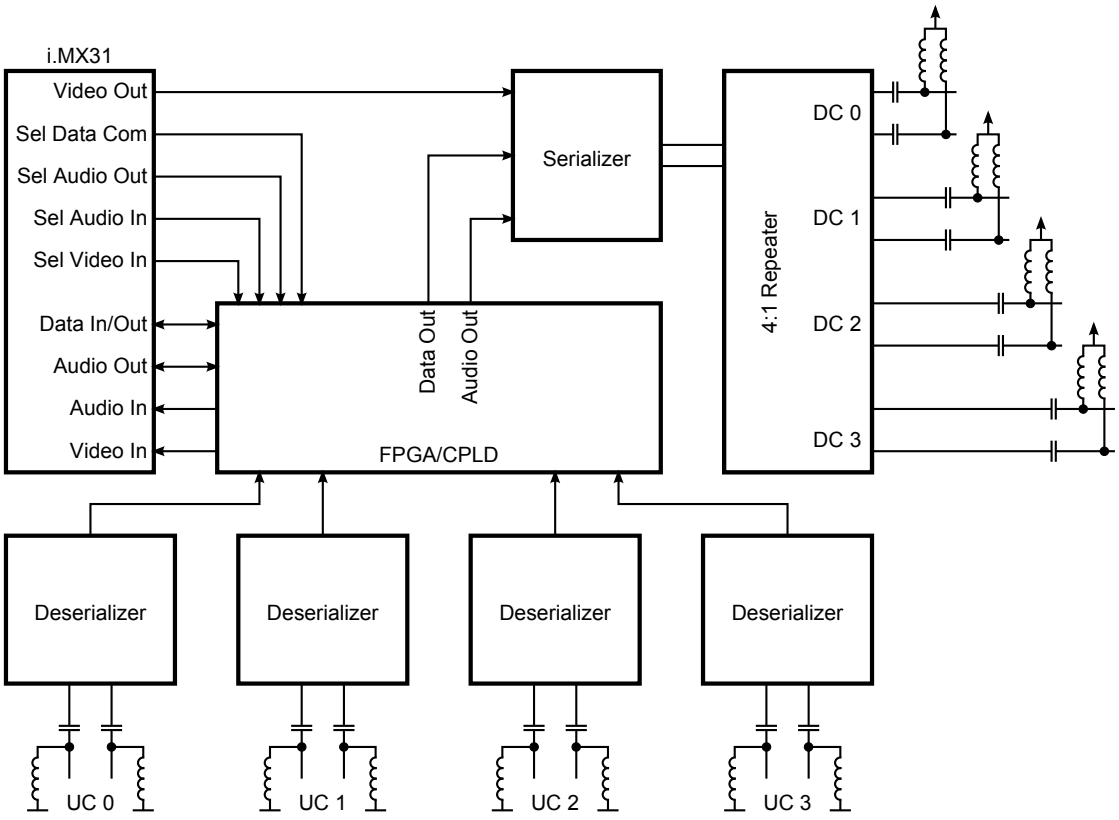


Abbildung 3.7: Schematischer Aufbau des RPI-Buskonzepts

4 Bauteilrecherche

Aufgrund der hohen Anforderungen an den RPI-Bus, muss nach geeigneten Bauteilen zur Umsetzung des unter Punkt 3 vorgestellten Konzepts gesucht werden. Dazu sollen die bereits erhaltenen Erkenntnisse als Entscheidungsgrundlage für die gesuchten Bauteile dienen.

4.1 SerDes Auswahl

Die gesuchten Serializer und Deserializer müssen ein paralleles Interface mit mindestens 24 Ein- bzw. Ausgänge, sowie ein serielles Interface mit einem differentiellen Leitungspaar besitzen. Daher kommen nur zwei der unter Punkt 2.2 vorgestellten SerDes Architekturen in Frage. Da jedoch bei 8b/10b SerDes die Nutzdatenrate aufgrund der notwendigen Kodierung geringer ist als die bei Embedded Clock SerDes, wird die Verwendung von Embedded Clock SerDes bevorzugt. Die Auswahl, der in Fragen kommenden SerDes kann nun auch noch durch das Kriterium, dass eine gleichspannungsfreie Signalübertragung möglich ist, verkleinert werden.

Zusammengefasst können folgende Anforderungen für die Serializer und Deserializer festgehalten werden:

- paralleles Interface: 24 Bit;
  serielles Interface: ein differentielles Leitungspaar
- Embedded Clock Architektur oder 8b/10b SerDes
- Taktrate von bis zu 30 MHz
- gleichspannungsfreie Signalübertragung
- niedrige Leistungsaufnahme ⇒ LVDS als serielles Interface

4.1.1 SerDes von MAXIM

Die Serializer und Deserializer von MAXIM besitzen allesamt eine sehr geringe Stromaufnahme. Die in Tabelle 4.1 angeführten Werte der Stromaufnahme beziehen sich auf den Worst Case und entsprechen den Werten, wenn das Bauteil mit der maximalen Taktfrequenz betrieben wird. Der Serializer MAX9247 besitzt zusätzlich

die Möglichkeit der Preemphasis, wodurch durch höhere Umladeströme schnellere Pegelwechsel auf dem differentiellen Leitungspaar erreicht werden und somit die Signalqualität verbessert wird. Nachteilig wirkt sich jedoch bei den von MAXIM angebotenen Komponenten die Notwendigkeit eines Referenztakts aus, welcher dem Deserializer die Synchronisation mit dem Serializer ermöglicht.

	Interfaces		**Taktrate**	**Strom**	**Temp.**	**Funktionen**
	Rx	**Tx**	MHz	mA	°C	
MAX9247 MAX9248	27 Bit LVCMOS	1 LVDS	2.5 - 42	70	-40 - 105	AC-Kopplung mögl., fixe Preemphasis, Referenztakt
	1 LVDS	27 Bit LVCMOS	2.5 - 42	135	-40 - 105	
MAX9217 MAX9218	27 Bit LVCMOS	1 LVDS	3 - 35	70	-40 - 85	AC-Kopplung mögl., keine Preemphasis, Referenztakt
	1 LVDS	27 Bit LVCMOS	3 - 35	70	-40 - 85	

Tabelle 4.1: SerDes des Halbleiterherstellers MAXIM

4.1.2 SerDes von Intersil

Der Halbleiterhersteller Intersil bietet nur einen Serializer an, welcher den Anforderungen, die an den RPI-Bus gestellt werden, gerecht wird. Dieser Serializer kann jedoch auch durch eine entsprechende Konfiguration als Deserializer eingesetzt werden. Im Vergleich zu den von MAXIM zur Verfügung gestellten integrierten Schaltkreisen, besitzt der ISL34341 eine wesentlich höhere Stromaufnahme, wobei es sich auch hier, bei denen in der Tabelle 4.2 angeführten Werten, wieder um den Worst Case handelt. Zusätzlich wird auch bei diesem Chip ein Referenztakt zur Taktrückgewinnung beim Deserializer benötigt, wodurch dieser IC aus der Auswahl zur Verwendung beim RPI-Bus ausscheidet.

	Interfaces		**Taktrate**	**Strom**	**Temp.**	**Funktionen**
	Rx	**Tx**	MHz	mA	°C	
ISL34341 ISL34341	24 Bit LVCMOS	1 LVDS	6 - 40	136	-40 - 85	AC-Kopplung mögl., prog. Preemphasis, Referenztakt
	1 LVDS	24 Bit LVCMOS	6 - 40	170	-40 - 85	

Tabelle 4.2: SerDes des Halbleiterherstellers Intersil

4.1.3 SerDes von Texas Instruments

Die Produkte von National Semiconductors bzw. Texas Instruments besitzen alle in etwa die gleiche Stromaufnahme, wobei es sich auch hier wieder um Worst Case Betrachtungen handelt. Die SerDes Paare unterscheiden sich hauptsächlich in der Treiberstärke und im spezifizierten Temperaturbereich. Während die Treiberstärke der Bauteile beim RPI-Bus aufgrund der kurzen Übertragungswege irrelevant ist, kann der Temperaturbereich, in dem die SerDes spezifiziert sind, zur Auswahl beitragen.

	Interfaces		**Taktrate**	**Strom**	**Temp.**	**Funktionen**
	Rx	**Tx**	MHz	mA	°C	
DS99R101 DS99R102	24 Bit LVCMOS	1 LVDS	3 - 40	85	0 - 70	AC-Kopplung mögl., PTO, embedded clock
	1 LVDS	24 Bit LVCMOS	3 - 40	95	0 - 70	
DS99R103 DS99R104	24 Bit LVCMOS	1 LVDS	3 - 40	90	-40 - 85	AC-Kopplung, Preemphasis, embedded clock
	1 LVDS	24 Bit LVCMOS	3 - 40	95	-40 - 85	
DS90C241 DS90C124	24 Bit LVCMOS	1 LVDS	5 - 35	70	-40 - 105	AC-Kopplung, Preemphasis, embedded clock
	1 LVDS	24 Bit LVCMOS	5 - 35	85	-40 - 105	
DS90UR241 DS90UR124	24 Bit LVCMOS	1 LVDS	5 - 43	90	-40 - 105	AC-Kopplung, Preemphasis, embedded clock, PTO
	1 LVDS	24 Bit LVCMOS	5 - 43	105	-40 - 105	

Tabelle 4.3: SerDes des Halbleiterherstellers Texas Instruments

Da das RPI-Bussystem in Geräten Anwendung finden sollte, welche auch bei extremen Umwelteinflüssen eingesetzt werden, sind Komponenten erforderlich, welche auch für Temperaturen unter 0 °C spezifiziert sind und somit scheiden die ICs DS99R101 und DS99R102 aus.

4.1.4 Entscheidungsfindung

Wie aus den Betrachtungen 4.1.1 bis 4.1.3 ersichtlich sein soll, entfallen die SerDes der Halbleiterhersteller MAXIM und Intersil aus der näheren Auswahl, da diese die Notwenigkeit eines Referenztaktes beim Deserializer besitzen. Bezogen auf den spezifizierten Temperaturbereich kommen somit nur die SerDes DS90C241/DS90C124,

sowie DS90UR241/DS90UR124 in die engere Auswahl. Da jedoch die SerDes DS90UR241/DS90UR124 zusätzliche Funktionen, wie Preemphasis und Progressive Turn On (PTO) besitzen, fällt die endgültige Entscheidung auf diese beiden Bauteile.

Serializer - DS90UR241

Dieser Serializer von Texas Instruments bzw. National Semiconductors wurde speziell zur Übertragung von Displaydaten über ein differentielles Leitungspaar entwickelt. Er stellt dazu ein 24 Bit breites parralleles Interface zur Verfügung, wobei 18 Eingänge für die RGB-Daten, drei Eingänge für die Steuersignale VSYNC, HSYNC und DE (Data Enable) und drei Eingänge zur allgemeinen Verwendung bestimmt sind.

Durch ein propritäres Kodierungsverfahren ermöglicht der DS90UR241 die Übertragung eines gleichspannungsfreien Signals, sowie eine reduzierte EMV-Emission, da die abgestrahlte Störung über einen weiten Frequenzbereich aufgeteilt wird. Zusätzlich bietet dieser Chip zwei Möglichkeiten zur Verbesserung der Signalqualität an. Mittels Preemphasis und einer höheren Differenzspannung können Qualitätsverluste des Differenzsignals, hervorgerufen durch verlustbehaftete Übertragungswege, kompensiert werden. Weiters lässt sich beim DS90UR241 Serializer die Art der Taktflanke, mit der die parallelen Eingangsdaten eingelesen werden, durch eine externe Konfiguration einstellen. Über die Beschaltung eines weiteren Eingangs des DS90UR241 lässt sich dieser in einen Power Down Modus versetzen, wodurch seine Stromaufnahme auf einige wenige μA reduziert werden kann (vgl. [17], S.1ff).

In Abbildung 4.1 ist der vom DS90UR241 erzeugte Bitstrom für ein eingelesenes 24 Bit Wort abgebildet. Neben den 24 Bits des parallelen Interfaces werden noch zwei Bits zur Taktrückgewinnung, sowie zwei Kontrollbits übertragen. Diese Kontrollbits benötigt der Deserializer, um die empfangenen Daten korrekt interpretieren zu können. Da jedes Datenpaket im seriellen Bitstrom mit einem High-Bit beginnt und mit einem Low-Bit endet, ergeben sich wie unter Punkt 2.2.3 bereits angeführt, periodische ⌐-Übergänge im seriellen Bitstrom, anhand dessen der Deserializer das eingebettete Taktsignal extrahieren kann.

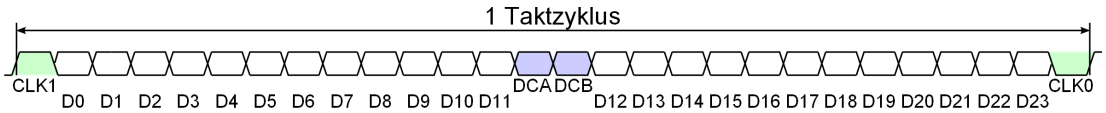


Abbildung 4.1: LVDS-Bitstrom eines 24 Bit Datenpakets (vgl. [17], S.22)

Zwischen dem Einlesen der Daten des parallelen Interfaces und der Ausgabe des serialisierten Bitstroms verstreicht eine gewisse Zeitdauer t_{SD}, wobei Texas Instruments den mathematischen Zusammenhang zur Berechnung des Serializer-Delays im dazugehörigen Datenblatt angibt. Diese Zeitdauer ist ausschließlich von

der Taktfrequenz – mit der, der Serializer betrieben wird – abhängig und berechnet sich aus Formel 4.1.

$$t_{SD} = 3.5 \cdot \frac{1}{\underbrace{f_{CLK}}_{\text{in MHz}}} + 10\,\text{ns} \tag{4.1}$$

Abbildung 4.2 zeigt den Zusammenhang zwischen den zu serialisierenden Daten, dem Taktsignal und dem seriellen Bitstrom. Die parallel vorliegenden Daten werden mit einer ↓-Flanke des Taktsignals eingelesen, die schließlich nach dem Verstreichen der Verzögerungszeit t_{SD} zusammen mit den Takt- und Kontrollbits als serieller Bitstrom am LVDS Interface zur Verfügung stehen.

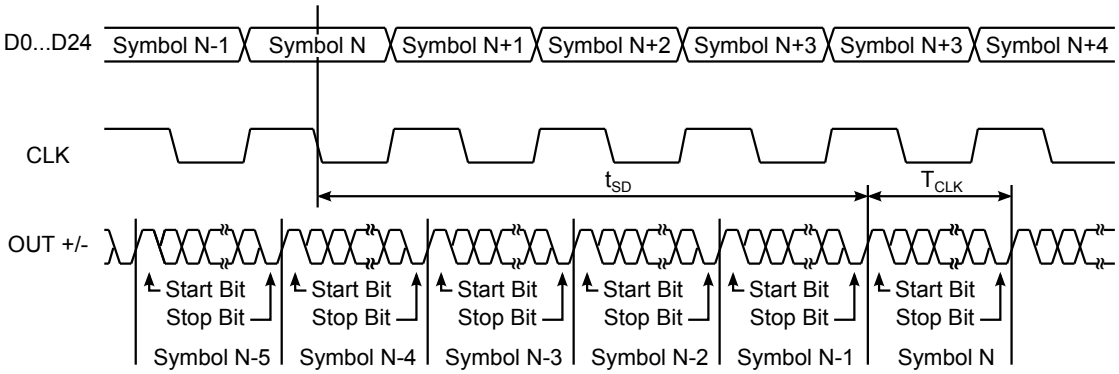


Abbildung 4.2: Zeitverzögerung zwischen dem Einlesen der parallelen Eingänge und der Übertragung des ersten Bits des korrespondierenden Datenpakets im seriellen Bitstrom (vgl. [17], S.9)

Deserializer - DS90UR124

Der Deserializer DS90UR124 entspricht dem Gegenstück de Serializers DS90UR241. Er stellt als Eingangsschnittstelle ein LVDS Interface zur Verfügung, welches den seriell übertragenen Bitstrom einliest und gibt diesen schließlich parallel auf 24 LVCMOS Ausgängen aus.

Bezüglich der Ausgangsschnittstelle können beim DS90UR124 verschiedene Konfigurationen vorgenommen werden. Die Daten können durch entsprechende Einstellung mit ↓- oder ↑-Flanken des rückgewonnenen Taktsignals ausgegeben werden und mittels der Funktion Progressiv Turn On können zwei verschiedene Varianten der Datenausgabe eingestellt werden. Die Daten werden in beiden Fällen auf drei 8 Bit Gruppen aufgeteilt, wobei diese drei Gruppen leicht zeitversetzt geschaltet werden. Da dadurch immer nur acht Ausgänge gleichzeitig geschaltet werden, wird die dynamische Stromaufnahme des Deserializers verringert und in weiterer Folge auch die leitungsgeführte Störemission. Der Unterschied zwischen den beiden Modi der

Progressiv Turn On Funktion findet sich in den eingefügten Verzögerungszeiten zwischen dem Umschalten der einzelnen Gruppen.

Zusätzlich kann durch externe Beschaltung des DS90UR124 die Stromergiebigkeit der LVCMOS Ausgänge konfiguriert werden, wobei bei mit höheren Ausgangsströme auch die Störemissionen ansteigen.

Wie beim DS90UR241 Serializer besteht auch beim Deserializer die Möglichkeit, die integrierte Schaltung in einen Power Down Modus zu versetzen, sodass die Stromaufnahme des Chips auf wenige μA reduziert wird (vgl. [17], S.1ff).

Zwischen dem Einlesen des seriellen Bitstroms am LVDS Interface und der Ausgabe der korrespondierenden Daten an der parallelen Schnittstelle, entsteht auch beim Deserializer eine Verzögerungszeit, die Texas Instruments im Datenblatt des DS90UR124 mit dem mathematischen Zusammenhang 4.2 angibt.

$$t_{DD} = \left[5 + \frac{5}{56}\right] \cdot \underbrace{\frac{1}{f_{CLK}}}_{\text{in MHz}} + 8\,\text{ns} \tag{4.2}$$

4.2 LVDS-Repeater Auswahl

Wie unter Punkt 3.4 bereits beschrieben, kann der Downchannel als Broadcast Kanal entworfen werden. Das bedeutet, dass der gleiche Bitstrom an alle vier Anschlusspunkten des RPI-Busses weitergeleitet werden kann. Dazu stellt Texas Instruments bzw. National Semiconductors einige LVDS Repeater zur Verfügung.

Anders als bei den SerDes muss nun auf ausreichend hohen Datendurchsatz geachtet werden. Im Worst Case werden Videodaten für eine Displayauflösung von 800×600 px und einer Bildwiederholfrequenz von 60 fps übertragen. Mittels dieser Parameter kann der erforderliche Pixeltakt abgeschätzt werden. Die Datenrate des vom Serializer erzeugten Bitstromes ist schließlich um den Faktor 28 höher als der Pixeltakt, da wie in Abbildung 4.1 ersichtlich, während eines Taktzyklus 28 Bit vom Serializer ausgegeben werden. Der LVDS-Repeater muss also in der Lage sein, Daten mit der nachfolgend abgeschätzten Datenrate übertragen zu können.

$$\begin{aligned} f_{PIXCLK} &\approx 800\,\text{px} \cdot 600\,\text{px} \cdot 60\,\text{fps} \\ &\approx 30\,\text{MHz} \\ Datenrate &\approx f_{PIXCLK} \cdot 28 \\ &\approx 1\,\text{Gb/s} \end{aligned}$$

Diese Anforderung, welche an den LVDS-Repeater gestellt wird, verkleinert die

Auswahl an geeigneten Bauteilen enorm. Bei näherer Betrachtung der Komponenten von Texas Instruments kommen nur noch zwei dieser Repeater für die Verwendung beim RPI-Bus in Frage. In Tabelle 4.4 sind diese beiden Schaltkreise gegenübergestellt. Trotz höherer Stromaufnahme des DS90BR204 fällt die Entscheidung zur Verwendung beim RPI-Bussystem auf diesen, da dieser Funktionen zur Signalaufbereitung zur Verfügung stellt.

	Interfaces		**Throughput**	**Strom**	**Temp.**	**Funktionen**
	Rx	**Tx**	MB/s	mA	°C	
DS90BR254	2	4	1500	135	-40 - 85	LVDS, LVPECL, CML kompatibel
DS90BR204	2	4	3125	185	-40 - 85	LVDS, LVPECL, CML kompatibel; Preemphasis; Equalization

Tabelle 4.4: LVDS-Repeater von Texas Instruments

4.3 Auswahl der Koppelinduktivitäten

Bevor Überlegungen bezüglich der Koppelinduktivitäten getroffen werden, wird die LVDS-Übertragung mit einem stark vereinfachten Modell simuliert.

Da die Ausgangstreiber von LVDS-Sendern mit einer geschalteten Stromquelle arbeiten und aus [17] hervorgeht, dass der Serializer DS90UR241 an einem 100 Ω Widerstand eine Differenzspannung von 500 mV erzeugt, wird angenommen, dass der DS90UR241 einen Ausgangsstrom von 5 mA treibt. Zusätzlich befindet sich in [17] ein typisches Anwendungsbeispiel, wie der Serializer zu beschalten ist.

Der Ausgangstreiber des DS90UR241 wird mittels einer pulsförmigen Konstantstromquellen realisiert und mittels 100 Ω differentiell abgeschlossen, sowie kapazitiv mit 100 nF vom weiteren Übertragungsweg entkoppelt.

Da das differentielle Signal auf einer Platine zu einem Steckverbinder geführt werden muss, an dem mittels Kabel die Wearables angeschlossen werden können, wird auch ein Simulationsmodel dieser Leiterbahnen miteinbezogen. Auch die Leiterbahnen, welche beim Peripheriegerät das differentielle Signal zum Deserializer führen werden mit demselben Modell simuliert.

Da bezüglich des Kabels, mit dem die Wearables an den RPI-Bus angeschlossen werden, keine Informationen vorliegen, wird dieses als Verlustlose Leitung mit

einem Differenzwiderstand von 100 Ω – jeweils Z_0=50 Ω gegen GND – und einem Laufzeitdelay von 5 ns – entspricht einer Kabellänge von ca. 1 m – nachgebildet.

Schlussendlich wird das differentielle Signal wieder kapazitiv entkoppelt und mit 100 Ω abgeschlossen. Abbildung 4.3 zeigt das Simulationsmodell in LTSpice, welches für die ersten Betrachtungen herangezogen wird. Anzumerken ist noch, dass die folgenden Betrachtungen und Simulationen nur für den Downchannel durchgeführt werden, da sich der Upchannel identisch verhält.

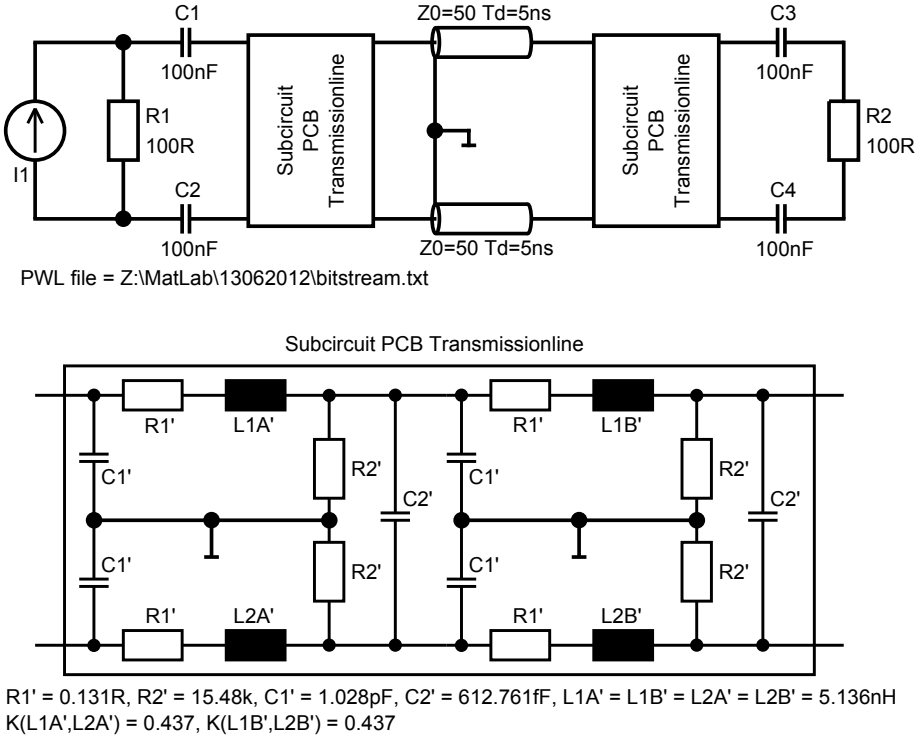


Abbildung 4.3: oben: Gesamtsimulationsmodell für die LVDS-Übertragung ohne Koppelinduktivitäten zur Einspeisung der Phantomspannung;
unten: Detailansicht des nachgebildeten Leitungspaares mit den Werten der Widerstands-, Leitwerts-, Kapazitäts- und Induktivitätsbelägen

4.3.1 Simulationsmodell für differentielle Leiterbahnen

Bei der LVDS-Übertragung liegt ein differentieller Widerstand von 100 Ω zwischen den beiden differentiellen Leitungen vor. Dieser Widerstand hängt vom Wellenwiderstand der Leiterbahnen, deren geometrischen Abmessungen und der Anordnung auf der Platine ab. Für eine genaue Bestimmung des Wellenwiderstandes einer Leiterbahn, sowie des differentiellen Widerstandes eines Leitungspaares müssten sogenannte 2D-Field-Solver-Simulationsprogramme verwendet werden. Es existieren jedoch auch Näherungsformeln, mit denen sowohl der Wellenwiderstand, als auch der differentielle Widerstand von Leiterbahnen abgeschätzt werden kann.

Abbildung 4.4 zeigt den Querschnitt einer Single-Ended Leiterbahn und einem differentiellen Leiterbahnenpaar.

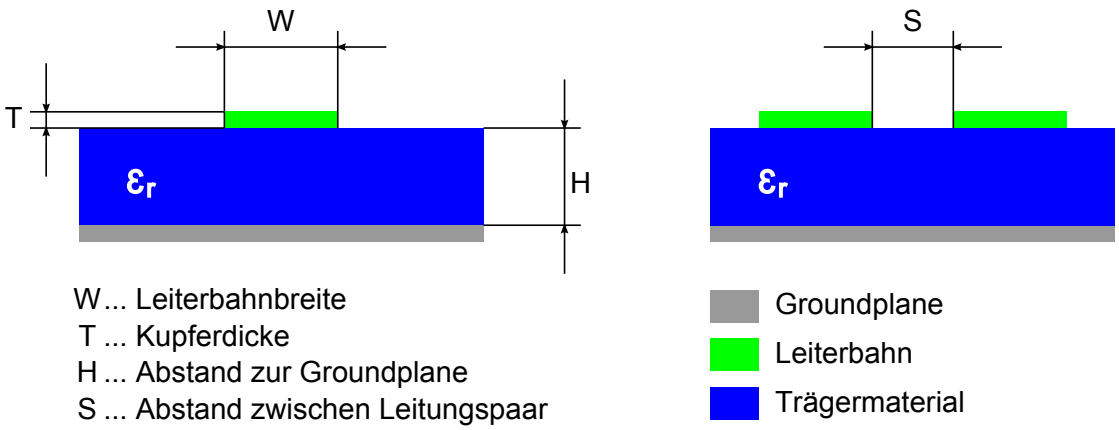


Abbildung 4.4: links: Querschnitt einer Single-Ended Leiterbahn;
rechts: Querschnitt eines Leiterbahnenpaares; (vgl. [7], S.39)

Mit den Formeln 4.3 und 4.4 können nun impedanzkontrollierte Leiterbahnen berechnet werden. Die Parameter ε_r, H, W, T und S entsprechen unter anderem den in Abbildung 4.4 eingezeichneten Abmessungen und sind zum Teil selbst wählbar oder vom Platinenhersteller vorgegeben.

$$Z_0 = \frac{87}{\sqrt{\epsilon_r + 1.41}} \cdot \ln\left[\frac{5.98 \cdot H}{0.8 \cdot W + T}\right] \tag{4.3}$$

$$Z_{diff} = 2 \cdot Z_0 \cdot \left[1 - 0.48 \cdot e^{-0.96 \cdot \frac{S}{H}}\right] \tag{4.4}$$

Die Platine für den Prototypen des RPI-Busses wird bei PCB-POOL gefertigt und somit ergeben sich folgende fixierte Parameter bzw. Einschränkungen bei den frei wählbaren Größen.

Parameter	**Wert**	**Bemerkung**
ε_r	4.5	Standard-Industriequalität FR4
H	0.38 mm	Abstand zwischen äußerem Layer und darunterliegenden inneren Layer
W	min. 0.15 mm	von PCB-POOL kleinst möglich herstellbare Leiterbahnbreite
T	0.035 mm	Kupferdicke der Leiterbahnen

Fortsetzung auf nächster Seite

Tabelle 4.5 – *fortgesetzt von vorheriger Seite*

Parameter	**Wert**	**Bemerkung**
S	min. 0.15 mm	von PCB-POOL kleinst möglich herstellbarer Leiterbahnabstand

Tabelle 4.5: fixierte und wählbare Parameter zur Berechnung impedanzkontrollierter Leiterbahnen mit Einschränkungen durch den Leiterplattenhersteller (vgl. [18])

Aus Tabelle 4.5 ist ersichtlich, dass nur zwei Parameter zur Realisierung impedanzkontrollierter Leiterbahnen bedingt frei wählbar sind. Da jedoch der Abstand zwischen zwei Leiterbahnen eines differentiellen Leiterpaares möglichst gering sein sollte, sodass sich einerseits Störungen auf beide Leitungen gleich auswirken und andererseits durch die enge Kopplung der Leitungen weniger Störungen ausgesendet werden, wird auch der Parameter S mit dem Minimalwert von 0.15 mm fixiert. Es bleibt somit nur noch die Leiterbahnbreite W zur freien Auswahl über.

Abbildung 4.5 zeigt den Verlauf des Differenzwiderstandes eines Leiterbahnpaares in Abhängigkeit der Leiterbahnbreite. Anhand dieses Diagramms kann nun die notwendige Leiterbahnbreite für einen Differenzwiderstand von 100 Ω abgelesen werden.

Unter zur Hilfenahme des Programms IC-Emc und den ermittelten Parametern für ε_r, H, W, T und S kann nun ein Simulationsmodell für ein differentielles Leiterbahnenpaar ermittelt werden, welches in Abbildung 4.3 mit dem Subcircuit PCB Transmissionline dargestellt ist.

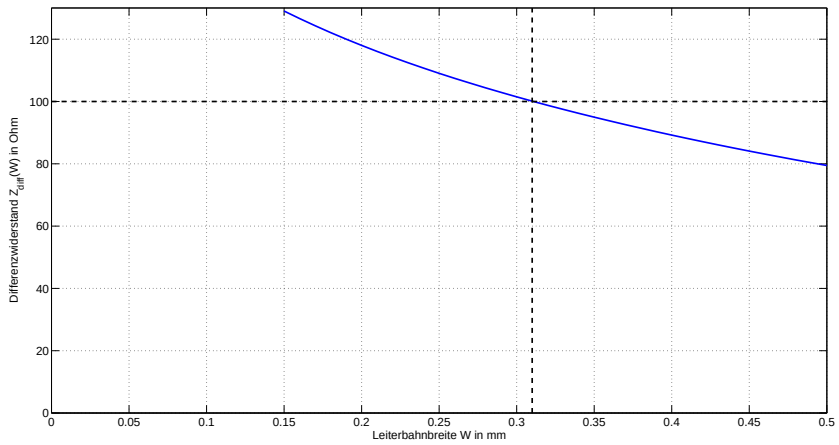


Abbildung 4.5: Differenzwiderstand eines Leiterbahnpaares in Abhängigkeit der Leiterbahnbreite; Z_{diff}=100.1 Ω bei W = 0.31 mm

4.3.2 Simulation der LVDS-Übertragung

Da der DS90UR241 ein proprietäres Kodierungsverfahren anwendet, um einen nahezu gleichspannungsfreien Bitstrom erzeugen zu können, können bezüglich der Generierung des Bitstromes keine Rückschlüsse getroffen werden. Um die Simulation jedoch möglichst genau an die Realität anpassen zu können, wird in MATLAB ein zufälliger Bitstrom mit dem Mittelwert 0 generiert, welcher in weiterer Folge als Input für den LVDS Ausgangstreiber des Serializers (pulsförmige Stromquelle) dient.

Abbildung 4.6 zeigt das erste von 10 serialisierten Symbolen, welche mit einer Taktfrequenz von 5 MHz mittels des Serializers eingelesen werden. Die Dauer eines Symbols im seriellen Bitstrom beträgt somit 200 ns und die Bitdauer bzw. das Unit-Interval, welches in Abbildung 4.6 mittels der rot strichlierter Linien dargestellt ist, ist mit 7.143 ns gegeben. Berechnet werden Symboldauer und Unit-Interval nach 4.5 und 4.6.

$$t_{sym} = \frac{1}{f_{clk}} = \frac{1}{5\,\mathrm{MHz}} = 200\,\mathrm{ns} \tag{4.5}$$

$$UI = \frac{t_{sym}}{28} = \frac{200\,\mathrm{ns}}{28} = 7.143\,\mathrm{ns} \tag{4.6}$$

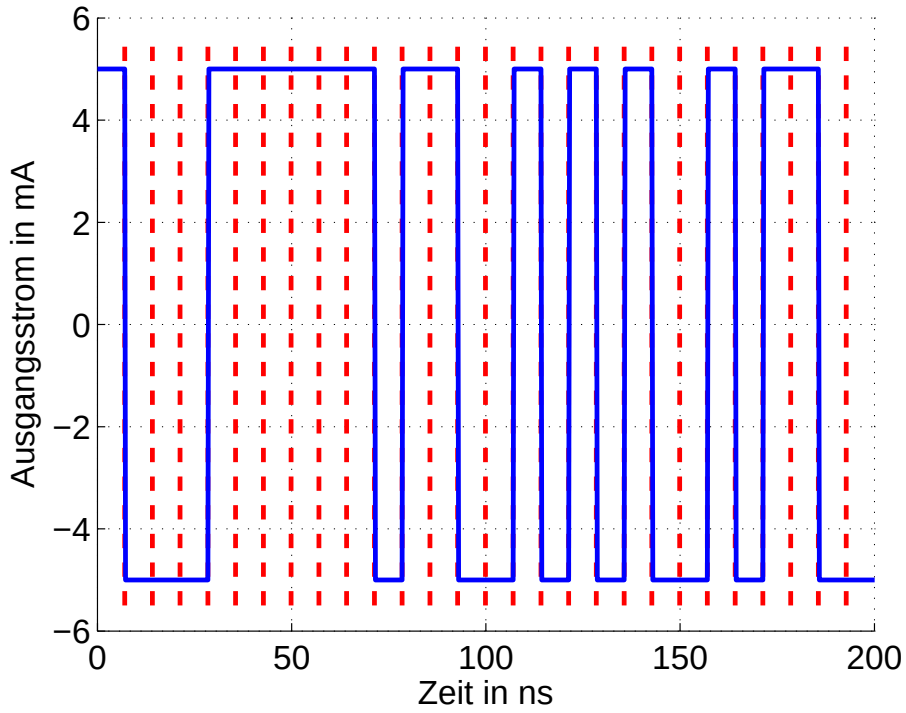


Abbildung 4.6: Die ersten 200 ns des in MATLAB erzeugten zufälligen Bitstromes

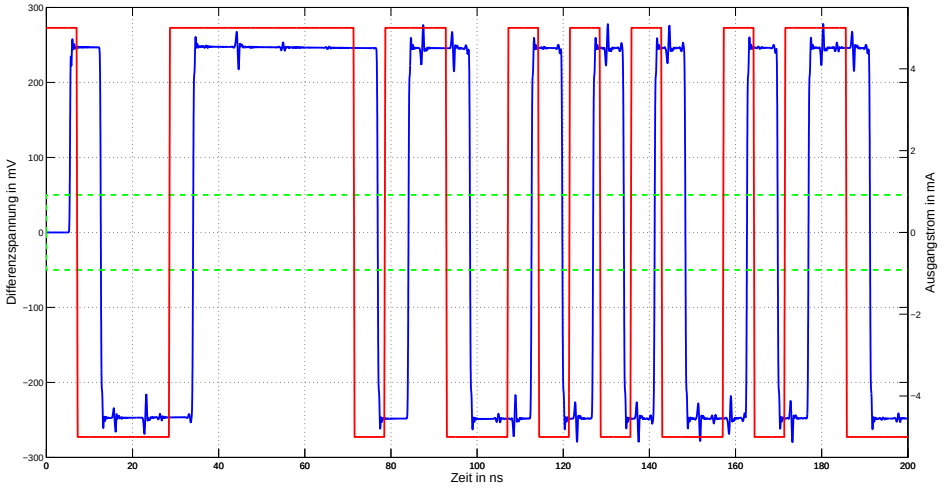


Abbildung 4.7: Simulationsergebnis der ersten 200 ns des unter 4.3 dargestellten Simulationsmodell; die rote Kurve entspricht dem Ausgangsstrom des LVDS-Treibers; die grünen Linien stellen die Schaltschwellen des LVDS-Empfängers dar; der blaue Kurvenverlauf zeigt die resultierende Differenzspannung am 100 Ω Widerstand beim LVDS-Empfänger

In Abbildung 4.7 sind der Verlauf des Ausgangsstroms des Serializers, sowie der Verlauf der Differenzspannung am Deserializer dargestellt. Zusätzlich sind die Schaltschwellen des Deserializers bei ±50 mV eingezeichnet, welche von der Differenzspannung niemals unterschritten werden dürfen, da sonst ein Übertragungsfehler auftritt.

Beim Verlauf der Differenzspannung ist auch zu erkennen, dass diese dem Ausgangsstrom um das Laufzeitdelay des Kabels nacheilt. Die Auslenkung der Differenzspannung befindet sich in etwa zwischen ±250 mV, da sich der vom Serializer getriebene Strom zu gleichen Teilen durch R_1 und R_2 aufteilt (siehe Abbildung 4.3).

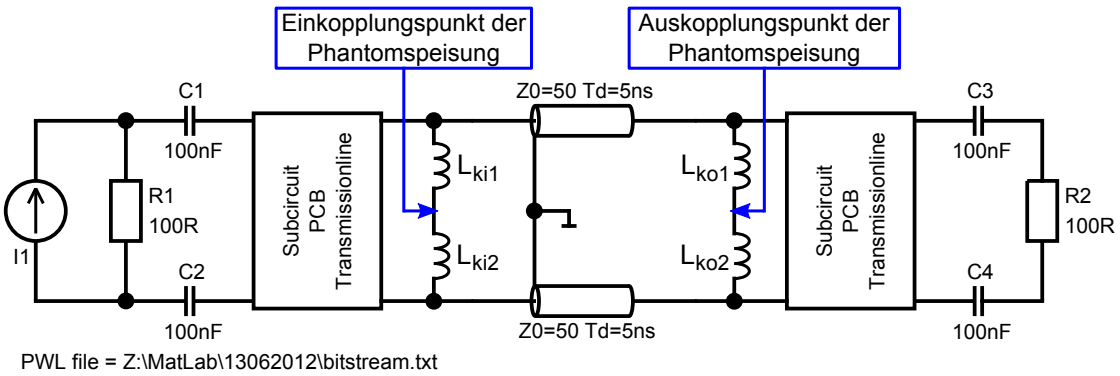


Abbildung 4.8: Simulationsmodell für die LVDS-Übertragung mit Berücksichtigung der Koppelinduktivitäten zur Einspeisung der Phantomspannung auf die Busleitungen

Werden nun die Koppelinduktivitäten zur Einspeisung der Phantomspannung in das Simulationsmodell 4.3 eingefügt, so befinden sich diese zwischen den beiden differentiellen Leitungen. Abbildung 4.8 zeigt das Simulationsmodell der LVDS-Übertragung mit Berücksichtigung der Koppelinduktivitäten. Die beiden Induktivitäten L_{ki1} und L_{ki2} dienen zur Einkopplung der Phantomspeisung auf die Busleitungen und mittels den Induktivitäten L_{ko1} und L_{ko2} wird die Phantomspannung bei den Wearables wieder ausgekoppelt.

Die Koppelinduktivitäten L_{ki1}, L_{ki2}, L_{ko1} und L_{ko2} müssen nun einige Eigenschaften aufweisen um beim RPI-Bus verwendet werden zu können. So dürfen sie für Gleichspannungssignale nur einen sehr geringen Eigenwiderstand aufweisen, sodass die Spannungsabfälle an ihnen möglichst gering gehalten werden und somit nahezu die gesamte eingespeiste Phantomspannung an den Wearables ausgekoppelt werden kann.

Zusätzlich müssen sie einer Strombelastung von 250 mA standhalten, da in der Spezifikation des RPI-Busses festgehalten wurde, dass die Wearables eine Stromaufnahme von bis zu 500 mA besitzen dürfen. Da jeweils zwei Leitungen zur Spannungs- und Stromversorgung der Peripherie zur Verfügung stehen und diese als gleichwertig angenommen werden können, werden bei einer maximalen Strombelastung der Busleitungen jeweils 250 mA über die Leitungen des differentiellen Leitungspaares fließen.

Aus der Sicht des Differenzsignals befinden sich die Koppelinduktivitäten parallel zum Differenzwiderstand. Die Koppelinduktivitäten müssen demnach für das Differenzsignal über einen gewissen Frequenzbereich eine ausreichend hohe Impedanz darstellen, sodass durch diese schaltungstechnische Maßnahme die Flankensteilheit des Differenzsignals nicht wesentlich beeinflusst wird. Zusätzlich werden die Koppelinduktivitäten durch den vom Serializer getriebenen Strom permanent umgeladen, wodurch der Spannungspegel des Differenzsignals nach jedem Schaltspiel nach einer Exponentialfunktion abklingt. Dieser Abklingvorgang muss langsam genug erfolgen, sodass die schon erwähnten Schaltschwellen des Deserializers nie unterschritten werden.

Eine hohe Impedanz bei Induktivitäten geht immer mit einer hohen Eigeninduktivität einher, welche sowohl von der Windungsanzahl (N), als auch vom Kernmaterial (μ_r) abhängig ist. Wird die Eigeninduktivität durch die Verwendung eines ferromagnetischen Kerns vergrößert, so nehmen die Wirkleistungsverluste der Induktivität bei hohen Frequenzen durch Wirbelströme im Kernmaterial zu. Es wären somit Luftspulen erforderlich, welche aufgrund der schlechten magnetischen Leitfähigkeit von Luft, eine hohe Windungsanzahl besitzen müssten. Somit wären Koppelinduktivitäten mit großen Bauformen erforderlich, um die nötige Eigeninduktivität zu erreichen. Große Bauformen sind im Hinblick auf den RPI-Bus jedoch unerwünscht, da Wearable Computing Geräte in ihren Abmessungen möglichst klein gehalten werden sollten. Zusätzlich nehmen die parasitären Effekte mit größeren Bauformen zu, wodurch die Eigenresonanzfrequenz der Induktivität in einen niedrigere Frequenzbereich verschoben wird (vgl. [19], o.S.).

Zusammengefasst können nun folgende Eigenschaften für die Koppelinduktivitäten festgehalten werden:

- niedriger Gleichspannungswiderstand R_{DC}
- Gleichstrombelastbarkeit $\geq 250\,mA$
- ausreichend große Eigeninduktivität L
- ausreichend hohe Eigenresonanzfrequenz f_r

Würth Elektronik ist einer der größten Hersteller induktiver Bauelemente in Europa. Das Sortiment umfasst Spulen, stromkompensierte Drosseln für Datenleitungen und Versorgungsleitungen, EMV-Ferrite und vieles mehr. Zusätzlich bietet Würth Elektronik eine umfassende Bibliothek an Simulationsmodellen ihrer induktiven Bauelemente für LTSpice an. Daher wird in weiter Folge im Produktkatalog von Würth Elektronik nach geeigneten Koppelinduktivitäten für den RPI-Bus gesucht und die Simulation der LVDS-Übertragung mit einem ähnlichen Modell der gewählten Induktivität für die kleinst- und größtmögliche Taktfrequenz des Serializers durchgeführt. Die kleinstmögliche Taktfrequenz liegt bei 5 MHz, wobei sich, wie schon gezeigt, eine Symboldauer von 200 ns und ein Unit-Interval von 7.143 ns ergibt. Bei der größtmöglichen Taktfrequenz ergibt sich nach 4.5 eine Symboldauer von nur 23.256 ns und nach 4.6 ein Unit-Interval von 830.565 ps.

1 μH Koppelinduktivität

Zu Beginn werden die Auswirkungen – des Einbaus einer 1 μH Induktivität – auf das hochfrequente Datensignal untersucht. Dazu wird die Speicherdrossel 74402800 mit NiZn-Kernmaterial, einem maximalen Gleichstromwiderstand von 85 mΩ, einem Nennstrom von 1.5 A und einer Eigenresonanzfrequenz von 150 MHz zur Simulation verwendet. Abbildung 4.9 zeigt oben das Simulationsergebnis bei einer Taktfrequenz von 5 MHz und unten das Ergebnis bei 43 MHz. Der rote Kurvenverlauf entspricht jeweils dem Ausgangsstrom und die blaue der Differenzspannung am Deserializer. Grün eingezeichnet sind wieder die Schaltschwellen des Deserializers zu sehen.

Bei der Simulation mit einer Taktfrequenz von 5 MHz ist deutlich zu sehen, dass die Differenzspannung bei ca. 90 ns die obere Schaltschwelle des Deserializers unterschreitet und es wird somit zu einem Übertragungsfehler kommen. In der Realität wird der verlauf der Differenzspannung voraussichtlich noch schlechter sein, da weder störende Einflüsse der Steckverbinder noch Einflüsse des Übertragungskabels berücksichtigt werden. Für hohe Taktfrequenzen des Serializers ist die oben erwähnte Speicherdrossel zum Einkoppeln der Phantomspannung jedoch scheinbar geeignet.

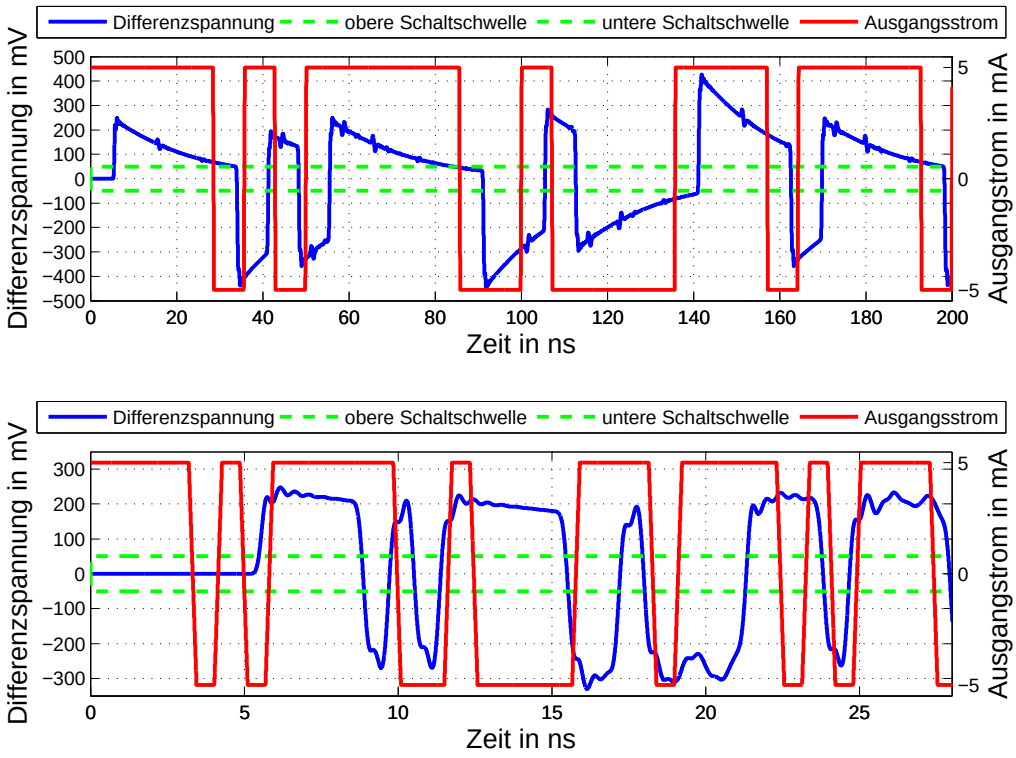


Abbildung 4.9: Simulation der Auswirkung von 1 μH Koppelinduktivitäten auf das Datensignal;
oben: Simulationsergebnis bei einer Taktfrequenz von 5 MHz;
unten: Simulationsergebnis bei einer Taktfrequenz von 43 MHz;

10 μH Koppelinduktivität

Verwendet man jedoch die Speicherdrossel 744028100, welche eine Eigeninduktivität von 10 μH, einen maximalen Gleichstromwiderstand von 0.67 Ω, einen Nennstrom von 0.45 A und eine Eigenresonanzfrequenz von 45 MHz aufweist, so ist aus Abbildung 4.10 zu erkennen, dass diese Drossel wahrscheinlich sowohl für niedrige als auch für hohe Taktfrequenzen des Serializers geeignet ist.

Nachteilig wirkt sich bei dieser Speicherdrossel jedoch der relativ hohe Gleichstromwiderstand aus. Besitzt ein Peripheriegerät die höchstzulässige Stromaufnahme von 500 mA, so entsteht über den Strompfad der Phantomspeisung an den Koppelinduktivitäten ein Spannungsabfall von 0.67 V, wodurch möglicherweise Spannungsversorgungsgrenzen von integrierten Schaltkreisen auf dem Peripheriegerät unterschritten werden.

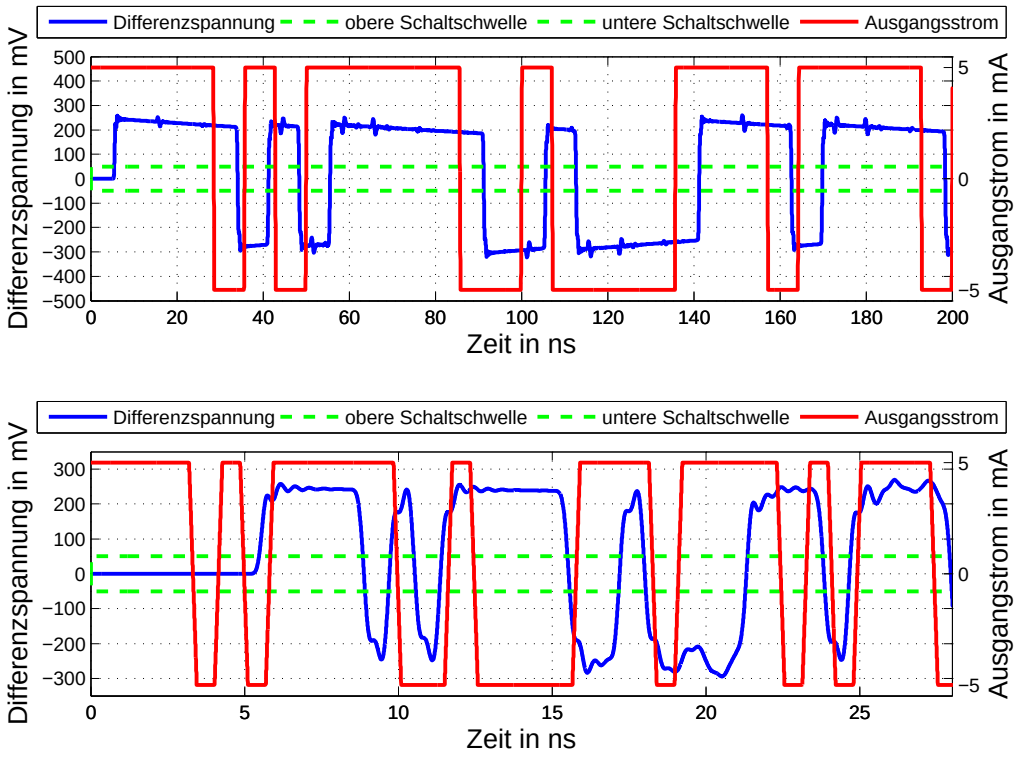


Abbildung 4.10: Simulation der Auswirkung von 10 μH Koppelinduktivitäten auf das Datensignal;
oben: Simulationsergebnis bei einer Taktfrequenz von 5 MHz;
unten: Simulationsergebnis bei einer Taktfrequenz von 43 MHz;

Stromkompensierte Drossel

Zur Einkopplung der Phantomspannung auf die Datenleitungen des RPI-Busses wären auch stromkompensierte Drosseln denkbar. Diese Drosseln bestehen aus zwei gleichen Wicklungen, welche um einen gemeinsamen Kern gewickelt sind – siehe Abbildung 4.11. Für Gleichtaktströme stellt eine stromkompensierte Drossel eine sehr hohe Induktivität dar, da sich die vom Gleichtaktstrom erzeugten Magnetfelder im gemeinsamen Kernmaterial addieren – Gegentaktströme werden durch die stromkompensierte Drossel hingegen kaum beeinflusst.

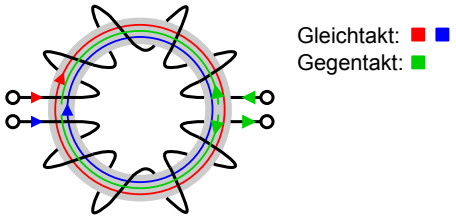


Abbildung 4.11: Funktionsprinzip einer stromkompensierten Drossel (vgl. [19])

Die stromkompensierte Drossel muss so in die Schaltung eingefügt werden, dass sie im Hinblick auf das hochfrequente Datensignal im Gleichtakt betrieben wird und gegensinnig vom Versorgungsstrom der Wearables durchflossen wird. Abbildung 4.12 zeigt das Simulationsmodell für die Verwendung von stromkompensierten Drosseln zum Ein- und Auskoppeln der Phantomspannung.

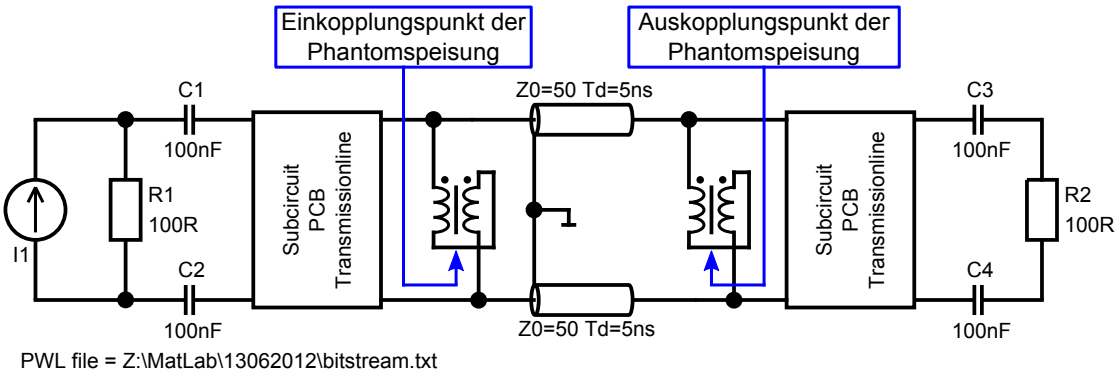


Abbildung 4.12: Simulationsmodell für die LVDS-Übertragung mit stromkompensierten Drosseln zum Ein- und Auskoppeln der Phantomspannung

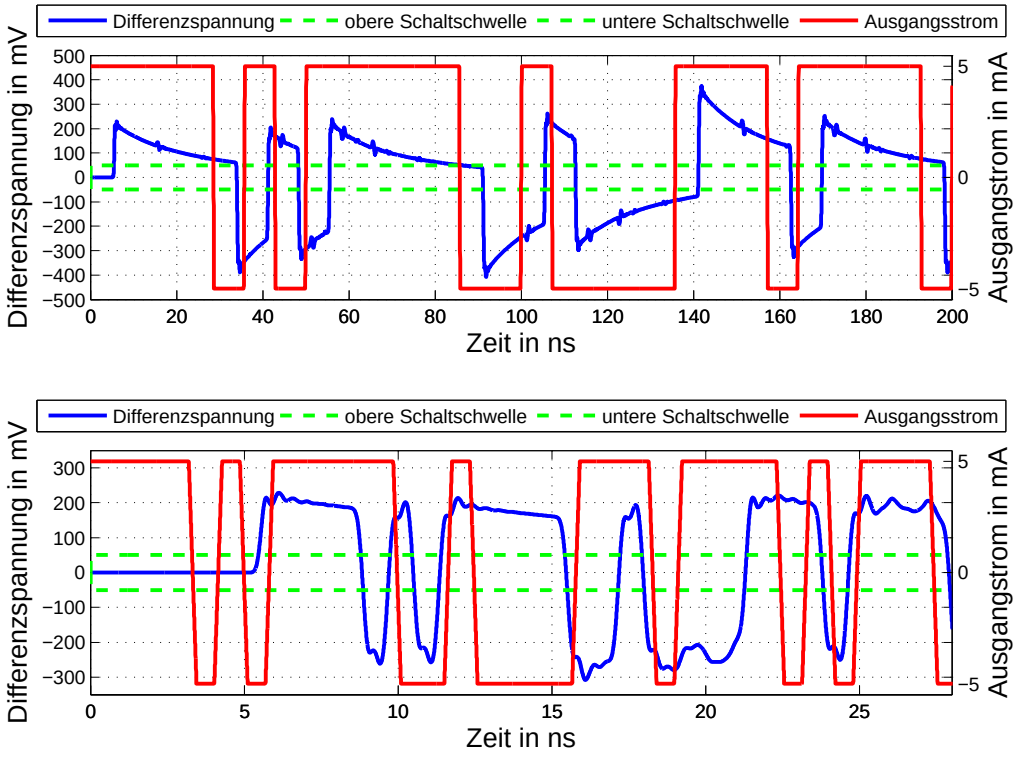


Abbildung 4.13: Simulation der Auswirkung von stromkompensierten Drosseln auf das Datensignal;
oben: Simulationsergebnis bei einer Taktfrequenz von 5 MHz;
unten: Simulationsergebnis bei einer Taktfrequenz von 43 MHz;

Abbildung 4.13 zeigt das Simulationsergebnis bei der Verwendung der stromkompensierten Drossel 744231121 von Würth Elektronik. Die elektrischen Eigenschaften

dieser Drossel sind gegeben mit einem Gleichstromwiderstand von 0.3 Ω, einer Nennstrombelastbarkeit von 0.37 A und einer Impedanz von ≥ 100 Ω ab 80 MHz im Gleichtaktbetrieb. Es ist zu erkennen, dass die stromkompensierte Drossel bei niedrigen Taktfrequenzen des Serializers ungeeignet ist, da bei der Übertragung von mehreren gleichen Bits die Schaltschwellen des Deserializers unterschritten werden.

SMD-Ferrite

Auch die Verwendung von SMD-Ferrite wäre zur Einkopplung der Phantomspannung denkbar. Diese induktiven Bauelemente werden üblicherweise zur Unterdrückung von hochfrequenten, leitungsgebundenen Störungen eingesetzt und sind in Multilayer-Technologie aufgebaut, wobei als Ferritmaterial eine besondere Nickel-Zink Legierung dient, sodass bei hohen Frequenzen die Impedanz des Bauteils maßgeblich durch einen resistiven Anteile bestimmt wird (vgl. [19]), o.S.). Für Gleichspannungen hingegen, stellt ein SMD-Ferrit jedoch nur einen sehr kleinen Widerstand dar und scheint somit bestens zur Einkopplung der Phantomspannung auf die Busleitungen geeignet zu sein.

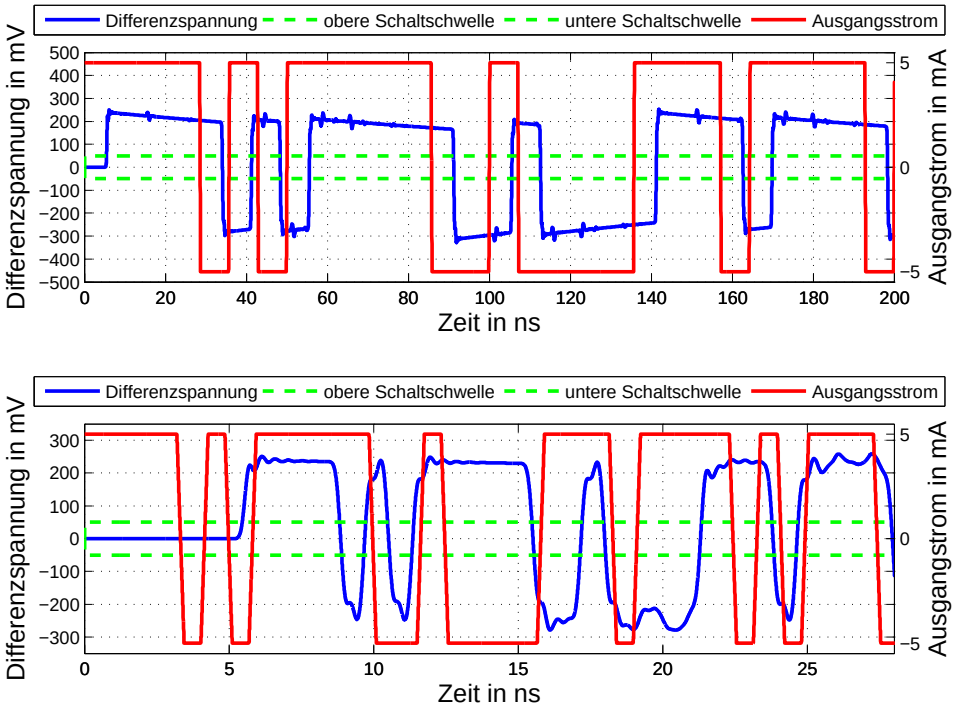


Abbildung 4.14: Simulation der Auswirkung von SMD-Ferrite auf das Datensignal; oben: Simulationsergebnis bei einer Taktfrequenz von 5 MHz; unten: Simulationsergebnis bei einer Taktfrequenz von 43 MHz;

Abbildung 4.14 zeigt das Simulationsergebnis des Modells 4.8, bei dem für die Koppelelemente L_{ki1}, L_{ki2}, L_{ko1} und L_{ko2} jeweils der SMD-Ferrit 742792097 von

Würth Elektronik eingesetzt wird. Dieser SMD-Ferrit weist einen Gleichspannungswiderstand von 300 mΩ, eine Nennstrombelastbarkeit von 1 A und eine Impedanz von $\geq 100\,\Omega$ ab 2 MHz auf. Es ist zu erkennen, dass sich dieses Bauteil sowohl bei niedrigen als auch bei hohen Taktfrequenzen des Serializers zur Einkopplung der Phantomspannung eignet.

4.3.3 Entscheidungsfindung

Aus den – unter Punkt 4.3.2 – erhaltenen Ergebnissen ist ersichtlich, dass sich sowohl Speicherdrosseln mit einer entsprechend hohen Eigeninduktivität, als auch SMD-Ferrite mit geeignetem Impedanzverlauf als Koppelelemente eignen. Bei genauerer Betrachtung dieser beiden Bauelemente ist auffallend, dass Speicherdrosseln mit vergleichbar niedrigem Gleichspannungswiderstand und ausreichender Nennstrombelastbarkeit wesentlich größere Bauteilabmessungen aufweisen als SMD-Ferrite. Zusätzlich sind SMD-Ferrite preislich günstiger, wodurch die Entscheidung zum einkoppeln der Phantomspannung auf den SMD-Ferrit 742792097 von Würth Elektronik fällt.

Bauteil	**Typ**	$\mathbf{R_{DC}}$	**Abmessungen**	**Preis bei Farnell**
744028100	Speicherdrossel	0.67 Ω	2.8 mm × 2.8 mm	0.974€
742792097	SMD-Ferrit	0.3 Ω	2.0 mm × 1.2 mm	0.205€

Tabelle 4.6: Vergleich der Speicherdrossel 744028100 mit dem SMD-Ferrit 742792097 im Hinblick auf Bauteilgröße, Preis und Gleichspannungswiderstand

5 Konzeptausführung

Es wird ein erster Prototyp – in weiterer Folge LVDS-Testboard genannt – entwickelt, welcher zum Testen der ausgewählten Hardware-Komponenten dienen soll. Über den Downchannel werden bei diesem Prototypen jedoch nur die Signale zur Videoausgabe übertragen. Über den Upchannel kann ein einstellbares Bitmuster gesendet werden, welches mittels LEDs visualisiert wird. Auch die Einkopplung der Phantomspeisung, sowie verschiedene Strombelastungen der Busleitungen sollen mittels dieses Prototyps getestet werden.

Der Prototyp soll dabei anstatt der TFT-Display-Adapterplatine an das i.MX31 Developementboard angeschlossen werden und selbst den Anschluss der Adapterplatine ermöglichen.

5.1 Schaltungsentwurf

In weiterer Folge werden Ausschnitte des erstellten Stromlaufplanes beschrieben. Das Original des Stromlaufplanes befindet sich im Anhang A, sowie auf einer, dieser Masterarbeit beliegenden DVD.

5.1.1 Downchannel

Abbildung 5.1 zeigt die notwendigen Schaltungskomponenten zur Serialisierung des Videosignals des i.MX31. Diese Bauteile bilden die Sendeeinrichtung für den Downchannel. Abbildung 5.2 zeigt die Empfangseinrichtung für den Downchannel. Diese Baugruppen sollen später in den Weareables verbaut werden.

Steckverbinder `X1` (Abb. 5.1)
Der 72-polige Steckverbinder dient zur Verbindung des Prototypen mit dem i.MX31 Developementboard. Die zum Testen der Videoübertragung relevanten Signale `ipu0` bis `ipu17`, `hsync`, `dtmg`, `vctrl`, sowie der Pixeltakt `pixclk` werden von `X1` abgezweigt und dem Serializer `IC4` zugeführt. Auch die Spannungsversorgung für die integrierten Schaltungen des LVDS-Testboards wird mit `X1` vom Developementboard abgegriffen.

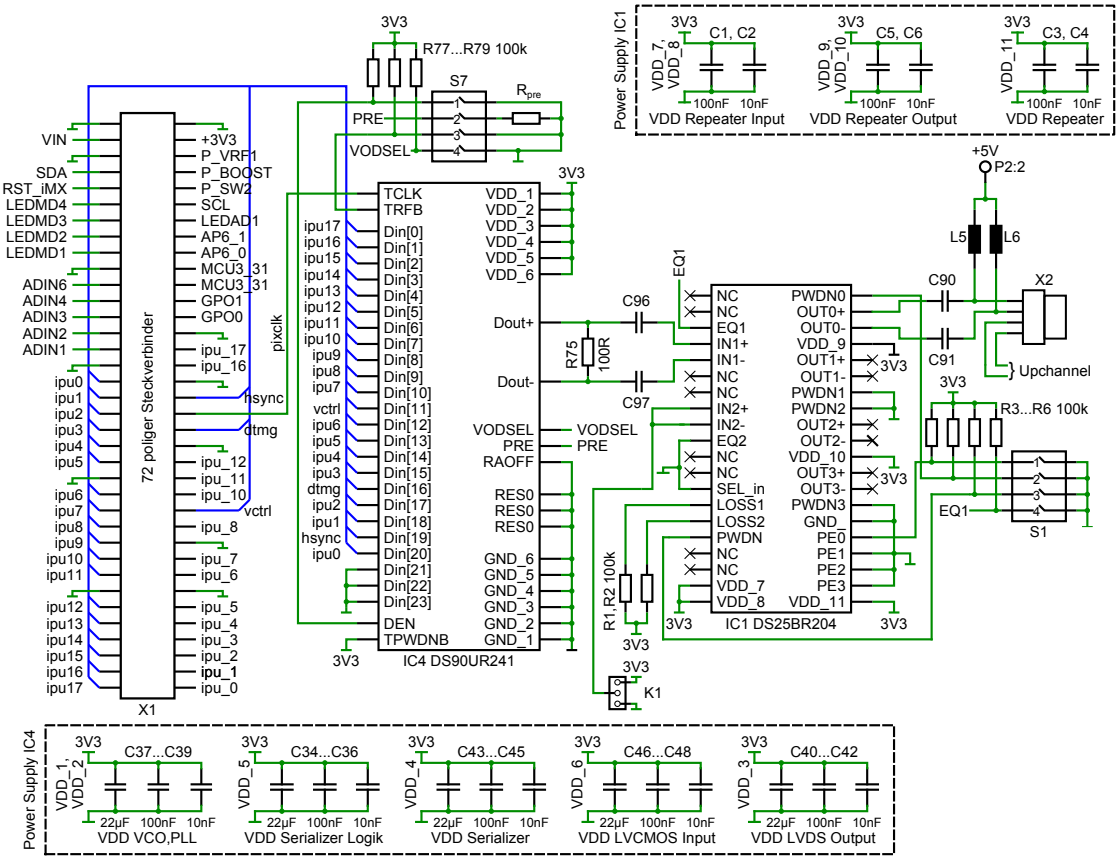


Abbildung 5.1: Schaltplanausschnitt für die Sendeeinrichtung des Downchannels

USB Typ A Stecker `X2` (Abb. 5.1) - **USB Typ A Buchse** `X3` (Abb. 5.2)
Die Steckverbinder `X2` und `X3` dienen zur Verbindung der Sendeseite mit der Empfangsseite des Downchannels bzw. zur Verbindung der Empfangsseite mit der Sendeseite des Upchannels über ein USB-Kabel. Somit kann das Hot-Plugging von Busteilnehmern durch einfaches Entfernen und Anschließen des Kabels getestet werden.

Steckverbinder `X4` (Abb. 5.2)
Der 72-polige Steckverbinder `X4` dient zum Anschluss der TFT-Display-Adapterplatine. Die vom Deserializer `IC5` deserialisierten Videosignale (`ld0` - `ld17`, `dvctrl`, `ddtmg`, `dhsync`, `dpixclk`) werden an die entsprechenden Pins von `X4` geführt, sodass ein an die Adapterplatine angeschlossenes TFT-Display richtig funktioniert.

Serializer `IC4` (Abb. 5.1)
`IC4` wird mit dem Pixeltakt `pixclk` getaktet und serialisiert die Datensignale `ipu0` bis `ipu17`, sowie die Steuersignale `hsync`, `dtmg` und `vctrl`. Die drei Eingänge des Serializers, welche in weiterer Folge zur Übertragung von Audio- und Steuerdaten verwendet werden sollen, sind bei diesem Entwurf mit GND verbunden. Die LVDS-Ausgänge des Serializers sind mit `R75` abgeschlossen und über die Kapazitäten `C96` und `C97` gleichspannungsmäßig vom LVDS-Repeater `IC1` entkoppelt.

Spannungsversorgung von `IC4` (Abb. 5.1)
Der Serializer verfügt über sechs separate Spannungsversorgungsanschlüsse, die den internen Baugruppen des Serializers zugeordnet sind. Alle Spannungsversorgungsanschlüsse sollten mit einer Parallelschaltung von drei Kapazitäten (22 μF, 100 nF, 10 nF) gepuffert werden, wobei die Spannungsversorgung der Phasenregelschleife (PLL) und des spannungsgesteuerten Oszillators (VCO) mit den selben Kapazitäten gepuffert werden kann.

LVDS-Repeater `IC1` (Abb. 5.1)
Der LVDS-Repeater verfügt über zwei differentielle Eingangsschnittstellen. Mittels einer Select-Leitung, die permanent auf GND liegt, werden die Eingänge `IN1+` und `IN1-` ausgewählt. Die Eingänge der zweiten Eingangsschnittstelle sind kurzgeschlossen und können über einen Jumper (`K1`) entweder auf Masse- oder VDD-Potential gelegt werden, um einer Fehlfunktion des LVDS-Repeaters vorzubeugen. `IC1` verfügt über vier differentielle Ausgangsschnittstellen, die alle über separate `PWDNn`-Eingänge deaktiviert werden können. Die drei nicht verwendeten Ausgangsschnittstellen werden durch die Beschaltung der Eingänge `PWDN1`, `PWDN2` und `PWDN3` mit Massepotential permanent deaktiviert. Die differentiellen Ausgänge `OUT0+` und `OUT0-` sind über die Koppelkapazitäten `C90` und `C91` mit dem Stecker `X2` verbunden.

Spannungsversorgung von `IC1` (Abb. 5.1)
Auch der LVDS-Repeater verfügt über mehrere separate Spannungsversorgungsanschlüsse, wobei National Semiconductors hier keine Auskunft über eine interne Baugruppenzugehörigkeit angibt. Aufgrund der Positionierung der Anschlüsse beim Chipgehäuse kann jedoch vermutet werden, dass jeweils ein Spannungsversorgungsanschluss für die differentiellen Eingangsstufen gedacht ist. Zwei weitere Anschlüsse dienen vermutlich zur Versorgung der vier Ausgangstreiber und ein weiterer Anschluss könnte für die restlichen Baugruppen im Chip gedacht sein. Da jedoch nur eine Eingangsschnittstelle und eine Ausgangsschnittstelle verwendet wird, werden nicht alle Spannungsversorgungsanschlüsse separat mit Pufferkapazitäten versehen. Aus dem Referenzdesign für den LVDS-Repeater (vgl. [20]. S.8) wurde entnommen, dass die Spannungsversorgungsanschlüsse mit einer Parallelschaltung von 100 nF und 10 nF Kapazitäten gepuffert werden sollen.

Deserializer `IC5` (Abb. 5.2)
Die LVDS-Eingänge des Deserializers werden mit den Kondensatoren `C98` und `C99` kapazitiv entkoppelt und mit dem Widerstand `R76` abgeschlossen. `IC5` deserialisiert schließlich die über den Downchannel übertragenen Videosignale. Die nicht verwendeten Ausgänge des parallelen Interfaces vom Deserializer werden nicht angeschlossen. Die Kontrollpins des Built-In-Selftests des Deserializers werden so beschaltet, dass dieser inaktiv ist. Die LED `D26` visualisiert den Zustand der PLL und leuchtet, wenn das Taktsignal aus dem seriellen Bitstrom extrahiert werden kann.

Spannungsversorgung `IC5` (Abb. 5.2)
Der Deserializer verfügt über acht separate Spannungsversorgungsanschlüsse, die

den internen Baugruppen des Deserializers zugeordnet sind. Alle Spannungsversorgungsanschlüsse sollten wieder mit einer Parallelschaltung von drei Kapazitäten (22 μF, 100 nF, 10 nF) gepuffert werden, wobei aus Platzgründen auf dem Layout auf die Pufferkapazitäten für die LVDS-Eingangsstruktur verzichtet werden musste.

Koppelkapazitäten C90**,** C91**,** C96 **und** C97 (Abb. 5.1)
Die Koppelkapazitäten zur gleichspannungsmäßigen Entkopplung der Signalleitungen müssen eine Kapazität von 100 nF sowie ein NPO oder X7R Dielektrikum aufweisen. Zusätzlich sollte eine möglichst kleine Bauform bei den Koppelkapazitäten verwendet werden, da kleinere Bauformen mit geringeren parasitären Effekten einhergehen.

SMD-Ferrite L1**,** L2 (Abb. 5.2) **und** L5**,** L6 (Abb. 5.1)
Die SMD-Ferrite L5 und L6 dienen zum Einkoppeln der an P2:1 angelegten Gleichspannung auf die Busleitungen des Downchannels. Mit den SMD-Ferriten L1 und L2 wird die Gleichspannung auf der Empfangsseite des Downchannels über P1:1 wieder ausgekoppelt.

Schalter S1 (Abb. 5.1)
Mittels des vier-poligen Schalters S1 erfolgt die Konfiguration des LVDS-Repeaters. Die entsprechenden Eingänge von IC1 liegen bei geöffneten Schaltern über die Pullup-Widerstände R3 bis R6 auf VDD-Potential und werden durch das schließen der Schalter auf GND gezogen.

Pos.		Konfiguration des LVDS-Repeaters IC1mittels Schalter S1
1	On	PE0=L; Preemphasis für LVDS-Ausgang OUT0+,OUT0- deaktiviert
	Off	PE0=H; Preemphasis für LVDS-Ausgang OUT0+,OUT0- aktiviert
2	On	PWDN0=L; LVDS-Ausgang OUT0+,OUT0- deaktiviert
	Off	PWDN0=H; LVDS-Ausgang OUT0+,OUT0- aktiviert
3	On	PWDN=L; LVDS-Repeater deaktiviert
	Off	PWDN=H; LVDS-Ausgang OUT0+,OUT0- aktiviert
4	On	EQ1=L; Signalaufbereitung für LVDS-Eingang IN1+, IN1- deaktiviert
	Off	EQ1=H; Signalaufbereitung für LVDS-Eingang IN1+, IN1- aktiviert

Tabelle 5.1: Schalterstellungen zur Konfiguration des LVDS-Repeaters IC1

Schalter S7 (Abb. 5.1)
Der vier-polige Schalter S7 ermöglicht die Konfiguration des Serializers IC4. Bei geöffneter Schalterstellung liegen die Eingänge TRFB, VODSEL und DEN über die Pullup-Widerstände R77 bis R79 auf VDD-Potential. Bei geschlossener Schalterstellung werden die Eingänge von IC4 auf GND gezogen. Der PRE-Eingang des Serializers hingegen floatet bei geöffneter Schalterstellung, während bei geschlossener Schalterstellung über den Widerstand R_{pre} ein Strom zur Preemphasis eingeprägt wird.

Pos.		Konfiguration des Serializers IC4 mittels Schalter S7
1	On	DEN=L; LVDS-Ausgänge in TRI-STATE; PLL läuft
	Off	DEN=H; LVDS-Ausgänge sind aktiviert
2	On	PRE=on; Preemphasis wird über Widerstand eingestellt
	Off	PRE=off; Preemphasis-Eingang ist offen
3	On	TRFB=L; paralleles Interface wird mit ⌞-Taktflanke eingelesen
	Off	TRFB=H; paralleles Interface wird mit ⌜-Taktflanke eingelesen
4	On	VODSEL=L; Ausgangsdifferenzspannung gleich ±500 mV
	Off	VODSEL=H; Ausgangsdifferenzspannung gleich ±900 mV

Tabelle 5.2: Schalterstellungen zur Konfiguration des Serializers IC4

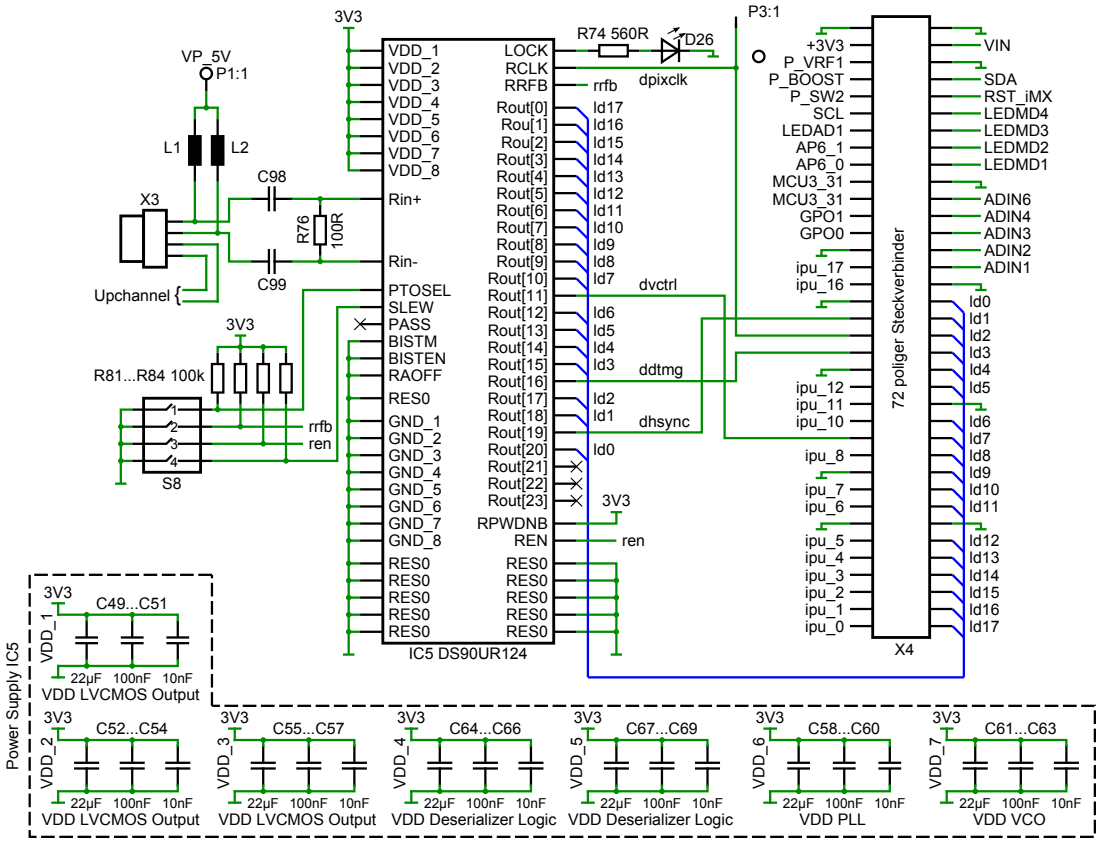


Abbildung 5.2: Schaltplanausschnitt für die Empfangseinrichtung des Downchannels

Schalter S8 (Abb. 5.2)
Der vier-polige Schalter S8 ermöglicht die Konfiguration des Deserializers IC4. Bei geöffneter Schalterstellung liegen die Eingänge RRFB, PTOSEL, REN und SLEW über die Pullup-Widerstände R81 bis R84 auf VDD-Potential. Bei geschlossener Schalterstellung werden die entsprechenden Eingänge von IC5 auf GND gezogen.

Pos.	Konfiguration des Deserializers IC5 mittels Schalter S8	
1	On	PTOSEL=L; Ausgangsgruppen schalten immer gleich
	Off	PTOSEL=H; Spread-Mode
2	On	RRFB=L; LVCMOS-Ausgänge sind mit ⌝-Taktflanke gültig
	Off	RRFB=H; LVCMOS-Ausgänge sind mit ⌟-Taktflanke gültig
3	On	REN=L; LVCMOS-Ausgänge in TRI-STATE; PLL läuft
	Off	REN=H; LVCMOS-Ausgänge sind aktiviert
4	On	SLEW=L; niedrige LVCMOS-Ausgangstreiberstärke (2 mA)
	Off	SLEW=H; hohe LVCMOS-Ausgangstreiberstärke (4 mA)

Tabelle 5.3: Schalterstellungen zur Konfiguration des Deserializers IC5

5.1.2 Upchannel

In Abbildung 5.3 sind die notwendigen Baugruppen für die Sende- und Empfangseinrichtung des Upchannels dargestellt.

Schalter S2, S3 und S4
Mittels der zwei zehn-poligen Schalter S2 und S3, sowie des vier-poligen Schalters S4 kann ein 24-stelliges Bitmuster eingestellt werden, welches über den Upchannel übertragen wird.

Leuchtdioden D1 bis D24
Diese Leuchtdioden dienen zur Darstellung des an den Schaltern S2, S3 und S4 eingestellten Bitmusters, sowie zur Verifikation der Datenübertragung über den Upchannel. Die Vorwiderstände für die Leuchtdioden wurden mit 560 Ω gewählt.

Serializer IC3
IC3 kann entweder mit dem des Deserializers IC5 rückgewonnenen Takts, oder mit einem externen Takt, getaktet werden und serialisiert das an den Schaltern S2, S3 und S4 eingestellte Bitmuster. Die LVDS-Ausgänge des Serializers sind wieder mit 100 Ω differentiell abgeschlossen und kapazitiv über C94 und C95 vom weiteren Übertragungsweg entkoppelt.

Spannungsversorgung von IC3
Die Spannungsversorgung des Serializers IC3 erfolgt identisch wie bei IC4.

Schalter S5
Der vier-polige Schalter S5 dient zur Konfiguration des Serializers IC3. Bei geöffneter Schalterstellung liegen die Steuerleitungen vodsel, den und trfb des Serializers über Pullup-Widerstände auf VDD-Potential und werden beim Schließen der Schalterpositionen auf Masse gezogen. Über den Widerstand R_{pre} wird bei geschlossener Schalterstellung ein Strom zur Preemphasis eingeprägt, während bei geöffneter Schalterposition die Preemphasis deaktiviert ist.

Pos.		Konfiguration des Serializers IC3 mittels Schalter S5
1	On	VODSEL=L; Ausgangsdifferenzspannung gleich ±500 mV
	Off	VODSEL=H; Ausgangsdifferenzspannung gleich ±900 mV
2	On	PRE=on; Preemphasis wird über Widerstand eingestellt
	Off	PRE=off; Preemphasis-Eingang ist offen
3	On	DEN=L; LVDS-Ausgänge in TRI-STATE; PLL läuft
	Off	DEN=H; LVDS-Ausgänge sind aktiviert
4	On	TRFB=L; parallels Interface wird mit ⌝-Taktflanke eingelesen
	Off	TRFB=H; paralleles Interface wird mit ⌞-Taktflanke eingelesen

Tabelle 5.4: Schalterstellungen zur Konfiguration des Serializers IC3

Deserializer IC2
Die LVDS-Eingänge des Deserializers sind kapazitiv mittels C92 und C93 entkoppelt und über den Widerstand R35 differentiell abgeschlossen. IC2 deserialisiert das über den Upchannel übertragene Bitmuster und treibt mit seinen LVCMOS-Ausgängen die Leuchtdioden D1 bis D24. Über die LED D25 wird der Zustand der PLL visualisiert. Der Built-In-Selftest von IC2 ist ebenfalls – wie beim Deserializer des Downchannels – permanent deaktiviert.

Spannungsversorgung von IC2
Auch beim Deserializer teilen sich die Baugruppen VCO und PLL die Pufferkapazitäten. Zusätzlich wird für die drei LVCMOS-Ausgangsbänke nur eine Parallelschaltung von Pufferkapazitäten vorgesehen, da aufgrund der manuellen Einstellung des zu übertragenden Bitmusters mit sehr langsamen Schaltvorgängen der LVCMOS-Ausgänge gerechnet werden kann und somit die Pufferkapazitäten nur selten Strom zur Verfügung stellen müssen.

Schalter S6
Der vier-polige Schalter S6 dient zur Konfiguration des Deserializers IC2. Bei geöffneter Schalterstellung liegen die Eingänge rrfb, ptosel, ren und slew über die Pullup-Widerstände R70 bis R73 auf VDD-Potential, während die entsprechenden Eingänge durch das Schließen der Schalterpositionen auf Masse gezogen werden.

Pos.		Konfiguration des Deserializers IC2 mittels Schalter S6
1	On	SLEW=L; niedrige LVCMOS-Ausgangstreiberstärke (2 mA)
	Off	SLEW=H; hohe LVCMOS-Ausgangstreiberstärke (4 mA)
2	On	REN=L; LVCMOS-Ausgänge in TRI-STATE; PLL läuft
	Off	REN=H; LVCMOS-Ausgänge sind aktiviert
3	On	RRFB=L; LVCMOS-Ausgänge sind mit ⌝-Taktflanke gültig
	Off	RRFB=H; LVCMOS-Ausgänge sind mit ⌞-Taktflanke gültig
4	On	PTOSEL=L; Ausgangsgruppen schalten immer gleich
	Off	PTOSEL=H; Spread-Mode

Tabelle 5.5: Schalterstellungen zur Konfiguration des Deserializers IC2

SMD-Ferrite L3, L4, L7 und L8
Die SMD-Ferrite L3 und L4 dienen zum Einkoppeln des Stromrückflusses der Phantomspeisung auf die Busleitungen. Über die SMD-Ferrite L7 und L8 wird dieser Strom wieder aus den Busleitungen ausgekoppelt und zur Phantomspannungsquelle zurück geführt. Es besteht somit ein geschlossener Gleichstrompfad: Phantomspannungsquelle → L5, L6 → Busleitungen des Downchannels → L1, L2 → Last → L3, L4 → Busleitungen des Upchannels → L7, L8 → Phantomspannungsquelle.

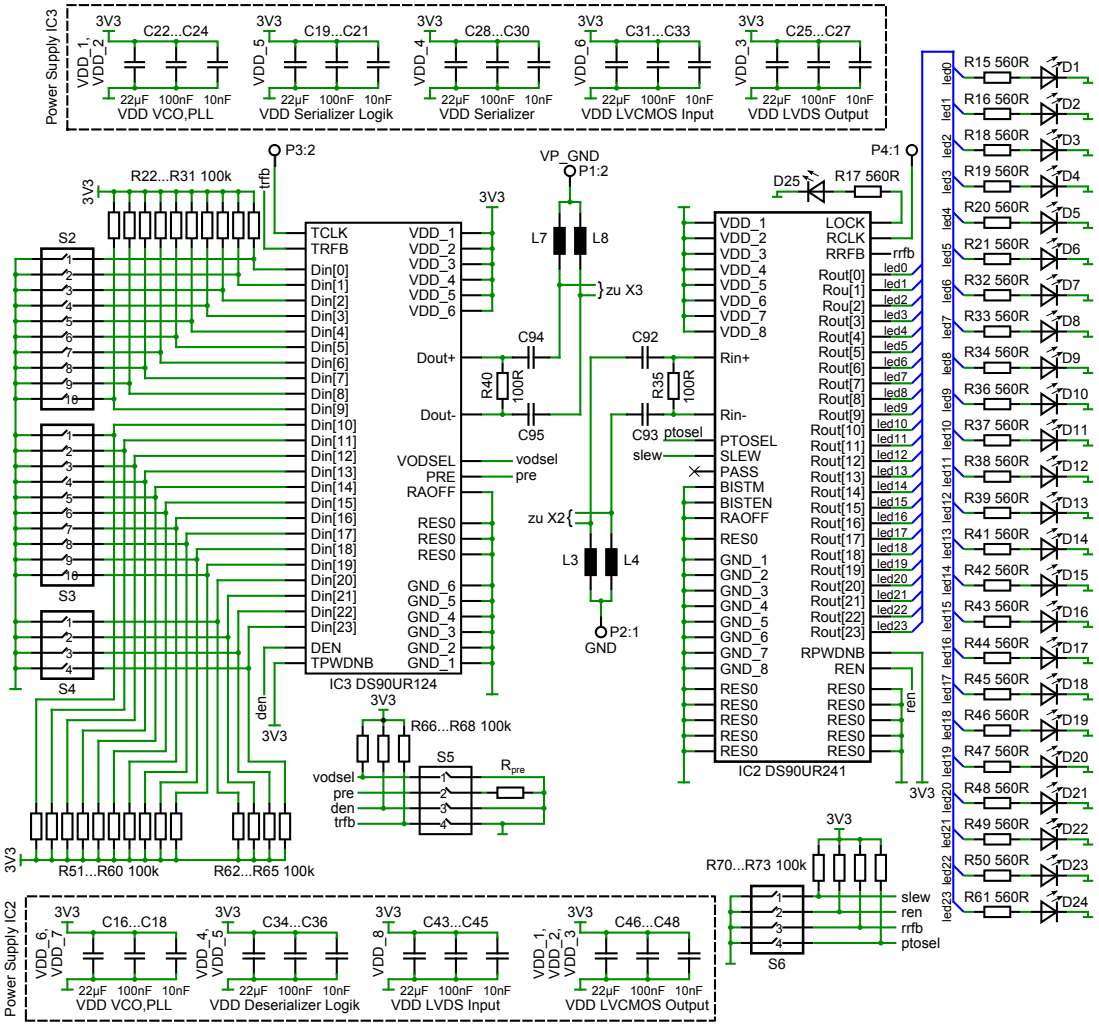


Abbildung 5.3: Schaltungskomponenten für den Upchannel

5.1.3 Schaltbare Belastung der Phantomspannung

Um die Auswirkungen verschiedener Strombelastungen der Phantomspeisung auf die Datenübertragung zu untersuchen, wird ein schaltbares Widerstandsnetzwerk eingeplant. Das Widerstandsnetzwerk, bestehend aus R7 bis R14 kann über den Jumper JP1 von der Phantomspeisung getrennt werden, sodass die Datenübertragung über den Down- und Upchannel auch bei Leerlauf der Phantomspeisung getestet werden kann.

Zum Ein- bzw. Ausschalten der Lastwiderstände werden N-Kanal MOS-Transistoren als Low-Side Schalter verwendet, da diese bessere Eigenschaften aufweisen als P-Kanal Transistoren als High-Side Schalter. Beim verwendeten Transistortyp handelt es sich um die integrierte Schaltung PMGD370XN von NXP, welche zwei N-Kanal MOS-Transistoren auf demselben Substrat aufweist. Die Transistoren erlauben das Schalten von maximal 0.74 A und weisen bei einer Steuerspannung von 5 V einen On-Widerstand $R_{on} \leq 400\,\text{m}\Omega$ auf (vgl. [21], S.1).

Die Ansteuerung der einzelnen Transistoren erfolgt über den Steckverbinder P5, wobei sichergestellt werden muss, dass die Gates der Transistoren immer mit einer definierten Steuerspannung beschaltet sind.

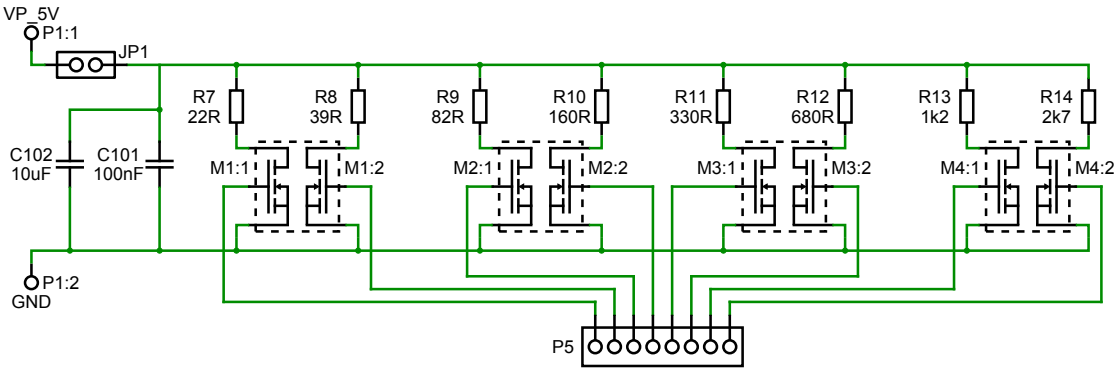


Abbildung 5.4: Über NMOS-Transistoren schaltbare Belastung der Phantomspeisung

5.2 Bauteildimensionierungen

In den folgenden Abschnitten werden die notwendigen Bauteildimensionierungen vorgenommen.

5.2.1 Dimensionierung der LED Vorwiderstände

Die LEDs auf dem LVDS-Testboard werden allesamt von LVCMOS-Ausgängen der Deserializer angesteuert. Laut Datenblatt des Deserializers liefern die LVCMOS-Ausgänge einen maximalen High-Pegel von 3.3 V und können je nach Konfiguration einen Ausgangsstrom von ±2 mA oder ±4 mA treiben. Die LEDs, welche zur Visualisierung der PLL-Zustände und des über den Upchannel übertragenen Bitmusters verwendet werden besitzen laut Datenblatt [22] bei einem Stromfluss von 1 mA eine Vorwärtsspannung von maximal 2.8 V.

$$R_V = \frac{V_{OH,max} - V_{F,max}}{I_F} = \frac{3.3\,\text{V} - 2.8\,\text{V}}{1\,\text{mA}} = 500\,\Omega$$

Es werden Widerstände mit 560 Ω zur Strombegrenzung durch die Leuchtdioden verwendet um die Strombelastung der LVCMOS-Ausgänge von ±2 mA auch bei typischen Betriebsbedingungen nicht zu überschreiten.

$$I_{typ} = \frac{V_{OH,typ} - V_{F,typ}}{R_V} = \frac{3.0\,\mathrm{V} - 2.1\,\mathrm{V}}{560\,\Omega} = 1.6\,\mathrm{mA}$$

5.2.2 Dimensionierung der schaltbaren Strombelastung

Die Widerstände des schaltbaren Widerstandsnetzwerks werden so dimensioniert, dass ein Stromfluss von 500 mA auftritt, sobald alle Transistoren eingeschaltet sind. Der Stromfluss durch die einzelnen Transistoren soll dabei binär gewichtet sein, wodurch sich folgende Teilströme durch die einzelnen Widerstände ergeben.

I_{R7}	I_{R8}	I_{R9}	I_{R10}	I_{R11}	I_{R12}	I_{R13}	I_{R14}
250 mA	125 mA	62.5 mA	31.3 mA	15.6 mA	7.8 mA	3.9 mA	2 mA

Tabelle 5.6: binär gewichtete Teilströme durch das Widerstandsnetzwerk

Die Widerstände des schaltbaren Netzwerks werden nun so dimensioniert, dass sich bei einer Phantomspannung VP_5V = 5 V in etwa die in Tabelle 5.6 angeführten Teilströme ergeben. Da jedoch durch die Spannungsabfälle an den SMD-Ferriten, welche von der Strombelastung der Phantomspeisung abhängig sind, nicht die vollständige eingespeiste Phantomspannung am Widerstandsnetzwerk zur Verfügung steht wird die Berechnung des entsprechenden Widerstandswert jeweils für zwei Fälle betrachtet. Im ersten Fall sind alle Widerstände eingeschaltet und es fließt der maximale Strom über die Busleitungen und im zweiten Fall ist nur der aktuell betrachtete Widerstand aktiv. Zusätzlich wird auch der nicht ideale On-Widerstand der Transistoren in die Dimensionierung miteinbezogen.

$$R_{i,min} = \frac{VP\_5V - (500\,\mathrm{mA} \cdot R_{ferrit} \cdot 2)}{I_{Ri}} - R_{on} \tag{5.1}$$

$$R_{7,min} = \frac{5\,\mathrm{V} - (500\,\mathrm{mA} \cdot 0.3\,\Omega \cdot 2)}{250\,\mathrm{mA}} - 0.4\,\Omega = 18.4\,\Omega$$

$$R_{i,max} = \frac{VP\_5V - (I_{Ri} \cdot R_{ferrit} \cdot 2)}{I_{Ri}} - R_{on} \tag{5.2}$$

$$R_{7,max} = \frac{5\,\mathrm{V} - (250\,\mathrm{mA} \cdot 0.3\,\Omega \cdot 2)}{250\,\mathrm{mA}} - 0.4\,\Omega = 19\,\Omega$$

In Tabelle 5.7 sind die mittels der Formeln 5.1 und 5.2 berechneten Widerstandswerte und die tatsächlich gewählten Widerstände gegenübergestellt.

	R7	R8	R9	R10	R11	R12	R13	R14
max.	19 Ω	39 Ω	79 Ω	159 Ω	319 Ω	639 Ω	1279 Ω	2559 Ω
min.	18.4 Ω	37.2 Ω	74.8 Ω	150 Ω	300.4 Ω	601.2 Ω	1202.8 Ω	2406 Ω
gewählt	18 Ω	39 Ω	82 Ω	150 Ω	330 Ω	680 Ω	1200 Ω	2700 Ω

Tabelle 5.7: Berechnete und schließlich gewählte Widerstandswerte für das schaltbare Widerstandsnetzwerk

Mit den aus der E12-Reihe gewählten Widerständen lässt sich unter Berücksichtigung der Serienwiderstände der SMD-Ferrite und der On-Widerstände der Transistoren eine maximale Strombelastung der Phantomspeisung von in etwa 490 mA einstellen.

Abbildung 5.5 zeigt ein Widerstand/Strom-Diagramm für eine am Widerstand auftretende Verlustleistung von 125 mW, wobei es sich hier um die maximal zulässige Verlustleistung P_{tot} von konventionellen Widerständen handelt. Es ist ersichtlich, dass für die Widerstände R7 bis R10 die Strombelastung zu hoch ist um Widerstände verwenden zu können, welche für eine Verlustleistung von 125 mW konzipiert sind. Deshalb werden für R7 bis R10, Leistungswiderstände mit der maximal zulässigen Verlustleistung von 1 W verwendet.

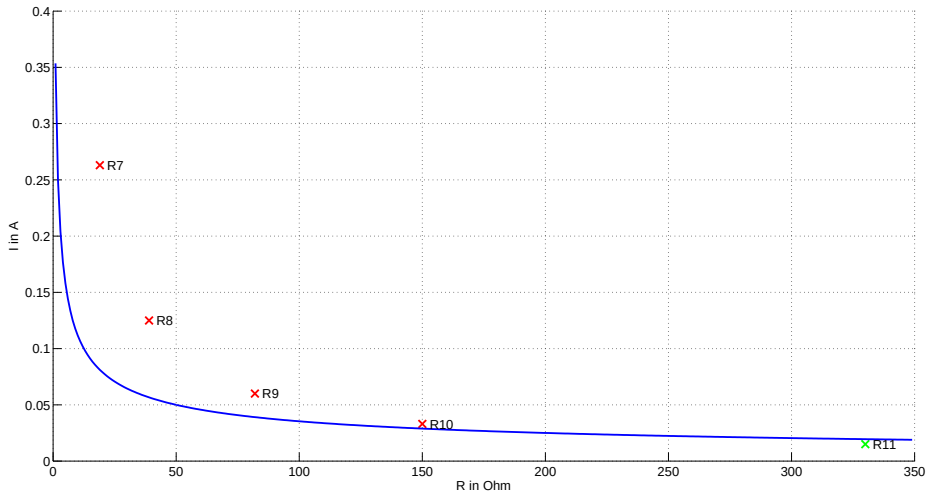


Abbildung 5.5: Widerstand/Strom-Diagramm für P_{tot} = 125 mW und den tatsächlich zu erwartenden maximalen Strombelastungen für die Widerstände R7 bis R11; es ist zu erkennen, dass bei den Widerständen R7 bis R10 mit der zu erwartenden Strombelastung die Verlustleistung von 125 mW überschritten wird

5.3 Layout

Das Layout für das LVDS-Testboard wird für eine vierlagige Platine erstellt. Dabei werden die beiden inneren Lagen für die Spannungsversorgung der Schaltung vorgesehen, wobei eine Lage für das Bezugspotential und eine Lage für die Betriebsspannung reserviert wird. Mittels dieser Anordnung lassen sich induktivitätsarme Zuleitungen zu den Spannungsversorgungsanschlüssen der integrierten Schaltungen realisieren. Abbildung 5.6 zeigt das Layout der beiden inneren Lagen. Links ist der Layer für die Betriebsspannung zu sehen und rechts der Layer für das Bezugspotential. Bei den rot gekennzeichneten Bereichen handelt es sich um Layoutfehler, da hier die Durchkontaktierungen zu nahe aneinander gesetzt wurden und somit ein langer Spalt in der Massefläche entsteht. Dadurch wird gegebenenfalls die effektive Fläche einer Leiterschleife vergrößert und es kann somit zu einer Verringerung der Störfestigkeit, sowie zu einer Erhöhung von Störemission kommen. Die Leiterbahnführung auf der Massefläche, welche dieselbe in der Mitte teilt, dürfte jedoch nach ausreichenden Überlegungen kein Problem darstellen, da keine signalführenden Leiterbahnen über den dadurch entstehenden Spalt geführt werden.

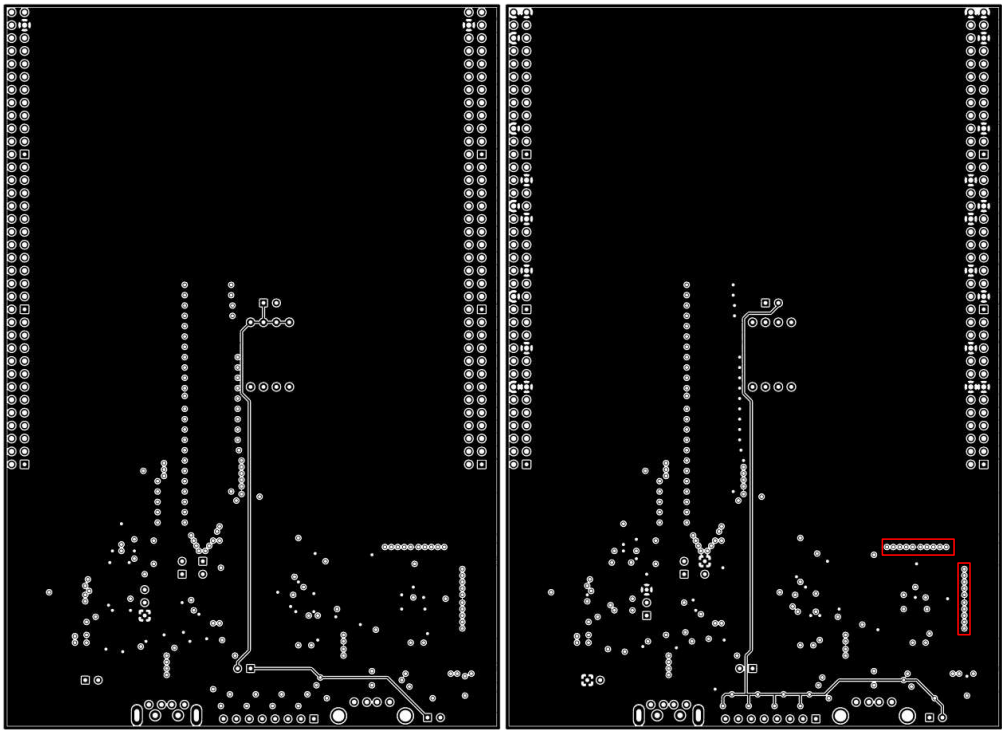


Abbildung 5.6: links: Layout der betriebsspannungsführenden Lage; rechts: Layout der Lage für das Bezugspotential

Alle Signalleitungen werden auf den beiden äußeren Lagen der Platine verlegt. Dadurch lässt sich die Eigeninduktivität der Signalleitungen – hervorgerufen durch eine geringe Anzahl an benötigten Vias – klein halten, wodurch die schnellen Signalanstiegs- und Signalabfallszeiten weniger beeinflusst werden.

Abbildung 5.7 zeigt links den Toplayer und rechts den Bottomlayer der Platine. Die im rot gekennzeichneten Bereich horizontal verlegten Leiterbahnen befinden sich über dem vorhin erwähnten Spalt in der Massefläche. Diese Signalleitungen werden jedoch von der Display-Adapterplatine nicht kontaktiert und somit entstehen auch keine Leiterschleifen.

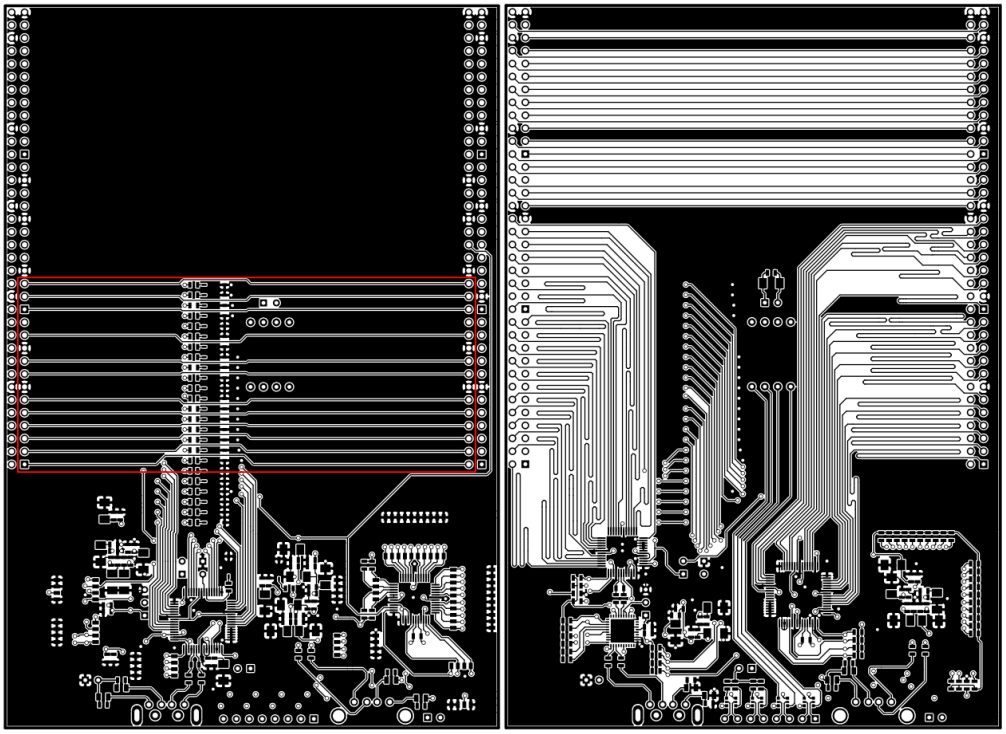


Abbildung 5.7: links: Layout des Toplayers, der zur Signalführung des Upchannels verwendet wird; rechts: Layout des Bottomlayers, der zur Signalführung des Downchannels verwendet wird

Die Pufferkapazitäten für die integrierten Schaltkreise werden so nahe wie möglich bei den vorgesehenen Spannungsversorgungsanschlüssen platziert, damit die Eigeninduktivität der Zuleitungen möglichst gering gehalten wird. Zusätzlich entstehen durch die örtliche Nähe der Pufferkapazitäten beim Chip keine Leiterschleifen, welche in weiterer Folge das Abstrahlen von Störungen – hervorgerufen durch die Umladeströme der Kondensatoren – ermöglichen.

Bei der Signalführung der Displaysignale vom Steckverbinder `X2` zum Serializer `IC4`, sowie vom Deserializer `IC5` zum Steckverbinder `X4` wird besonderes Augenmerk auf die gleiche Länge aller Leiterbahnen gelegt, um sicherzustellen dass alle Signale durch das gleiche Laufzeitdelay verzögert werden. Auch bei der Signalführung der differentiellen Signale wird darauf geachtet, dass beide Leitungen die gleiche Länge aufweisen.

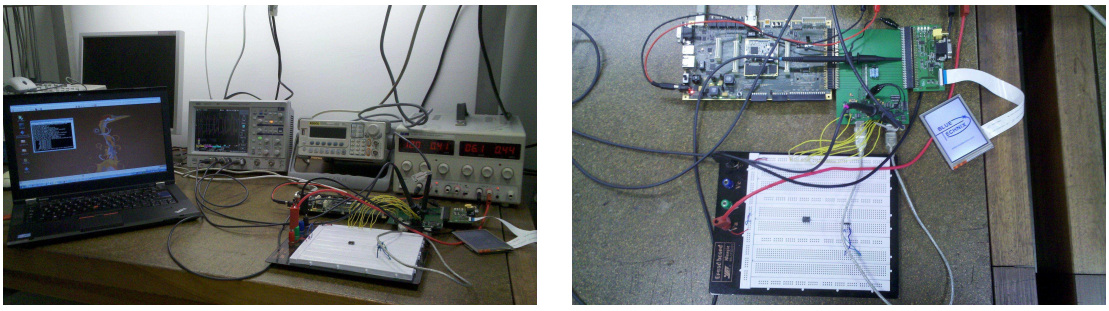
Die differentiellen Abschlusswiderstände der LVDS-Übertragung werden so nahe wie möglich bei den differentiellen Ausgängen der Serializer bzw. den differentiellen Eingängen der Deserializer platziert, um die resultierenden Stichleitungen vom Abschlusswiderstand zum Chip so klein wie möglich zu halten [17].

5.4 Messungen

Es werden einige Messungen mittels des LVDS-Testboards durchgeführt, um einerseits die Auswirkungen der SMD-Ferrite, sowie den Einfluss von verschieden Strombelastungen der Phantomspeisung auf die differentielle Signalübertragung untersuchen zu können. Dazu wird das i.MX31 Developementboard über USB und Ethernet mit einem Linux-Rechner verbunden. Auf dem Linux-Rechner befindet sich das Kernel-Image und das Dateisystem des Developementboards. Beim Booten des Developmentboards wird das Kernel-Image über Ethernet geladen und ein angeschlossenes Display zeigt bereits einen Bootscreen an. Das LVDS-Testboard wird mit den Steckverbindern `X2` und `X4` zwischen das Developementboard und die Display-Adapterplatine geschaltet und mit einem USB-Kabel wird schließlich die Verbindung zwischen den Sende- und Empfangseinheiten des Down- und Upchannels hergestellt. Die Abbildungen 5.8(a) und 5.8(b) zeigen den Messaufbau und in Tabelle 5.8 sind die verwendeten Messgeräte angeführt.

Gerät	Hersteller	Bezeichnung
Labornetzgerät	TTi	EL302T Triple Power Supply
Oszilloskop	LeCroy	waveJet 354 500 MHz Oscilloscope
Funktionsgenerator	RIGOL	DG1022 Function/Arbitrary Waveform Generator

Tabelle 5.8: Verwendete Geräte zum Betrieb und zur Vermessung des LVDS-Testboards



(a) (b)

Abbildung 5.8: (a) zeigt den Messaufbau mit allen verwendeten Geräten; (b) zeigt eine Detailansicht des Messaufbaus

Bei allen Messungen wird jeweils mit dem Oszilloskop der Spannungsverlauf des positiven (Kanal 1 – `RIN+`) und negativen (Kanal 2 – `RIN-`) Differenzeingangs des Deserializers vom Downchannel aufgezeichnet. Getriggert wird das Oszilloskop mit dem vom Deserializer rückgewonnenen Taktsignal (Kanal 3 – `RCLK`). Da der Deserializer seine Differenzeingänge mit einer Gleichspannung von 1.8 V beaufschlagt, werden die Messsignale an Kanal 1 und Kanal 2 mittels einer AC-Kopplung

aufgenommen, sodass der zur Verfügung stehende Messbereich optimal ausgenützt werden kann. Als Spannungsauflösung werden an Kanal 1 und 2, 100 mV/Div verwendet und als zeitliche Auflösung werden 50 ns/Div eingestellt. Die Messsignale werden schließlich im CSV-Dateiformat abgespeichert und mittels MATLAB ausgewertet.

5.4.1 Ohne SMD-Ferrite

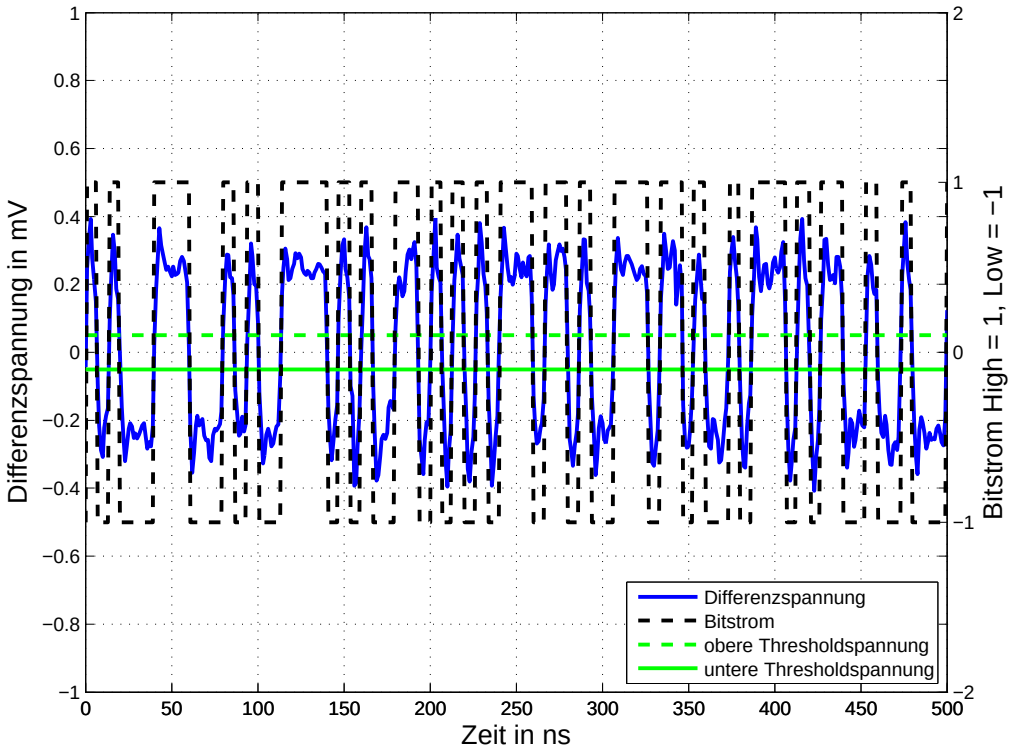


Abbildung 5.9: Resultierender Signalverlauf aus der Differenzbildung des Messsignals an Kanal 1 und Kanal 2 bei nicht vorhandenen SMD-Ferriten

Bei der ersten Betrachtung des Differenzsignals am Deserializer des Downchannel sind noch keine SMD-Ferrite verbaut wodurch mit der bestmöglichen Signalqualität am Deserializer zu rechnen ist.

5.4.2 Jeweils ein SMD-Ferrit

Es werden nun die vorgesehenen SMD-Ferrite eingebaut, um deren Auswirkung auf das Differenzsignal zu untersuchen. In weiterer Folge wird die Phantomspeisung durch das schaltbare Widerstandsnetzwerk immer stärker belastet, bis die Signalqualität unbrauchbar wird und somit auch die Datenübertragung über den Downchannel teilweise ausfällt.

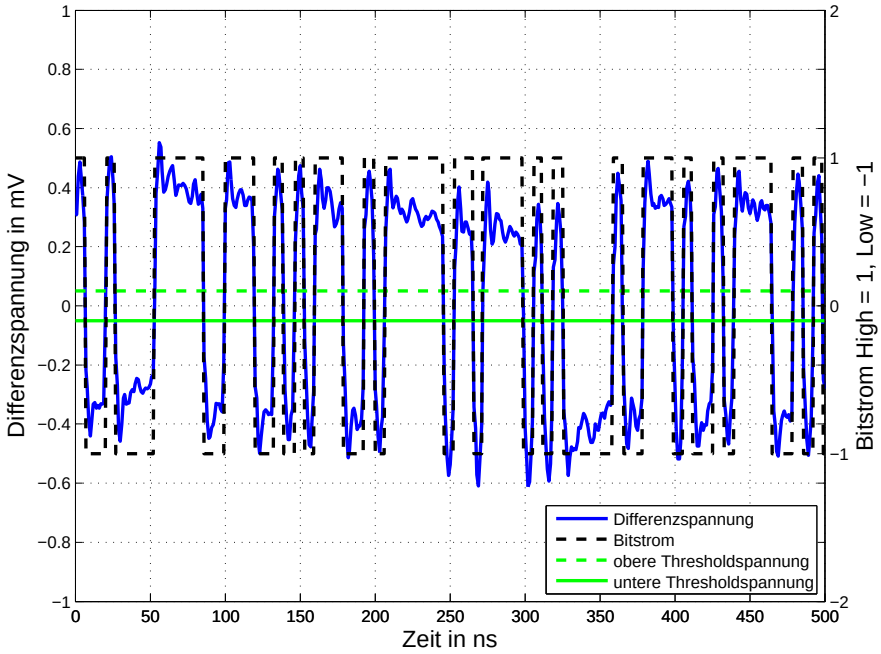


Abbildung 5.10: Resultierender Signalverlauf aus der Differenzbildung des Messsignals an Kanal 1 und Kanal 2 bei jeweils einem SMD-Ferrit zur Ein- und Auskopplung der Phantomspannung

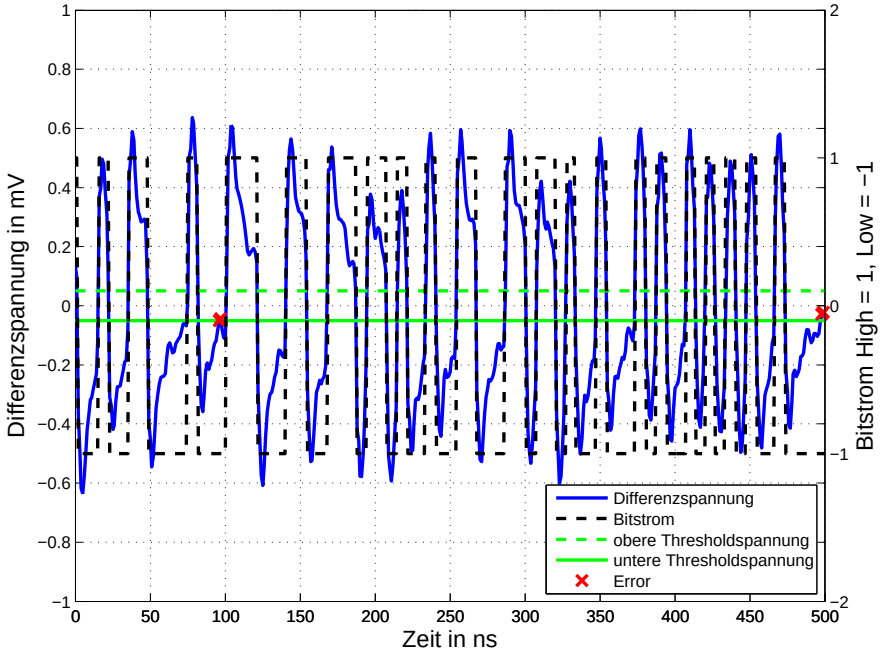


Abbildung 5.11: Resultierender Signalverlauf aus der Differenzbildung des Messsignals an Kanal 1 und Kanal 2 bei jeweils einem SMD-Ferrit zur Ein- und Auskopplung der Phantomspannung und einer Strombelastung von 500 mA; rot gekennzeichnete Stellen markieren die aufgetretenen Übertragungsfehler

Abbildung 5.10 zeigt den Verlauf der Differenzspannung sowie den dazugehörigen Bitstrom bei eingebauten SMD-Ferriten, jedoch unbelasteter Phantomspeisung. Es ist zu erkennen, dass die Differenzspannung nach einem Pegelwechsel nach einer Exponential-Funktion abklingt. Dies lässt sich aufgrund des induktiven Verhaltens der SMD-Ferrite erklären.

Mit zunehmender Strombelastung steigt auch die Anzahl von Übertragungsfehlern an, die sich als Pixelfehler am Display nachweisen lassen. Ab einer gewissen Strombelastung hat die Qualität des Differenzsignals so stark abgenommen, dass der Deserializer nicht mehr in der Lage ist, das eingebettete Taktsignal zu extrahieren und die Übertragung fällt vollständig aus.

In Abbildung 5.11 ist deutlich zu erkennen, dass aufgrund des zusätzlichen Stromflusses – hervorgerufen durch die Belastung der Phantomspeisung – die exponentiellen Abklingvorgänge beim Pegelwechsel des Differenzsignals wesentlich schneller erfolgen. Vor allem in Bereichen, in denen mehrere gleiche Bits hintereinander übertragen werden kommt es somit zu Übertragungsfehlern, da die Differenzspannung die notwendigen Schwellspannungen von $\pm 50\,\mathrm{mV}$ an den Eingängen des Deserializers unterschreitet. Diese Bereiche sind in Abbildung 5.11 rot gekennzeichnet.

Wird die Strombelastung der Phantomspeisung wieder reduziert, so ist ab einem Laststrom von 125 mA wieder eine zuverlässige Signalübertragung gegeben.

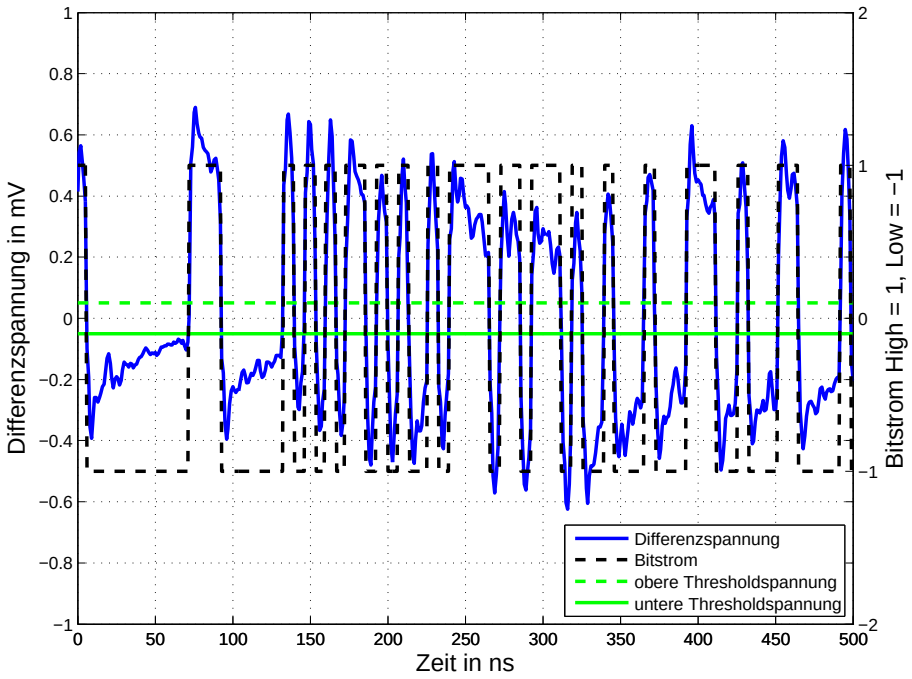


Abbildung 5.12: Resultierender Signalverlauf aus der Differenzbildung des Messsignals an Kanal 1 und Kanal 2 bei jeweils einem SMD-Ferrit zur Ein- und Auskopplung der Phantomspannung und einer Strombelastung von 125 mA

5.4.3 Jeweils zwei SMD-Ferrite

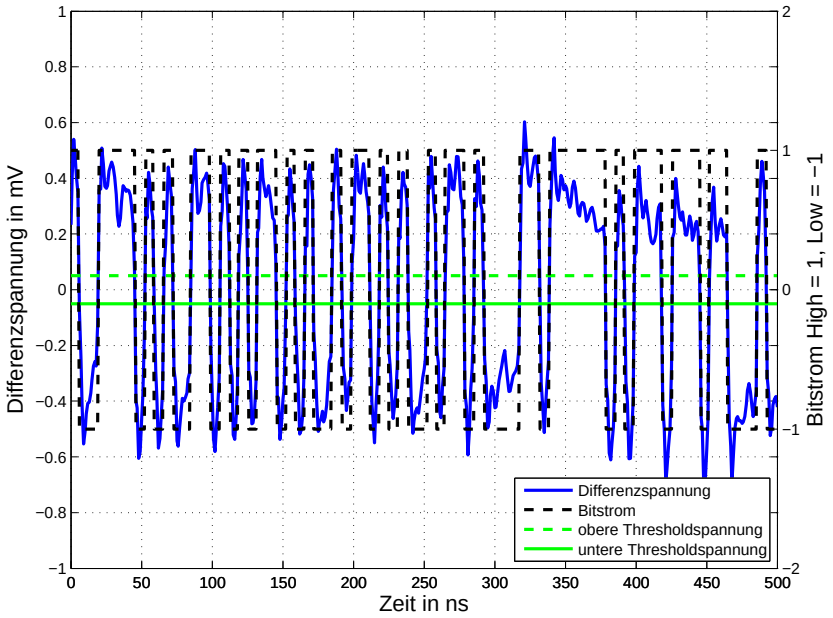


Abbildung 5.13: Resultierender Signalverlauf aus der Differenzbildung des Messsignals an Kanal 1 und Kanal 2 bei jeweils zwei SMD-Ferriten zur Ein- und Auskopplung der Phantomspannung und einer Strombelastung von 250 mA

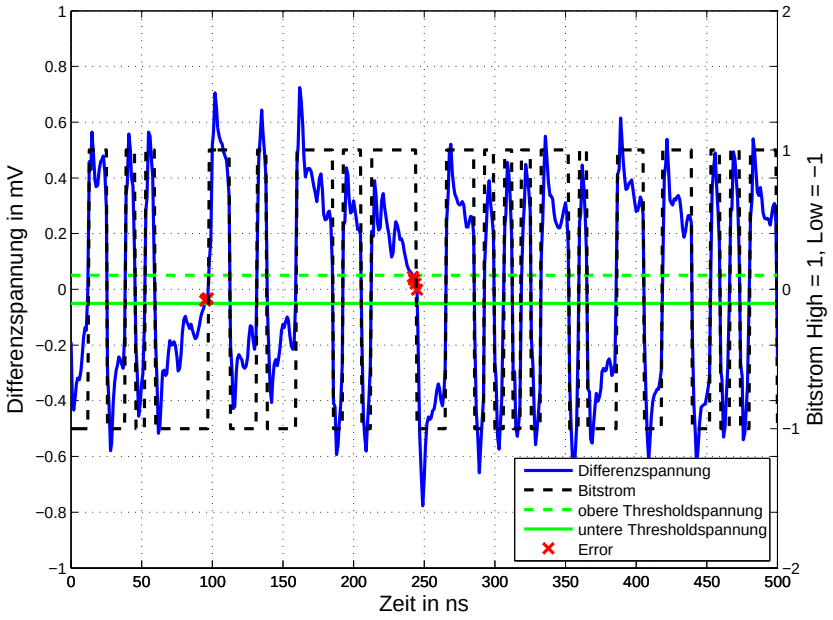


Abbildung 5.14: Resultierender Signalverlauf aus der Differenzbildung des Messsignals an Kanal 1 und Kanal 2 bei jeweils zwei SMD-Ferriten zur Ein- und Auskopplung der Phantomspannung und einer Strombelastung von 500 mA; rot gekennzeichnete Stellen markieren die aufgetretenen Übertragungsfehler

Um die exponentiellen Abklingvorgänge beim Pegelwechsel der Differenzspannung zu minimieren werden nun jeweils zwei SMD-Ferrite zum Einkoppeln der Phantomspannung verbaut, wodurch die Induktivität verdoppelt wird und somit die Umladevorgänge verlangsamt werden. Durch diese schaltungstechnische Maßnahme sind nun Lastströme bis zu 250 mA möglich, ohne dass es zu Übertragungsfehlern kommt. Bei einer Belastung der Phantomspeisung von 500 mA ist jedoch weiterhin keine zuverlässige Datenübertragung gegeben.

5.4.4 Jeweils drei SMD-Ferrite

Aus Abbildung 5.14 geht hervor, dass mit jeweils zwei SMD-Ferrite zum Einkopplen der Phantomspannung die geforderte Spezifikation von einem 500 mA Laststrom noch nicht erfüllt werden kann. Deshalb wird die Koppelinduktivität durch das Hinzufügen von jeweils einem weiteren SMD-Ferrit nochmals erhöht. Abbildung 5.15 zeigt den Verlauf der Differenzspannung bei einem Laststrom von 500 mA und jeweils drei verbauten SMD-Ferriten. Es ist deutlich zu erkennen, dass nun auch bei einer maximalen Strombelastung der Phantomspeisung eine zuverlässige Signalübertragung gegeben ist.

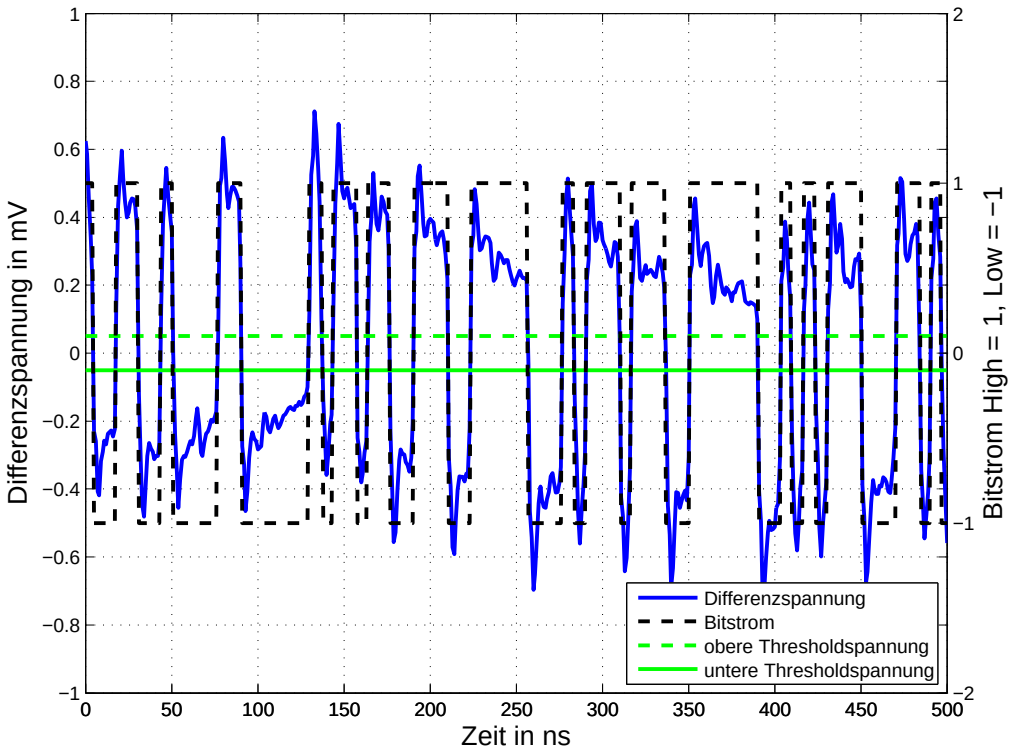


Abbildung 5.15: Resultierender Signalverlauf aus der Differenzbildung des Messsignals an Kanal 1 und Kanal 2 bei jeweils drei SMD-Ferriten zur Ein- und Auskopplung der Phantomspannung und einer Strombelastung von 500 mA

5.5 Zusammenfassung

Aufgrund des zusätzlichen Stromflusses auf den Datenleitungen – hervorgerufen durch die Belastung der Phantomspeisung – klingt die Differenzspannung mit zunehmender Stromstärke nach einem Pegelwechsel aufgrund des induktiven Verhaltens der SMD-Ferrite immer schneller ab. Um diesen Abklingvorgang zu verlangsamen, wird die Koppelinduktivität so weit erhöht, dass durch den Abklingvorgang die geforderten Schwellspannungen von $\pm 50\,\mathrm{mV}$ von der Differenzspannung nicht unterschritten werden.

Wird die Preempahsis des Serializers `IC3` oder des LVDS-Repeaters `IC1` aktiviert, so wird der Stromfluss auf den Datenleitungen nochmals erhöht, da die differentiellen Ausgänge der integrierten Schaltungen nun einen höheren Strom zum schnelleren Umalden des Übertragungskabels zur Verfügung stellen. Dadurch können auch bei langen und verlustbehafteten Kabeln steile Flanken beim Pegelwechsel der Leitung erreicht werden [17]. Aufgrund der Koppelinduktivitäten wird das Abklingen der Differenzspannung somit nochmals beschleunigt. Das bedeutet, dass die Datenübertragung schon bei niedrigeren Lastströmen ausfällt. Die erhöhte Induktivität von jeweils drei verwendeten SMD-Ferriten reicht jedoch aus, sodass auch bei aktivierter Preemphasis noch eine zuverlässige Signalübertragung gegeben ist.

6 Remote Peripheral Interface Bus

Aus den in Kapitel 5 erhaltenen Erkenntnissen lässt sich zeigen, dass die in Kapitel 4 ausgewählten Bauteile zur Umsetzung des RPI-Buskonzepts geeignet sind. Der RPI-Bus soll jedoch zur gleichzeitigen Übertragung von Video-, Audio- und Steuerdaten – sowohl über den Down-, als auch über den Upchannel – in der Lage sein. Zur Übertragung der Videoausgabedaten sollen wie beim LVDS-Testboard bereits angewendet, die Signale des Display Interfaces vom i.MX31 serialisiert werden. Somit sind bereits 21 der 24 parallelen Eingänge des Serializers vergeben. Die Übertragung der Audioausgabe erfolgt im Slave-Modus, wodurch nur ein Datensignal über den Downchannel übertragen werden muss. Die Steuerdaten (`SCK` und `WS`) der Audioübertragung werden hingegen über den Upchannel übertragen. Es stehen also noch zwei Eingänge des Serializers zur Verfügung, um Steuerdaten zu den Wearables übertragen zu können. Als Datenschnittstelle soll ein SPI des i.MX31 verwendet werden, wodurch drei weitere Signale ($\overline{\texttt{SS}}$, `SCLK` und `MOSI`) serialisiert werden müssen. Durch eine entsprechende Signalzusammenfassung wie unter Punkt 3.4 bereits angedeutet, lässt sich die Anzahl an benötigten Signalleitungen zur Übertragung von Steuerdaten über den Downchannel auf zwei reduzieren. Dazu werden die Signale $\overline{\texttt{SS}}$ und `MOSI` durch ein CPLD auf ein Signal zusammengefasst.

Über den Upchannel werden die Videoeingabedaten so übertragen, dass das CMOS Sensor Interface des i.MX31 Developementboards kontaktiert werden kann. Zusätzlich werden sowohl die für die Audioausgabe als auch für die Audioeingabe notwendigen Steuerdaten über den Upchannel übertragen. Ein weiterer Eingang des Serializers des Upchannels wird für die Gegenrichtung der Steuerdatenübertragung verwendet. Somit werden bei diesem Serializer nur 16 der 24 parallelen Eingänge benötigt.

Es wird ein weiterer Prototyp entworfen, wobei nun die Übertragung aller geforderten Daten möglich sein soll. Dieser Prototyp – im folgenden RPI-Testboard genannt – ermöglicht das Senden der Video- und Audioausgabedaten, sowie der Steuerdaten. Eine zweite Testplatine – im weiteren als Wearable-Testboard bezeichnet – wurde im Rahmen einer Seminararbeit [23] entworfen und dient schließlich zur Anbindung der Peripherie (Display, Headset, etc.).

6.1 Schaltungsentwurf

In weiterer Folge werden Ausschnitte der erstellten Stromlaufpläne beschrieben. Das Original der entsprechenden Stromlaufpläne befindet sich auf der beliegenden DVD, sowie im Anhang A dieser Arbeit.

6.1.1 Spannungsversorgung

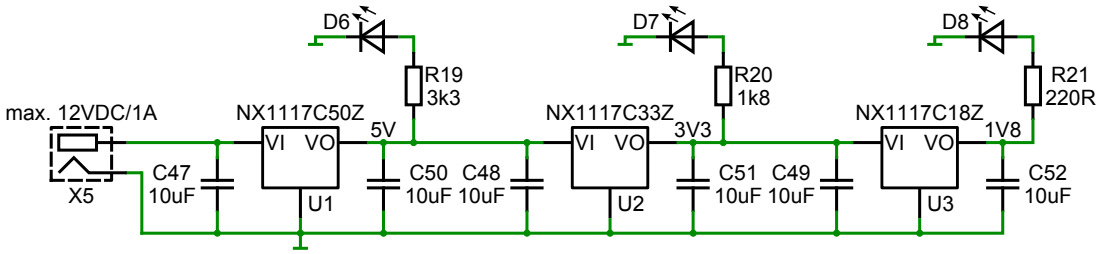


Abbildung 6.1: Schaltungskomponenten für die Spannungsversorgung des RPI-Testboards

Abbildung 6.1 zeigt die für die Spannungsversorgung des RPI-Testboards notwendigen Schaltungskomponenten. Als Stromeingang dient der Steckverbinder `X5`, der eine Nennspannung von 12 V und einen Nennstrom von 1 A aufweist. Die Eingangsspannung von `X5` wird mit dem Festspannungsregler `U1` auf eine 5 V Gleichspannung geregelt. Diese Gleichspannung entspricht der – über die Busleitungen mitübertragenen – Phantomspannung. Zwei weitere Spannungsregler `U2` und `U3` erzeugen die für die integrierten Schaltungen notwendigen Betriebsspannungen von 3.3 V und 1.8 V. Alle drei Spannungsregler werden ein- und ausgangsseitig mit 10 μF Kondensatoren stabilisiert. Die Leuchtdioden `D6` bis `D8` dienen jeweils als Power-LED für eine der drei Spannungsdomänen.

6.1.2 CPLD

Es wird ein CPLD benötigt um das Zusammenfassen der Signale $\overline{\texttt{SS}}$ und `MOSI` von der SPI-Schnittstelle zu ermöglichen.

Das CPLD muss des Weiteren ein Multiplexing der über die vier Upchannels übertragenen Daten vornehmen. Dies ist erforderlich, da der i.MX31 immer nur von einer Quelle, Video-, Audio- oder Steuerdaten verarbeiten kann. Deshalb werden alle Signale der vier Upchannels zum CPLD geführt, das schließlich die Signale zur Videoeingabe, Audioein- bzw. ausgabe oder Dateneingabe eines Kanals für den i.MX31 zur Verfügung stellt.

Die aktiven Datenquellen werden beim RPI-Testboard über einen acht-poligen DIP-Schalter selektiert, wobei jeweils zwei Schalter zum Auswählen von einer der vier Quellen notwendig sind.

Das CPLD muss somit eine große Anzahl an I/O-Pins aufweisen, wobei in Tabelle 6.1 alle Signalleitungen angeführt sind, die mittels des CPLDs weiterverarbeitet werden müssen.

Da bereits Erfahrungen mit der CoolRunner-II CPLD-Generation der Firma Xilinx gesammelt werden konnten, fällt die Entscheidung auf das CPLD vom Typ XC2C256, welches in einem TQ144-Gehäuse 118 User-I/Os besitzt und somit den geforderten Anforderungen gerecht wird. Die I/Os des CPLDs können so konfiguriert werden, dass sie mit dem LVCMOS33 Standard – und somit mit den LVCMOS Ein- und ausgängen der anderen integrierten Schaltungen – kompatibel sind (vgl. [24]).

Xilinx bietet eine große Anzahl an Anwendungsbeschreibungen an, sodass Fehler beim Schaltungsentwurf, sowie beim Digitaldesign vermieden werden können. Aus [25] gehen die zwölf wichtigsten Punkte hervor, die beim Schaltungsentwurf für das CPLD beachtet werden müssen.

Typ	**Anzal**	**Funktion**
Input	1	`lcdc_pclk` zum Takten der sequentiellen Logik des CPLDs
Input	3	$\overline{\texttt{SS}}$, `SCLK` und `MOSI` zur Datenausgabe über ein SPI des i.MX31
Output	2	`rpi_sclk` und `rpi_mosi_ss` zur Übertragung der Datenausgabe über den RPI-Bus
Input	4	`rpi_0_miso` ... `rpi_3_miso` zum Empfangen der Dateieingabe über die vier Upchannels
Output	1	`MISO` zur Dateneingabe am SPI des i.MX31
Output	11	`CSI_PCLK`, `CSI_VSYNC`, `CS_HSYNC` sowie `CSI_D0` bis `CSI_D7` zur Videoeingabe beim i.MX31
Input	44	jeweils `rpi_i_pclk`, `rpi_i_vsync`, `rpi_i_hsync` sowie `rpi_i_d0` bis `rpi_i_d7` pro Upchannel zum Empfangen von Videoeingabedaten
Output	3	`SCKi`, `WSi` und `SDi` zur Audioeingabe beim i.MX31
Input	12	jeweils `rpi_i_scki`, `rpi_i_wsi` und `rpi_i_sdi` pro Upchannel zum Empfangen von Audioeingabedaten
Input	1	`SDo` zur Audioausgabe durch den i.MX31
Output	2	`SCKo` und `WSo` zur Steuerung der Audioausgabe des i.MX31
Output	1	`rpi_sdo` zur Übertragung der Audioausgabedaten über den Downchannel

Fortsetzung auf nächster Seite

Tabelle 6.1 – *fortgesetzt von vorheriger Seite*

Typ	**Anzal**	**Funktion**
Input	8	jeweils `rpi_i_scko` und `rpi_i_wso` pro Upchannel zur Steuerung der Audioausgabe des i.MX31
Input	2	`video_in_sel_0` und `video_in_sel_1` zur Auswahl des Kanals, dessen Videoeingabedaten zum i.MX31 geführt werden sollen
Input	2	`audio_in_sel_0` und `video_in_sel_1` zur Auswahl des Kanals, dessen Audioeingabedaten zum i.MX31 geführt werden sollen
Input	2	`data_in_sel_0` und `data_in_sel_1` zur Auswahl des Kanals, dessen Eingabedaten zum i.MX31 geführt werden sollen
Input	2	`audio_out_sel_0` und `audio_out_sel_1` zur Auswahl des Kanals, der die Audioausgabe des i.MX31 steuern soll
Output	4	`status0` bis `status3`, um Debugginformationen des Digitaldesigns zur Verfügung stellen zu können
Summe	**102**	**benötigte User-I/Os**

Tabelle 6.1: Zusammenfassung der Ein- und Ausgänge des CPLDs

Beim richtigen Schaltungsentwurf für das CPLD ist es essentiell, alle Spannungsversorgungsanschlüsse mit separaten Pufferkapazitäten zu versehen, um zusätzliche Energie bei den schnellen Schaltvorgängen des CPLDs zur Verfügung stellen zu können (vgl. [25], S.2).

Des weiteren müssen alle Eingänge mit einem fixen Zustand (High oder Low) beschaltet werden, wobei jedoch die Möglichkeit besteht, die Input-Buffer Zellen über einen programmierbaren Schalter auf GND zu legen. So wird einerseits die Stromaufnahme des CPLDs und andererseits das Rauschen im Logikkern des Bauteils reduziert (vgl. [25], S.2)

Außerdem sollten Pulldown-Widerstände an den Eingängen des CPLDs vermieden werden, da beim Einschalten des Schaltkreises die I/Os im Tri-State sind und der externe Pulldown-Widerstand dem Verhalten der Eingangszelle entgegenwirken kann (vgl [25], S.3).

Werden Leuchtdioden mittels des CPLDs getrieben, so sind diese mit der Anode an die Betriebsspannung anzuschließen. Die LED wird also mit einem Low-Pegel am CPLD-Ausgang eingeschaltet. Hierbei ist zusätzlich zu beachten, dass die Strombelastung eines Ausgangs des CPLDs den Wert von 8 mA nicht überschreitet (vgl. [26], S.2).

Aufgrund dieser Anwendungsbeschreibungen werden die Eingänge – zur Auswahl der Datenquellen – mit Pullup-Widerständen beschaltet und können mit dem Schalter `S1` auf Low gezogen werden. Die vier reservierten Ausgänge `status0` bis `status3`

werden über Vorwiderstände mit LEDs beschaltet, die wie in [26] beschrieben mit der Anode an die Betriebspannung angeschlossen werden. Alle Spannungsversorgungsanschlüsse des CPLDs werden mit einer Parallelschaltung von 10 nF und 100 nF Kondensatoren gepuffert, wobei die Werte für die Pufferkapazitäten aus [27] entnommen wurden.

Das CPLD XC2C256 besitzt zusätzlich die Möglichkeit von In-System Programming (ISP) über eine JTAG-Schnittstelle. Dazu werden die vier Signale – `TCK`, `TMS`, `TDI` und `TDO` – des JTAG TAPs des CPLDs von außen über die Pinleiste `P1` zugänglich gemacht. Wichtig ist hierbei noch, dass dem Programmiergerät (Download Cable) eine Referenzspannung des CPLDs zur Verfügung gestellt wird (vgl. [27], S.1).

6.1.3 Downchannel

Der Downchannel beim RPI-Testboard wird ähnlich wie bereits beim LVDS-Testboard entworfen. Die Signale für die Videoausgabe werden über den 40-poligen FFC-Steckverbinder `X4` – der über ein Flachbandkabel mit der Display-Adapterplatine vom i.MX31 Developementboard verbunden wird – eingelesen und dem Serializer `IC2` zugeführt. Die Signale für die Übertragung der Steuerdaten werden an die Pinleiste `X2` angeschlossen und durch das CPLD `IC1` vorverarbeitet, bevor sie dem Serializer zugeführt werden. Das Signal zur Audioausgabe wird ebenfalls durch das CPLD geleitet und wird über die Pinleiste `X1` an das RPI-Testboard angeschlossen.

Die differentiellen Ausgänge des Serializers `IC2` werden wieder mit einem LVDS-Repeater verbunden, der schließlich die Schnittstellen der Downchannels, der vier möglichen Peripheriegeräte, zur Verfügung stellt. Die differentiellen Ausgänge des LVDS-Repeaters sind bereits am Chip mit 100 Ω differentiell abgeschlossen und werden wieder kapazitiv mit 100 nF vom weiteren Übertragungsweg entkoppelt. Über jeweils drei SMD-Ferrite wird die Phantomspannung auf die Busleitungen der Downchannels eingekoppelt, die über die Micro USB Typ B Buchsen `X10` bis `X13` von der Peripherie kontaktiert werden können.

Die Konfiguration der integrierten Schaltkreise `IC2` und `IC3` erfolgt beim RPI-Testboard mittels Einbau von 0 Ω Widerständen, wobei die entsprechenden Eingänge der Chips entweder mit High oder Low beschaltet werden können. Im Gegensatz zum LVDS-Testboard lässt sich beim Serializer nun nur noch die Art der Taktflanke, bei der die parallelen Eingänge eingelesen werden, einstellen. Beim LVDS-Repeater kann die Preemphasis aller Ausgangstreiber, sowie die Input-Equalization der Eingangsschnittstelle konfiguriert werden.

Die Spannungsversorgung des Serializers `IC2` wird identisch wie beim LVDS-Testboard durchgeführt. Beim LVDS-Repeater werden diesmal jedoch alle Spannungsversorgungsanschlüsse mit der gleichen Parallelschaltung von Pufferkapazitäten, wie sie beim Serializer verwendet wird, versehen.

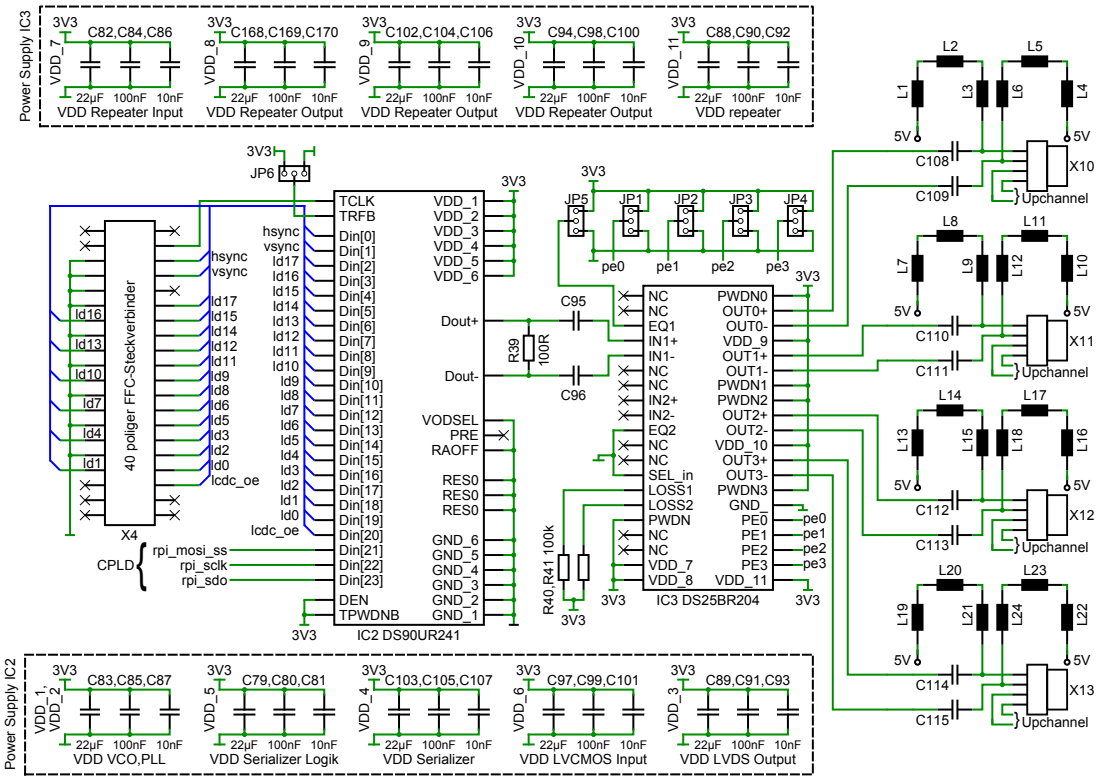


Abbildung 6.2: Schaltungskomponenten für die Downchannels des RPI-Testboards

6.1.4 Upchannel

Die vier Upchannels des RPI-Testboards sind nahezu identisch aufgebaut. Sie unterscheiden sich jedoch in der Anbindung an das CPLD `IC1`, welches das Multiplexing der Datenquellen vornimmt. Da für die Upchannels die gleichen Serializer und Deserializer verwendet werden, aber nur 16 von 24 möglichen Signalen serialisiert werden müssen, werden die Signalausgänge bei den Deserializer-ICs so beschaltet, dass sich ein möglichst einfaches Routing auf der Leiterplatte ergibt. Dies wird erreicht, indem einige Signale – die über den Upchannel übertragen werden – an zwei verschiedenen Eingängen des Serializers angelegt werden und somit redundant übertragen werden. In Tabelle 6.2 sind die Pinbelegungen der Deserializer – für den jeweiligen Upchaennel – mit den dazugehörigen Signalleitungen angeführt.

Abgesehen von den parallelen Ausgängen der Deserializer sind die vier Upchannels aber völlig identisch aufgebaut. Alle Spannungsversorgungsanschlüsse der Deserializer werden wieder separat mit einer Parallelschaltung von 10 nF, 100 nF und 22 μF gepuffert.

Der Status der PLL von den einzelnen Deserializer-ICs wird – wie beim LVDS-Testboard bereits ausgeführt – wieder mit Hilfe von jeweils einer Leuchtdiode visualisiert.

Signal	**Serializer**		**Deserializer**			
	Eingang	Eingang alt.	Upch. 0	Upch.l 1	Upch. 2	Upch. 3
MISO	Din[0]	Din[23]	Rout[0]	Rout[0]	Rout[0]	Rout[23]
SDi	Din[1]	Din[22]	Rout[1]	Rout[1]	Rout[1]	Rout[22]
WSi	Din[2]	Din[21]	Rout[2]	Rout[2]	Rout[2]	Rout[21]
SCKi	Din[3]	Din[20]	Rout[3]	Rout[3]	Rout[3]	Rout[20]
WSo	Din[4]	Din[19]	Rout[4]	Rout[4]	Rout[4]	Rout[19]
SCKo	Din[5]	Din[18]	Rout[5]	Rout[5]	Rout[5]	Rout[18]
CSI_HSYNC	Din[6]	Din[17]	Rout[17]	Rout[6]	Rout[17]	Rout[6]
CSI_VSYNC	Din[7]	Din[16]	Rout[16]	Rout[7]	Rout[16]	Rout[7]
D2	Din[8]	—	Rout[8]	Rout[8]	Rout[8]	Rout[8]
D3	Din[9]	—	Rout[9]	Rout[9]	Rout[9]	Rout[9]
D4	Din[10]	—	Rout[10]	Rout[10]	Rout[10]	Rout[10]
D5	Din[11]	—	Rout[11]	Rout[11]	Rout[11]	Rout[11]
D6	Din[12]	—	Rout[12]	Rout[12]	Rout[12]	Rout[12]
D7	Din[13]	—	Rout[13]	Rout[13]	Rout[13]	Rout[13]
D8	Din[14]	—	Rout[14]	Rout[14]	Rout[14]	Rout[14]
D9	Din[15]	—	Rout[15]	Rout[15]	Rout[15]	Rout[15]

Tabelle 6.2: unterschiedliche Pinbelegungen der Serializer und Deserializer der vier Upchannels des RPI-Busses

Die Konfiguration der Deserializer lässt sich ebenfalls wie beim Serializer oder LVDS-Repeater vom Downchannel über 0 Ω Widerstände vornehmen, welche die entsprechenden Steuerpins der Deserializer entweder auf einen High- oder Low-Pegel legen.

Der Built-In-Selftest der Deserializer wird auch beim RPI-Testboard permanent deaktiviert, indem die entsprechenden Pins der Deserializer auf Massepontential gelegt werden.

Die differentiellen Eingänge der Deserializer sind über 100 nF Kondensatoren von den Micro USB Typ B Buchsen X10 bis X13 kapazitiv entkoppelt und mittels 100 Ω Widerständen differentiell abgeschlossen. Zwischen den Micro USB Typ B Buchsen und den nachfolgenden Koppelkapazitäten wird jeweils wieder über drei SMD-Ferrite die Phantomspannung ausgekoppelt. Abbildung 6.3 zeigt den Schaltplanausschnitt für den Upchannel mit der Nummer 0.

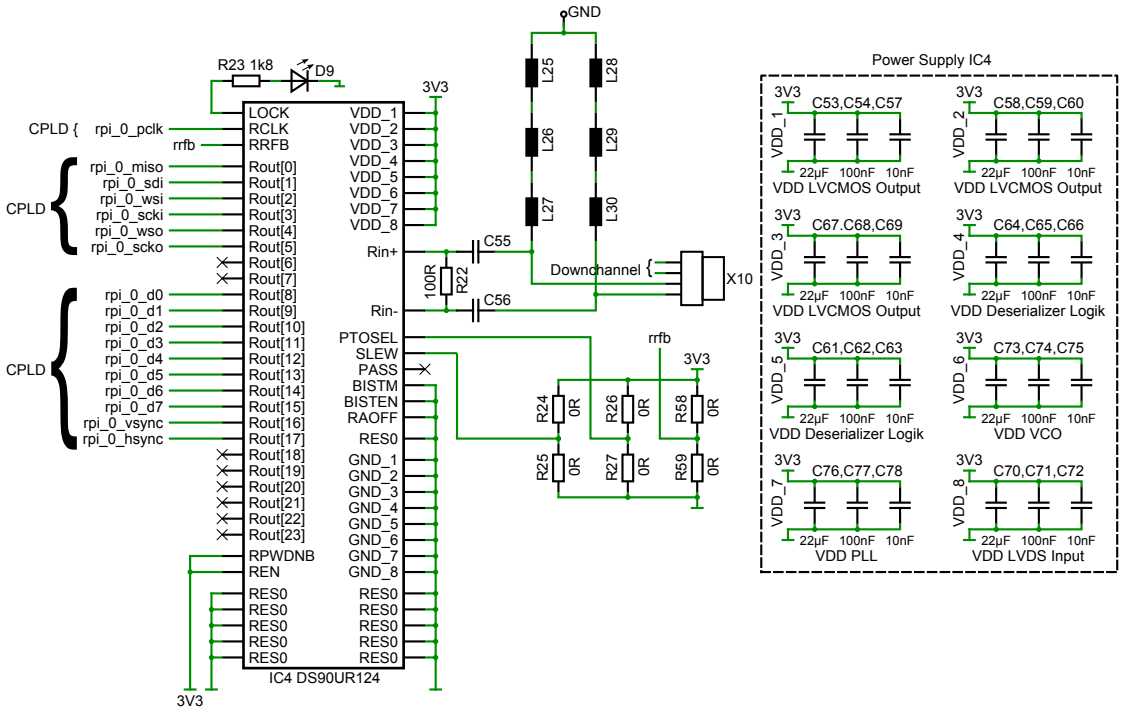


Abbildung 6.3: Schaltungskomponenten für den Upchannel 0 des RPI-Testboards

6.2 Dimensionierungen

Auch beim RPI-Testboard müssen einige Dimensionierungen, wie beispielsweise der maximale Laststrom der Gesamtschaltung und die LED-Vorwiderstände, durchgeführt werden.

6.2.1 Maximaler Betriebsstrom des RPI-Testboards

Bei den Spannungsreglern ist zu beachten, dass die maximale Verlustleistung P_{tot} nicht überschritten wird. Die maximale Verlustleistung lässt sich mittels der im Datenblatt (vgl. [28], S.3) angegebenen Formel 6.1 bestimmen.

$$P_{tot} = \frac{T_j - T_{amb}}{R_{th(j-a)}} \tag{6.1}$$

Bei T_j handelt es sich um die maximale Sperrschichttemperatur des Spannungsreglers und ist mit 150 °C gegeben. T_{amb} entspricht der Umgebungstemperatur und wird mit 40 °C angenommen. Der Temperaturwiderstand ($R_{th(j-a)}$) zwischen Sperrschicht und Bauteilumgebung wird im Datenblatt mit 150 K/W angegeben.

Somit lässt sich unter der Annahme, dass das RPI-Testboard mittels eines 6 V Akkus betrieben wird, der maximal zulässige Betriebsstrom ermitteln, sodass die maximale Verlustleistung des 5 V Spannungsreglers nicht überschritten wird.

$$\begin{aligned} P_{tot} &= \frac{150\,°\mathrm{C} - 40\,°\mathrm{C}}{150\,\mathrm{K/W}} = 0.733\,\mathrm{W} \\ P_{tot} &= (V_{in,max} - V_{out}) \cdot I_{max} \\ \Rightarrow I_{max} &= \frac{P_{tot}}{V_{in} - V_{out}} = \frac{0.733\,\mathrm{W}}{6\,\mathrm{V} - 5\,\mathrm{V}} = 733\,\mathrm{mA} \end{aligned} \tag{6.2}$$

Der gesamte Betriebsstrom des RPI-Testboards und auch der von den Peripheriegeräten, wird vom 5 V-Spannungsregler zur Verfügung gestellt, wodurch aufgrund des maximalen Laststroms von 733 mA die geforderten Spezifikationen (500 mA Laststrom bei allen Peripheriegeräten) nicht erfüllt werden können. Da es sich jedoch beim RPI-Testboard lediglich um einen Prototypen handelt, wird auf den Entwurf eines Spannungsversorgungskonzept verzichtet. Ein möglicher Ansatz wäre jedoch die Verwendung eines Step-Down Konverters, der für den notwendigen Laststrom ausgelegt ist.

Der 3.3 V Spannungsregler dient zur Spannungsversorgung der integrierten Schaltkreise auf dem RPI-Testboard und muss somit die Betriebsströme für das CPLD `IC1`, den Serializer `IC2`, den LVDS-Repeater `IC3` und die vier Deserializer `IC4` bis `IC7` zur Verfügung stellen. Die Summe der Betriebsströme dieser ICs kann nach Betrachtung der entsprechenden Datenblätter mit ca. 410 mA abgeschätzt werden und somit kann gezeigt werden, dass der 3.3 V Spannungsregler ohne zusätzlichen Kühlkörper betrieben werden kann.

$$\begin{aligned} P_{tot} &= (V_{in} - V_{out}) \cdot I \\ 0.733\,\mathrm{W} &\geq (5\,\mathrm{V} - 3.3\,\mathrm{V}) \cdot 0.410\,\mathrm{A} \approx 0.7\,\mathrm{W} \end{aligned}$$

Der 1.8 V Spannungsregler stellt lediglich die Betriebsspannung für die interne Logik des CPLDs zur Verfügung, wobei dessen Stromaufnahme stark von der Taktfrequenz abhängig ist. Diese kann mit maximal 30 MHz abgeschätzt werden, wodurch sich beim CPLD eine typische Stromaufnahme von 11.68 mA (vgl. [24], S.2) ergibt. Nach Formel 6.2 ergibt sich beim 1.8 V Spannungsregler eine Verlustleistung von 17.5 mW, wodurch auch dieser ohne zusätzlichen Kühlkörper betrieben werden kann.

Würden sich die Spannungsregler – hervorgerufen durch den Anstieg der Verlustleistung über den maximal zulässigen Wert P_{tot} – zu stark erwärmen, so schalten sich diese automatisch ab und es kommt zu keiner Zerstörung des Bauteils (vgl. [28], S.1).

Werden nun die abgeschätzten Lastströme der 3.3 V und 1.8 V Spannungsdomäne addiert, so kann ein Wert für den maximal zulässigen Laststrom ($I_{p,max}$) der

Phantomspeisung berechnet werden:

$$\begin{aligned} I_{p,max} &= I_{max} - I_{3V3} - I_{1V8} \\ I_{p,max} &= 733\,\text{mA} - 410\,\text{mA} - 11.68\,\text{mA} \\ I_{p,max} &\approx 300\,\text{mA} \end{aligned}$$

Da jedoch ein Peripheriegerät, das mit einem Display ausgestattet ist, eine Stromaufnahme $\geq 300\,\text{mA}$ aufweisen wird, wird der 5 V Spannungsregler mit einem zusätzlichen Kühlkörper – mit einem thermischen Widerstand von 55 K/W – versehen. Der maximal zulässige Laststrom durch den 5 V Spannungsregler erhöht sich somit auf:

$$\begin{aligned} P_{tot} &= \frac{T_j - T_{amb}}{R_{th(j-sp)} + R_{th,KK}} \\ P_{tot} &= \frac{150\,^\circ\text{C} - 40\,^\circ\text{C}}{20\,\text{K/W} + 55\,\text{K/W}} \\ P_{tot} &= 1.466\,W \end{aligned}$$

$$I_{max} = \frac{1\,\text{V}}{1.466\,\text{W}} \approx 1.4\,\text{A}$$

Durch den Kühlkörper erhöht sich der maximal zulässige Laststrom auf 1.4 A, wobei weiterhin darauf zu achten ist, dass der Betriebsstrom des RPI-Testboards den Wert von 1 A nicht überschreitet, da der Spannungsregler nur bis zu einem Laststrom von 1 A spezifiziert ist.

6.2.2 LED-Vorwiderstände

Beim RPI-Testboard werden Leuchtdioden mit einem Vorwärtsstrom von 0.5 mA und einer Vorwärtsspannung von 1.6 V (max. 1.9 V) verwendet. Somit können die Vorwiderstände für die Leuchtdioden wieder wie folgt berechnet werden.

Power-LED 5 V Spannungsdomäne

$$R_V = \frac{5\,\text{V} - V_{F,typ}}{I_F} = \frac{5\,\text{V} - 1.6\,\text{V}}{0.5\,\text{mA}} = 6.8\,\text{k}\Omega \Rightarrow 6.8\,\text{k}\Omega \text{ gewählt}$$

Power-LED 3.3 V Spannungsdomäne

$$R_V = \frac{3.3\,\text{V} - V_{F,max}}{I_F} = \frac{3.3\,\text{V} - 1.6\,\text{V}}{0.5\,\text{mA}} = 3.4\,\text{k}\Omega \Rightarrow 3.3\,\text{k}\Omega \text{ gewählt}$$

Power-LED 1.8 V Spannungsdomäne

$$R_V = \frac{1.8\,\mathrm{V} - V_{F,max}}{I_F} = \frac{5\,\mathrm{V} - 1.6\,\mathrm{V}}{0.5\,\mathrm{mA}} = 400\Omega \Rightarrow 390\,\Omega \text{ gewählt}$$

6.2.3 Impedanzkontrollierte Leiterbahn

Die Leiterbahnführung der differentiellen Signale von den Down- und Upchannels wird beim RPI-Testboard – wie im Abschnitt 4.3.1 bereits erwähnt – impedanzkontrolliert durchgeführt. Dazu werden die Leiterbahnen eines differentiellen Leiterpaares mit einem minimalen Abstand zueinander verlegt und entsprechend der Formeln 4.3 und 4.4 wird die notwendige Leiterbahnbreite berechnet, sodass sich ein differentieller Widerstand von 100 Ω zwischen den beiden Leiterbahnen ergibt – siehe Tabelle 4.5 und Abbildung 4.5.

6.3 Layout

Das Layout für das RPI-Testboard wird wieder für eine vierlagige Platine erstellt, sodass die inneren Lagen zur Spannungsversorgung der Schaltung verwendet werden können und die äußeren Lagen zur Signalführung zur Verfügung stehen.

Abbildung 6.4 zeigt das Layout der beiden inneren Lagen der Platine. Links ist der Layer für die Betriebsspannungen zu sehen, wobei die Kupferfläche der 3.3 V Spannungsdomäne entspricht. Die Leiterbahn, die den Spannungseingang des RPI-Testboards mit dem 5 V Spannungsregler verbindet, kann mit einem Strom von bis zu 1 A belastet werden und muss somit dementsprechend dicker gewählt werden. PCB-POOL gibt in den Spezifikationen [18], bei einer Leiterbahnbreite von 0.5 mm und einer zulässigen Erwärmung um 10 °C, eine Strombelastbarkeit von 1 A an. Da sich diese Leiterbahn aber auf einem Zwischenlayer befindet und dadurch die Wärme – im Vergleich zu den äußeren Lagen – nicht so gut abgeleitet werden kann, wird eine Leiterbahnbreite von 0.8 mm gewählt. Auch die Leiterbahnen für die Phantomspeisung werden mit einer Breite von 0.8 mm gewählt.

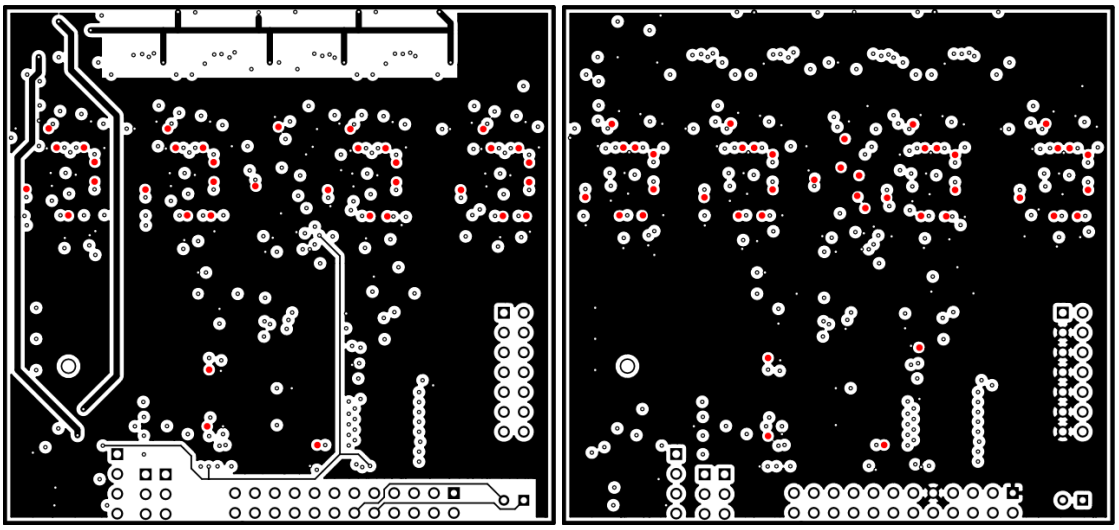


Abbildung 6.4: links: Layer zur Verteilung der Betriebsspannungen 5 V, 3.3 V und 1.8 V; rechts: Layer des gemeinsamen Bezugspotentials

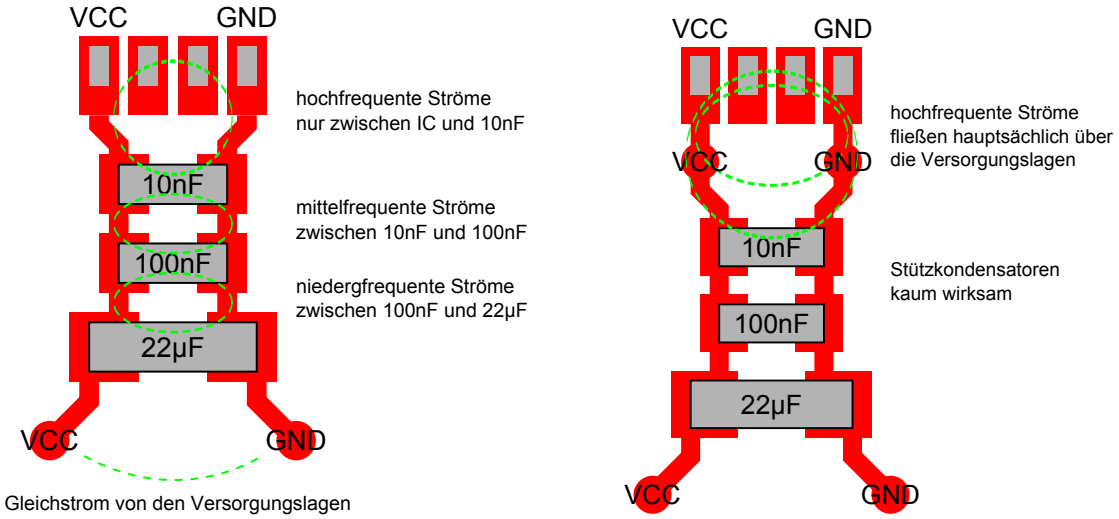


Abbildung 6.5: (a) korrektes Layout – die Kapazitäten sind induktivitätsarm mit dem IC verbunden und bilden nur kleine Leiterschleifen;
(b) schlechtes Layout – durch die Vias zwischen Kondensator und IC erhöht sich die Induktivität der Zuleitungen und es entsteht eine große Leiterschleife über die Layer von VCC und GND (vgl. [29], o.S.)

Die in Abbildung 6.4 rot eingefärbten Durchkontaktierungen weisen jeweils dasselbe Potential wie die entsprechende Kupferlage auf, sind jedoch von dieser isoliert. Dies hat den Grund, da es sich hierbei um die Zuleitungen der Stützkondensatoren zu den Spannungsversorgungsanschlüssen der integrierten Schaltungen handelt und somit die zusätzlich benötigte Energie bei den Schaltvorgängen der Schaltkreise

direkt aus den Kondensatoren entnommen wird. Durch diese Maßnahme ergibt sich für die Pufferkapazitäten ein Layout wie in Abbildung 6.5(a) dargestellt.

Die 10 nF Kondensatoren werden so nahe wie möglich bei den entsprechenden Spannungsversorgungsanschlüssen des ICs platziert. Die Kontakte der 10 nF Kondensatoren sind direkt mit den Spannungsversorgungsanschlüssen verbunden und es fließen somit nur zwischen diesem Kondensator und dem IC hochfrequente Ströme, wodurch es zu einer Reduktion der EMV-Emissionen – im Vergleich zu einem Layout wie in Abbildung 6.5(b) gezeigt – kommt.

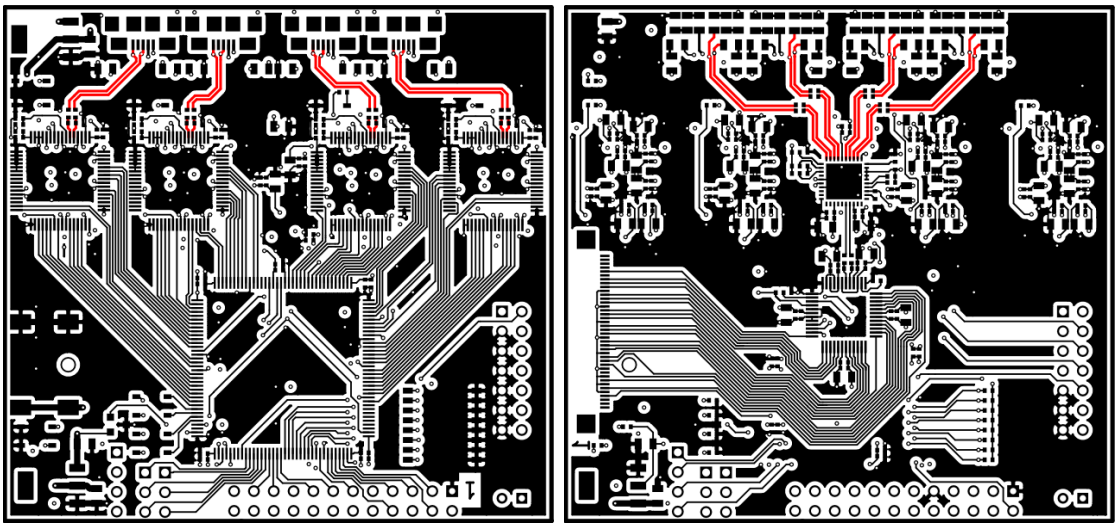


Abbildung 6.6: links: Toplayer wird hauptsächlich zur Signalführung der vier Upchannels verwendet; rechts: Bottomlayer wird hauptsächlich zur Signalführung des Downchannels verwendet

Abbildung 6.6 zeigt links das Layout für den Toplayer und rechts das Layout für den Bottomlayer des RPI-Testboards. Auf der Oberseite der Platine werden alle für die Upchannels des RPI-Busses notwendigen Bauelemente platziert und untereinander verbunden. Die vier Downchannels befinden sich hingegen auf der Unterseite des RPI-Testboards. Somit lassen sich wieder induktivitätsarme Verbindungen zwischen den einzelnen Bauelementen herstellen, da die Leiterbahnführung ohne einen Layerwechsel vorgenommen wird.

Die rot eingefärbten Leiterbahnen in Abbildung 6.6 sind impedanzkontrollierte Leiterbahnen und führen die differentiellen Signale für die verschiedenen Down- und Upchannels.

Bei der Leiterbahnführung der parallelen Eingangssignale des Serializers bzw. der parallelen Ausgangssignale der Deserializer wird wieder darauf geachtet, dass alle Leiterbahnen einer Datenquelle in etwa gleich lang sind, sodass es durch ein Laufzeitdelay zwischen den einzelnen Signalen zu keinem Fehlverhalten kommen kann.

7 Datenübertragung über den RPI-Bus

Die Datenkommunikation beim RPI-Bus soll über SPI stattfinden. Da jedoch die verwendeten Serializer und Deserializer nicht ausreichend viele parallele Ein- bzw. Ausgänge aufweisen, muss das Serial Peripheral Interface modifiziert werden, sodass zur Übertragung über den Downchannel nur zwei Leitungen – also zwei Bits im serialisierten Bitstrom – notwendig sind.

Dazu werden mittels des CPLDs auf dem RPI-Testboard die Signale $\overline{\texttt{SS}}$ und `MOSI` zu einem Signal zusammengefasst, wodurch sich jedoch die zeitlichen Zusammenhänge der Signale des SPIs ändern. Schlussendlich muss sichergestellt werden, dass die Timing-Anforderungen des SPI-Masters und eines SPI-Slaves nicht verletzt werden. Daher werden nun genauere Betrachtungen in Bezug auf den SPI-Master angestellt.

7.1 SPI-Master

Als SPI-Master kann eine von drei CSPI Schnittstellen des i.MX31 verwendet werden. Alle drei Schnittstellen verfügen über ein separates Kontrollregister `CONREG`, anhand dessen sich das Interface einstellen lässt. Abbildung 7.1 zeigt den Aufbau dieser Kontrollregister. In Tabelle 7.1 sind die Funktionen der einzelnen Registerfelder beschrieben, wobei die grau eingefärbte Spalte die für den RPI-Bus verwendeten Konfigurationswerte beinhaltet.

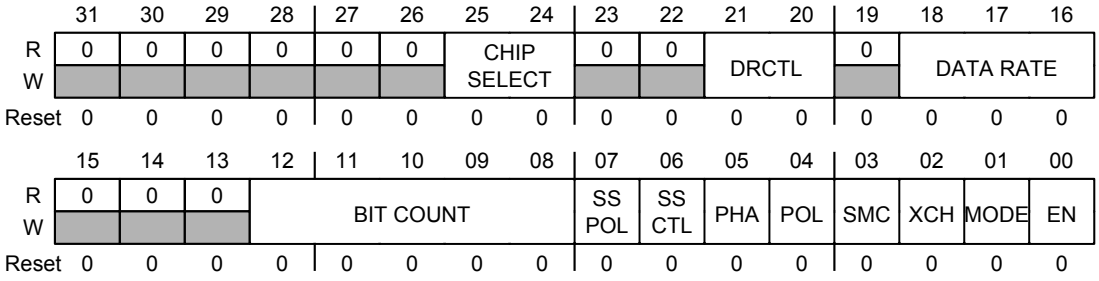


Abbildung 7.1: Aufbau der SPI Kontrollregister des i.MX31; auf grau hinterlegte Registerfelder kann nicht zugegriffen werden

Registerfeld	**Beschreibung**	**RPI**
31-26	reserviert	000000b
25-24 CHIP SELECT	zur Auswahl einer der vier $\overline{\text{SS}}$n Ausgänge; bei Beginn der Datenübertragung wird nur der ausgewählte $\overline{\text{SS}}$n Ausgang aktiviert	00b
23-22	reserviert	00b
21-20 DRCTL	Zum Aktivieren des SPI_RDY Eingangs; wird beim RPI-Bus nicht verwendet	00b
19	reserviert	0b
18-16 DATA RATE	Zum Einstellen der Baudrate von SCLK; mit diesem Registerfeld wird das Teilerverhältnis für die Eingangstaktfrequenz festgelegt	111b
15-13	reserviert	000b
12-08 BIT COUNT	Zum Einstellen der Framelänge eines SPI-Datentransfers	00111b
07 SSPOL	Zum Einstellen der Polarität der $\overline{\text{SS}}$n Ausgänge	0b
06 SSCTL	$\overline{\text{SS}}$n Ausgang bleibt zwischen einzelnen SPI-Transfers aktiviert oder wird invertiert	1b
05 PHA	Zum Einstellen der Taktflanke bei der die Daten an MOSI und MISO gültig sind	1b
04 POL	Zum Einstellen der Polarität von SCLK	1b
03 SMC	Zur Einstellung der Startbedingung eines SPI-Bursttransfers	0b
02 XCH	Zum Starten eines SPI-Bursttransfers bei SMC = 0b; 1b → startet SPI-Burst; wird nach Datentransfer automatisch gelöscht	1b
01 MODE	Zur Master oder Slave Konfiguration	1b
00 EN	Zum Aktivieren der SPI Schnittstelle	1b

Tabelle 7.1: Beschreibung der Registerfelder der SPI Kontrollregister und die zu verwendenden Initialisierungswerte für den RPI-Bus

Die in Tabelle 7.1 angeführten Einstellungen konfigurieren eine SPI Schnittstelle des i.MX31 als Master. Zur Datenübertragung wird der Modus 3 (POL = 1, PHA = 1) verwendet, bei dem SCLK im Ruhezustand einen logischen High-Pegel aufweist und die Daten an MOSI und MISO mit einer ⌟⌜-Flanke von SCLK übernommen

werden. Zum Selektieren eines Slaves wird der Ausgang $\overline{\text{SS0}}$ verwendet, der in weiterer Folge vom CPLD mit dem Signal `MOSI` verknüpft wird. Die Polarität von $\overline{\text{SS0}}$ wird so festgelegt, dass mit einem Low-Pegel der Slave selektiert wird. Als Teiler für die Eingangstaktfrequenz wird 512 verwendet, wodurch sich – bei einer maximalen Eingangstaktfrequenz von 62.5 MHz (vgl. [15], S.3-43) – die kleinst mögliche Baudrate von 122 kb/s für das Serial Peripheral Interface ergibt. Als Framelänge eines SPI-Transfers werden 8 Bit festgelegt.

Wird ein Bursttransfer eingeleitet – das bedeutet, dass mehrere Datenpakete mit einer Größe von jeweils acht Bit hintereinander übertragen werden – so nimmt das Signal $\overline{\text{SS0}}$ zwischen den einzelnen Datenpakete wieder einen High-Pegel an, bis das nächste Byte zur Übertragung bereit steht. Das Ende eines Bursttransfers tritt ein, sobald die Sendebuffer des SPI Interfaces leer sind. Gestartet wird der Bursttransfer mit dem Setzen des XCH-Bits im Kontrollregister.

Die SPI Schnittstellen des i.MX31 verfügen über ein weiteres Register (`PERIODREG`), mit dem die Dauer (Waitstates) zwischen den einzelnen Datenpakete bei Bursttransfers eingestellt werden können. Abbildung 7.2 zeigt den Aufbau dieser Register. In Tabelle 7.2 werden die Registerfelder beschrieben, sowie in der grau eingefärbten Spalte die beim RPI-Bus zu verwendenden Initialisierungswerte angeführt.

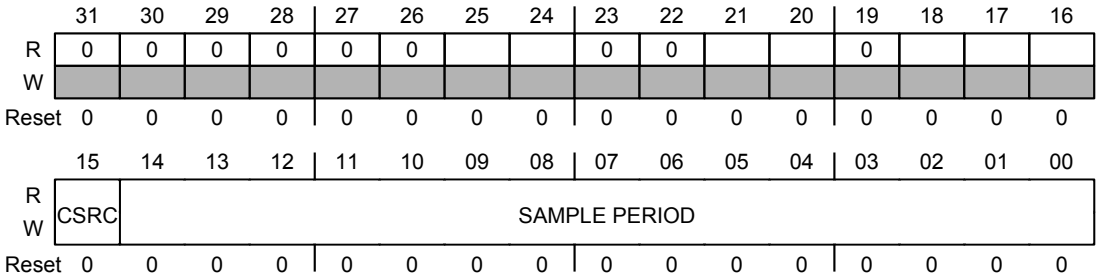


Abbildung 7.2: Aufbau der SPI Sample Period Kontrollregister des i.MX31; auf grau hinterlegte Registerfelder kann nicht zugegriffen werden

Registerfeld	**Beschreibung**	**RPI**
31-16	reserviert	`0000h`
15 CSCR	Zur Auswahl des zu verwendenden Taktsignals für den Waitstate-Counter	`0b`
14-00 SAMPLE PERIOD	Endwert des Waitstate-Counters; erreicht der Zähler diesen Wert, so wird das nächste Datenpaket eines SPI-Bursttransfers übertragen	`0001h`

Tabelle 7.2: Beschreibung der Registerfelder der SPI Sample Period Kontrollregister und die zu verwendenden Initialisierungswerte für den RPI-Bus

Das Sample Period Kontrollregister (`PERIODREG`) wird im Hinblick auf den RPI-Bus so konfiguriert, dass ein Taktzyklus von `SCLK` zwischen den einzelnen Datenpaketen eines Bursttransfers liegt. Somit wird einerseits einem langsameren Slave die

Möglichkeit eingeräumt, die empfangenen Daten zu verarbeiten (vgl. [15], S.24-19) und andererseits wird die Zeit zwischen zwei aufeinanderfolgenden Datenpakete zur Signalmodifikation am RPI-Testboard und zur Signalrekonstruktion am Peripheriegerät benötigt.

Abbildung 7.3 zeigt das Timing-Diagramm der SPI Schnittstellen des i.MX31 im Master Mode und der Konfiguration `POL` = 1, `PHA` = 1. In [15] wird angegeben, dass bei dieser Konfiguration die Daten im Senderegister der SPI Schnittstelle mit jeder ⅂-Flanke von `SCLK` am `MOSI` Ausgang bitweise hinaus geschoben werden. Das bedeutet, dass die Daten an `MOSI` kurz nach einer ⅂-Flanke und für die Dauer eines Taktzyklus von `SCLK` gültig sind.

Die Daten am `MISO` Eingang werden hingegen mit einer ⅃-Flanke an `SCLK` eingelesen, wobei diese mindestens 5 ns vorher und 6 ns danach einen gültigen und stabilen Zustand aufweisen müssen (vgl. [16], S.36f).

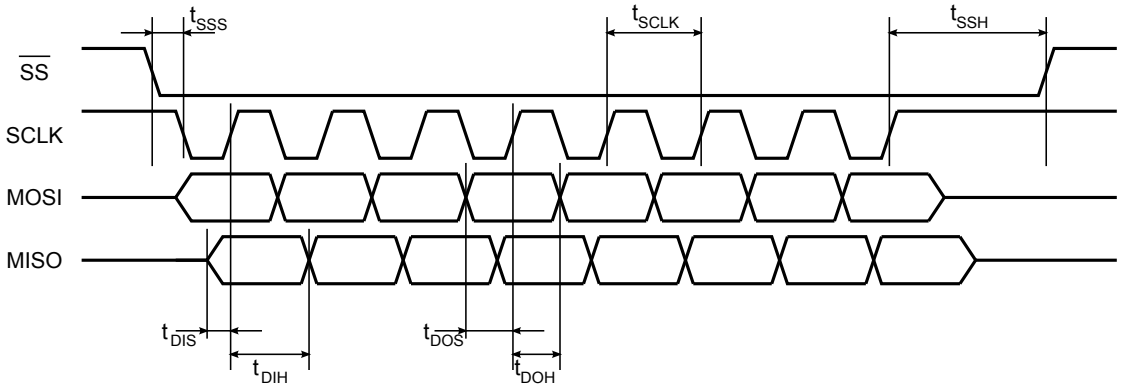


Abbildung 7.3: Timing der SPI Schnittstelle als Master und im Übertragungsmodus 3 mit eingezeichneten Timing-Parametern (vgl. [16], S.36)

Parameter	**Symbol**	**Min.**	**Max.**
Periodendauer von `SCLK`	t_{SCLK}	60 ns	8192 ns
$\overline{SS}$ Setup-Zeit	t_{SSS}	25 ns	—
$\overline{SS}$ Hold-Zeit	t_{SSH}	25 ns	—
`MOSI` Setup-Zeit	t_{DOS}	5 ns	—
`MOSI` Hold-Zeit	t_{DOH}	5 ns	—
`MISO` Setup-Zeit	t_{DIS}	6 ns	—
`MISO` Hold-Zeit	t_{DIH}	5 ns	—

Tabelle 7.3: SPI Timing-Parameter aus Abbildung 7.3 (vgl. [16], S.37)

Tabelle 7.3 gibt Auskunft über die Werte der in Abbildung 7.3 eingezeichneten Timing-Parameter. Anzumerken ist hierbei, dass die minimale Setup- und Hold-Zeit des `MOSI` Signals für die Konfiguration im Slave-Modus gilt, bei dem dieses Signal ein Eingangssignal ist.

Aufgrund der minimalen Setup-Zeit des `MISO` Signals können nun Überlegungen angestellt werden, wie viel Zeit die Datenübertragung über den RPI-Bus in Anspruch nehmen darf, sodass t_{DIS} eingehalten wird.

Die Signale $\overline{SS}$, `SCLK` und `MOSI` werden dem CPLD des RPI-Testboards zugeführt, das die Signalmodifikation von $\overline{SS}$ und `MOSI` vornimmt, wodurch alle Signale verzögert werden. Anschließend werden die Signale des Serializers und des LVDS-Repeaters verzögert. Aufgrund der Übertragung über ein Kabel kommt eine weitere Laufzeitverzögerung von etwa 5 ns/m hinzu. Beim Peripheriegerät wird der serialisierte Bitstrom bei der Rekonstruktion in ein paralleles 24 Bit Wort erneut verzögert.

Das Signal `MISO` des SPI-Slaves, welches üblicherweise im SPI-Modus 3 kurz nach einer ⌞-Flanke von `SCLK` gültig ist, muss anschließend über den Upchannel zum SPI-Master übertragen werden, wodurch sich weitere Verzögerungen durch den Serializer, den Deserializer, sowie durch die Laufzeitverzögerung am Übertragungskabel ergeben.

Mit Gleichung 7.1 wird versucht, all diese Verzögerungszeiten zu addieren, wobei Tabelle 7.4 eine Beschreibung der Zeitparameter beinhaltet.

$$\begin{aligned} t_{TD} \geq\ & t_{?1} + t_{SD,DC} + t_{DR} + t_D + t_{DD,DC} + ... \\ & ... + t_{?2} + t_{SD,UC} + t_D + t_{DD,UC} + t_{CPLD} \end{aligned} \tag{7.1}$$

Parameter	**Symbol**
Verzögerung von `SCLK` durch das CPLD	$t_{?1}$
Serializer-Delay des Downchannels (siehe Formel 4.1)	$t_{SD,DC}$
LVDS-Repeater-Delay	t_{DR}
Signallaufzeit auf der Übertragungsleitung	t_D
Deserializer-Delay des Downchannels (siehe Formel 4.2)	$t_{DD,DC}$
Setup-Zeit von `MISO` des SPI Slaves	$t_{?2}$
Serializer-Delay des Upchannels	$t_{SD,UC}$
Deserializer-Delay des Upchannels	$t_{DD,UC}$
Signalverzögerung durch das Multiplexing von `MISO` im CPLD	t_{CPLD}

Tabelle 7.4: Beschreibung der Zeitparameter aus Gleichung 7.1

Die gesamte Verzögerungszeit – wie in Gleichung 7.1 angeführt – darf nun einen bestimmten Wert nicht überschreiten, sodass die Setup-Zeit von MISO beim i.MX31 eingehalten werden kann. Berechnen lässt sich die maximal zulässige Verzögerung t_{TD} mittels Gleichung 7.2.

$$\begin{aligned} t_{TD} &\leq 0.5 \cdot t_{SCLK,max} - t_{DIS} \\ t_{TD} &\leq 0.5 \cdot 8192\,\text{ns} - 6\,\text{ns} = 4140\text{ns} \end{aligned} \tag{7.2}$$

Um nun einen Richtwert für die Summe der unbekannten Zeitparameter $t_{?1}$ und $t_{?2}$ aus Gleichung 7.1 zu erhalten, müssen Überlegungen bezüglich der minimalen Taktfrequenzen der SerDes von Down- und Upchannel angestellt werden.

Der Downchannel wird mit dem Taktsignal der Videoausgabe getaktet, wobei das Display der Adapterplatine des i.MX31 Developementboards typischer Weise mit einem Pixeltakt von 5.33 MHz angesteuert wird (vgl. [30], S.5). Somit wird als minimale Taktfrequenz für den Downchannel – bedingt durch die verwendeten SerDes – 5 MHz festgelegt.

Beim Upchannel wird jenes Taktsignal zum Ansteuern der SerDes verwendet, welches die höchste Frequenz aufweist und wird somit bei Peripheriegeräten – welche eine Videoeingabeeinheit besitzen – dem Pixeltakt des CMOS Sensors entsprechen. Für alle anderen Geräte wird nun eine minimale Taktfrequenz von 20 MHz für die SerDes des Upchannels festgelegt, sodass auch die – im Vergleich zu den Nutzdaten – hochfrequenten Audiosignale übertragen werden können.

Mit diesen gewählten minimalen Taktfrequenzen der SerDes-Paare des Down- und Upchannels lässt sich nun ein Wert für die Summe der unbekannten Zeitparameter $t_{?1}$ und $t_{?2}$ berechnen, wobei die Verzögerungszeit des LVDS-Repeaters aus [31] entnommen wurde. Die Verzögerungszeiten der Serializer und Deserializer wurden mit Hilfe der Formeln 4.1 und 4.2 für die entsprechenden minimalen Taktfrequenzen berechnet, wodurch nun für die unbekannten Zeitparameter und das Multiplexing des Signals MISO in Summe 1945.9 ns zur Verfügung stehen.

$$\begin{aligned} 4140 &\geq t_{?1} + 710\,\text{ns} + 0.6\,\text{ns} + 5\,\text{ns} + 1026\,\text{ns} + ... \\ &... \quad + t_{?2} + 185\,\text{ns} + 5\,\text{ns} + 262.5\,\text{ns} + t_{CPLD} \\ 1945.9 &\geq t_{?1} + t_{?2} + t_{CPLD} \end{aligned} \tag{7.3}$$

7.2 Digitaldesign zur Datenübertragung am RPI-Testboard

In diesem Abschnitt wird auf das Digitaldesign des CPLDs am RPI-Testboard eingegangen und erläutert, wie die Signale MOSI und $\overline{\text{SS}}$ des Serial Peripheral Interfaces zusammengefasst werden. Der Sourcecode für das Digitaldesign befindet sich im Anhang B.

7.2.1 Synchronisation der SPI Schnittstelle

Da die Signale der SPI Schnittstelle nicht synchron zum Pixeltakt der Videoausgabe sind und dieses Taktsignal aber zum Einlesen der Daten beim Serializer verwendet wird, müssen die Signale $\overline{\texttt{SS}}$, `MOSI` und `SCLK` erst mit dem Pixeltakt synchronisiert werden.

Werden diesbezüglich keine Vorkehrungen getroffen, so kann es zu einer Verletzung der Setup- und Hold-Zeiten des Input-Latches vom Serializer kommen. Abbildung 7.4 zeigt ein Timing, bei dem das Signal `SCLK` der SPI Schnittstelle nicht mit dem Pixeltakt synchronisiert wird, bevor es dem Serializer zugeführt wird. Es ist zu erkennen, dass bei der zweiten ⎍-Flanke von `PIXCLK` das Signal `SCLK` während der Setup-Zeit t_{DIS} nicht stabil ist und somit kann der Ausgang des Latches für eine gewisse Zeit einen metastabilen Zustand annehmen und kippt anschließend entweder auf Low oder High.

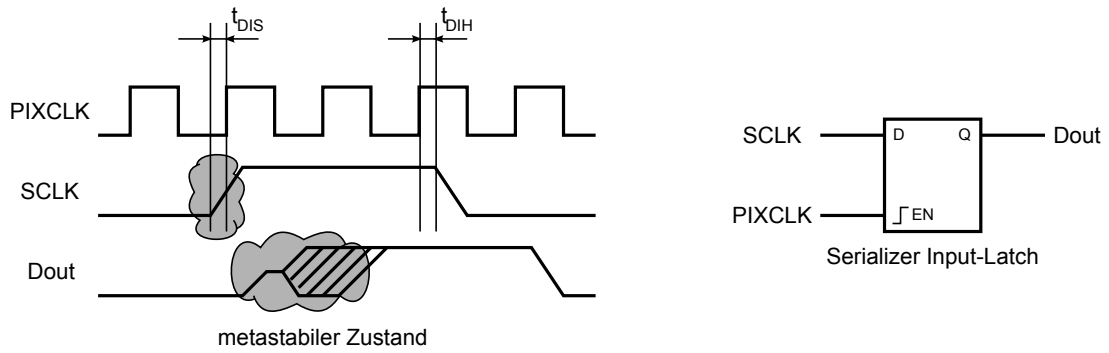


Abbildung 7.4: Verhalten eines Flipflop- oder Latchausganges wenn die Setup- oder Hold-Zeiten beim Eingang verletzt werden. (vgl. [32], o.S.)

Auch nach einer Synchronisation der Signale kann es zu einem metastabilen Zustand kommen, jedoch wird die Wahrscheinlichkeit, dass ein solcher auftritt, drastisch verringert. Mittels Gleichung 7.4 (vgl. [33], S.2) lässt sich die mittlere Zeitdauer zwischen zwei auftretenden metastabilen Zuständen bei einem Flipflop berechnen.

$$MTBF = \frac{e^{(t_r/\tau)}}{f_{clk} \cdot f_{async} \cdot T_0} \tag{7.4}$$

Die Parameter τ und T_0 sind von der Prozesstechnologie des Flipflops abhängig und sind beim CPLD XC2C256 mit $\tau = 90.3\,\mathrm{E}^{-12}\,\mathrm{s}$ und $T_0 = 1.98\,\mathrm{E}^{13}\,\mathrm{s}$ gegeben (vgl. [33], S.1). Bei der Zeitdauer t_r handelt es sich um die Zeit, die dem Ausgang des Flipflops zur Verfügung steht, um wieder einen gültigen Wert zu erlangen, bis er wieder verwendet wird. Diese Zeitdauer setzt sich aus der Verzögerungszeit des Filpflops (t_{COI}), der Setup-Zeit (t_{SUI}) des nachfolgenden Flipflops, der Laufzeit (t_{PD}) des Signals vom Ausgang des ersten Flipflops zum Eingang des zweiten Flipflops und der Periodendauer des Taktsignals (T_{CLK}) – mit dem die beiden Flipflops getaktet werden – zusammen. Gleichung 7.5 zeigt den mathematischen Zusammenhang von t_r (vgl. [32], o.S.).

$$t_r = T_{CLK} - (t_{COI} + t_{PD} + t_{SUI}) \tag{7.5}$$

Zum Synchronisieren der Signale des SPI Interfaces mit dem Pixeltakt der Videoausgabe werden im CPLD nun zwei Flipflops in serie geschaltet. Diese Flipflops werden sowohl bei einer ⌐-Flanke als auch bei einer ¬-Flanke des Pixeltakts getriggert, wodurch die Signale des Serial Peripheral Interfaces bei einem Pixeltakt von 5 MHz schließlich mit 10 MHz ($T_{CLK} = 100\,ns$) abgetastet werden. Die Setup- und Verzögerungszeit von Flipflops im XC2C256 sind laut [24] mit $t_{SUI} \geq 1.8\,ns$ und $t_{COI} \leq 0.7\,ns$ gegeben. Die Signallaufzeit im CPLD zwischen dem Ausgang des ersten Flipflops und dem Eingang des zweiten Flipflops ist mit $t_F \leq 3\,ns$ und $t_{LOGI1} \leq 1.1\,ns$ gegeben. Mittels dieser Werte und Gleichung 7.5 lässt sich nun die Zeitdauer berechnen, die dem Ausgang eines Flipflops zur Verfügung steht um wieder einen stabilen Wert zu erlangen, bevor der Ausgangswert weiterverarbeitet wird.

$$t_r = 100\,ns - (0.7\,ns + 3\,ns + 1.1\,ns + 1.8\,ns) = 93.4\,ns$$

Werden zwei in Reihe geschaltete Flipflops zum Synchronisieren von `SCLK` verwendet, so lässt sich die MTBF nach Gleichung 7.6 errechnen (vgl. [34], S.2f).

$$\begin{aligned} MTBF &= \frac{e^{(t_r/\tau)}}{f_{async} \cdot T_0} \cdot \frac{e^{(t_r/\tau)}}{f_{clk} \cdot T_0} \\ &= \frac{e^{(93.4\,ns/90.3\,ps)}}{122\,kHz \cdot 1.98\,E^{13}\,s} \cdot \frac{e^{(93.4\,ns/90.3\,ps)}}{10\,MHz \cdot 1.98\,E^{13}\,s} \\ &= 53.45\,E^{858}\,s \\ &\approx 17\,E^{842}\,\text{Milliarden Jahre} \\ &\quad \text{Alter der Erde: 4.55 Milliarden Jahre} \end{aligned} \tag{7.6}$$

Abbildung 7.5 zeigt die Synchronizer-Struktur zum Synchronisieren des Signals `SCLK`, wobei die Signale `MOSI` und $\overline{\texttt{SS}}$ mit der gleichen Struktur synchronisiert werden. Rechts von der strichlierten Linie ist das Signal `SCLK_sync` synchron zum Pixeltakt.

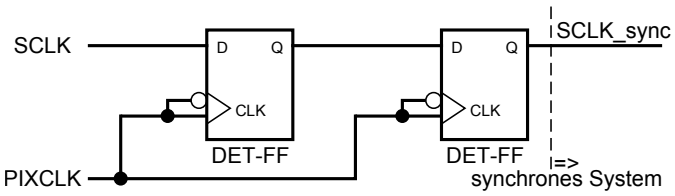


Abbildung 7.5: zweistufige Synchronizer-Struktur mit Dual-Edge Triggered Flipflops

7.2.2 Modifikation von `MOSI_sync` und $\overline{\texttt{SS}}$`_sync`

Nach der Synchronisation der SPI Signale mit dem Pixeltakt werden die Signale `MOSI_sync` und $\overline{\texttt{SS}}$`_sync` zum Signal `MOSI_SS` zusammengefasst. Dazu wird das

Signal `MOSI_sync` nach einer ⌝⌞-Flanke von $\overline{\texttt{SS}}$`_sync` für eine Periodendauer des Pixeltakts auf einen Low-Pegel, eine Periodendauer lang auf einen High-Pegel und nochmals für die Dauer einer Pixeltaktperiode auf einen Low-Pegel gelegt. Anschließend nimmt das Signal `MOSI_SS` den aktuellen Wert des `MOSI_sync` Signals an. Unabhängig vom aktuellen Wert dieses Signals treten durch die Modifikation bei jedem neu eingeleiteten SPI-Transfer – hervorgerufen durch einen ⌝⌞-Übergang an $\overline{\texttt{SS}}$ – zwei ⌝⌞-Übergänge auf, bevor das `MOSI_SS` Signal zum Datentransport verwendet wird.

Aufgrund dieser Signalmodifikation eilt nun das Taktsignal `SCLK_sync` des Serial Peripheral Interfaces, dem im `MOSI_SS` Signal integrierten Slave-Select Signal vor. Deshalb wird das `SCLK_sync` Signal um sechs weitere Taktperioden vom Pixeltakt verzögert, sodass dieses auch nach der Rekonstruktion der Signale `MOSI` und $\overline{\texttt{SS}}$ dem Slave-Select Signal auf jedem Fall nacheilt.

Während das synchronisierte Slave-Select Signal einen Low-Pegel aufweist, werden die ⌟⌜-Flanken des verzögerten SPI-Taktsignals `SCLK_delay` mitgezählt. Sobald das synchronisierte Slave-Select Signal wieder einen High-Pegel aufweist und der Taktflankenzähler übergelaufen ist, wird das `MOSI_SS` Signal erneut modifiziert.

Anschließend wird unterschieden, ob das Signal `MOSI_Sync` beim letzten ⌝⌞-Übergang von `SCLK_delay` einen Low- oder High-Pegel aufwies. War das zuletzt übertragene Datenbit an `MOSI_Sync` ein High-Bit, dann wird das `MOSI_SS` Signal wieder mit einem Low-Puls, einem High-Puls und einem weiteren Low-Puls – mit jeweils der Dauer eine Pixeltaktperiode – beaufschlagt, bevor es den Ruhezustand High erreicht. War jedoch das letzte Datenbit an `MOSI_Sync` ein Low-Bit, so wird das Signal `MOSI_SS` zuerst auf High gesetzt, bevor es mit den vorhin erwähnten Pulsen beaufschlagt wird.

Da die Flipflops der Synchronizer und die der Delayline des SPI-Taktsignals, bei einem Reset des CPLDs mit logisch 0 initialisiert werden, jedoch alle Signale einen Ruhezustand von logisch 1 aufweisen, müssen diese Flipflops erst mit korrekten Werten initialisiert werden. Dazu werden nach einem Reset acht Taktzyklen vom Pixeltakt abgewartet, um sicherzustellen, dass alle Flipflops mit korrekten Werten geladen sind. Erst dann ist das CPLD bereit um die Modifikation von `MOSI` und $\overline{\texttt{SS}}$ vornehmen zu können. Das CPLD generiert dann ein Ready-Signal um dem i.MX31 mitteilen zu können, dass es für die Datenkommunikation bereit ist.

Der vorhin beschriebene Vorgang zur Modifikation der Signale `MOSI` und $\overline{\texttt{SS}}$ wird mittels einer FSM im CPLD implementiert. Abbildung 7.6 zeigt das Zustandsübergangsdiagramm der FSM zum Erzeugen des Signals `MOSI_SS`. Bei den Zuständen `MOSI_SS_0`, `MOSI_SS_2`, `MOSI_DS_0` und `MOSI_DS_2` wird das Signal `MOSI_SS` jeweils für einen Taktzyklus des Pixeltakts auf Low gesetzt. Bei den Zuständen `MOSI_SS_1`, `MOSI_DS_1` und `MOSI_DS_3` hingegen auf High. Im Zustand `IDLE` und `WAIT` folgt das `MOSI_SS` Signal dem Pegel des synchronisierten `MOSI_sync` Signals.

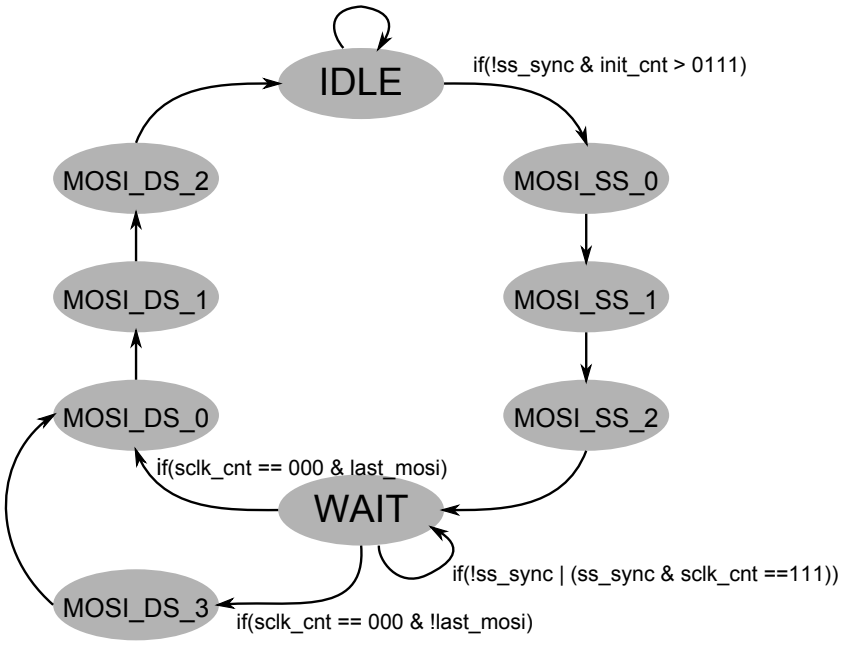


Abbildung 7.6: Zustandsübergangsdiagramm der FSM zur Erzeugung des kombinierten `MOSI_SS` Signals

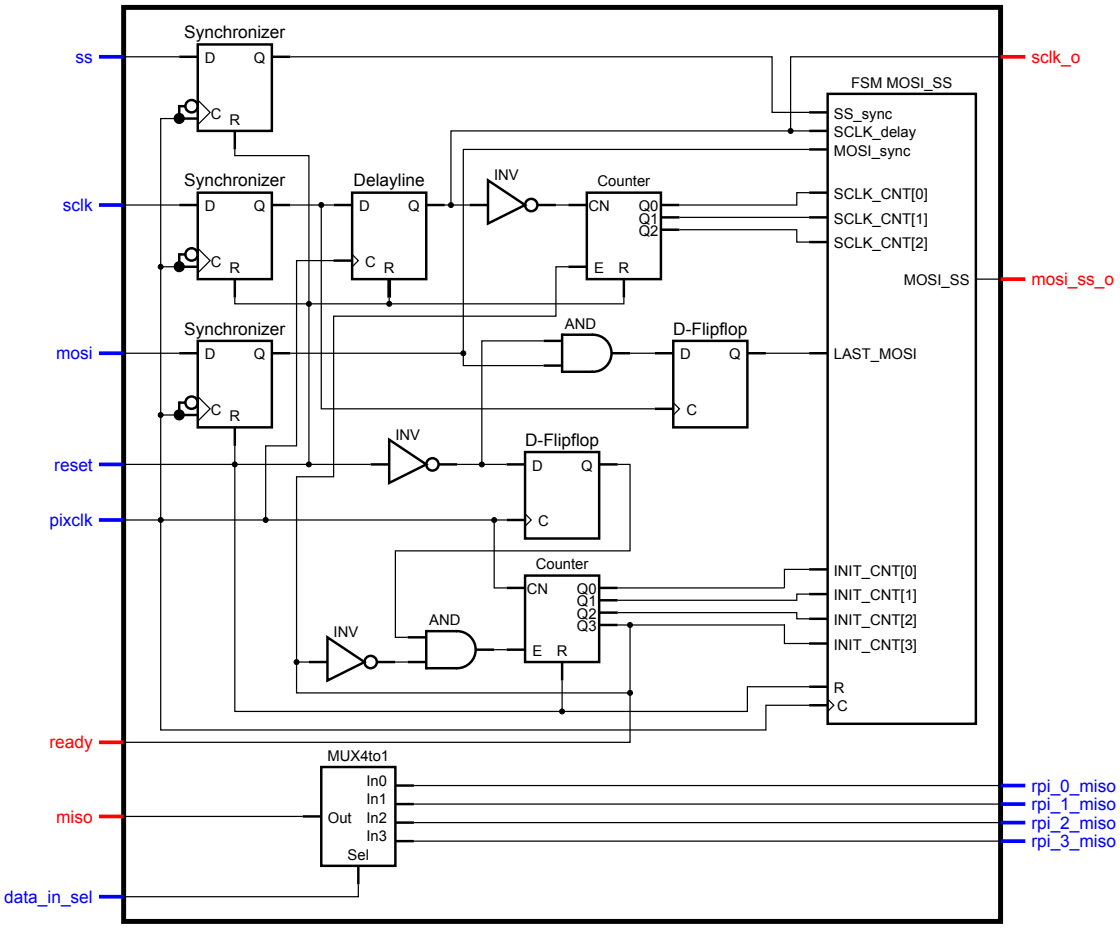


Abbildung 7.7: Digitaldesign der Signalmodifikation von $\overline{\texttt{SS}}$ und `MOSI`

Abbildung 7.7 zeigt den Aufbau des Digitaldesigns im CPLD zur Modifikation des SPI, um es über den Downchannel übertragen zu können. Blau eingefärbte Anschlüsse entsprechen Eingängen des CPLDs, rot gekennzeichnete Anschlüsse sind hingegen Ausgänge des CPLDs. Diese Grafik wurde leicht vereinfacht aus dem Tech Viewer des Xilinx ISE Design Suites, in dem das Digitaldesign für das CPLD durchgeführt wurde, entnommen.

7.3 Timing der modifizierten SPI Schnittstelle

Abbildung 7.8 zeigt die zeitlichen Zusammenhänge der Eingangssignale des CPLDs und dessen generierten Ausgangssignale. Zusätzlich ist auch der Verlauf des Pixeltakts – mit dem die Signale im CPLD synchronisiert werden – zu sehen. In Tabelle 7.5 sind die im Timing eingezeichneten Zeitparameter beschrieben und es werden auch deren Größenordnungen – bei einer minimalen Taktfrequenz des Downchannels von 5 MHz – angegeben.

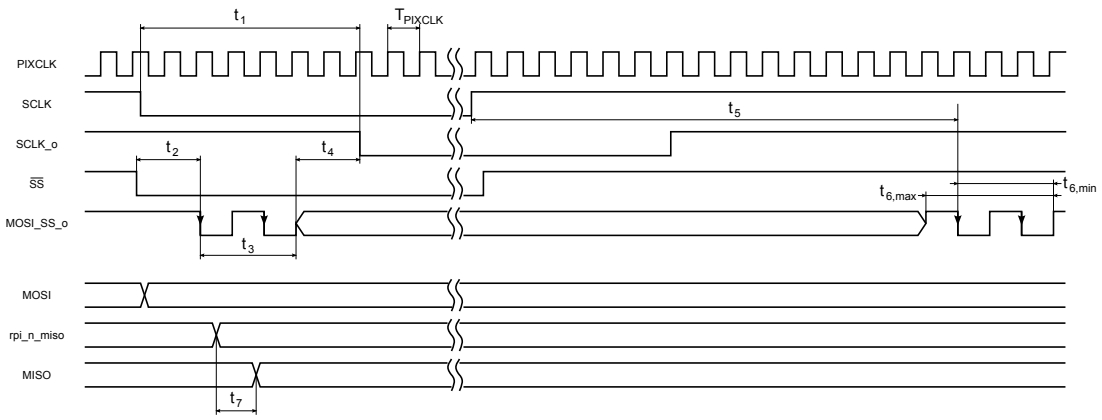


Abbildung 7.8: Timing-Diagramm der modifizierten SPI Schnittstelle

Parameter	**Symbol**	**Min.**	**Max.**
Periodendauer des Pixeltakts	T_{PIXCLK}	200 ns	
Zeitverzögerung zwischen `SCLK` und dem Ausgangssignal `SCLK_o` des CPLDs	t_1	1206 ns	1406 ns
Zeitverzögerung zwischen $\overline{SS}$ und dem modifizierten Ausgangssignal `MOIS_SS` des CPLDs	t_2	411.3 ns	611.3 ns
Dauer der Signalmodifikation für das in `MOSI_SS_o` integrierte Select-Kommando	t_3	600 ns	
`MOSI_SS_o` gültig vor `SCLK_o`	t_4	194.7 ns	394.7 ns
Verzögerungszeit zwischen letzter ⌟-Flanke von `SCLK` und erster ⌞-Flanke des Deselect-Kommandos	t_5	1586.3 ns	2186.3 ns

Fortsetzung auf nächster Seite

Tabelle 7.5 – *fortgesetzt von vorheriger Seite*

Parameter	**Symbol**	**Min.**	**Max.**
Dauer der Signalmodifikation für das in `MOSI_SS_o` integrierte Deselect-Kommando	t_6	600 ns	800 ns
Verzögerungszeit zwischen `rpi_n_miso` Ausgang eines Upchannels zum `MISO` Eingang des i.MX31	t_7	7.5 ns	

Tabelle 7.5: Beschreibung der Zeitparameter aus Abbildung 7.8

7.3.1 Ermittlung der Timing-Parameter

Das Signal `SCLK` wird zu Beginn mit dem Pixeltakt synchronisiert und anschließend mit Hilfe eines Schieberegisters, welches mit jeder ⌋⌈-Flanke von `PIXCLK` getriggert wird, um weitere sechs Taktperioden verzögert. Der Minimalwert für t_1 ergibt sich dann, wenn das Eingangsignal `SCLK` gleichzeitig mit einer ⌋⌈-Flanke an `PIXCLK` den Pegel wechselt und ein eventuell auftretender metastabiler Zustand richtig aufgelöst wird (d.h., dass der Ausgang des betroffenen Flipflops bereits auf den aktuellen Wert von `SCLK` kippt, anstatt zurück auf den alten Wert von `SCLK`).

$$\begin{aligned} t_{1,min} &= 6 \cdot T_{PIXCLK} + t_{GCK} + t_{COI} + t_{OUT} \\ t_{1,min} &= 6 \cdot 200\,\text{ns} + 2.7\,\text{ns} + 0.7\,\texttt{ns} + 2.6\,\text{ns} \\ t_{1,min} &= 1206\,\text{ns} \end{aligned}$$

Der maximale Wert für t_1 ergibt sich hingegen dann, wenn das Taktsignal `SCLK` ebenfalls mit einer ⌋⌈-Flanke an `PIXCLK` den Pegel wechselt und der eventuell aufgetretene metastabile Zustand jedoch falsch aufgelöst wird (.d.h., dass der Ausgang des betroffenen Flipflops auf den alten Wert von `SCLK` kippt).

$$\begin{aligned} t_{1,max} &= 7 \cdot T_{PIXCLK} + t_{GCK} + t_{COI} + t_{OUT} \\ t_{1,max} &= 7 \cdot 200\,\text{ns} + 2.7\,\text{ns} + 0.7\,\text{ns} + 2.6\,\text{ns} \\ t_{1,max} &= 1406\,\text{ns} \end{aligned}$$

Der Zeitparameter t_1 entspricht zusätzlich dem in Gleichung 7.3 angeführten unbekannten Parameter $t_{?1}$.

Die Parameter t_{GCK} und t_{OUT} sind ebenfalls wie der Parameter t_{COI} – der schon zur Berechnung der MTBF benötigt wurde – Zeitparameter des CPLDs und wurden aus [24] entnommen. Beim Parameter t_{GCK} handelt es sich um das Global Clock Buffer Delay und bei t_{OUT}, um das Output Buffer Delay. Die Summer aus den Parametern t_{GCK}, t_{OUT} und t_{COI} entspricht bei einem Flipflop des CPLDs der Zeitdauer vom Auftreten einer triggernden Taktflanke bis zum Zeitpunkt, bei dem der Ausgang des Flipflops einen gültigen und stabilen Wert erreicht (vgl. [35], S.3).

Das Signal $\overline{\texttt{SS}}$ wird ebenfalls erst mit dem Signal `PIXCLK` synchronisiert und anschließend der FSM zur Signalmodifikation zugeführt, die ebenfalls mit ⌐-Übergängen an `PIXCLK` getriggert wird. Unter der Berücksichtigung des Auftretens von Setup-Zeit Verletzungen der Synchronizer-Struktur kann auch hier wieder ein Minimal- und Maximalwert für t_2 angegeben werden. Der Minimalwert lässt sich somit anhand folgenden mathematischen Zusammenhangs berechnen:

$$\begin{aligned} t_{2,min} &= 2 \cdot T_{PIXCLK} + t_{GCK} + t_{COI} + ... \\ &\quad ... + \overbrace{t_F + t_{LOGI1} + t_{LOGI2} + t_{PDI}}^{t_{PD2'}} + t_{OUT} \\ t_{2,min} &= 2 \cdot 200\,\text{ns} + 2.7\,\text{ns} + 0.7\,\text{ns} + ... \\ &\quad ... + 3\,\text{ns} + 1.1\,\text{ns} + 0.5\,\text{ns} + 0.7\,\text{ns} + 2.6\,\text{ns} \\ t_{2,min} &= 411.3\,\text{ns} \end{aligned}$$

Der Maximalwert für t_2 ist gleich, wie beim Zeitparameter t_1, um eine Taktperiode des Pixeltakts größer als der Minimalwert und ergibt sich somit zu $t_{2,\text{max}} = 611.3\,\text{ns}$.

Das Signal `MOSI_`$\overline{\texttt{SS}}$`_o` ändert erst 11.3 ns nach einer ⌐-Flanke an `PIXCLK` den Wert, da dieses Signal sowohl mittels sequentieller, als auch mittels kombinatorischer Logik erzeugt wird und somit die Verzögerungszeiten t_F, t_{LOGI1}, t_{LOGI2} und t_{PDI} hinzukommen (vgl. [35], S.7).

Die Signalmodifikation von $\overline{\texttt{SS}}$ und `MOSI` nimmt aufgrund der FSM genau drei Taktzyklen von `PIXCLK` in Anspruch, wodurch sich t_3 immer mit 600 ns ergibt.

Der Minimalwert für den Zeitparameter t_4 ergibt sich entweder dann, wenn das $\overline{\texttt{SS}}$ Signal gleichzeitig mit einer ⌐-Flanke an `PIXCLK` von High auf Low geht und ein eventuell entstandener metastabiler Zustand falsch aufgelöst wird, oder wenn das Signal `SCLK` gleichzeitig mit einer ⌐-Flanke von `PIXCLK` den Pegel wechselt und der metastabile Zustand ebenfalls falsch aufgelöst wird. In diesem Fall eilt das Signal `SCLK_o` dem Signal `MOSI_`$\overline{\texttt{SS}}$`_o` nur in etwa eine Taktperiode des Pixeltakts vor.

$$\begin{aligned} t_{4,min} &= T_{PIXCLK} - t_{PD2'} \\ t_{4,min} &= 200\,\text{ns} - 5.3\,\text{ns} \\ t_{4,min} &= 194.7\,\text{ns} \end{aligned}$$

Beim Maximalwert von t_4 werden die metastabilen Zustände jedoch richtig aufgelöst und das Signal `SCLK_o` eilt somit dem Signal `MOSI_`$\overline{\texttt{SS}}$`_o` beinahe zwei Taktperioden des Pixeltakts vor.

$$\begin{aligned} t_{4,max} &= 2 \cdot T_{PIXCLK} - t_{PD2'} \\ t_{4,max} &= 2 \cdot 200\,\text{ns} - 5.3\,\text{ns} \\ t_{4,max} &= 394.7\,\text{ns} \end{aligned}$$

Der Minimalwert für den Zeitparameter t_5 ergibt sich dann, wenn die letzte ⌐-Flanke eines Bursts an `SCLK` gleichzeitig mit einer ⌐-Flanke an `PIXCLK` auftritt,

der metastabile Zustand beim Synchronizer für das SPI-Taktsignal jedoch richtig aufgelöst wird und das letzte Datenbit an `MOSI` einen High-Pegel aufweist.

$$\begin{aligned} t_{5,min} &= 8 \cdot T_{PIXCLK} + t_{GCK} + t_{COI} + t_{PD2'} + t_{OUT} - t_{SSH,min} \\ t_{5,min} &= 8 \cdot 200\,\text{ns} + 2.7\,\text{ns} + 0.7\,\text{ns} + 5.3\,\text{ns} + 2.6\,ns - 25\,\text{ns} \\ t_{5,min} &= 1586.3\,\text{ns} \end{aligned}$$

Beim Maximalwert für t_5 wird der vorhin erwähnte metastabile Zustand bei der entsprechenden Synchronizer-Struktur falsch aufgelöst und das letzte Datenbit an `MOSI` weist einen Low-Pegel auf.

$$\begin{aligned} t_{5,max} &= 11 \cdot T_{PIXCLK} + t_{GCK} + t_{COI} + t_{PD2'} + t_{OUT} - t_{SSH,min} \\ t_{5,max} &= 11 \cdot 200\,\text{ns} + 2.7\,\text{ns} + 0.7\,\text{ns} + 5.3\,\text{ns} + 2.6\,ns - 25\,\text{ns} \\ t_{5,max} &= 2186.3\,\text{ns} \end{aligned}$$

Auch der Zeitparameter t_6 ist abhängig vom Wert des letzten Datenbits an `MOSI` bei einem SPI-Transfer. Ist dieses Bit ein High-Bit, so wird das Deselect-Kommando mittels drei Zuständen von der FSM generiert, wodurch sich $t_{6,\text{min}}$ mit $3 \cdot T_{PIXCLK} = 600\,\text{ns}$ ergibt. Ist das letzte Datenbit jedoch ein Low-Bit, so durchläuft die FSM einen vierten Zustand und $t_{6,\text{max}}$ beträgt somit 800 ns.

Zwischen dem Deselektieren des SPI-Slaves durch einen ⌟⌜-Übergang an $\overline{\texttt{SS}}$ und dem Ende des dazugehörigen Deselect-Kommandos verstreicht eine Zeitdauer von 2786.3 ns (entspricht der Summe aus $t_{5,\text{max}}$ und $t_{6,\text{min}}$). Es muss jedoch sichergestellt werden, dass der nächste Datentransfer – eingeleitet durch einen ⌜⌞-Übergang an $\overline{\texttt{SS}}$ – erst dann beginnt, wenn das Deselect-Kommando abgeschlossen ist, wodurch zumindest ein Waitstate zwischen zwei SPI-Transfers erforderlich wird.

Die Zeitdauer t_7 entspricht der Verzögerungszeit im CPLD (vgl. t_{CPLD} in Tabelle 7.5) durch das Multiplexing des `MISO` Signals von den vier Upchannels und ergibt sich mit 7.5 ns. Dieser Wert entspricht dem Multi Product Term Propagation Delay T_{PD2} und wurde aus [35] entnommen.

Mit diesen neu erhaltenen Zeitparametern t_1 und t_7 kann nun ein Wert für den noch unbekannten Zeitparameter $t_{?2}$ aus Gleichung 7.3 bestimmt werden.

$$\begin{aligned} 1945.9\,\text{ns} &\geq t_{1,max} + t_{?2} + t_7 \\ 1945.9\,\text{ns} &\geq 1406\,\text{ns} + t_{?2} + 7.5\,\text{ns} \\ 532.4\,\text{ns} &\geq t_{?2} \end{aligned}$$

Das bedeutet, dass beim Peripheriegerät 532.4 ns zur Verfügung stehen, bevor das Signal `MISO` am entsprechenden Eingang des Upchannel-Serializers einen gültigen Wert aufweisen muss.

7.4 Digitaldesign zur Datenübertragung bei Peripheriegeräten

Beim Peripheriegerät stehen nach dem Deserializer des Downchannels die Signale `SCLK` und `MOSI_SS` zur Verfügung. Mit einem CPLD am Peripheriegerät können nun wieder die für SPI notwendigen Signale rekonstruiert werden.

Gleich wie beim RPI-Testboard, muss das CPLD, das über den Upchannel zu übertragende Signal `MISO` mit dem Taktsignal des Serializers synchronisieren, sodass die Setup- und Hold-Zeiten desselbigen nicht verletzt werden.

7.4.1 Synchronisation von `MISO`

Die Synchronisation des `MISO` Signals des SPI-Slaves wird mit der gleichen Synchronizer-Struktur, wie in Abbildung 7.5 gezeigt, durchgeführt. Da jedoch die minimale Taktfrequenz des Upchannels um den Faktor vier höher ist als die des Downchannels ergibt sich hier eine geringere MTBF. Wird bei den Peripheriegeräten das gleiche CPLD wie am RPI-Testboard verwendet, so kann die MTBF folgendermaßen berechnet werden:

$$\begin{aligned} t_r &= 25\,\text{ns} - (0.7\,\text{ns} + 3\,\text{ns} + 1.1\,\text{ns} + 1.8\,\text{ns}) \\ &= 18.4\,\text{ns} \end{aligned}$$

$$\begin{aligned} MTBF &= \frac{e^{(18.4\,\text{ns}/90.3\,\text{ps})}}{122\,\text{kHz} \cdot 1.98\,\text{E}^{13}\,\text{s}} \cdot \frac{e^{(18.4\,\text{ns}/90.3\,\text{ps})}}{40\,\text{MHz} \cdot 1.98\,\text{E}^{13}\,\text{s}} \\ &= 508.718\,\text{E}^{135}\,\text{s} \\ &\approx 16\,\text{E}^{120}\,\text{Milliarden Jahre} \end{aligned}$$

7.4.2 Signalrekonstruktion des Serial Peripheral Interface

Um das für den SPI-Slave notwendige $\overline{\text{SS}}$ Signal erzeugen zu können, muss das CPLD die beiden ⅂-Übergänge im `MOSI_SS` Signal detektieren.

Durch die Verzögerung des `SCLK` Signals im CPLD des RPI-Testboards wird sichergestellt, dass sich dieses während der beiden ⅂-Flanken des Select-Kommandos noch im Ruhezustand befindet. Aufgrund der zeitlichen Verschiebung zwischen `SCLK` und den Daten in `MOSI_SS` finden jedoch auch die Pegelwechsel von den Daten nur während ⎍⎍-Pulsen von `SCLK` statt. Deshalb müssen auch hier wieder die ⅂-Übergänge von `SCLK` mitgezählt werden, sodass in weiterer Folge nach der Datenübertragung das Deselct-Kommando detektiert werden kann.

Abbildung 7.9 zeigt das Zustandsübergangsdiagramm der FSM zur Signalrekonstruktion im CPLD der Wearables. Das Signal `MOSI_SS` weist im Ruhezustand einen High-Pegel auf. Wird ein ↓-Übergang im `MOSI_SS` Signal erkannt, so wartet das CPLD auf eine weitere ↓-Flanke und setzt dann das $\overline{SS}$ Signal auf Low. Anschließend verharrt das CPLD für acht Taktzyklen von `SCLK` in einem Waitstate. Sobald der Taktzyklenzähler von `SCLK` übergelaufen ist, beginnt das CPLD wieder mit der Detektion von ↓-Übergängen im `MOSI_SS` Signal, wobei nach der zweiten ↓-Flanke der SPI-Slave durch einen ↑-Übergang an $\overline{SS}$ wieder deselektiert wird und das CPLD wartet erneut auf ein Select-Kommando.

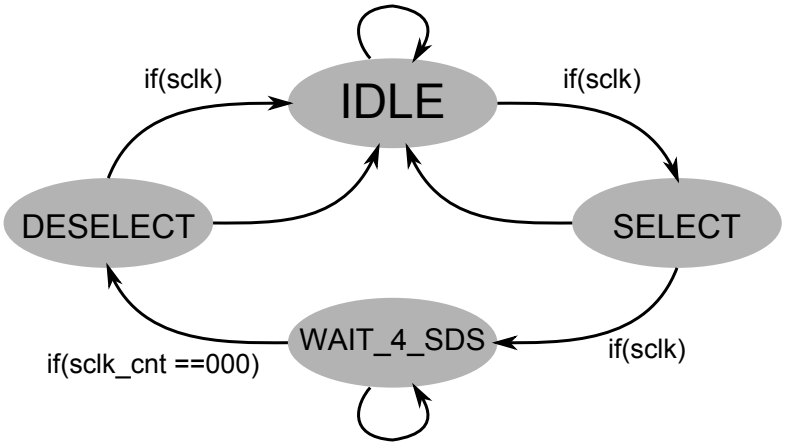


Abbildung 7.9: Zustandsübergangsdiagramm der FSM zur Signalrekonstruktion von $\overline{SS}$ und `MOSI`

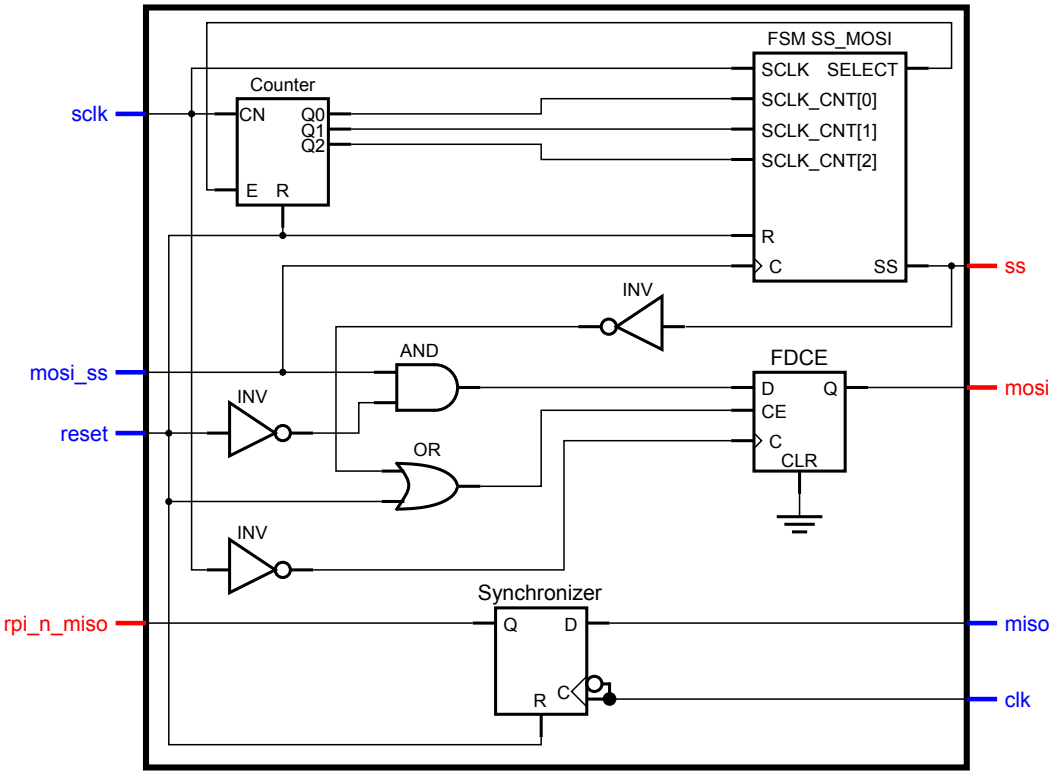


Abbildung 7.10: Digitaldesign der Signalrekonstruktion von $\overline{SS}$ und `MOSI`

In Abbildung 7.10 ist das Digitaldesign zur Signalrekonstruktion des Serial Peripheral Interfaces zu sehen. Auch diese Grafik wurde leicht vereinfacht aus dem Tech Viewer des Xilinx ISE Design Suites entnommen. Bei den rot gekennzeichneten Anschlüssen handelt es sich wieder um Ausgänge des CPLDs, während blaue Anschlüsse die Eingänge kennzeichnen.

Die Daten an `MOSI` werden mit jeder ⌝-Flanke von `SCLK` aktualisiert, wodurch nun der ursprüngliche zeitliche Zusammenhang zwischen `MOSI` und `SCLK` wieder gegeben ist. Die Daten an `MISO` werden wie bereits erwähnt mit dem Taktsignal des nachfolgenden Serializers synchronisiert und schließlich über den Upchannel übertragen.

7.5 Timing der rekonstruierten SPI Schnittstelle

Abbildung 7.11 zeigt das Timing-Diagramm der rekonstruierten Signale des Serial Peripheral Interfaces beim Peripheriegerät. Tabelle 7.6 gibt Auskunft über die im Timing-Diagramm eingezeichneten Zeitparameter, wobei anzumerken ist, dass der minimal und maximal Wert von t_7 nur für eine Serializer-Taktfrequenz von 20 MHz gültig ist. Die Parameter t_3 und t_5 hingegen sind nur für eine Frequenz von 5 MHz des Downchannels gültig.

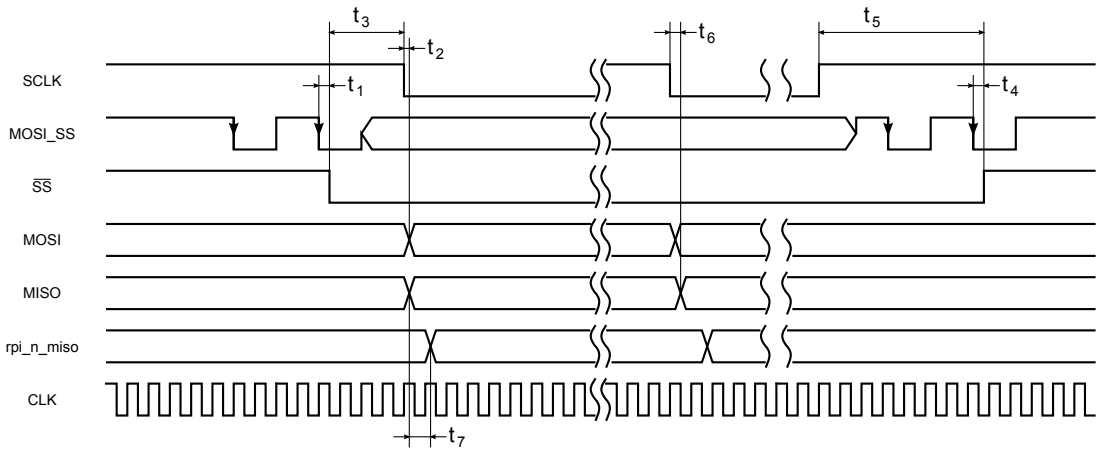


Abbildung 7.11: Timing-Diagramm der rekonstruierten SPI Schnittstelle beim Peripheriegerät

Parameter	**Symbol**	**Min.**	**Max.**
Zeitverzögerung zwischen zweiter ⌝-Flanke des Select-Kommandos in `MOSI_SS` und ⌝-Flanke von $\overline{\text{SS}}$	t_1		6 ns
`MOSI` Setup-Zeit	t_2		6 ns
$\overline{\text{SS}}$ Setup-Zeit (vgl. t_{SSS} in 7.3)	t_3	200 ns	400 ns

Fortsetzung auf nächster Seite

Tabelle 7.6 – *fortgesetzt von vorheriger Seite*

Parameter	**Symbol**	**Min.**	**Max.**
Zeitverzögerung zwischen letzter ↘-Flanke des Deselect-Kommandos in MOSI_SS und ↗-Flanke von $\overline{SS}$	t_4	6 ns	
$\overline{SS}$ Hold-Zeit (vgl. t_{SSH} in 7.3)	t_5	806 ns	1006 ns
MISO Setup-Zeit	t_6	?	
Verzögerung von MISO durch das CPLD	t_7	31 ns	56 ns

Tabelle 7.6: Beschreibung der Zeitparameter aus Abbildung 7.11

7.5.1 Ermittlung der Timing-Parameter

Der Zeitparameter t_1 entspricht der Verzögerung zwischen der Detektion der zweiten ↘-Flanke des Select-Kommandos und dem daraus folgendem ↘-Übergang an $\overline{SS}$. Das Slave-Select Signal wird von der FSM, welche mit jeder Flanke an MOSI_$\overline{SS}$ getriggert wird, erzeugt und somit ergibt sich t_1 zu:

$$\begin{aligned} t_1 &= t_{GCK} + t_{COI} + t_{OUT} \\ t_1 &= 2.7\,\text{ns} + 0.7\,\text{ns} + 2.6\,\text{ns} \\ t_1 &= 6\,ns \end{aligned}$$

Das in MOSI_$\overline{SS}$ eingebettete MOSI Signal wird mit jeder ↘-Flanke an SCLK ausgelesen, wodurch sich auch t_2 mit 6 ns ergibt.

Die Salve-Select Setup-Zeit ergibt sich bei der rekonstruierten SPI-Schnittselle am Peripheriegerät je nach Wert von t_4 aus Tabelle 7.5, zu $t_{3,\text{min}} = 200\,\text{ns}$ oder $t_{3,\text{max}} = 400\,\text{ns}$.

Auch die Verzögerung zwischen der letzten ↘-Flanke eines Deselect-Kommandos und dem tatsächlichen deselektieren des SPI-Slaves durch das von der FSM erzeugte Slave-Select Signal ergibt sich zu $t_4 = 6\,\text{ns}$.

Die Slave-Select Hold-Zeit ergibt sich je nach Wert von t_6 aus Tabelle 7.5 zu:

$$\begin{aligned} t_{5,min} &= T_{PIXCLK} + t_{6,min} + t_{GCK} + t_{COI} + t_{OUT} \\ t_{5,min} &= 200\,\text{ns} + 600\,\text{ns} + 2.7\,\text{ns} + 0.7\,\text{ns} + 2.6\,\text{ns} \\ t_{5,min} &= 806\,\text{ns} \end{aligned}$$

$$\begin{aligned} t_{5,max} &= T_{PIXCLK} + t_{6,max} + t_{GCK} + t_{COI} + t_{OUT} \\ t_{5,max} &= 200\,\text{ns} + 800\,\text{ns} + 2.7\,\text{ns} + 0.7\,\text{ns} + 2.6\,\text{ns} \\ t_{5,max} &= 1006\,\text{ns} \end{aligned}$$

Bei der verwendeten SPI-Konfiguration aktualisiert der SPI-Slave bei jeder ↳-Flanke an `SCLK` die Daten an `MISO`. Diese Daten werden vor der Übertragung über den Upchannel mit dessen Taktsignal synchronisiert und es kann wieder zu Setup-Zeit Verletzungen bei der Synchronizer-Struktur kommen. Je nach Auflösung eines eventuell aufgetretenen metastabilen Zustands ergibt sich für den Zeitparameter t_7 einer der folgenden Werte:

$$\begin{aligned} t_{7,min} &= 0.5 \cdot T_{UC\_CLK} + t_{GCK} + t_{COI} + t_{OUT} \\ t_{7,min} &= 0.5 \cdot 50\,\text{ns} + 2.7\,\text{ns} + 0.7\,\text{ns} + 2.6\,\text{ns} \\ t_{7,min} &= 31\,\text{ns} \end{aligned}$$

$$\begin{aligned} t_{7,max} &= 1 \cdot T_{UC\_CLK} + t_{GCK} + t_{COI} + t_{OUT} \\ t_{7,max} &= 1 \cdot 50\,\text{ns} + 2.7\,\text{ns} + 0.7\,\text{ns} + 2.6\,\text{ns} \\ t_{7,max} &= 56\,\text{ns} \end{aligned}$$

Die Summer der Zeitparameter t_6 und $t_{7,\text{max}}$ entspricht schließlich dem noch unbekannten Zeitparameter $t_{?2}$ und es kann nun die maximal zulässige Setup-Zeit von `MISO` des SPI-Slaves bestimmt werden.

$$\begin{aligned} 532.4\,\text{ns} &\geq t_6 + t_{7,max} \\ 532.4\,\text{ns} &\geq t_6 + 56\,\text{ns} \\ 476.4\,\text{ns} &\geq t_6 \end{aligned}$$

Das bedeutet, dass der SPI-Slave spätestens 476.4 ns nach einem ↳-Übergang von `SCLK` gültige Daten an `MISO` zur Verfügung stellen muss, sodass eine funktionierende Datenverbindung über den RPI-Bus besteht. Da die Setup-Zeit von `MISO` des SPI-Slaves jedoch im 10 ns-Bereich angenommen werden kann, steht auch noch ausreichend viel Zeit für eventuell unberücksichtigte Verzögerungszeiten – wie z.B. die Signallaufzeitverzögerung auf den Leiterbahnen – zur Verfügung.

8 Audioübertragung über den RPI-Bus

Die Audioübertragung soll beim RPI-Bus über I²S Interfaces stattfinden, wobei jeweils eine I²S Schnittstelle für die Audioausgabe und eine für die Audioeingabe benötigt wird. Wie bereits im Kapitel 3 festgehalten wurde, wird der i.MX31 sowohl für die Audioeingabe, als auch für die Audioausgabe im Slave-Modus betrieben, wodurch ein am RPI-Bus angeschlossenes Peripheriegerät zwei I²S Master zur Audioübertragung aufweisen muss.

MAXIM bietet beispielsweise die integrierten Schaltkreise MAX9851 und MAX9853 an, wobei es sich um Stereo Audiocodecs handelt, die unter anderem den Betrieb eines Headsets ermöglichen und zwei I²S Schnittstellen zur digitalen Audioübertragung aufweisen (vgl. [36], S.1ff). Es werden nun Betrachtungen angestellt, wie mit Hilfe des MAX9851 die Audioübertragung durchgeführt werden kann.

8.1 Timing der I²S Schnittstellen des MAX9851

In diesem Abschnitt wird das Timing der I²S Schnittstellen des Audiocodecs am Peripheriegerät betrachtet. Abbildung 8.1 zeigt das Timing-Diagramm und in Tabelle 8.1 werden die Größenordnungen der im Timing eingezeichneten Zeitparameter angegeben.

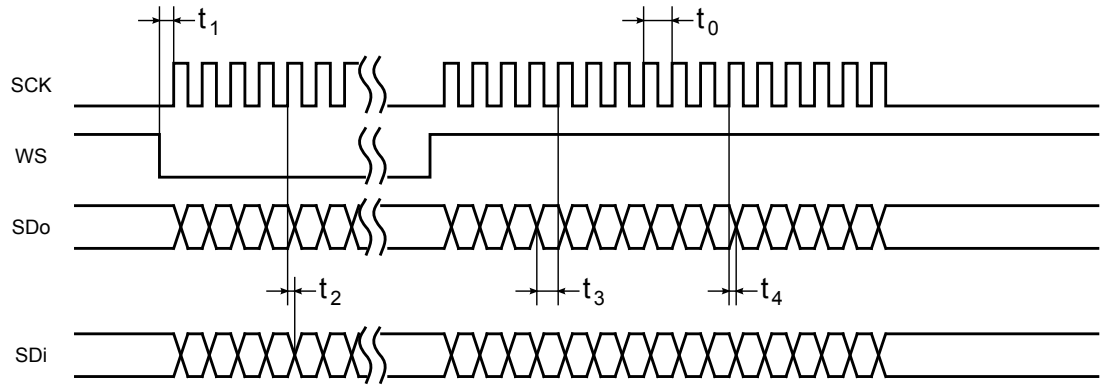


Abbildung 8.1: Timing-Diagramm der I²S Schnittstellen des MAX9851 im Master Modus (vgl. [36], S.33)

Bei den in Abbildung 8.1 dargestellten Signalen SCK, WS und SDi handelt es sich um Ausgangssignale des MAX9851. Das Signal SDo ist hingegen das Audioausgabesignal des i.MX31, das vom Audiocodec eingelesen werden muss und in weiterer Folge von diesem wieder in ein analoges Signal umgewandelt wird.

Parameter	**Symbol**	**Min.**	**Max.**
Periodendauer von SCK	t_0	310 ns	
WS vor SCK	t_1	30 ns	—
SDi Setup-Zeit	t_2	—	35 ns
SDo Setup-Zeit	t_3	30 ns	—
SDo Hold-Zeit	t_4	5 ns	—

Tabelle 8.1: I²S Timing-Parameter aus Abbildung 8.1 (vgl. [36], S.14)

Zur Audioausgabe am Peripheriegerät muss sichergestellt werden, dass die digitalen Audiodaten des i.MX31 spätestens 30 ns vor einer ⌋⌈-Flanke von SCK gültig sein müssen und ihren Zustand für weitere 5 ns beibehalten. Zusätzlich garantiert der MAX9851, dass spätestens 35 ns nach einem ⌋⌈-Übergang an SCK die digitalen Audioeingangsdaten gültig sind.

8.2 Timing der SSI Schnittstellen des i.MX31

In erster Linie muss sichergestellt werden, dass die Timing-Anforderungen der Synchronous Serial Interfaces des i.MX31 nicht verletzt werden. Abbildung 8.2 zeigt das Timing der SSIs des i.MX31 im I²S Modus, wobei die Signale SCK, WS und SDi Eingangssignale sind, SDo jedoch ein Ausgangssignal. In Tabelle 8.2 sind die im Timing eingezeichneten Zeitparameter beschrieben und es werden wiederum deren Größenordnungen angegeben.

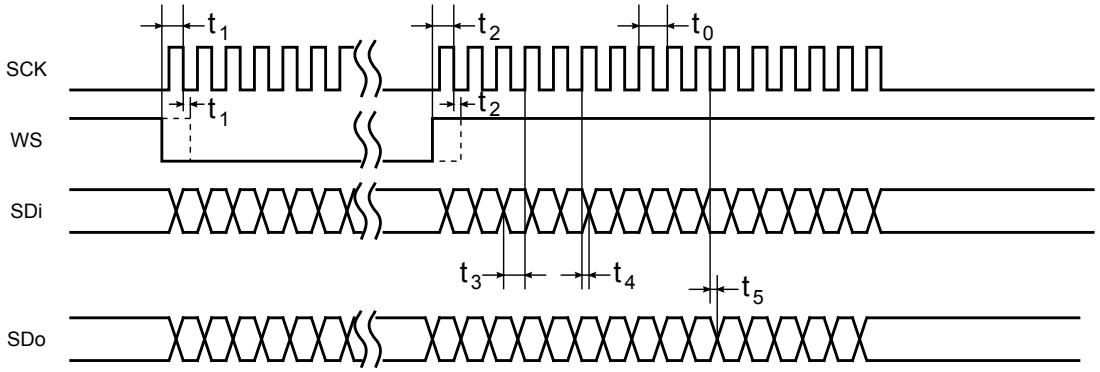


Abbildung 8.2: Timing-Diagramm der SSI Schnittstellen des i.MX31 im I²S Modus (vgl. [16], S.101ff)

Parameter	Symbol	Min.	Max.
Periodendauer von SCK	t_0	81.4 ns	—
WS_L vor SCK_L	t_1	-10 ns	15 ns
WS_H vor SCK_L	t_2	-10 ns	15 ns
SDi Setup-Zeit	t_3	10 ns	—
SDi Hold-Zeit	t_4	2 ns	—
SDo Setup-Zeit	t_5	—	15 ns

Tabelle 8.2: SSI Timing-Parameter aus Abbildung 8.2 (vgl. [16], S.101ff)

Aus Abbildung 8.2 und Tabelle 8.2 geht hervor, dass das Signal WS frühestens 15 ns vor der ersten ⎿-Flanke eines Bursts an SCK aber spätestens 15 ns danach einen Pegelwechsel aufweisen muss. Die Audioeingangsdaten müssen mindestens 10 ns vor einer ⌋⌈-Flanke von SCK und für weitere 2 ns danach einen gültigen Wert aufweisen. Zusätzlich garantiert das SSI des i.MX31, dass spätestens 15 ns nach einem ⎿-Übergang an SCK die Audioausgangsdaten an SDo gültig sind.

8.3 Digitaldesign zur Audioeingabe

8.3.1 Peripheriegerät

Da die Signale der I^2S Schnittstellen des Audiocodecs nicht synchron zum Taktsignal des Serializers des Upchannel sind, müssen diese wieder mittels Synchronizer synchronisiert werden, sodass auch hier die Setup-Zeit des Input-Latches des Serializers nicht verletzt wird.

In Kapitel 7 wurde eine Mindestaktfrequenz von 20 MHz für die SerDes der Upchannels festgelegt. Zur Synchronisation der Audiodaten mit dem Taktsignal des Upchannels (UC_CLK) werden dieses Mal bei den Synchronizer-Strukturen jedoch keine Dual-Edge-Triggered Flipflops verwendet, sondern D-Flipflops, die mit einer ⎿-Flanke des Taktsignals des Upchannels getriggert werden, wodurch die synchronisierten Daten stets bei einem ⌋⌈-Übergang des besagten Taktsignals stabil sind. Abbildung 8.3 zeigt oben das Timing der des MAX9851 generierten Signale LRCLK (Links-/Rechts-Kanalinformation – vgl. WS), BCLK (Taktsignal der digitalen Audiodaten – vgl. SCK) und SDIN (digitale Audiodaten – vgl. SDi). In der Mitte sind die Verläufe der synchronisierten Signale zusehen, die schließlich über den Upchannel übertragen werden und unten sind die vom Deserializer des Upchannels rekonstruierten Signale dargestellt.

Es ist zu erkennen, dass beim Signal LRCLK bei der ersten ⎿-Flanke des Taktsignals UC_CLK die Setup-Zeit der entsprechenden Synchronizer-Struktur verletzt

wird. Dadurch kann es kurzzeitig zu einem metastabilen Zustand kommen und das synchronisierte Signal `WSi_sync` behält je nach Auflösung des metastabilen Zustandes für eine weitere Taktperiode von `UC_CLK` den alten Wert bei. Dadurch kann das Signal `rpi_n_WSi` dem Signal `rpi_n_SCKi` um eine ganze Taktperiode von `UC_CLK` voreilen, und somit können die Zeitparameter t_1 bzw. t_2 (siehe Tabelle 8.2) der SSI Schnittstellen des i.MX31 verletzt werden.

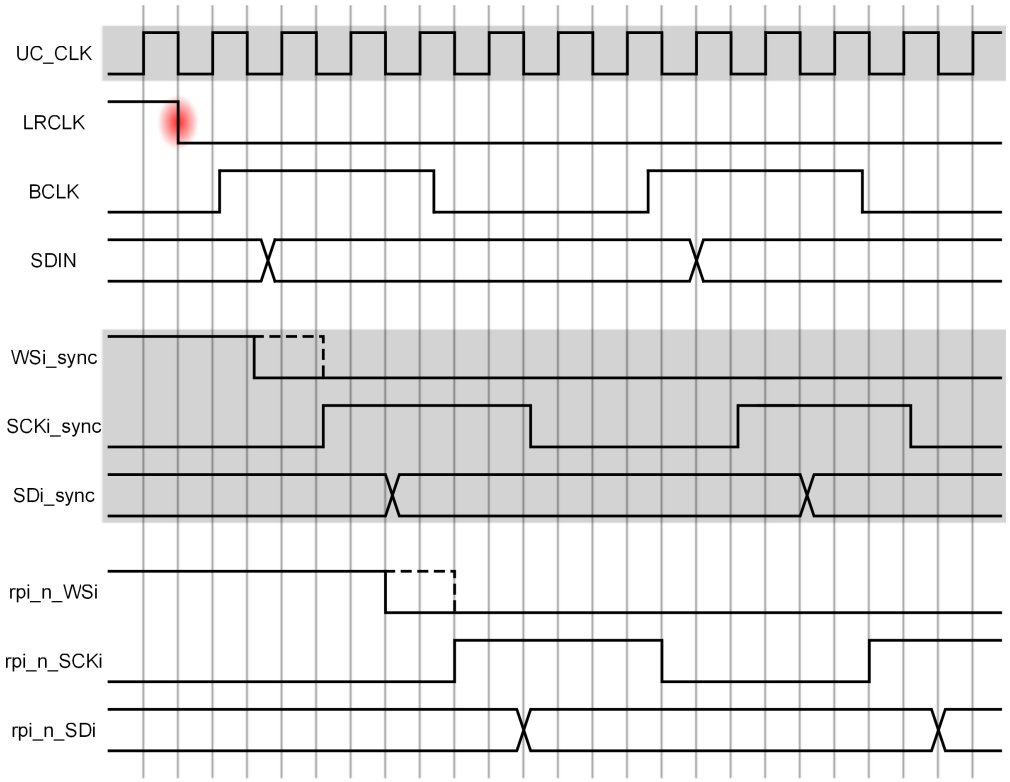


Abbildung 8.3: Gegenüberstellung der Signale des Audiocodecs, der synchronisierten Signale des CPLDs, sowie der Ausgangssignale eines Upchannel-Deserializers

8.3.2 RPI-Testboard

Beim RPI-Testboard wird mittels Multiplexer im CPLD eine der vier digitalen Audioschnittstellen ausgewählt und zu einem Serial Peripheral Interface des i.MX31 durchgeschleift. Da die Signale `rpi_n_WSi` und `rpi_n_SCKi` nach der Übertragung über den Upchannel entweder gleichzeitig den Pegel wechseln oder `rpi_n_WSi` sogar eine Taktperiode des Upchannel-Taktsignals dem Signal `rpi_n_SCKi` voreilt, dieses jedoch maximal 10 ns vor einer ⌞-Flanke an `rpi_n_SCKi` den Pegel wechseln darf, muss das Signal `rpi_n_WSi` entsprechend verzögert werden, sodass die Parameter t_1 bzw. t_2 aus Tabelle 8.2 nicht verletzt werden. Um dies zu gewährleisten wird das Signal `rpi_n_WSi` erneut mit dem Signal `rpi_n_SCKi` synchronisiert, indem das vom Multiplexer ausgewählte Signal `rpi_n_wsi` mit jeder ⌞-Flanke an `rpi_n_SCKi` abgetastet wird, wodurch sich nun ein definiertes Delay von 5.1 ns zwischen einem Pegelwechsel an `SCKi` und dem tatsächlichen Ausgangssignal `WSi` ergibt.

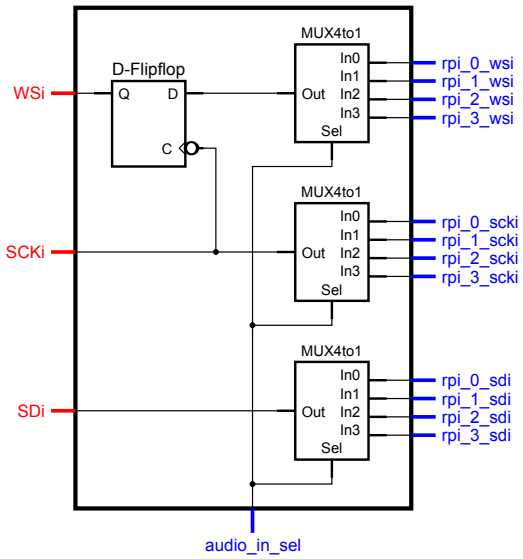


Abbildung 8.4: Digitaldesign zur Synchronisation der Audioeingabesignale nach der Übertragung über den Upchannel

Abbildung 8.4 zeigt das Digitaldesign zur Signalmodifikation der – über einen Upchannel übertragenen – Steuer- und Datensignale, sodass diese vom i.MX31 weiterverarbeitet werden können. Bei den blau eingefärbten Anschlüssen des Digitaldesigns handelt es sich wieder um Eingänge des CPLDs am RPI-Testboard, wobei rot eingefärbte Anschlüsse, Ausgänge des CPLDs repräsentieren.

8.4 Timing der Audioeingabesignale

Das Timing-Diagramm 8.5 zeigt die Verläufe der vom CPLD ausgegebenen Signale zur Audioeingabe, welche schließlich dem SSI des i.MX31 zugeführt werden. In Tabelle 8.3 sind die Größenordnungen der im Timing eingezeichneten Zeitparameter angeführt. Anzumerken ist hierbei, dass die Werte t_0, t_1, t_2, t_5 und t_6 von der Taktfrequenz des Upchannels abhängig sind und nur für eine Frequenz von 20 MHz gültig sind.

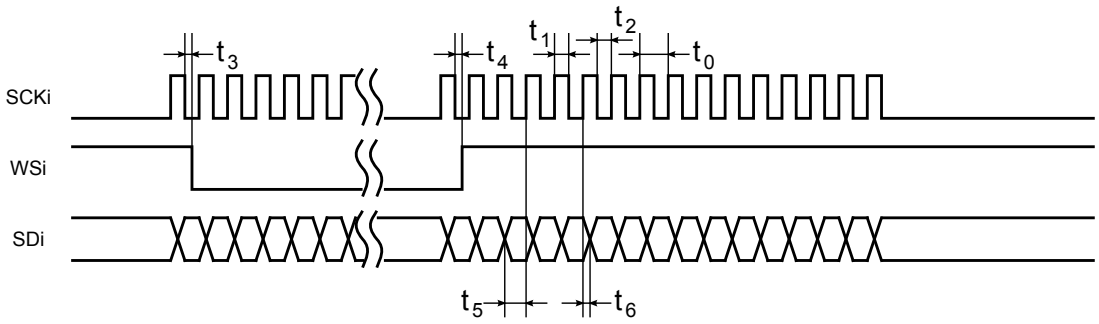


Abbildung 8.5: Timing-Diagramm der Audioeingabesignale nach der Übertragung über den Upchannel und der erneuten Synchronisation durch das CPLD am RPI-Testboard

Parameter	Symbol	Min.	Max.
Periodendauer von SCKi	t_0	300 ns	350 ns
High-Dauer von SCKi	t_1	150 ns	200 ns
Low-Dauer von SCKi	t_2	150 ns	200 ns
WSi_L vor $SCKi_L$	t_3	—	5.1 ns
WSi_H vor $SCKi_L$	t_4	—	5.1 ns
SDi Setup-Zeit	t_5	0 ns	50 ns
SDi Hold-Zeit	t_6	100 ns	200 ns

Tabelle 8.3: Beschreibung der Zeitparameter aus Abbildung 8.5

8.4.1 Ermittlung der Timing-Parameter

Da das Taktsignal BCLKi beim Peripheriegerät mit dem Taktsignal des Upchannels synchronisiert wird, und es dadurch zu Setup-Zeit Verletzungen bei der Synchronizer-Struktur kommen kann, weist das Audioeingabe-Taktsignal SCKi am RPI-Testboard einen Jitter von 50 ns auf. Dieser Jitter entsteht dadurch, da beim entsprechenden Synchronizer am Peripheriegerät oftmals die Setup-Zeit verletzt wird und somit gegebenenfalls metastabile Zustände auftreten. Dadurch kann der aktuelle Wert des synchronisierten Signals SCKI_sync für eine weitere Taktperiode von UC_CLK beibehalten, oder auch sofort aktualisiert werden. Somit ergibt sich die maximale Low- bzw. High-Dauer von SCKi mit:

$$\begin{aligned} t_{1,max} = t_{2,max} &= 4 \cdot T_{UC\_CLK} \\ t_{1,max} = t_{2,max} &= 4 \cdot 50\,\text{ns} \\ t_{1,max} = t_{2,max} &= 200\,\text{ns} \end{aligned}$$

Wird die Setup-Zeit der Synchronizer-Struktur zum Synchronisieren von BCLKi jedoch nicht verletzt, oder werden die eventuell aufgetretenen metastabilen Zustände bei einer Verletzung der Setup-Zeit bereits mit dem neuen Wert von BCLKi aufgelöst, so ergibt sich der minimale Wert für die Low- bzw. High-Dauer von SCKi zu:

$$\begin{aligned} t_{1,min} = t_{2,min} &= 3 \cdot T_{UC\_CLK} \\ t_{1,min} = t_{2,min} &= 3 \cdot 50\,\text{ns} \\ t_{1,min} = t_{2,min} &= 150\,\text{ns} \end{aligned}$$

Für die Taktperiode von SCKi (t_0) ergibt sich somit typischerweise ein Wert von 300 ns, jedoch ein Maximalwert von 350 ns.

Die Setup-Zeit von WSi (t_3 und t_4) ergibt sich durch die erneute Synchronisation mit dem Signal SCKi mit 5.1 ns und errechnet sich folgendermaßen:

$$\begin{aligned} t_3 = t_4 &= t_F + t_{CT} + t_{COI} \\ t_3 = t_4 &= 3\,\text{ns} + 1.4\,\text{ns} + 0.7\,\text{ns} \\ t_3 = t_4 &= 5.1\,\text{ns} \end{aligned}$$

Beim Parameter t_{CT} handelt es sich um das Control Term Delay des CPLDs und wurde aus [35] entnommen.

Durch die Synchronisation der digitalen Audioausgabesignale kann die Bitgrenze der über einen Upchannel übertragenen Audioausgabesignale – je nach Phasenlage der Signale BCLKi und SDIN zu UC_CLK – gleichzeitig mit einer ⌋⌈-Flanke von rpi_n_SCKi oder eine Taktperiode von UC_CLK danach, auftreten. Die maximale und minimale Setup-Zeit für SDi ergibt sich somit zu:

$$\begin{aligned} t_{5,max} &= 1 \cdot T_{UC\_CLK} \\ t_{5,max} &= 1 \cdot 50\,\text{ns} \\ t_{5,max} &= 50\,\text{ns} \end{aligned}$$

$$\begin{aligned} t_{5,min} &= 0 \cdot T_{UC\_CLK} \\ t_{5,min} &= 0 \cdot 50\,\text{ns} \\ t_{5,min} &= 0\,\text{ns} \end{aligned}$$

Die maximale und minimale Hold-Zeit des Signals SDi lässt sich nun mittels der maximalen und minimalen Setup-Zeit von SDi, sowie den maximalen und minimalen Low-Zeiten von SCKi berechnen:

$$\begin{aligned} t_{6,max} &= t_{2,max} - t_{5,min} \\ t_{5,max} &= 200\,\text{ns} - 0\,\text{ns} \\ t_{5,max} &= 200\,\text{ns} \end{aligned}$$

$$\begin{aligned} t_{6,min} &= t_{2,min} - t_{5,max} \\ t_{6,min} &= 150\,\text{ns} - 50\,\text{ns} \\ t_{6,min} &= 100\,\text{ns} \end{aligned}$$

8.5 Digitaldesign zur Audioausgabe

Die Steuersignale WSo und SCKo zur Audioausgabe – die vom Audiocodec am Peripheriegerät generiert werden – werden identisch wie die Steuersignale zur Audioeingabe synchronisiert und über den Upchannel übertragen. Im folgenden Abschnitt wird auf das Digitaldesign zur Audioausgabe am RPI-Testboard eingegangen.

8.5.1 RPI-Testboard

Da die Mindesttaktfrequenz des Downchannels mit 5 MHz festgelegt wurde und die digitalen Audiodaten eine ähnlich hohe Frequenz aufweisen, muss besonders großes Augenmerk auf die Synchronisation der Audioausgabesignale – mit dem Taktsignal des Downchannels – gelegt werden.

Mittels Multiplexer werden wieder die Steuersignale zur Audioausgabe von einem der vier Upchannels ausgewählt. Vom ausgewählten Taktsignal `scko_int` – zur Audioausgabe – werden zu Beginn die ⌝⌞-Übergänge gezählt, wobei dieser Zähler bei jedem High-Pegel am ausgewählten Links-/Rechts-Kanal Signal `wso_int` zurückgesetzt wird, bis während eines Low-Pegels an `wso_int` 16 ⌝⌞-Übergänge an `scko_int` detektiert werden und dadurch der Zähler gleichzeitig deaktiviert wird. Das MSB dieses Zählers (`enable_sdo`) signalisiert nun, dass mit der Audioübertragung über den Downchannel begonnen werden kann und auch der Eingang zum Einlesen der digitalen Audioausgabedaten wird erst jetzt aktiviert. Durch diesen Initialisierungsvorgang gehen jedoch bis zu zwei Audiosamples verloren, die zum Peripheriegerät übertragen werden sollen.

Das ausgewählte Taktsignal `scko_int` wird vom Modul `wso_scko_synchronizer` mit einer 3-stufigen Synchronizer-Struktur erneut mit dem Taktsignal des Upchannels synchronisiert, während das `wso_int` Signal mittels einer 2-stufigen Synchronizer-Struktur einerseits mit dem Taktsignal des Upchannels (`wso_uc_sync`) und andererseits mit dem Taktsignal des Downchannels (`wso_dc_sync`) synchronisiert wird.

Das synchronisierte Taktsignal zur Audioausgabe `SCKo` entspricht dem tatsächlichen Signal, welches dem i.MX31 zugeführt wird. Die Signale `wso_uc_sync` und `wso_dc_sync` werden zur Synchronisation der zu übertragenden Audioausgangsdaten benötigt. Zusätzlich wird das Signal `wso_uc_sync` mit jeder ⌝⌞-Flanke von `SCKo` abgetastet, sodass die Timing-Anforderungen des SSI vom i.MX31 erfüllt werden können.

Vier Bufferregister übernehmen die Synchronisation der Audioausgangsdaten mit dem Taktsignal des Downchannels. Diese Register weisen eine Breite von 20 Bit auf und werden einerseits mit dem Taktsignal `SCKo` befüllt und andererseits mit dem Taktsignal des Downchannels ausgelesen. Zusätzlich werden diese Register vor jedem neuen Befüllen mit einem Initialwert von `0x0000F` geladen, wodurch nach dem Ladevorgang, das entsprechende Register mit dem digitalen Audiosample und vier vorstehenden High-Bits geladen ist. Mittels diesen vier High-Bits kann schließlich beim Peripheriegerät der Beginn eines neuen Audiosamples detektiert werden.

Die Auswahl des Registers, welches geladen werden soll, welches ausgelesen werden soll, sowie welches mit dem Initialwert geladen werden soll, wird durch das Modul `sdo_buffer_controller` vorgenommen. Dieses Modul steuert einerseits den Demultiplexer, der die digitalen Audioausgangsdaten zu einem der Bufferregister

durchschleift, sowie den Multiplexer, der eines der Bufferregister zum Auslesen auswählt. Auch die Taktsignale und die Signale zum Initialisieren der Register werden von diesem Modul erzeugt.

Die ausgelesenen Daten der Register werden schließlich mit jeder ⅂-Flanke des Downchannel-Taktsignals abgetastet, sodass die Setup-Zeit des Input-Latches des nachfolgenden Serializers nicht verletzt wird.

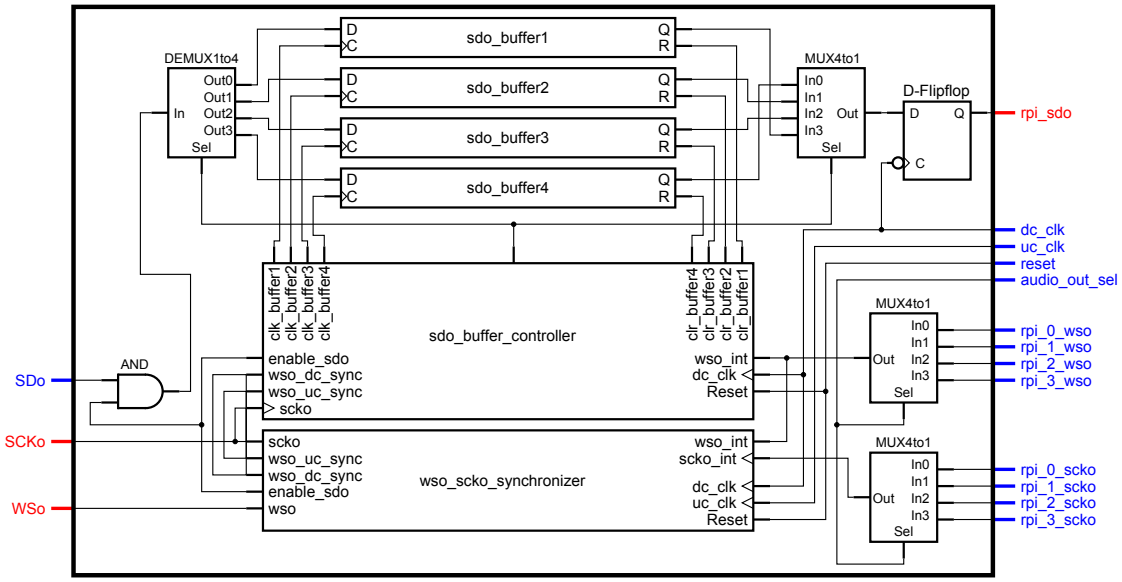


Abbildung 8.6: Digitaldesign zur Synchronisation der Audioausgangsdaten zur Übertragung über den Downchannel

Abbildung 8.6 zeigt das Digitaldesign zur Synchronisation der digitalen Audioausgangsdaten mit dem Taktsignal des Downchannels und wurde wieder leicht vereinfacht aus dem Tech Viewer des Xilinx ISE Design Suites übernommen. Rot gekennzeichnete Anschlüsse im Digitaldesigns entsprechen wieder Ausgängen des CPLDs, während blaue Anschlüsse die erforderlichen Eingänge repräsentieren.

Das Modul `sdo_buffer_controller` beinhaltet einen Zähler, der die Pegelwechsel an `wso_uc_sync` mitzählt, sobald das Signal `enable_sdo` einen High-Pegel aufweist. Bei diesem Zähler handelt es sich um einen zwei Bit breiten Zähler, dessen Zählerstand zum Selektieren des Bufferregister – welches ein- bzw. ausgelesen werden soll – dient.

Da das Signal `SCKo` aufgrund der verwendeten Synchronizer dem Signal `wso_uc_sync` um mindestens einen Taktzyklus des Upchannel-Taktsignals nacheilt, erfolgt die Auswahl eines Bufferregisters stets vor dem Anliegen eines Taktsignals an `SCKo`. Das ausgewählte Bufferregister wird schließlich mit dem Taktsignal `SCKo` geladen und sobald ein Pegelwechsel an `wso_uc_sync` auftritt und zusätzlich das Signal `wso_dc_sync` seinen Wert geändert hat, wird das Bufferregister mit dem Taktsignal des Downchannels ausgelesen. Gleichzeitig werden die ⅃-Übergänge des Downchannel-Taktsignals mitgezählt, wobei dieser Zähler nach der zwanzigsten

⌋-Flanke gestoppt wird. Dieses Disable-Signal – generiert aus dem aktuellen Zählerstand – wird durch ein D-Flipflop, das mit jeder ⌊-Flanke des Downchannel-Taktsignals getriggert wird, verzögert und dient schlussendlich dazu, dass das Bufferregister nicht mehr getaktet wird und in weiterer Folge keine ungültigen Werte ausgelesen werden.

Für jedes Bufferregister wird also ein Zähler zum Zählen der Taktflanken vom Downchannel-Taktsignal benötigt, sowie ein Filipflop zum verzögern des vorhin erwähnten Disable-Signals. Zusätzlich wird für jedes der vier Bufferregister ein Multiplexer zum Umschalten zwischen `SCKo` und dem Downchannel-Taktsignal benötigt. Diese Multiplexer werden mit dem Signal `wso_uc_sync` gesteuert.

Um sicherzustellen, dass die Bufferregister immer nur im ausgewählten Zustand beschrieben bzw. gelesen werden, werden die Taktsignale mittels zusätzlicher kombinatorischer Logik aktiviert bzw. deaktiviert. Auch die Reset-Signale für die Bufferregister, welche ein Laden der Register mit dem Initialwert `0x0000F` veranlassen, werden mittels kombinatorischen Verknüpfungen der Steuersignale generiert.

Abbildung 8.7 zeigt das Digitaldesign des Moduls `sdo_buffer_controller`, wobei auch hier die Eingänge des Moduls mittels blauen Anschlüssen und die Ausgänge durch rote Anschlüsse gekennzeichnet sind.

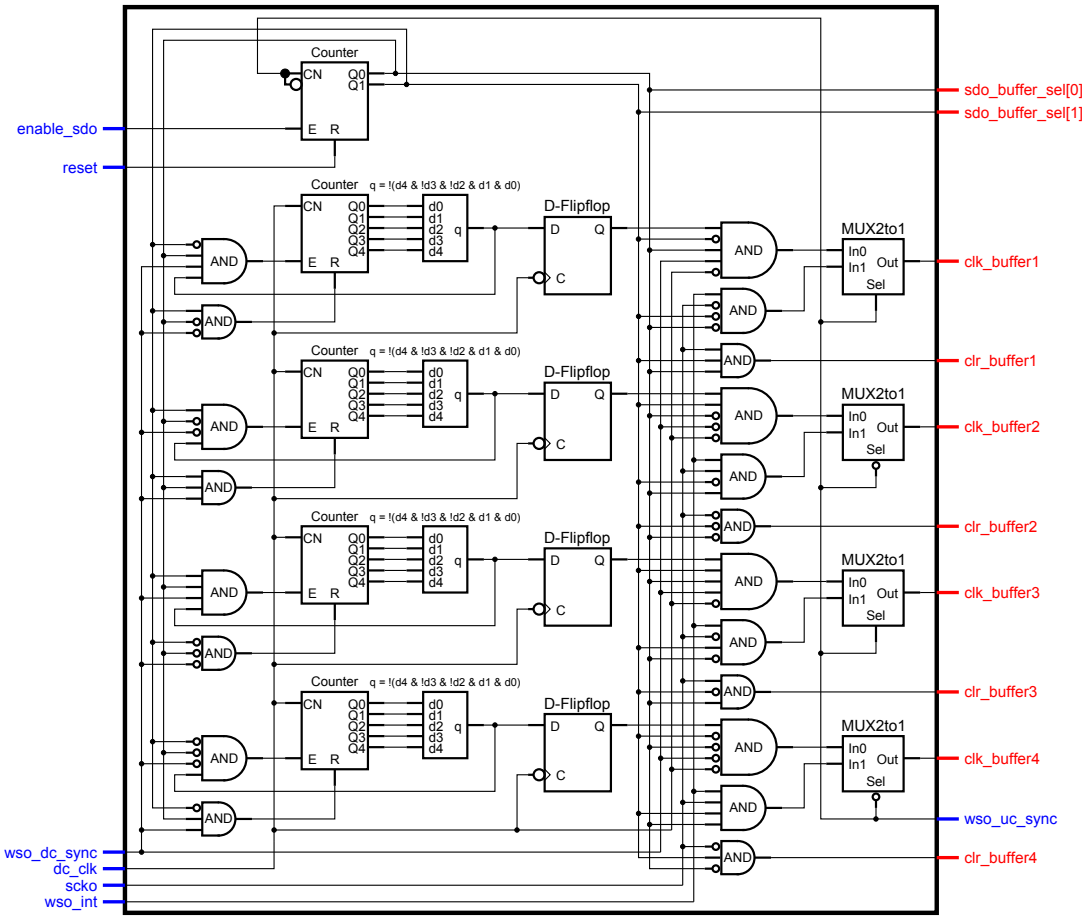


Abbildung 8.7: Digitaldesign des Moduls `sdo_buffer_controller`

8.5.2 Peripheriegerät

Die über den Downchannel übertragenen Audiodaten sind synchron zum Downchannel-Taktsignal und müssen nun mit denen vom MAX9851 generierten Signalen `BCLKo` und `LRCLKo` synchronisiert werden, sodass dieser die digitalen Audiodaten wieder in ein korrektes Analogsignal umwandeln kann.

Da das Signal – zur Übertragung der Audiodaten über den Downchannel – im Ruhezustand einen Low-Pegel aufweist und jedes Audiosample mit vier vorangehenden High-Bits eingeleitet wird, wird im CPLD am Peripheriegerät ein Zähler implementiert, der seinen Zählerstand immer dann erhöht, sobald das über den Downchannel übertragene Audiosignal während einer ⎍-Flanke des Downchannel-Taktsignals (`dc_clk`) einen High-Pegel aufweist.

Sobald dieser Zähler (`tx_start_counter`) während drei hintereinander folgenden ⎍-Übergängen an `dc_clk` seinen Zählerstand auf `11b` erhöht hat, wird ein weiterer Zähler (`tx_counter`) gestartet, der ebenfalls mit jeder ⎍-Flanke an `dc_clk` inkrementiert wird. Erreicht `tx_counter` den Wert `1111b` so ist das aktuelle Audiosample zu Ende und mittels `tx_start_counter` kann das nächste Audiosample detektiert werden.

Ein weiterer Zähler (`dc_counter`), der auch mit jeder ⎍-Flanke von `dc_clk` inkrementiert wird, ermöglicht das Verwerfen von Audiosamples, bei denen während der Initialbits ein Übertragungsfehler aufgetreten ist. Diese drei Zähler werden von einem separaten Modul (`counter_controller`) gesteuert, dessen FSM in Abbildung 8.8 abgebildet ist.

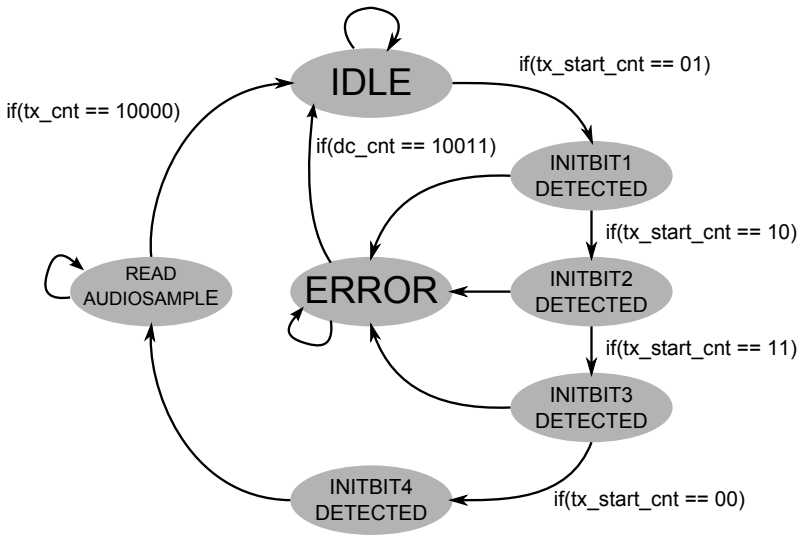


Abbildung 8.8: Zustandsübergangsdiagramm der FSM zur Erzeugung der Steuersignale für die Zähler `tx_start_counter`, `tx_counter` und `dc_counter`

Die FSM des Moduls `counter_controller` wird mit jedem ⎿-Übergang an `dc_clk` getriggert und befindet sich nach einem Reset im Zustand `IDLE`. In diesem Zustand ist nur der Zähler `tx_start_counter` aktiviert, und die FSM wartet im selbigen, bis während einer ⎍-Flanke an `dc_clk` ein High-Bit bei den digitalen Audioausgangsdaten detektiert wird und die FSM in den Zustand `INITBIT1 DETECTED` wechselt.

In diesem Zustand wird der Zähler `tx_counter` zurückgesetzt und der Zähler `dc_counter` wird aktiviert. Wird nun mit der nächsten ⎍-Flanke an `dc_clk` wieder ein High-Bit detektiert, so wechselt die FSM in den Zustand `INITBIT2 DETECTED`. Tritt jedoch ein Low-Bit während der nächsten ⎍-Flanke an `dc_clk` auf, so sind bei der Übertragung über den Downchannel Fehler aufgetreten und die FSM geht für die restliche Dauer des Audiosamples in den Zustand `ERROR`. In diesem Zustand ist nur der Zähler `dc_counter` aktiv, während der Zähler `tx_start_counter` zurückgesetzt wird. Sobald 19 weitere Taktzyklen vergangen sind, geht die FSM wieder vom Zustand `ERROR` in den Zustand `IDLE` über und es kann das nächste Audiosample eingelesen werden.

Wird das Audiosample jedoch Fehlerlos übertragen, so wird bei der nächsten ⎍-Flanke an `dc_clk` – sofern das Audioausgangssignal einen High-Pegel aufweist – vom Zustand `INITBIT2 DETECTED` zum Zustand `INITBIT3 DETECTED` gewechselt. In diesem Zustand wird nun der Zähler `tx_counter` aktiviert und die FSM wechselt in den Zustand `INITBIT4 DETECTED` falls auch bei der folgenden ⎍-Flanke an `dc_clk` bei den Audioausgangsdaten ein High-Bit detektiert wurde. Im Zustand `INITBIT4 DETECTED` wird nun der Zähler `tx_start_counter` zurückgesetzt, der Zähler `tx_counter` bleibt hingegen aktiviert. Die FSM wechselt dann in den Zustand `READ AUDIOSAMPLE` in dem sie verharrt, bis das gesamte Audiosample übertragen wurde und anschließend wieder in den Zustand `IDLE` zurückgeht.

Die übertragenen Audiodaten werden einstweilen in einem von vier Bufferregister zwischengespeichert, wobei mittels eines Demultiplexers entschieden wird, in welches der vier Register das aktuelle Audiosample geschrieben werden soll. Diese Register werden ähnlich wie beim CPLD am RPI-Testboard mit dem Downchannel-Taktsignal befüllt und mit dem des MAX9851 generierten Taktsignals `BCLKo` ausgelesen. Mittels eines Multiplexers wird entschieden, welches der vier Bufferregister schließlich ausgelesen werden soll, wobei dieses Signal nochmals mit dem Signal `BCLKo` synchronisiert werden muss. Der Ausgang dieses Synchronizers entspricht nun dem tatsächlichen Eingangssignal des Audiocodecs, der dieses Signal wieder in ein analoges Audiosignal umwandelt.

Die Steuerung des Demultiplexers, sowie des Multiplexers und auch die Erzeugung der Taktsignale für die vier Bufferregister wird wieder von einem separaten Modul (`sdo_buffer_controller`) durchgeführt. Dieses Modul besteht im wesentlichen aus zwei FSMs, wobei eine für das Laden der Bufferregister mit dem Taktsignal des Downchannels zuständig ist und die andere für das Auslesen der Register mit dem Taktsignal des Audiocodecs verantwortlich ist.

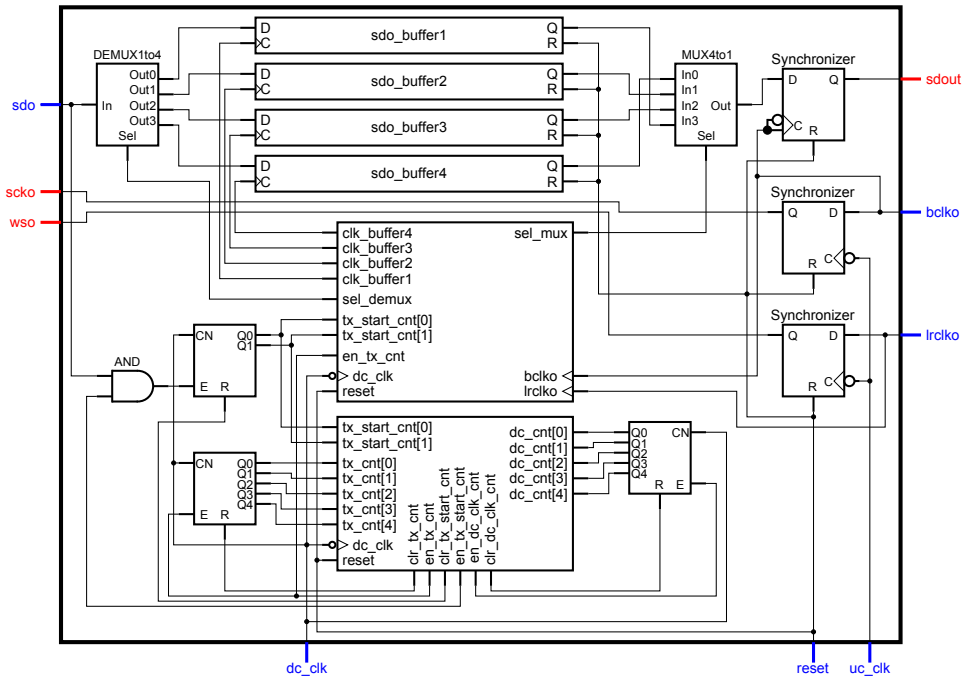


Abbildung 8.9: Digitaldesign zur Synchronisation der Audioausgangsdaten mit den Signalen des Audiocodecs MAX9851

Abbildung 8.9 zeigt das Digitaldesign zur Synchronisation der über den Downchannel übertragenen Audioausgangsdaten mit denen des MAX9851 generierten Steuersignalen. Blau gekennzeichnete Anschlüsse des Designs entsprechen wieder Eingängen des CPLDs am Peripheriegerät, während rote Anschlüsse die Ausgänge repräsentieren.

In den Abbildungen 8.10(a) und 8.10(b) sind die Zustandsübergangsdiagramme der vorhin erwähnten FSMs dargestellt, wobei sich links das Diagramm der FSM zum Laden der Bufferregister befindet und links das Diagramm der FSM zum Lesen der Register.

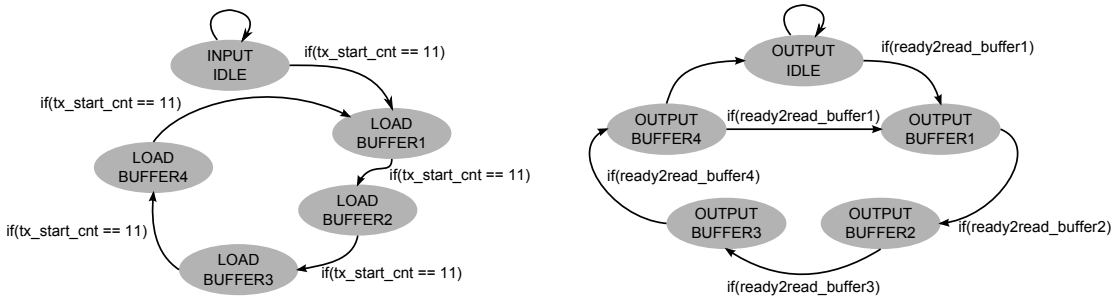


Abbildung 8.10: (a) zeigt das Zustandsübergangsdiagramm der FSM zum Einlesen der Bufferregister; (b) zeigt das Zustandsübergangsdiagramm der FSM zum Auslesen der Bufferregister

Die FSM – dargestellt in Abbildung 8.10(a) – wird mit jeder ⎿-Flanke an `dc_clk` getriggert und befindet sich nach einem Reset des CPLDs im Zustand `INPUT IDLE`. Sobald jedoch das erste Audiosample erkannt wurde, wechselt die FSM in den Zustand `LOAD BUFFER1` und das Downchannel-Taktsignal wird für das erste Bufferregister aktiviert. Dieses Taktsignal bleibt solange aktiv, bis der Zähler `tx_cnt` seinen Endwert erreicht hat und somit ein vollständiges Audiosample in das Bufferregister geladen wurde.

Die FSM verharrt nun solange im Zustand `LOAD BUFFER1` bis das nächste Audiosample detektiert wird und wechselt anschließend in den Zustand `LOAD BUFFER2`. In diesem Zustand wird nun das zweite Bufferregister zum Laden ausgewählt und auch das Downchannel-Taktsignal wird aktiviert. Ein weiteres Signal wird nun gesetzt, um der FSM zum Auslesen der Audiodaten mitzuteilen, dass das erste Bufferregister bereit ist um ausgelesen zu werden. Das Downchannel-Taktsignal für das zweite Bufferregister bleibt wieder solange aktiviert, bis der Zähler `tx_cnt` seinen Endwert erreicht hat.

Mit jeder weiteren Detektion eines neuen Audiosamples wechselt die FSM somit von Zustand zu Zustand, wobei immer ein Register neu geladen wird und der anderen FSM mitgeteilt wird, dass das im vorherigen Zustand geladene Register ausgelesen werden kann.

Die FSM – dargestellt in Abbildung 8.10(b) – wird mit jedem Pegelwechsel an `LRCLKo` getriggert und befindet sich nach einem Reset im Zustand `OUTPUT IDLE`. Sobald jedoch das Ready-Signal zum auslesen des ersten Bufferregister kommt, wechselt die FSM in den Zustand `OUTPUT BUFFER1`. Nun wird das Taktsignal `BCLKo` zum Auslesen des ersten Registers aktiviert und bleibt auch bis zum nächsten Zustandswechsel der FSM aktiv.

Liegt beim nächsten Pegelwechsel an `LRCLKo` im zweiten Bufferregister ein gültiges Audiosample vor, so wird das Taktsignal `BCLKo` im Zustand `OUTPUT BUFFER2` für das erste Register deaktiviert und gleichzeitig für das zweite Register aktiviert. Ist jedoch kein gültiges Audiosample im zweiten Bufferregister vorhanden, so geht die FSM in den Zustand `OUTPUT IDLE` zurück und wartet in diesem, bis das entsprechende Bufferregister mit korrekten Daten geladen ist.

Im fehlerlosen Fall rotiert die FSM schließlich in der Reihenfolge `OUTPUT BUFFER1` ⇒ `OUTPUT BUFFER2` ⇒ `OUTPUT BUFFER3` ⇒ `OUTPUT BUFFER4` ⇒ `OUTPUT BUFFER1`.

Da die Ausgangsdaten der Bufferregister immer mit ⎾-Flanken an `BCLKo` ausgelesen werden und diese aufgrund der Verzögerungszeiten des CPLDs erst kurz nach den ⎾-Übergängen gültig sind, müssen die Ausgangsdaten nochmals mit dem Taktsignal `BCLKo` synchronisiert werden, sodass die Timing-Anforderungen des Audiocodecs MAX9851 (vgl. t_3 in Tabelle 8.1) erfüllt werden können.

Dazu werden die aus den Bufferregistern ausgelesenen Audiosamples um ein vollständiges Sample (16 Taktzyklen an `BCLKo`) verzögert und schlussendlich mit jeder ⎿-Flanke an `BCLKo` vom CPLD ausgegeben.

8.6 Timing der Audioausgabesignale

Das Timing-Diagramm 8.11 zeigt die zeitlichen Zusammenhänge zwischen denen des MAX9851 generierten Signalen `BCLKo` und `LRCLKo` und den endgültigen Audioausgangsdaten, die über den Downchannel übertragen und mit Hilfe des CPLDs synchronisiert wurden. In Tabelle 8.4 werden wieder die Größenordnungen der im Timing eingezeichneten Zeitparameter angegeben.

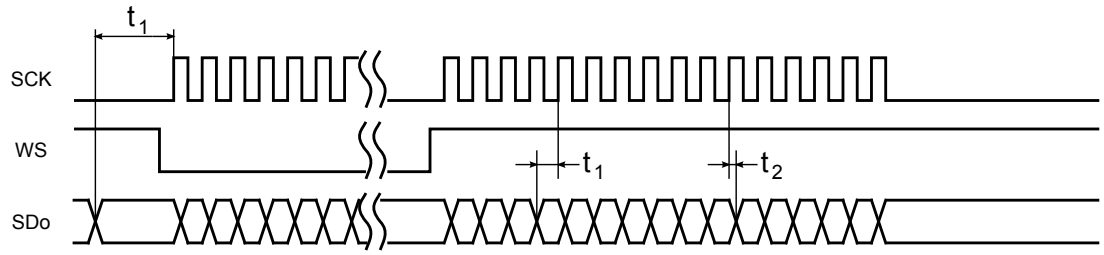


Abbildung 8.11: Timing-Diagramm der Audioausgabesignale nach der Übertragung über den Downchannel und der erneuten Synchronisation durch das CPLD am Peripheriegerät

Parameter	**Symbol**	**Min.**	**Max.**
`SDOUT` Setup-Zeit	t_1	149 ns	397 ns
`SDOUT` Hold-Zeit	t_3	161 ns	—

Tabelle 8.4: Beschreibung der Zeitparameter aus Abbildung 8.11

Vergleicht man die Werte der Setup- und Hold-Zeiten der übertragenen Audioausgangsdaten aus Tabelle 8.4 mit denen des Audiocodecs geforderten Setup- und Hold-Zeiten aus Tabelle 8.1, so ist ersichtlich, dass alle Timing-Anforderungen des I^2S Interfaces erfüllt werden.

8.6.1 Ermittlung der Timing-Parameter

Aufgrund der verwendeten Synchronizer-Struktur für das digitale Audioausgabesignal wird das erste Bit eines Audiosamples bereits bei der letzten ⎿-Flanke eines Bursts an `BCLKo` ausgegeben. Mit den ersten 15 ⎿-Falnken des nächsten Burts werden dann die restlichen Bits des Audiosamples ausgegeben. Da das Signal `BCLKo` zwischen zwei Bursts jedoch für 403 ns einen Ruhezustand (Low-Pegel) einnimmt, unterscheidet sich die Setup-Zeit von `SDOUT` im Falle des ersten Bits bei einem Burst an `BCLKo` zur Setup-Zeit eines Bits während eines Bursts, da die Zeitdauer zwischen einer ⎿-Flanke und einem ⎍-Übergang an `BCLKo` während eines Burts nur 155 ns beträgt.

Da der Ausgang der verwendeten Synchronizer-Struktur erst 6 ns (Summe aus t_{GCK}, t_{COI} und t_{OUT}) nach einer ⎿-Flanke an `BCLKo` gültig ist, muss dieses Delay noch von den 403 ns bzw. 155 ns abgezogen werden.

$$\begin{aligned} t_{1,max} &= 403\,\text{ns} - 6\,\text{ns} = 397\,\text{ns} \\ t_{1,min} &= 155\,\text{ns} - 6\,\text{ns} = 149\,\text{ns} \end{aligned}$$

Die minimale Hold-Zeit von `SDOUT` ergibt sich aus der Differenz der Taktperiode von `BCLKo` und der minimalen Setup-Zeit von `SDOUT`.

$$t_{2,min} = 310\,\text{ns} - 149\,\text{ns} = 161\,\text{ns}$$

9 Videoübertragung über den RPI-Bus

9.1 Videoausgabe

Wie im Kapitel 3 bereits erwähnt wurde, wird der Pixeltakt zur Videoausgabe zum Takten des Serializers des Downchannels verwendet. Die Signale zur Videoausgabe, welche dem Serializer zugeführt werden sind somit automatisch synchron zum Taktsignal des Serializers, wodurch keine weiteren Maßnahmen zur Signalsynchronisation vor der Übertragung notwendig sind.

Auch nach dem Deserializer am Peripheriegerät sind die digitalen Videosignale wieder synchron zum Pixeltakt und können daher sofort zum Ansteuern eines Displays verwendet werden.

9.2 Videoeingabe

Bei Peripheriegeräten – welche ein Kameramodul besitzen – wird der Pixeltakt zur Videoeingabe zum Takten des Serializers des entsprechenden Upchannels verwendet. Somit sind auch hier die Signale zur Videoeingabe automatisch synchron zum Taktsignal des Serializers.

Auch hier sind die Videosignale nach dem entsprechenden Deserializer am RPI-Testboard wieder synchron zum Pixeltakt. Da jedoch der i.MX31 nur ein CMOS Sensor Interface aufweist, muss einer der vier Upchannels zur Videoeingabe ausgewählt werden. Dazu werden die Ausgänge der vier Deserializer von den Upchannels – an denen die übertragenen Videosignale anliegen – zum CPLD am RPI-Testboard geführt, der schließlich mittels Multiplexer die Signale eines Upchannels zum CSI des i.MX31 durchschleift.

Abbildung 9.1 zeigt das erforderliche Digitaldesign zur Videoeingabe. Rote Anschlüssen des Designs repräsentieren hierbei wieder Ausgänge des CPLDs, während blaue Anschlüssen als Eingänge dienen.

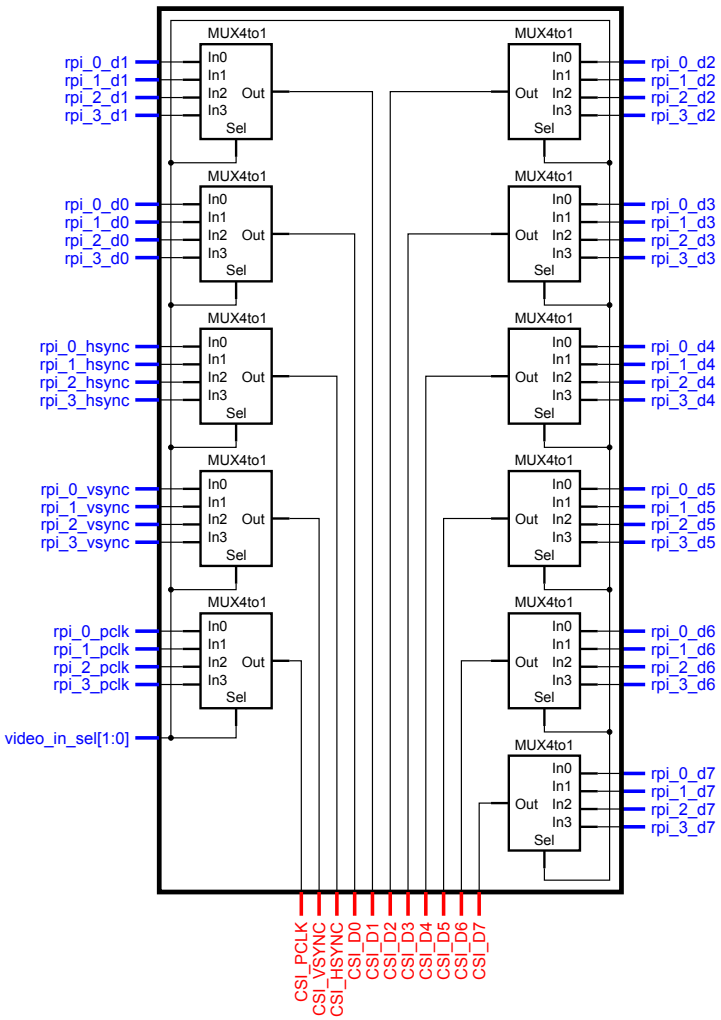


Abbildung 9.1: Digitaldesign zur Auswahl der Videoeingangssignale von einem der vier Upchannels

10 Kommunikationsprotokoll beim RPI-Bus

Der i.MX31 erhält über die Deserializer der vier Upchannels Auskunft darüber, an welchen Upchannels Geräte angeschlossen sind. Je nach dem, mit welchem der angeschlossenen Wearables er kommunizieren möchte, wird das entsprechende `MISO` Signal mittels des CPLDs ausgewählt. Somit besteht zwischen i.MX31 und Peripheriegerät eine transparente SPI Schnittstelle. Da jedoch der Downchannel beim RPI-Bus als Broadcast-Kanal entworfen wurde und somit immer alle Peripheriegeräte die vom i.MX31 gesendeten Daten empfangen, muss eine adressorientierte Datenkommunikation gewählt werden. Diesbezüglich wird vorgeschlagen, dass alle Peripheriegeräte mit einer fix eingestellten Geräteadresse versehen werden.

Problematisch zeigt sich hier bei noch, dass dem i.MX31 die Adressen der Peripheriegeräte zu Beginn nicht bekannt sind, weshalb ein Initialisierungsvorgang notwendig ist. Um diesen Initialisierungsvorgang einleiten zu können, ist es notwendig, Steuerzeichen zu übertragen, dessen Eindeutigkeit zu gewährleisten ist. Deshalb wird eine HEX zu ASCII Kodierung der Daten vorgeschlagen, sodass die Eindeutigkeit der Steuerzeichen immer gegeben ist.

10.1 Initialisierung von Wearables

Die Initialisierung von Wearables könnte beispielsweise in zwei Phasen erfolgen.

10.1.1 Initialisierungsphase 1

Der i.MX31 selektiert den `MISO` Eingang eines angeschlossenen Peripheriegeräts und schickt das Initialisierungskommando `0x07`. Alle Peripheriegeräte befinden sich währenddessen in einer Initialisierungsroutine und haben ihr Senderegister bereits mit ihrer High-Order Geräteadresse geladen, welche somit gleichzeitig mit dem Initialisierungskommando übertragen wird. Anschließend überträgt der i.MX31 ein IDLE-Byte `0x00`, sodass das Gerät auch die Low-Order Geräteadresse übertragen kann.

Der i.MX31 wiederholt schließlich High-Order und Low-Oorder Geräteadresse

gefolgt von einem weiteren IDLE-Byte. Je nach dem, ob High-Order und Low-Order Geräteadresse mit der eigenen Geräteadresse übereinstimmen, bestätigen die Peripheriegeräte während des IDLE-Bytes mit einem Acknowledge `0x06` oder Not-Acknowledge `0x15`. Das Gerät, welches die eigene Geräteadresse wieder empfangen hat, wartet nun auf die zweite Phase der Initialisierung, während die anderen Geräte auf das erneute Senden des Kommandos `0x07` warten.

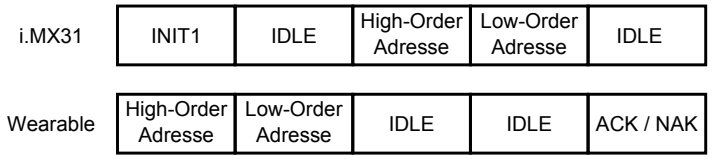


Abbildung 10.1: Datentransfer zwischen i.MX31 und Peripherie während Initialisierungsphase 1

10.1.2 Initialisierungsphase 2

Befindet sich ein Gerät bereits in der zweiten Initialisierungsphase, so reagiert es nur noch, wenn es mit der eigenen Geräteadresse angesprochen wird. Nach der Geräteadresse sendet der i.MX31 ein weiteres Initialisierungskommando `0x08` worauf das Peripheriegerät mit der Anzahl N an Befehlen antwortet, die es unterstützt. Der i.MX31 überträgt schließlich $2+2\times N+1$ IDLE-Bytes, während fortlaufend die vom Wearable unterstützen Befehle im ASCII-Format, gefolgt von einem weiteren Acknowledge, übertragen werden. Wird beim i.MX31 das Acknowledge empfangen und die Anzahl der gesendeten Befehle entspricht der ursprünglich angekündigten Anzahl N, so sendet der i.MX31 ein Acknowledge und teilt dem Peripheriegerät somit mit, dass es korrekt initialisiert wurde und die Initialisierungsphase 2 verlassen kann. Wird vom Peripheriegerät jedoch ein Not-Acknowledge empfangen, so wartet es erneut auf das Kommando `0x08`.

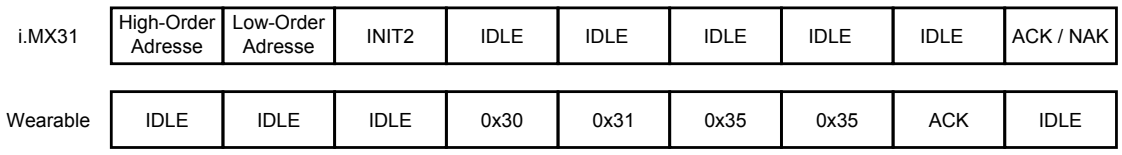


Abbildung 10.2: Datentransfer zwischen i.MX31 und Peripherie während Initialisierungsphase 2

In Abbildung 10.2 ist der Datentransfer zwischen i.MX31 und einem Peripheriegerät gezeigt, welches nur den Befehl `0x55` unterstützt, bei dem es sich beispielsweise um den Lesebefehl eines Sensorwerts handeln könnte.

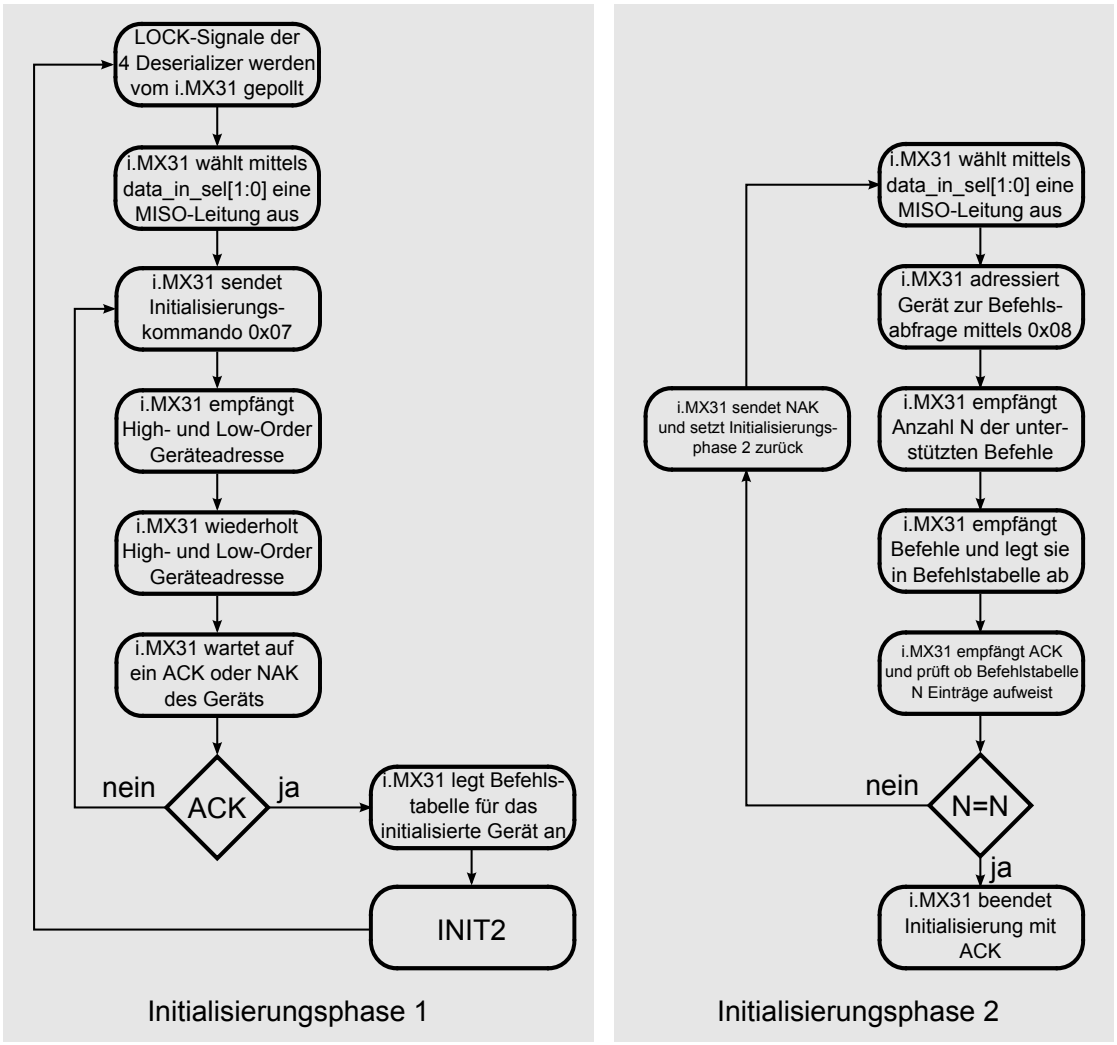


Abbildung 10.3: links: Flussdiagramm der Initialisierungsphase 1; rechts: Flussdiagramm der Initialisierungsphase 2

10.2 Kommunikation mit Wearables

Der Verbindungsaufbau mit einem Peripheriegerät beginnt wieder mit dem Auswählen der gewünschten `MISO` Leitung. Anschließend wird das Gerät mit seiner Geräteadresse angesprochen und ein entsprechender Befehl – den das Peripheriegerät unterstützt – wird übertragen. Das Peripheriegerät sendet währenddessen fortlaufend IDLE-Bytes, bis dieses den Befehl dekodiert hat und schließlich mit einem Acknowledge die Übertragung quittiert. Handelt es sich beim übertragenen Befehl beispielsweise um die Anforderung von Sensordaten, so werden nach der Dekodierung die angeforderten Daten übertragen und abschließend wird die Übertragung wieder mit einem Acknowledge vom Peripheriegerät quittiert.

Abbildung 10.4 zeigt den Datentransfer beim Auslesen eines Temperatursensors

mittels des Befehls `0x55`. Der Übertragene Temperaturwert ist in diesem Beispiel mit 26.3 °C gegeben.

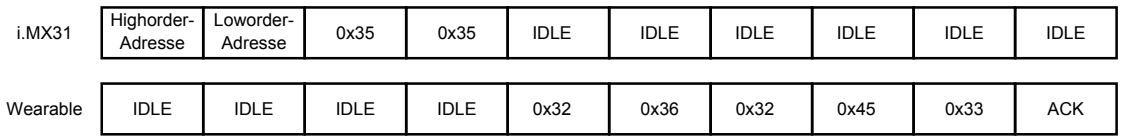


Abbildung 10.4: Datentransfer zwischen i.MX31 und Peripherie während des Auslesens eines Temperatursensors

11 Konklusion

In dieser Arbeit wurde in erster Linie ein Verfahren entwickelt, welches die Übertragung von digitalen Video-, Audio- und Steuerdaten in Wearable Computing Applikationen unter Zuhilfenahme der SerDes DS90UR241 und DS90UR124 von Texas Instruments ermöglicht. Zusätzlich wurde gezeigt, dass alle Timing-Anforderungen der verwendeten Schnittstellen durch das in dieser Arbeit entwickelte Verfahren eingehalten werden können.

Weiters wurde ein Konzept zur Phantomspeisung von Peripheriegeräten in Wearable Computing Applikationen entwickelt. Diese Phantomspannung befindet sich gleichzeitig mit den digitalen Video-, Audio- und Steuerdaten auf den Datenleitungen, welche die Peripheriegeräte mit der zentralen Recheneinheit verbinden. Es wurde gezeigt, dass mittels der SMD-Ferrite 742792097 von Würth Elektronik, ein Versorgungsstrom von 500 mA für die Peripheriegeräte zur Verfügung gestellt werden kann und trotzdem eine funktionierende Datenverbindung vorliegt.

11.1 Ausblick

- Detect-Circuit zur automatischen Erkennung angeschlossener Peripheriegeräte, sodass die Upchannels bei Bedarf in den Power-Down Modus versetzt werden können.

- Redesign der Upchannels, um Request-Leitungen von den Peripheriegeräten zum i.MX31 zur Verfügung stellen zu können. Mittels dieser Request-Leitungen soll auch den Wearables die Möglichkeit eingeräumt werden, einen Datentransfer einzuleiten.

- Entwurf eines Spannungsversorgungskonzepts.

- Entwurf eines weiteren Prototypen, basierend auf dem Modul TX27 von Ka-Ro electronics.

Literaturverzeichnis

[1] Kat Hannaford. Apple Hires Expert On "Wearable Technologies". http://gizmodo.com/5493613/apple-hires-expert-on-wearable-technologies. Abgerufen am 22/05/2012.

[2] Wolfgang Thronicke. Wearable Computing im Arbeitsumfeld. Technical report, Siemens AG, 2007.

[3] Diane Sieger. Anziehend. iX - MAGAZIN FÜR PROFESSIONELLE INFORMATIONSTECHNIK, August 2010.

[4] Mario Schwaiger und Marco Schwaiger. Firma — www.spintower.eu. http://www.spintower.eu/home.php?ll=de&page=firma. Abgerufen am 22/05/2012.

[5] BLUETECHNIX. *Quick Start Guide*. BLUTECHNIX.

[6] Texas Instruments. *LVDS Application and Data Handbook*. Texas Instruments, November 2002.

[7] Texas Instruments. *LVDS Owner's Manual*. Texas Instruments, 2008. 4.Edition.

[8] MAXIM. *Introduction to LVDS, PECL, and CML*. MAXIM Integrated Products, April 2008. 1.Edition.

[9] Nick Holland. *Interfacing Between LVPECL, VML, CML, and LVDS Levels*. Texas Instruments, Dezember 2002.

[10] TechChee.com. T-Gloves packs all everyday gadgets on your wrist. http://www.techchee.com/2009/12/01/t-glove-packs-all-everyday-gadgets-on-your-wrist/. Abgerufen am 29/02/2012.

[11] MarvGolden.com. David clark h8590 pro audio/video headset. http://www.marvgolden.com/david-clark-h8590-pro-audio-video-headset.html. Abgerufen am 08/03/2012.

[12] Sierra Wireless. Embedded SIM. http://www.sierrawireless.com/products

andservices/AirPrime/Wireless_Modules/Embedded_SIM.aspx. Abgerufen am 08/03/2012.

[13] ZOONAR. Telecom, Wissenschaft, Ulm, Baden-Wuerttemberg, Deutschland. http://www.zoonar.de/photo/telecom-wissenschaftsstadt-ulm-badenwuerttemberg-deutschland-telecom-sciencecity-ulm-badenwrttemberg-germany_600705.html. Abgerufen am 08/03/2012.

[14] en.wikipedia.org. Emergency medical services in Austria. http://en.wikipedia.org/wiki/File:Moderne_Leitstelle_Arbeitsplatz.jpg. Abgerufen am 14/08/2012.

[15] Freescale Semiconductor. *MCIMX31 and MCIMX31L Applications Processors Reference Manual*. Freescale Semiconductor, Juli 2006.

[16] Freescale Semiconductor. *MCIMX31 and MCIMX31L Multimedia Applications Processor*. Freescale Semiconductor, November 2011.

[17] National Semiconductors. *5-43 MHz DC-Balanced 24-Bit FPD-Link II Serializer and Deserializer Chipset*. National Semiconductors, Oktober 2011.

[18] Beta LAYOUT GmbH. service - spezifikationen — www.pcb-pool.com. http://www.pcb-specification.com/de. Abgerufen am 24/07/2012.

[19] H. Zenkner T. Brandner A. Gerfer, B. Rall. *Trilogie der Induktiven Bauelemete*. WÜRTH ELEKTRONIK, 2008. 4.Auflage.

[20] National Semiconductors. *3.125 Gbps 1:4 LVDS Buffer/Repeater with Transmit Pre-emphasis and Receive Equalization*. National Semiconductors, Dezember 2007.

[21] NXP Semiconductors. *Dual N-channel μTrenchMOS^{TM}extremely low level FET*. NXP Semiconductors, Febraur 2004. Revision 1.0.

[22] CML Innovative Technologies. *CMD28-21 Series SMT LEDs*. CML Innovative Technologies.

[23] Michael Hemetsberger. *Implementierung eines Video-Headsets basierend auf dem RPI-Bus*. Technische Universität Graz, Oktober 2012.

[24] Xilinx. *XC2C256 CoolRunner-II CPLD*, März 2007. Revision 3.2.

[25] Xilinx. *Bulletproof CPLD Design Practice*, Juni 2005. Revision 1.0.

[26] Xilinx. *Driving LEDs with Xilinx CPLDs*, April 2005. Revision 1.0.

[27] Xilinx. *A Quick JTAG ISP Checklist*, Dezember 2007. Revision 3.0.1.

[28] NXP Semiconductors. *Low-dropout Linear Regulators*. NXP Semiconductors, Juli 2011. Revision 1.0.

[29] Lothar Miller. Entkopplung. http://www.lothar-miller.de/s9y/categories/14-Entkopplung. Abgerufen am 31/07/2012.

[30] KAOHSIUNG HITACHI ELECZTRONICS CO.,LTD. *TX09D70VM1CCA*. KAOHSIUNG HITACHI ELECZTRONICS CO.,LTD, Jänner 2011. Revision 6.0.

[31] National Semiconductors. *3.125 Gbps 1:4 LVDS Repeater with Transmit Preemphasis and Receive Equalization*. National Semiconductors, Mai 2011. Revision 1.0.

[32] Leroy Davis. Digital Logic Metastability. http://www.interfacebus.com/Design_MetaStable.html. Abgerufen am 31/07/2012.

[33] Xilinx. *Metastability Characteristics for CoolRunner CPLDs*. Xilinx, Februar 2000. Revision 1.1.

[34] Ryan Donohue. Synchronization in Digital Logic Circuits. http://-www.stanford.edu/class/ee183/handouts_spr2003/synchronization_pres.pdf. Abgerufen am 31/07/2012.

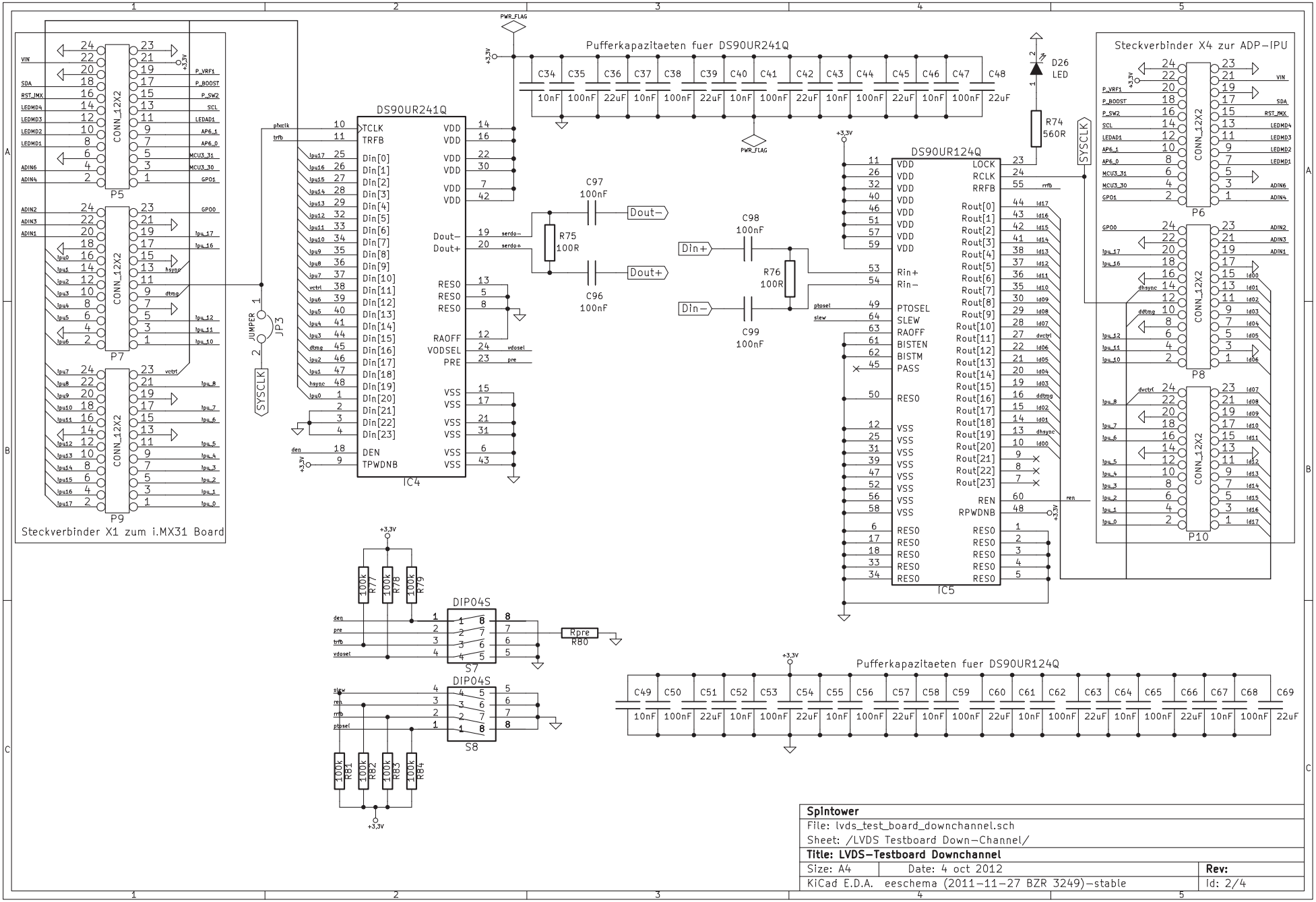
[35] Xilinx. *Understanding the CoolRunner-II Timing Model*. Xilinx, Februar 2003. Revision 1.5.

[36] MAXIM. *Stereo Audio CODECs with Microphone, DirectDrive Headphones, Speaker Amplifiers, or Line Outputs*. MAXIM, Juli 2007. Revision 2.0.

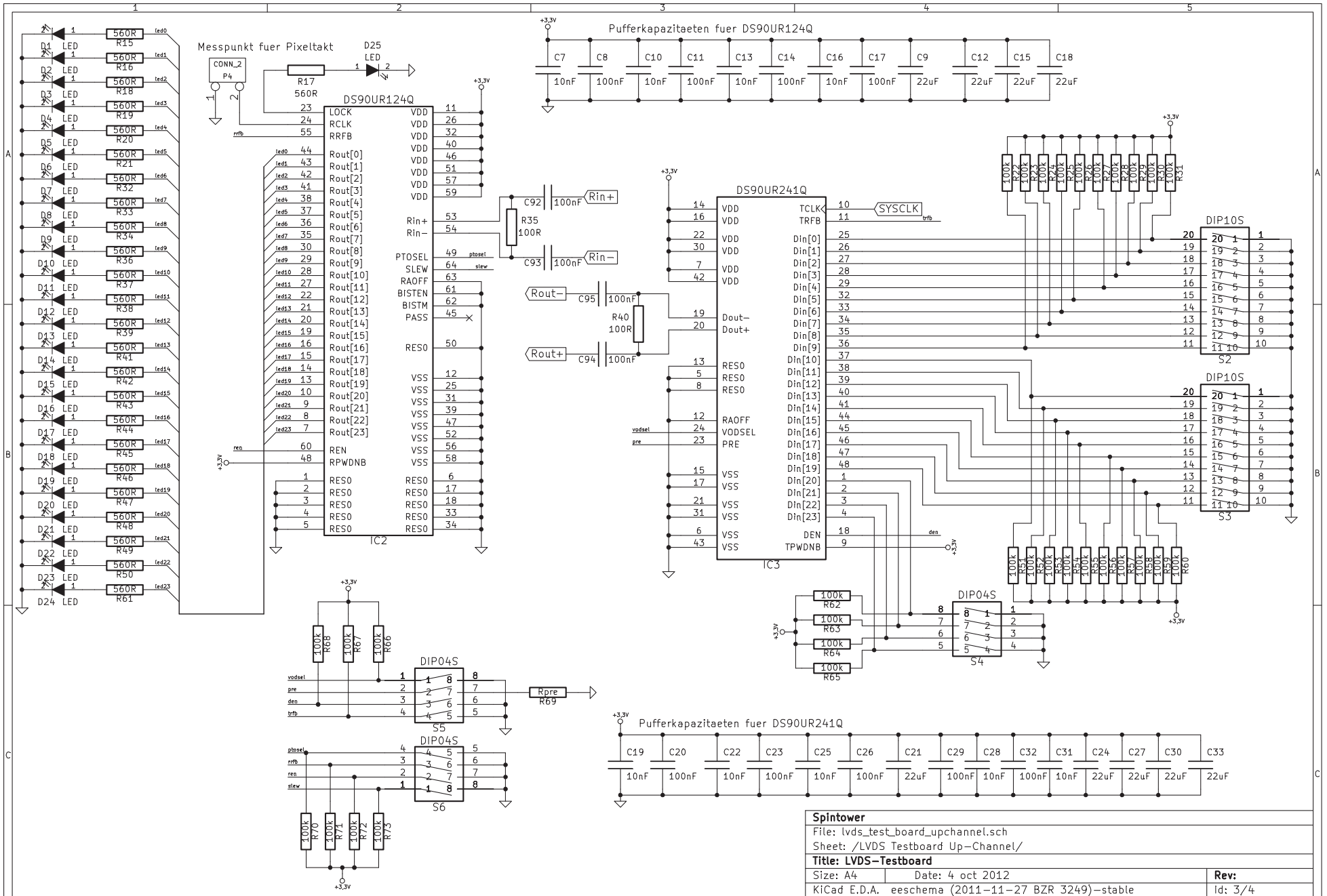
Anhang A

In diesem Abschnitt befinden sich die Originale der Stromlaufpläne des LVDS-Testboards, sowie des RPI-Testboards. Zusätzlich sind die Layouts aller Lagen der erstellten Platinen angefügt.

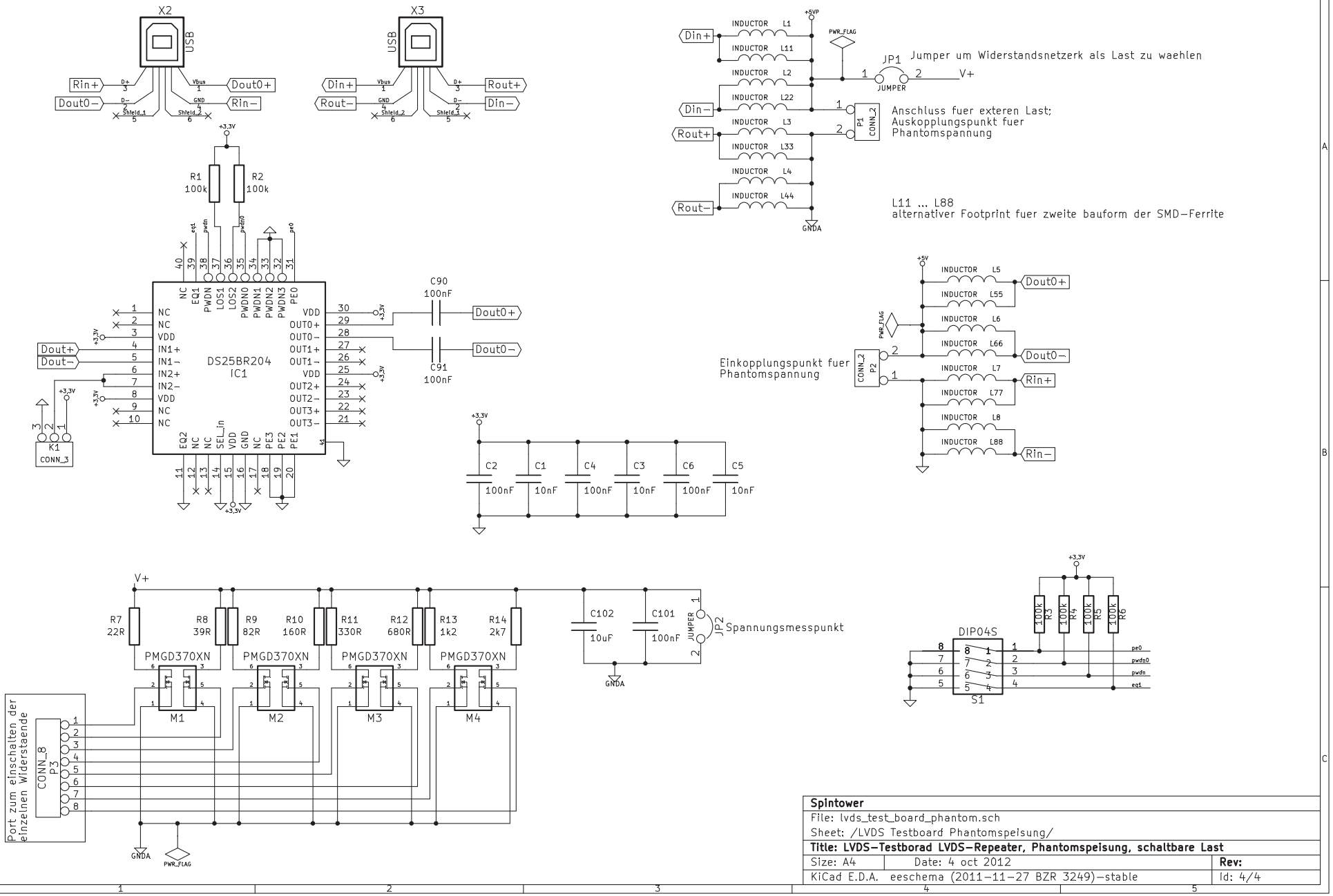
Die Stromlaufpläne und Layouts wurden mittels KiCad erstellt.



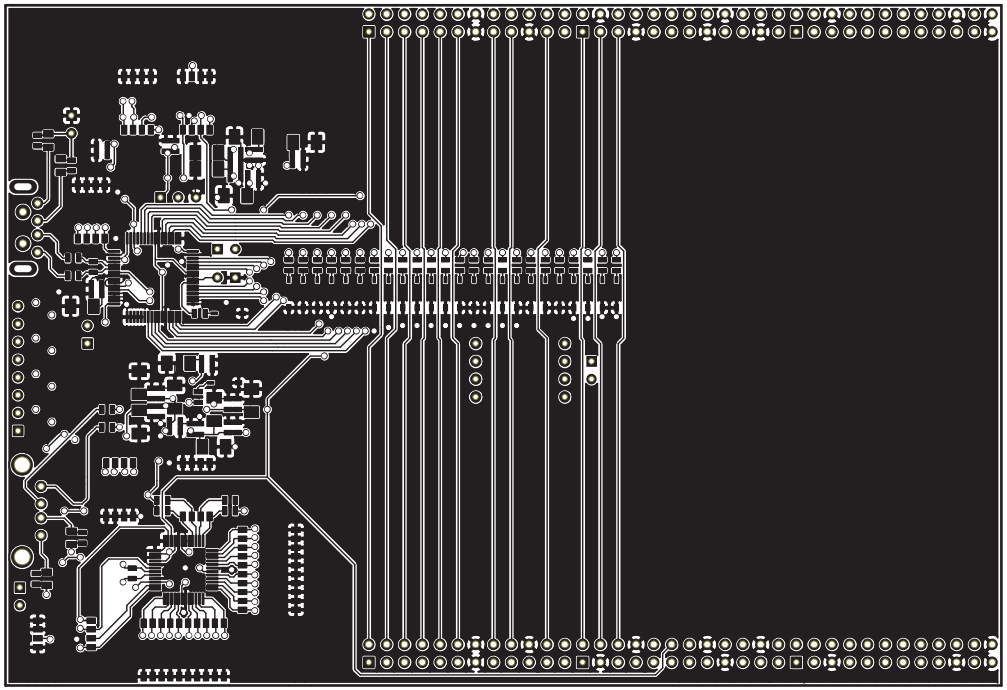
Spintower		
File: lvds_test_board_downchannel.sch		
Sheet: /LVDS Testboard Down−Channel/		
Title: LVDS−Testboard Downchannel		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A. eeschema (2011−11−27 BZR 3249)−stable		Id: 2/4



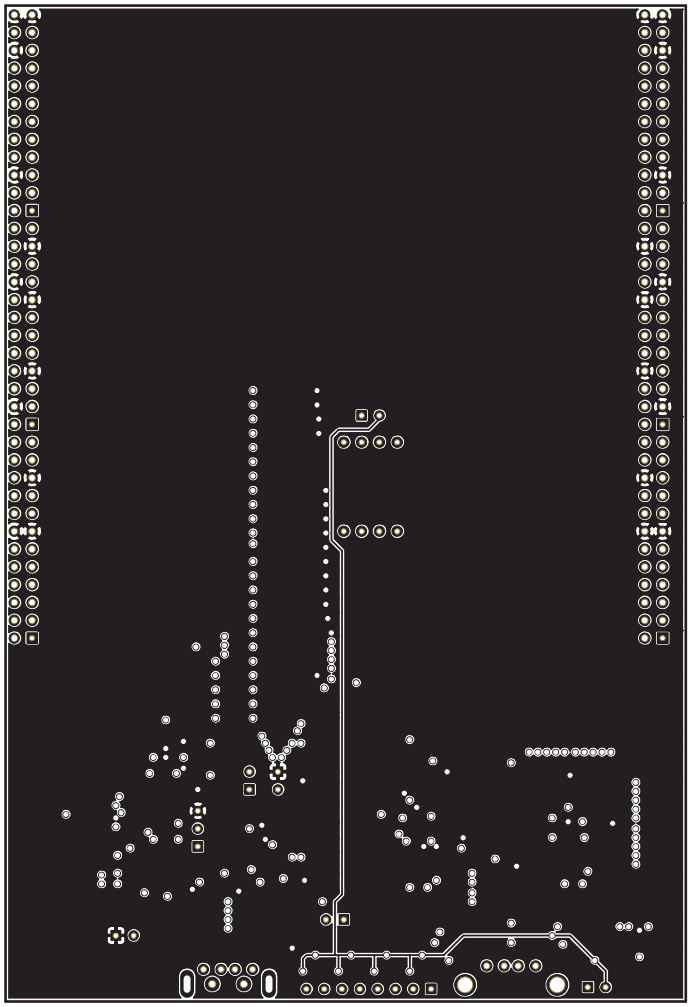
Spintower		
File: lvds_test_board_upchannel.sch		
Sheet: /LVDS Testboard Up-Channel/		
Title: LVDS-Testboard		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A. eeschema (2011-11-27 BZR 3249)-stable		Id: 3/4



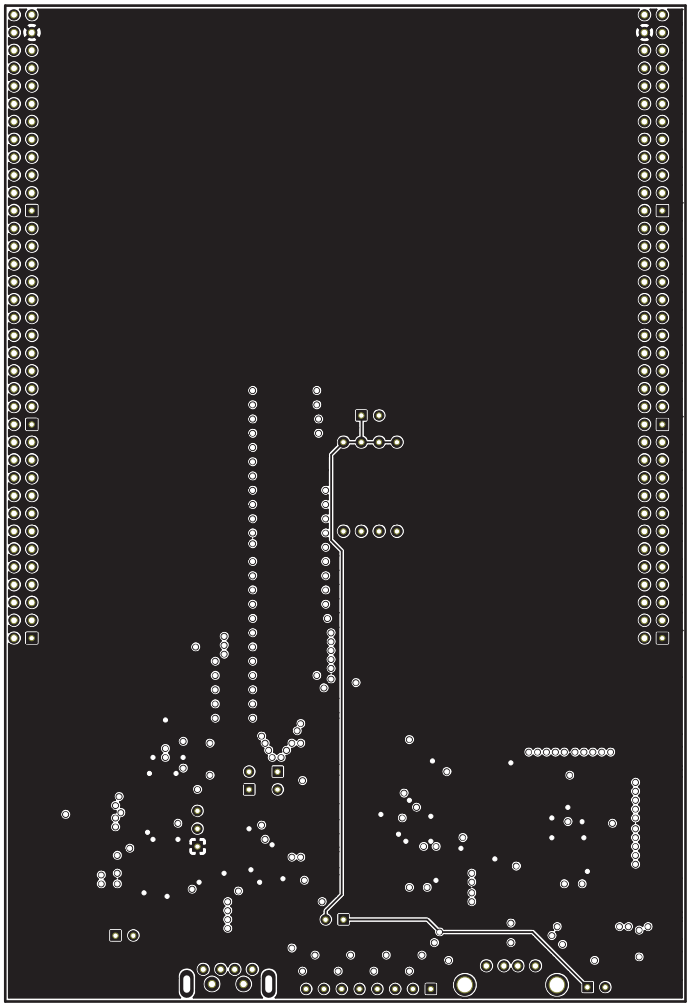
Spintower		
File: lvds_test_board_phantom.sch		
Sheet: /LVDS Testboard Phantomspeisung/		
Title: LVDS−Testborad LVDS−Repeater, Phantomspeisung, schaltbare Last		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A. eeschema (2011−11−27 BZR 3249)−stable		Id: 4/4



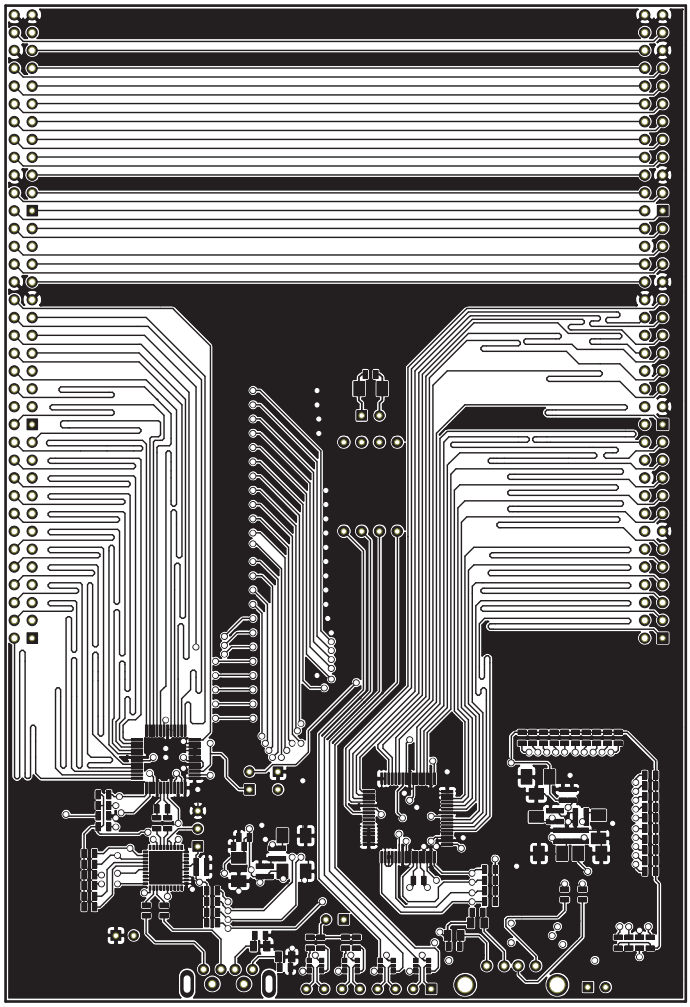
Spintower		
File: lvds_test_board_root.brd		
Sheet: 1/1		
Title: LVDS-Testborad Layer: Top		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A.	pcbnew (2011-11-27 BZR 3249)-stable	Id: 1/1



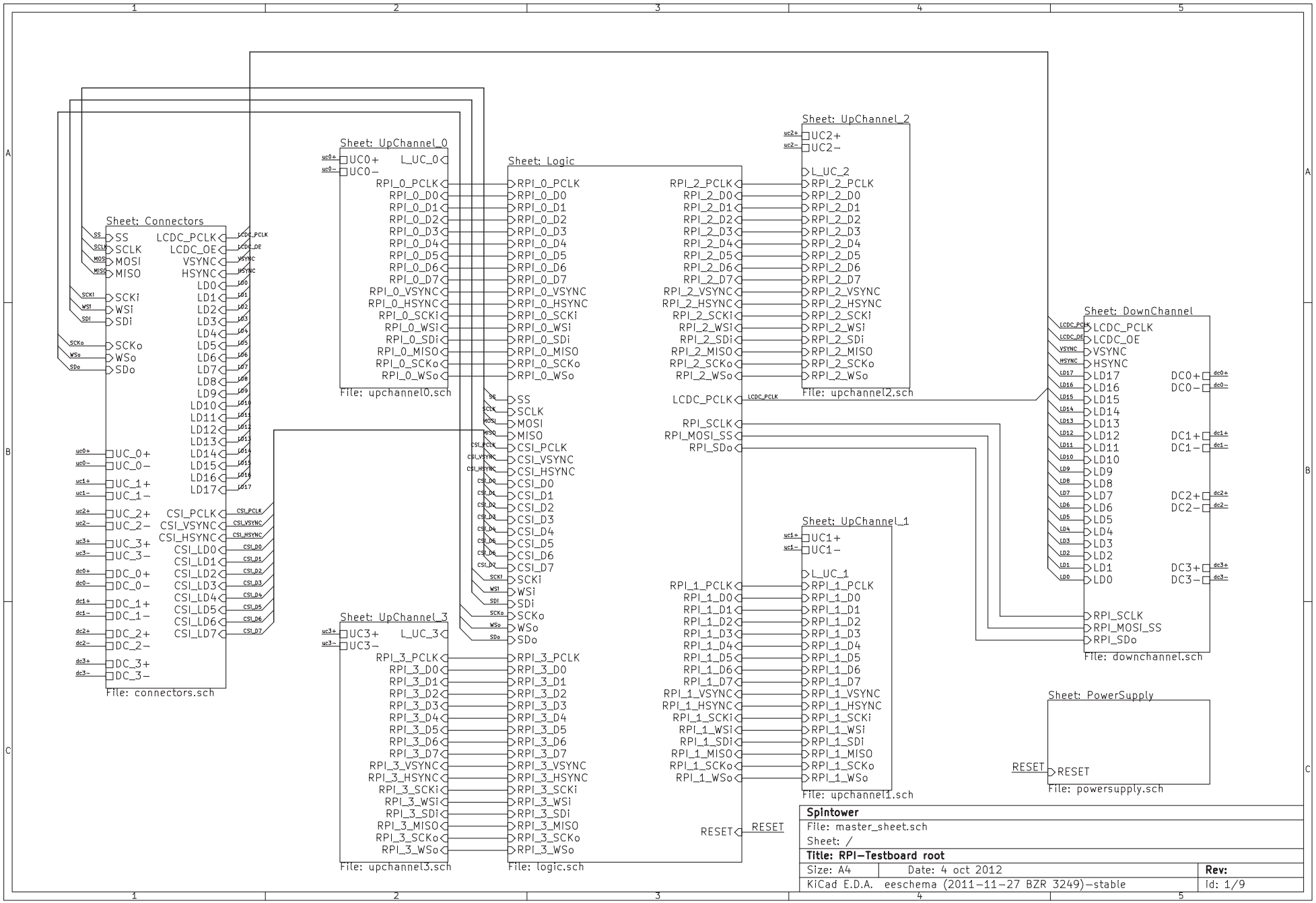
Spintower			
File: lvds_test_board_root.brd			
Sheet: 1/1			
Title: LVDS-Testborad Layer: GND			
Size: A4	Date: 4 oct 2012		**Rev:**
KiCad E.D.A. pcbnew (2011-11-27 BZR 3249)-stable			Id: 1/1

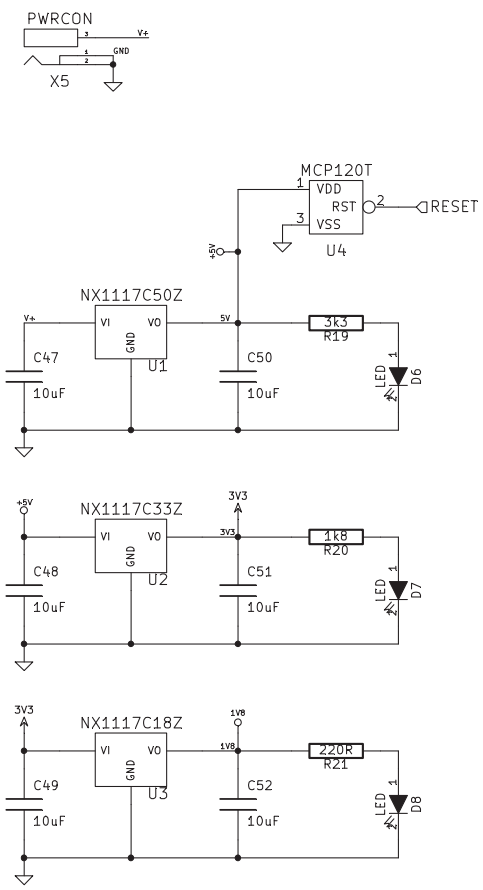


Spintower		
File: lvds_test_board_root.brd		
Sheet: 1/1		
Title: LVDS-Testborad Layer: VCC		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A. pcbnew (2011-11-27 BZR 3249)-stable		Id: 1/1

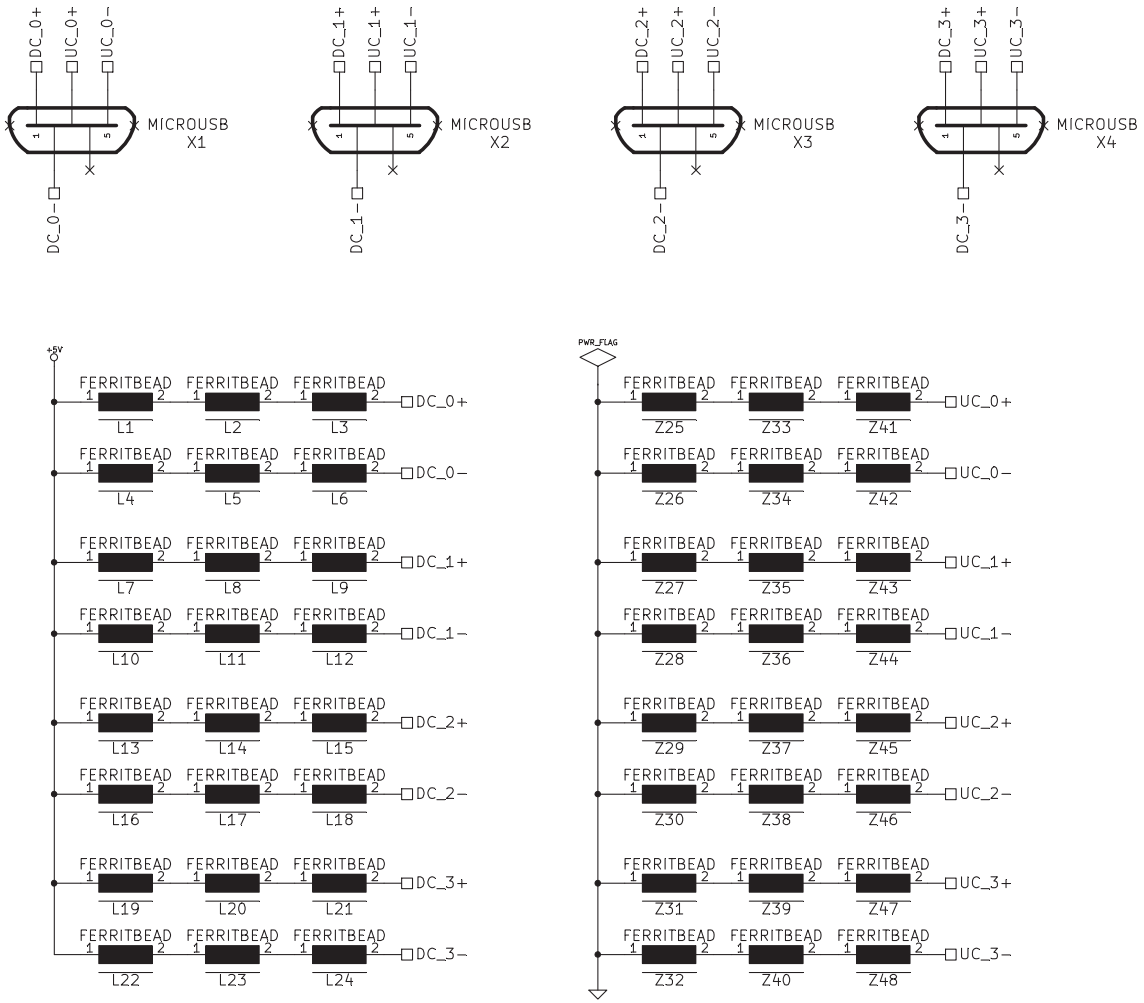


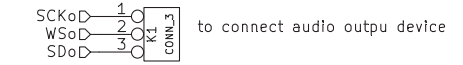
Spintower		
File: lvds_test_board_root.brd		
Sheet: 1/1		
Title: LVDS-Testborad Layer: Bottom		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A. pcbnew (2011-11-27 BZR 3249)-stable		Id: 1/1


Sheet: Connectors
File: connectors.sch
Sheet: UpChannel_0
File: upchannel0.sch
Sheet: Logic
File: logic.sch
Sheet: UpChannel_2
File: upchannel2.sch
Sheet: DownChannel
File: downchannel.sch
Sheet: UpChannel_1
File: upchannel1.sch
Sheet: UpChannel_3
File: upchannel3.sch
Sheet: PowerSupply
File: powersupply.sch
Spintower
File: master_sheet.sch
Sheet: /
Title: RPI-Testboard root
Size: A4
Date: 4 oct 2012
Rev:
KiCad E.D.A. eeschema (2011-11-27 BZR 3249)-stable
Id: 1/9

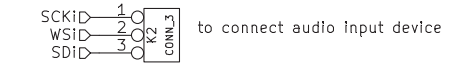


File: powersupply.sch		
Sheet: /PowerSupply/		
Title:		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A. eeschema (2011-11-27 BZR 3249)-stable		Id: 2/9

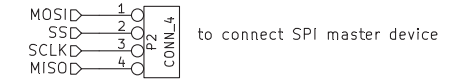




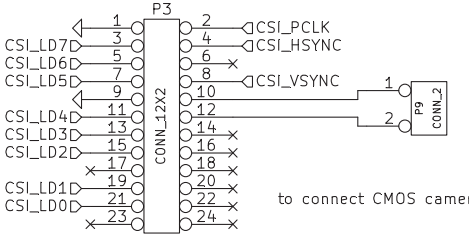
to connect audio outpu device



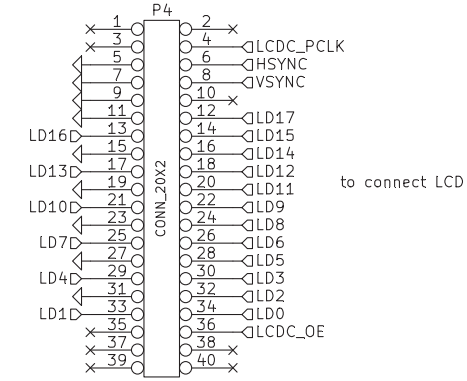
to connect audio input device



to connect SPI master device

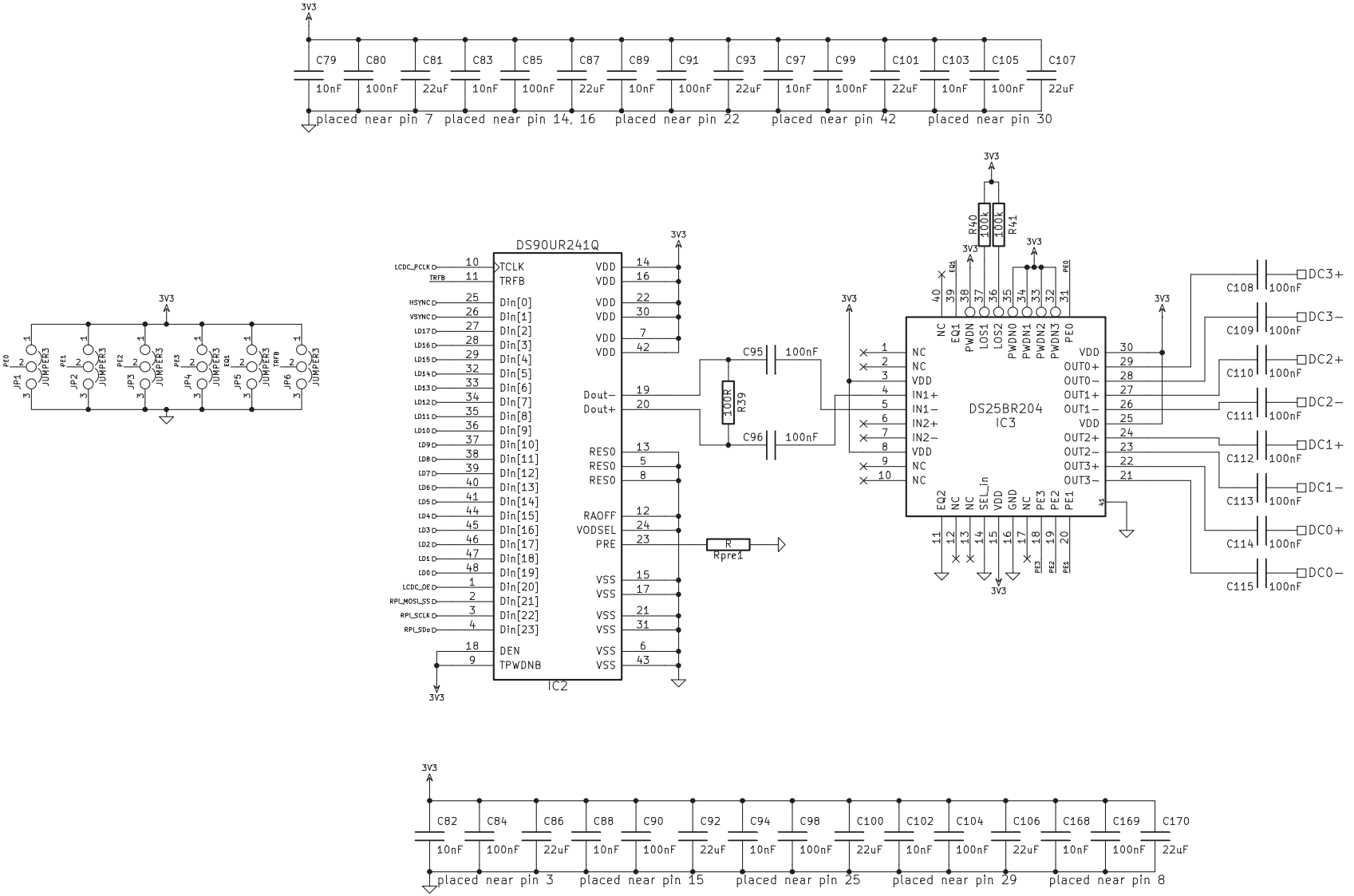


to connect CMOS camera

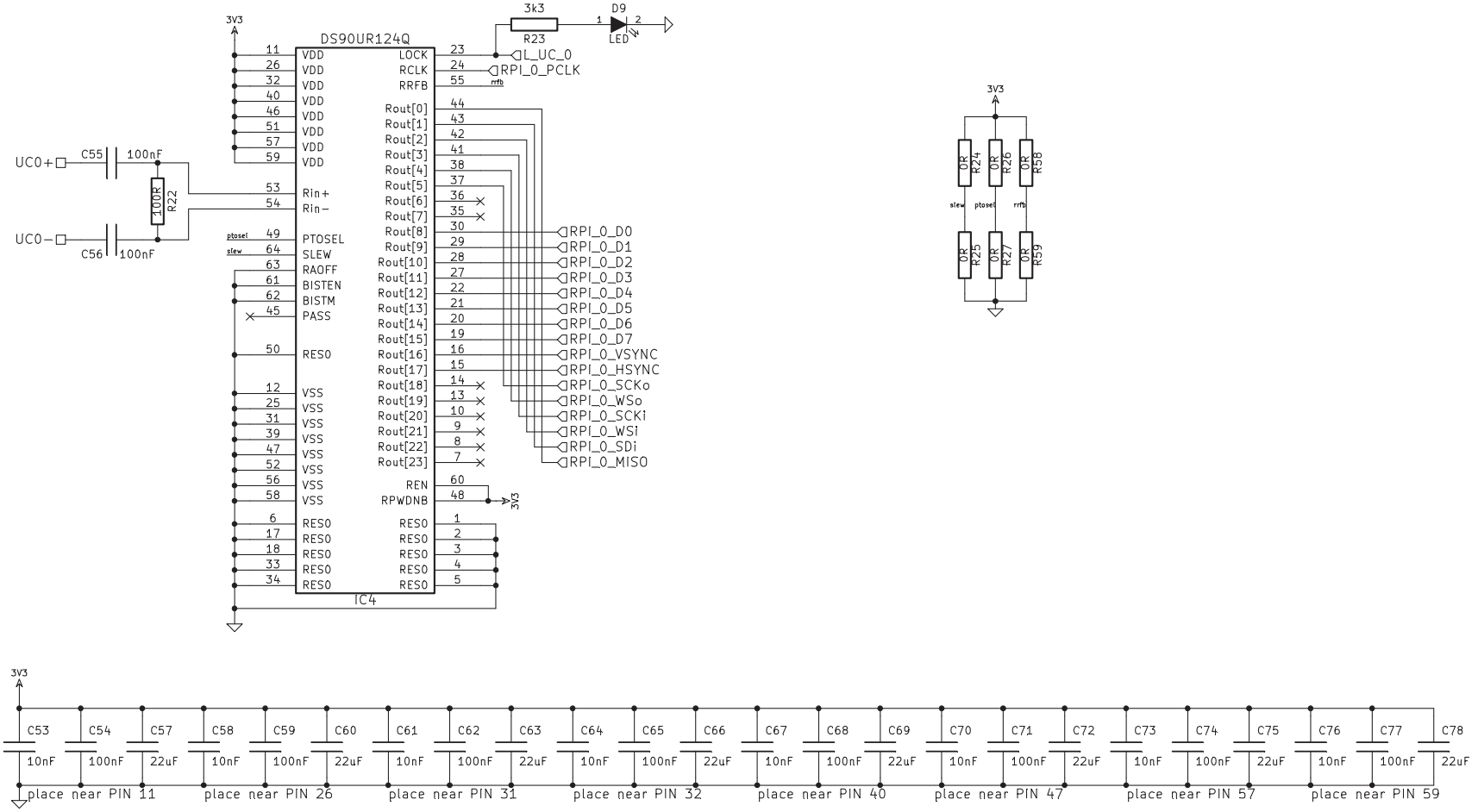


to connect LCD

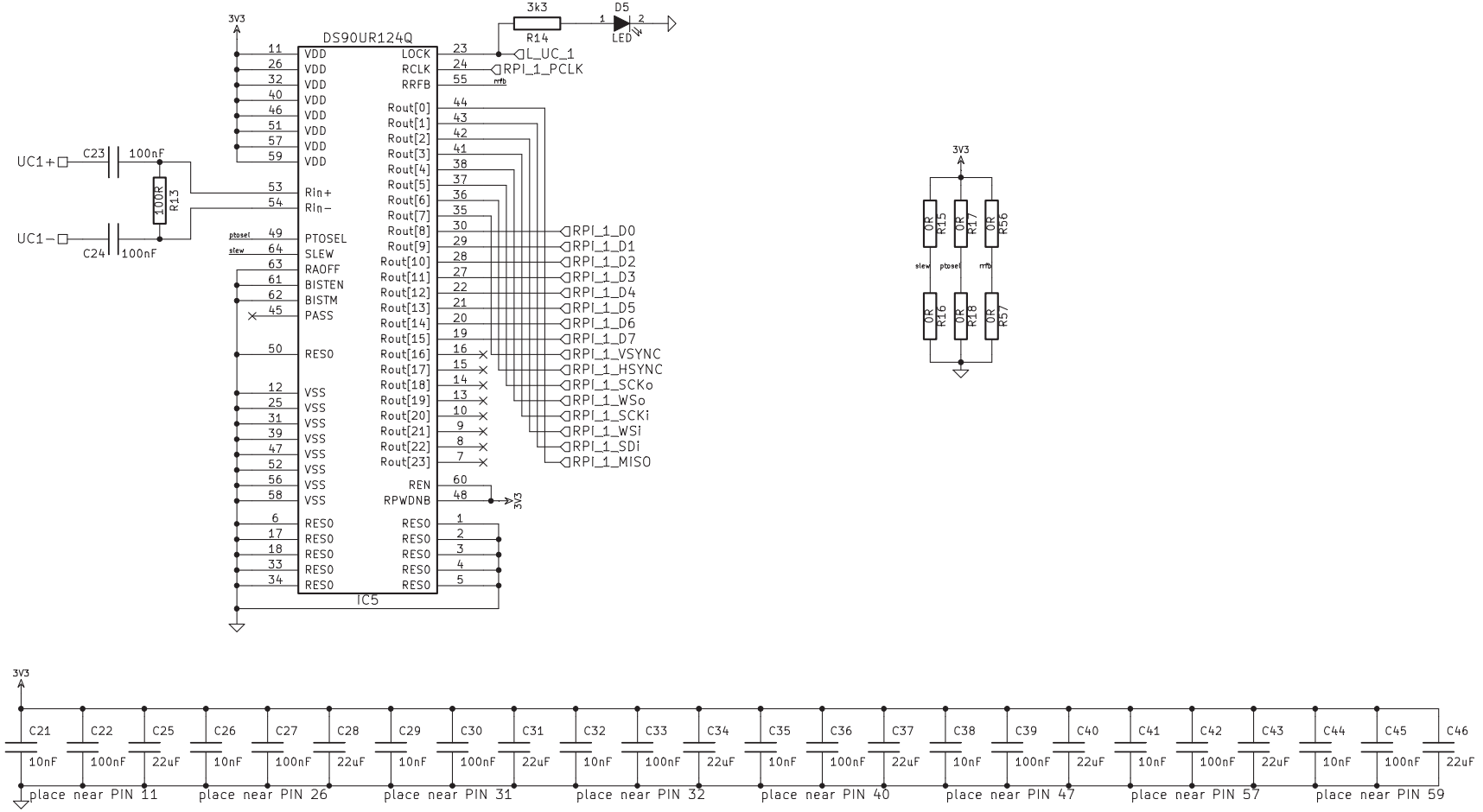
Spintower		
File: connectors.sch		
Sheet: /Connectors/		
Title: RPI-Testboard Steckverbinder		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A.	eeschema (2011-11-27 BZR 3249)-stable	Id: 3/9



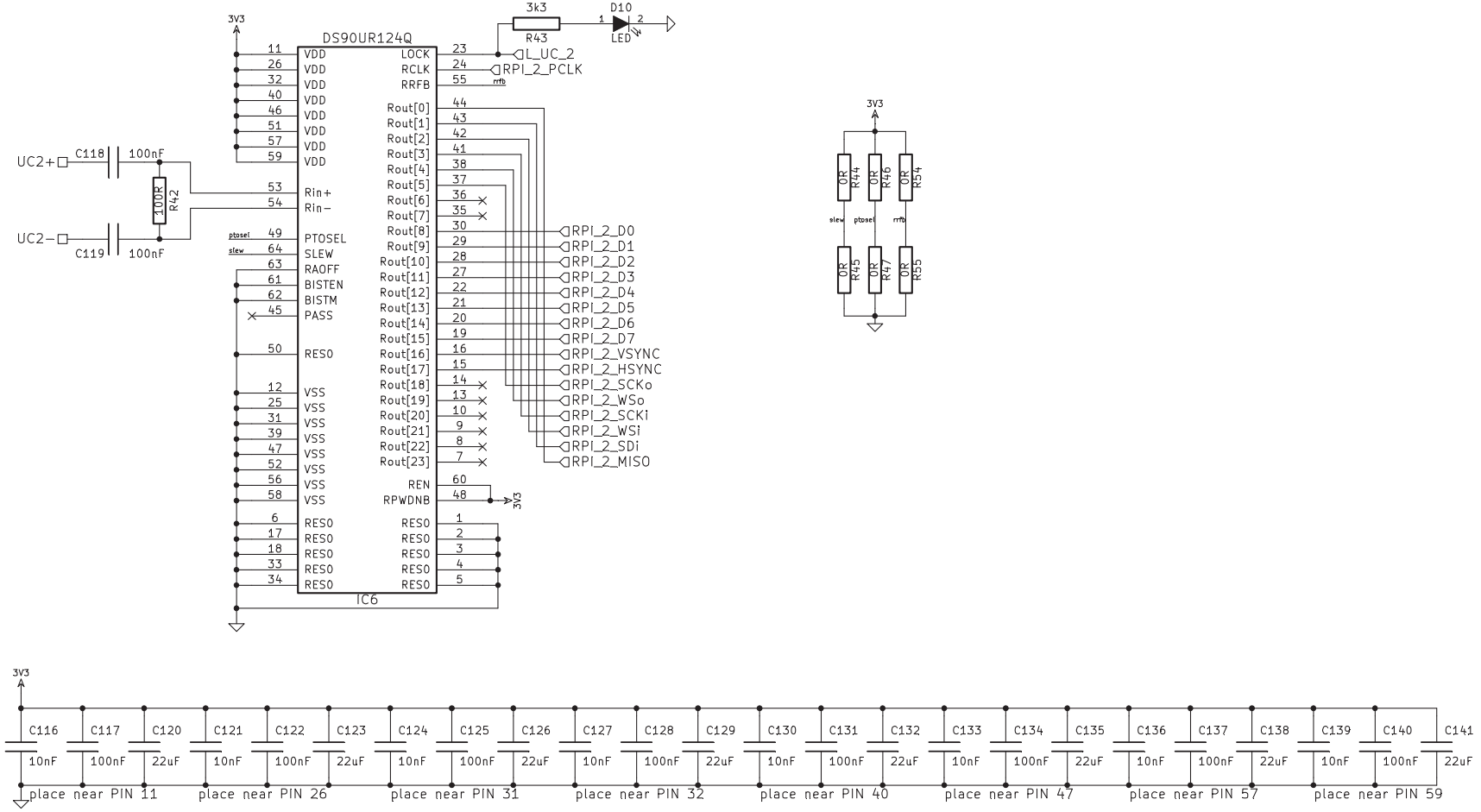
Spintower		
File: downchannel.sch		
Sheet: /DownChannel/		
Title: RPI−Testboard Downchannel		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A. eeschema (2011−11−27 BZR 3249)−stable		Id: 4/9



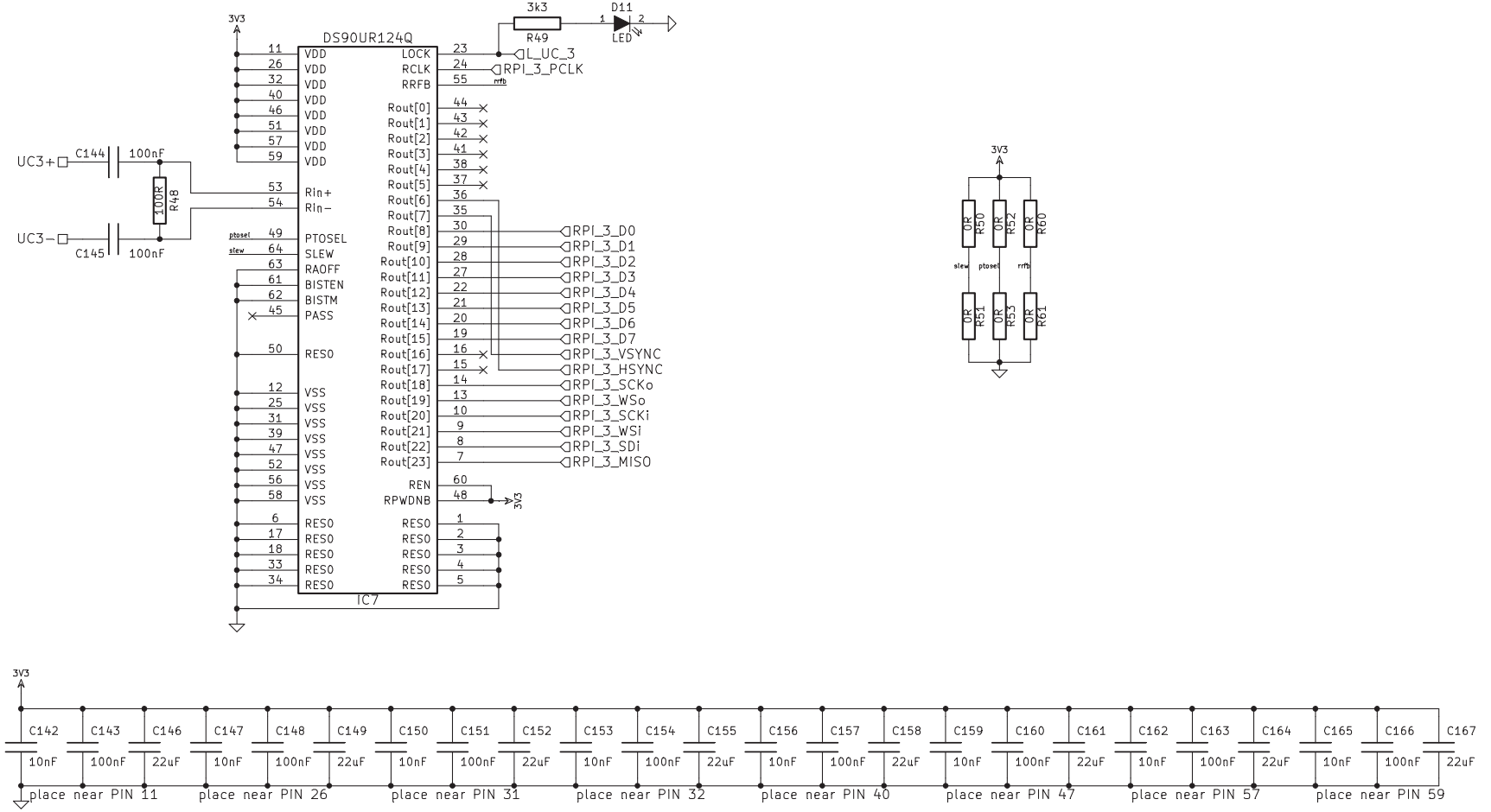
Spintower		
File: upchannel0.sch		
Sheet: /UpChannel_0/		
Title: RPI–Testboard Upchannel 0		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A.	eeschema (2011–11–27 BZR 3249)–stable	Id: 5/9



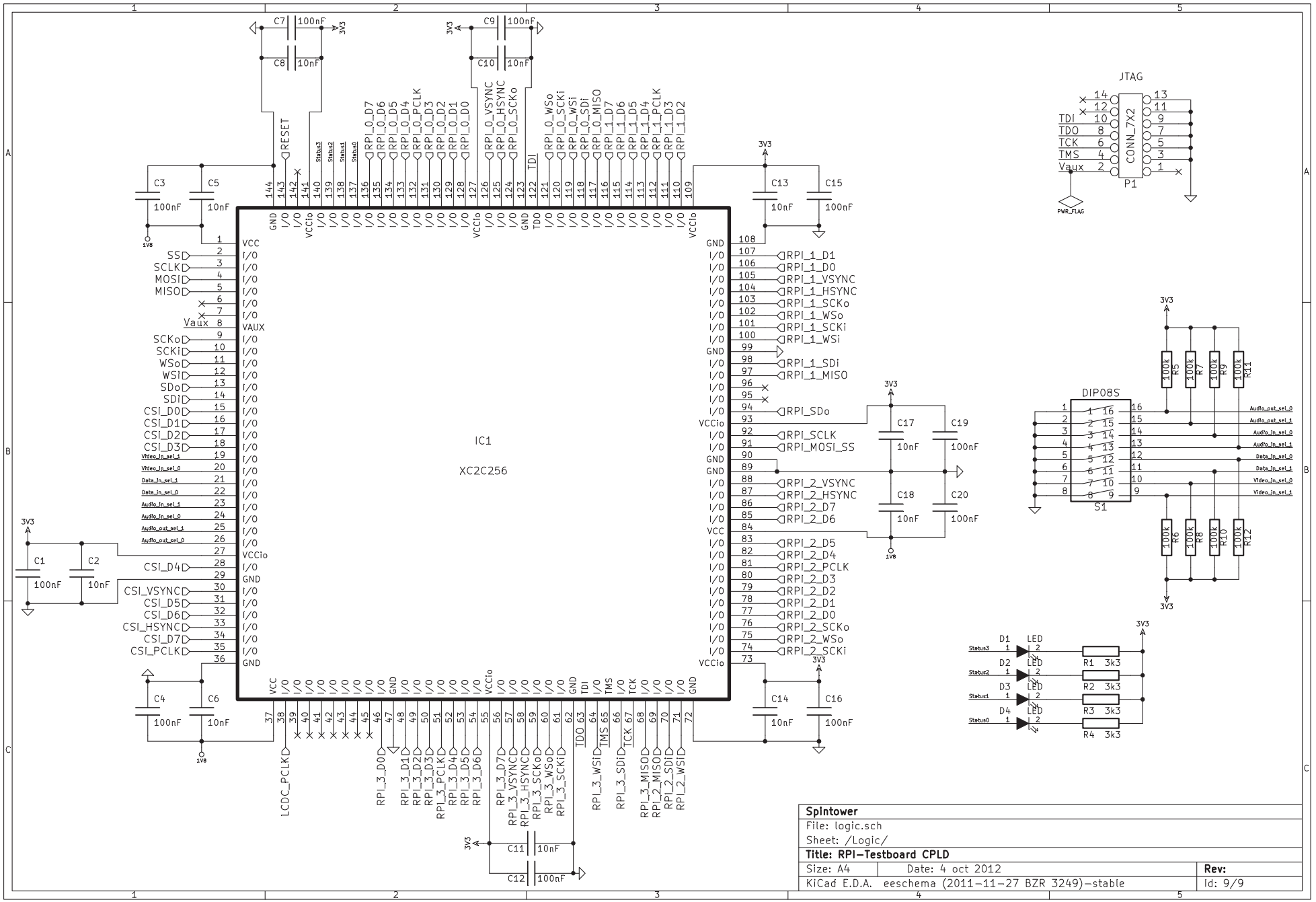
Spintower		
File: upchannel1.sch		
Sheet: /UpChannel_1/		
Title: RPI–Testboard Upchannel 1		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A.	eeschema (2011–11–27 BZR 3249)–stable	Id: 6/9

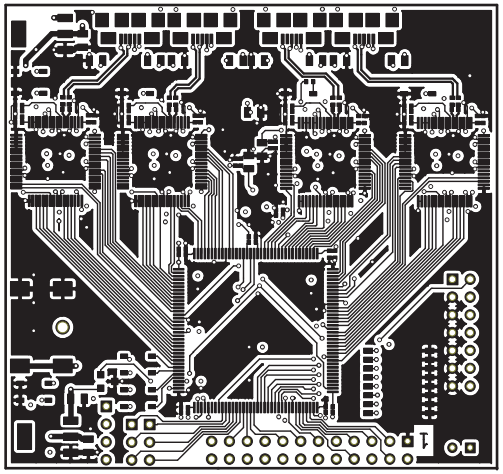


Spintower		
File: upchannel2.sch		
Sheet: /UpChannel_2/		
Title: RPI−Testboard Upchannel 2		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A.	eeschema (2011−11−27 BZR 3249)−stable	Id: 7/9

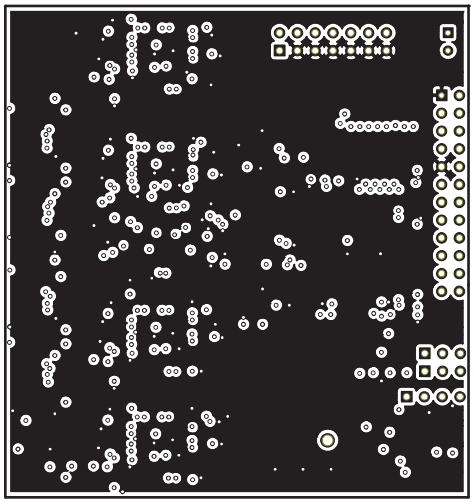


Spintower		
File: upchannel3.sch		
Sheet: /UpChannel_3/		
Title: RPI–Testboard Upchannel 3		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A. eeschema (2011–11–27 BZR 3249)–stable		Id: 8/9

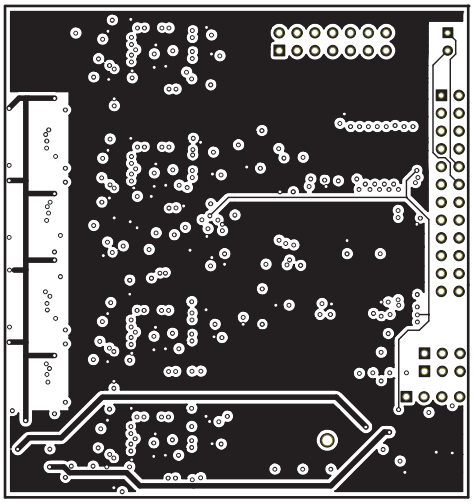

IC1
XC2C256
JTAG
CONN_7X2
P1
TDI
TDO
TCK
TMS
Vaux
PWR_FLAG
DIP08S
S1
SS
SCLK
MOSI
MISO
SCKo
SCKi
WSo
WSi
SDo
SDi
RESET
3V3
1V8
C1 100nF
C2 10nF
C3 100nF
C4 100nF
C5 10nF
C6 10nF
C7 100nF
C8 10nF
C9 100nF
C10 10nF
C11 10nF
C12 100nF
C13 10nF
C14 10nF
C15 100nF
C16 100nF
C17 10nF
C18 10nF
C19 100nF
C20 100nF
R1 3k3
R2 3k3
R3 3k3
R4 3k3
D1 LED
D2 LED
D3 LED
D4 LED
Status3
Status2
Status1
Status0
Spintower
File: logic.sch
Sheet: /Logic/
Title: RPI–Testboard CPLD
Size: A4
Date: 4 oct 2012
Rev:
KiCad E.D.A. eeschema (2011–11–27 BZR 3249)–stable
Id: 9/9



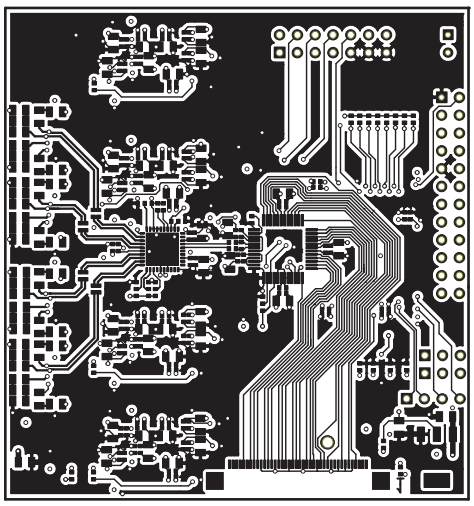
Spintower		
File: rpi.brd Sheet: 1/1		
Title: RPI-Testboard Layer: Top		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A.	pcbnew (2011-11-27 BZR 3249)-stable	Id: 1/1



Spintower
File: rpi.brd
Sheet: 1/1
Title: RPI-Testboard Layer:GND
Size: A4 | Date: 4 oct 2012 | Rev:
KiCad E.D.A. pcbnew (2011-11-27 BZR 3249)-stable | Id: 1/1



Spintower		
File: rpi.brd		
Sheet: 1/1		
Title: RPI-Testboard Layer: VCC		
Size: A4	Date: 4 oct 2012	**Rev:**
KiCad E.D.A. pcbnew (2011-11-27 BZR 3249)-stable		Id: 1/1



Spintower			
File: rpi.brd			
Sheet: 1/1			
Title: RPI-Testboard Layer: Bottom			
Size: A4	Date: 4 oct 2012		**Rev:**
KiCad E.D.A. pcbnew (2011-11-27 BZR 3249)-stable			Id: 1/1

Anhang B

In diesem Abschnitt befindet sich der Verilog-Sourcecode der erstellten Digitaldesigns. Die Programmierung wurde im XXXXXXVIIlinx ISE Design Suite 13.3 durchgeführt.

Fortsetzung auf nächster Seite

fortgesetzt von vorheriger Seite

Übersicht Anhang B – RPI-Testboard	
topmodule→audio_out_handler→sdo_buffer_controller→clk_sdobuffer3_mux	XXXIX
topmodule→audio_out_handler→sdo_buffer_controller→clk_sdobuffer4_mux	XXXIX
topmodule→audio_out_handler→sdo_buffer_demux	L
topmodule→audio_out_handler→sdo_buffer1	XXXIX
topmodule→audio_out_handler→sdo_buffer2	XXXIX
topmodule→audio_out_handler→sdo_buffer3	XXXIX
topmodule→audio_out_handler→sdo_buffer4	XXXIX
topmodule→audio_out_handler→sdo_output_mux	L
topmodule→video_in_handler	XXXIII
topmodule→video_in_handler→csi_pclk_mux	L
topmodule→video_in_handler→csi_vsync_mux	L
topmodule→video_in_handler→csi_hsync_mux	L
topmodule→video_in_handler→csi_d0_mux	L
topmodule→video_in_handler→csi_d1_mux	L
topmodule→video_in_handler→csi_d2_mux	L
topmodule→video_in_handler→csi_d3_mux	L
topmodule→video_in_handler→csi_d4_mux	L
topmodule→video_in_handler→csi_d5_mux	L
topmodule→video_in_handler→csi_d6_mux	L
topmodule→video_in_handler→csi_d7_mux	L

Übersicht Anhang B – Peripheriegerät	
topmodule	XL
topmodule→spi_recovery	XL
topmodule→spi_recovery→miso_synchronizer	LI
topmodule→spi_recovery→sclk_counter	LI
topmodule→spi_recovery→ss_mosi_reprocessing	XLII-XLIII
topmodule→audio_in_synchronizer	XLI
topmodule→audio_in_synchronizer→bclki_synchronizer	LII
topmodule→audio_in_synchronizer→lrclki_synchronizer	LII
topmodule→audio_in_synchronizer→sdin_synchronizer	LII
topmodule→audio_out_synchronizer	XLI-XLII
topmodule→audio_out_synchronizer→bclko_synchronizer	LII
topmodule→audio_out_synchronizer→lrclko_synchronizer	LII
topmodule→audio_out_synchronizer→sdo_buffer_controller	XLIII-XLVII
topmodule→audio_out_synchronizer→counter_controller	XLVII-XLIX
topmodule→audio_out_synchronizer→dc_clk_counter	LII
topmodule→audio_out_synchronizer→tx_start_counter	LII
topmodule→audio_out_synchronizer→tx_counter	LII
topmodule→audio_out_synchronizer→sdo_buffer_demux	L
topmodule→audio_out_synchronizer→sdo_buffer1	XLIX

Fortsetzung auf nächster Seite

fortgesetzt von vorheriger Seite

Übersicht Anhang B – Peripheriegerät	
topmodule→audio_out_synchronizer→sdo_buffer2	XLIX
topmodule→audio_out_synchronizer→sdo_buffer3	XLIX
topmodule→audio_out_synchronizer→sdo_buffer4	XLIX
topmodule→audio_out_synchronizer→sdo_out_mux	L
topmodule→audio_out_synchronizer→sdo_delay	XLIX

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////// TOPMDOLUE START /////////////////////////////////
module topmodule(lcdc_pclk,reset,
  data_in_sel, ss, sclk, mosi, miso, rpi_sclk, rpi_mosi_ss,
  rpi_0_miso, rpi_1_miso, rpi_2_miso, rpi_3_miso, audio_in_sel, wsi, scki, sdi,
  rpi_0_wsi, rpi_1_wsi, rpi_2_wsi, rpi_3_wsi,
  rpi_0_scki, rpi_1_scki, rpi_2_scki, rpi_3_scki,
  rpi_0_sdi, rpi_1_sdi, rpi_2_sdi, rpi_3_sdi,
  audio_out_sel, wso, scko, sdo, rpi_sdo,
  rpi_0_wso, rpi_1_wso, rpi_2_wso, rpi_3_wso,
  rpi_0_scko, rpi_1_scko, rpi_2_scko, rpi_3_scko,
  video_in_sel, csi_pclk, csi_vsync, csi_hsync, csi_d0, csi_d1, csi_d2,
  csi_d3, csi_d4, csi_d5, csi_d6, csi_d7,
  rpi_0_pclk, rpi_1_pclk, rpi_2_pclk, rpi_3_pclk,
  rpi_0_vsync, rpi_1_vsync, rpi_2_vsync, rpi_3_vsync,
  rpi_0_hsync, rpi_1_hsync, rpi_2_hsync, rpi_3_hsync,
  rpi_0_d0, rpi_1_d0, rpi_2_d0, rpi_3_d0, rpi_0_d1, rpi_1_d1, rpi_2_d1, rpi_3_d1,
  rpi_0_d2, rpi_1_d2, rpi_2_d2, rpi_3_d2, rpi_0_d3, rpi_1_d3, rpi_2_d3, rpi_3_d3,
  rpi_0_d4, rpi_1_d4, rpi_2_d4, rpi_3_d4, rpi_0_d5, rpi_1_d5, rpi_2_d5, rpi_3_d5,
  rpi_0_d6, rpi_1_d6, rpi_2_d6, rpi_3_d6, rpi_0_d7, rpi_1_d7, rpi_2_d7, rpi_3_d7,
  status0, status1, status2, status3);

  input lcdc_pclk, reset;
  output status0, status1, status2, status3;

  //SPI In- and Outputs
  input ss, sclk, mosi, rpi_0_miso, rpi_1_miso, rpi_2_miso, rpi_3_miso;
  input [1:0] data_in_sel;
  output miso, rpi_sclk, rpi_mosi_ss;

  //Audio-In In- and Outputs
  input rpi_0_wsi, rpi_1_wsi, rpi_2_wsi, rpi_3_wsi;
  input rpi_0_scki, rpi_1_scki, rpi_2_scki, rpi_3_scki;
  input rpi_0_sdi, rpi_1_sdi, rpi_2_sdi, rpi_3_sdi;
  input [1:0] audio_in_sel;
  output wsi, scki, sdi;

  //Audio-Out In- and Outputs
  input rpi_0_wso, rpi_1_wso, rpi_2_wso, rpi_3_wso, sdo;
  input rpi_0_scko, rpi_1_scko, rpi_2_scko, rpi_3_scko;
  input [1:0] audio_out_sel;
  output wso, scko, rpi_sdo;

  //Video-In In- and Outputs
  input rpi_0_pclk, rpi_1_pclk, rpi_2_pclk, rpi_3_pclk;
  input rpi_0_vsync, rpi_1_vsync, rpi_2_vsync, rpi_3_vsync;
  input rpi_0_hsync, rpi_1_hsync, rpi_2_hsync, rpi_3_hsync;
  input rpi_0_d0, rpi_1_d0, rpi_2_d0, rpi_3_d0;
  input rpi_0_d1, rpi_1_d1, rpi_2_d1, rpi_3_d1;
  input rpi_0_d2, rpi_1_d2, rpi_2_d2, rpi_3_d2;
  input rpi_0_d3, rpi_1_d3, rpi_2_d3, rpi_3_d3;
  input rpi_0_d4, rpi_1_d4, rpi_2_d4, rpi_3_d4;
  input rpi_0_d5, rpi_1_d5, rpi_2_d5, rpi_3_d5;
  input rpi_0_d6, rpi_1_d6, rpi_2_d6, rpi_3_d6;
  input rpi_0_d7, rpi_1_d7, rpi_2_d7, rpi_3_d7;
  input [1:0] video_in_sel;
  output csi_pclk, csi_vsync, csi_hsync, csi_d0, csi_d1;
  output csi_d2, csi_d3, csi_d4, csi_d5, csi_d6, csi_d7;

  spi_handler spi_handler (.dc_clk(lcdc_pclk),.reset(reset),
    .data_in_sel(data_in_sel),.ss(ss),.sclk(sclk),.mosi(mosi),
    .rpi_0_miso(rpi_0_miso),.rpi_1_miso(rpi_1_miso),.rpi_2_miso(rpi_2_miso),
    .rpi_3_miso(rpi_3_miso),.sclk_o(rpi_sclk),.mosi_ss_o(rpi_mosi_ss),
    .miso(miso),.ready(status0));

  audio_in_handler audio_in_handler(.audio_in_sel(audio_in_sel),
    .rpi_0_wsi(rpi_0_wsi),.rpi_1_wsi(rpi_1_wsi),.rpi_2_wsi(rpi_2_wsi),
    .rpi_3_wsi(rpi_3_wsi),.rpi_0_scki(rpi_0_scki),.rpi_1_scki(rpi_1_scki),
    .rpi_2_scki(rpi_2_scki),.rpi_3_scki(rpi_3_scki),.rpi_0_sdi(rpi_0_sdi),
    .rpi_1_sdi(rpi_1_sdi),.rpi_2_sdi(rpi_2_sdi),.rpi_3_sdi(rpi_3_sdi),
    .wsi(wsi),.scki(scki),.sdi(sdi));
```

```
  audio_out_handler audio_out_handler(.reset(reset),.dc_clk(lcdc_pclk),
    .rpi_0_pclk(rpi_0_pclk),.rpi_1_pclk(rpi_1_pclk),.rpi_2_pclk(rpi_2_pclk),
    .rpi_3_pclk(rpi_2_pclk),.audio_out_sel(audio_out_sel),
    .rpi_0_wso(rpi_0_wso),.rpi_1_wso(rpi_1_wso),.rpi_2_wso(rpi_2_wso),
    .rpi_3_wso(rpi_3_wso),.rpi_0_scko(rpi_0_scko),.rpi_1_scko(rpi_1_scko),
    .rpi_2_scko(rpi_2_scko),.rpi_3_scko(rpi_3_scko),.sdo(sdo),
    .wso(wso),.scko(scko),.rpi_sdo(rpi_sdo));

  video_in_handler video_in_handler(.video_in_sel(video_in_sel),
    .rpi_0_pclk(rpi_0_pclk),.rpi_1_pclk(rpi_1_pclk),.rpi_2_pclk(rpi_2_pclk),
    .rpi_3_pclk(rpi_3_pclk),.rpi_0_vsync(rpi_0_vsync),.rpi_1_vsync(rpi_1_vsync),
    .rpi_2_vsync(rpi_2_vsync),.rpi_3_vsync(rpi_3_vsync),
    .rpi_0_hsync(rpi_0_hsync),.rpi_1_hsync(rpi_1_hsync),
    .rpi_2_hsync(rpi_2_hsync),.rpi_3_hsync(rpi_3_hsync),
    .rpi_0_d0(rpi_0_d0),.rpi_1_d0(rpi_1_d0),.rpi_2_d0(rpi_2_d0),
    .rpi_3_d0(rpi_3_d0),.rpi_0_d1(rpi_0_d1),.rpi_1_d1(rpi_1_d1),
    .rpi_2_d1(rpi_2_d1),.rpi_3_d1(rpi_3_d1),.rpi_0_d2(rpi_0_d2),
    .rpi_1_d2(rpi_1_d2),.rpi_2_d2(rpi_2_d2),.rpi_3_d2(rpi_3_d2),
    .rpi_0_d3(rpi_0_d3),.rpi_1_d3(rpi_1_d3),.rpi_2_d3(rpi_2_d3),
    .rpi_3_d3(rpi_3_d3),.rpi_0_d4(rpi_0_d4),.rpi_1_d4(rpi_1_d4),
    .rpi_2_d4(rpi_2_d4),.rpi_3_d4(rpi_3_d4),.rpi_0_d5(rpi_0_d5),
    .rpi_1_d5(rpi_1_d5),.rpi_2_d5(rpi_2_d5),.rpi_3_d5(rpi_3_d5),
    .rpi_0_d6(rpi_0_d6),.rpi_1_d6(rpi_1_d6),.rpi_2_d6(rpi_2_d6),
    .rpi_3_d6(rpi_3_d6),.rpi_0_d7(rpi_0_d7),.rpi_1_d7(rpi_1_d7),
    .rpi_2_d7(rpi_2_d7),.rpi_3_d7(rpi_3_d7),.csi_pclk(csi_pclk),
    .csi_vsync(csi_vsync),.csi_hsync(csi_hsync),.csi_d0(csi_d0),.csi_d1(csi_d1),
    .csi_d2(csi_d2),.csi_d3(csi_d3),.csi_d4(csi_d4),.csi_d5(csi_d5),
    .csi_d6(csi_d6),.csi_d7(csi_d7));

  assign status1 = data_in_sel[1];
  assign status2 = audio_in_sel[1];
  assign status3 = video_in_sel[1];
endmodule
////////////////////////////// TOPMDOLUE END ////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////////////////////////////// SPI_HANDLER START ////////////////////////////////
module spi_handler (dc_clk, reset, data_in_sel, ss, sclk, mosi, rpi_0_miso,
                    rpi_1_miso, rpi_2_miso, rpi_3_miso, sclk_o, mosi_ss_o, miso,
              ready);

  input dc_clk, reset, ss, sclk, mosi;
  input rpi_0_miso, rpi_1_miso, rpi_2_miso, rpi_3_miso;
  input [1:0] data_in_sel;

  output sclk_o, mosi_ss_o, miso, ready;

  wire ss_sync, sclk_sync, mosi_sync;
  wire [3:0] init_cnt;
  wire [2:0] sclk_cnt;

  reg en_init_cnt;
  reg last_mosi;

  mux4to1 miso_mux (.sel(data_in_sel),.din0(rpi_0_miso),.din1(rpi_1_miso),
    .din2(rpi_2_miso),.din3(rpi_3_miso),.dout(miso));

  synchronizer_with_DETFF #(.DELAY(2)) ss_synchronizer (.clk(dc_clk),
    .reset(reset),.din(ss),.dout(ss_sync));

  synchronizer_with_DETFF #(.DELAY(2)) sclk_synchronizer (.clk(dc_clk),
    .reset(reset),.din(sclk),.dout(sclk_sync));

  synchronizer_with_DETFF #(.DELAY(2)) mosi_synchronizer (.clk(dc_clk),
    .reset(reset),.din(mosi),.dout(mosi_sync));

  delayline #(.DELAY(6)) sclk_delayline (.reset(reset),.clk(dc_clk),
    .din(sclk_sync),.dout(sclk_o));

  counter_async_clr #(.WIDTH(4)) init_counter (.clk(dc_clk),.reset(reset),
    .enable(en_init_cnt & !init_cnt[3]),.counter(init_cnt));
```

```
  counter_async_clr #(.WIDTH(3)) sclk_counter (.clk(!sclk_o),.reset(reset),
    .enable(init_cnt[3]),.counter(sclk_cnt));

  mosi_ss_processing mosi_ss_processing (.clk(dc_clk),.reset(reset),
    .init_cnt(init_cnt),.sclk_cnt(sclk_cnt),.ss(ss_sync),.sclk(sclk_o),
    .mosi(mosi_sync),.last_mosi(last_mosi),.mosi_ss(mosi_ss_o));

  always @(posedge dc_clk) begin
    if(reset)
      en_init_cnt <= 1'b0;
    else
      en_init_cnt <= 1'b1;
  end

  always @(posedge sclk_sync) begin
    if(reset)
      last_mosi <= 1'b0;
    else
      last_mosi <= mosi_sync;
  end

  assign ready = !init_cnt[3];

endmodule
////////////////////////////// SPI_HANDLER END ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////
////////////////////////////// AUDIO_IN_HANDLER START ////////////////////////////
module audio_in_handler(audio_in_sel,
                rpi_0_wsi, rpi_1_wsi, rpi_2_wsi, rpi_3_wsi,
                rpi_0_scki, rpi_1_scki, rpi_2_scki, rpi_3_scki,
                rpi_0_sdi, rpi_1_sdi, rpi_2_sdi, rpi_3_sdi,
                wsi, scki, sdi);

  input [1:0] audio_in_sel;
  input rpi_0_wsi, rpi_1_wsi, rpi_2_wsi, rpi_3_wsi;
  input rpi_0_scki, rpi_1_scki, rpi_2_scki, rpi_3_scki;
  input rpi_0_sdi, rpi_1_sdi, rpi_2_sdi, rpi_3_sdi;

  output reg wsi;
  output scki, sdi;

  mux4to1 wsi_mux (.sel(audio_in_sel),.din0(rpi_0_wsi),.din1(rpi_1_wsi),
    .din2(rpi_2_wsi),.din3(rpi_3_wsi),.dout(wsi_int));

  mux4to1 scki_mux (.sel(audio_in_sel),.din0(rpi_0_scki),.din1(rpi_1_scki),
    .din2(rpi_2_scki),.din3(rpi_3_scki),.dout(scki));

  mux4to1 sdi_mux (.sel(audio_in_sel),.din0(rpi_0_sdi),.din1(rpi_1_sdi),
    .din2(rpi_2_sdi),.din3(rpi_3_sdi),.dout(sdi));

  always @(negedge scki) begin
    if(wsi_int)
      wsi <= 1'b1;
    else
      wsi <= 1'b0;
  end

endmodule
////////////////////////////// AUDIO_IN_HANDLER END ///////////////////////////////
//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////
////////////////////////////// AUDIO_OUT_HANDLER START ////////////////////////////
module audio_out_handler(reset, dc_clk, rpi_0_pclk, rpi_1_pclk, rpi_2_pclk,
                rpi_3_pclk, audio_out_sel,
                rpi_0_wso, rpi_1_wso, rpi_2_wso, rpi_3_wso,
                rpi_0_scko, rpi_1_scko, rpi_2_scko, rpi_3_scko, sdo,
                wso, scko, rpi_sdo);
```

```
  input reset, dc_clk, rpi_0_pclk, rpi_1_pclk, rpi_2_pclk, rpi_3_pclk, sdo;
  input [1:0] audio_out_sel;
  input rpi_0_wso, rpi_1_wso, rpi_2_wso, rpi_3_wso;
  input rpi_0_scko, rpi_1_scko, rpi_2_scko, rpi_3_scko;

  output wso, scko;
  output reg rpi_sdo;

  wire uc_clk;
  wire wso_int, scko_int;
  wire wso_uc_sync, wso_dc_sync;
  wire enable_sdo;
  wire clr_buffer1, clr_buffer2, clr_buffer3, clr_buffer4;
  wire clk_buffer1, clk_buffer2, clk_buffer3, clk_buffer4;
  wire [1:0] sdo_buffer_sel;
  wire sdo_buffer1_in, sdo_buffer2_in, sdo_buffer3_in, sdo_buffer4_in;
  wire sdo_buffer1_out, sdo_buffer2_out, sdo_buffer3_out, sdo_buffer4_out;

  mux4to1 uc_clk_mux (.sel(audio_out_sel),.din0(rpi_0_pclk),.din1(rpi_1_pclk),
    .din2(rpi_2_pclk),.din3(rpi_3_pclk),.dout(uc_clk));

  mux4to1 wso_mux (.sel(audio_out_sel),.din0(rpi_0_wso),.din1(rpi_1_wso),
    .din2(rpi_2_wso),.din3(rpi_3_wso),.dout(wso_int));

  mux4to1 scko_mux (.sel(audio_out_sel),.din0(rpi_0_scko),.din1(rpi_1_scko),
    .din2(rpi_2_scko),.din3(rpi_3_scko),.dout(scko_int));

  wso_scko_synchronizer wso_scko_synchronizer(.upchannel_clk(uc_clk),
    .downchannel_clk(dc_clk),.reset(reset),.scko_int(scko_int),
    .wso_int(wso_int),.scko(scko),.wso(wso),.wso_uc_sync(wso_uc_sync),
    .wso_dc_sync(wso_dc_sync),.enable_sdo(enable_sdo));

  sdo_buffer_controller sdo_buffer_controller (.scko(scko),
    .downchannel_clk(dc_clk),.reset(reset),.wso_int(wso_int),
    .wso_uc_sync(wso_uc_sync),.wso_dc_sync(wso_dc_sync),.enable_sdo(enable_sdo),
    .clear_buffer1(clr_buffer1),.clear_buffer2(clr_buffer2),
    .clear_buffer3(clr_buffer3),.clear_buffer4(clr_buffer4),
    .clk_buffer1(clk_buffer1),.clk_buffer2(clk_buffer2),
    .clk_buffer3(clk_buffer3),.clk_buffer4(clk_buffer4),.sel(sdo_buffer_sel));

  demux1to4 sdo_buffer_demux (.sel(sdo_buffer_sel),.din(sdo & enable_sdo),
    .dout0(sdo_buffer1_in),.dout1(sdo_buffer2_in),.dout2(sdo_buffer3_in),
    .dout3(sdo_buffer4_in));

  audio_buffer #(.WIDTH(20)) sdo_buffer1 (.reset(clr_buffer1),.clk(clk_buffer1),
    .din(sdo_buffer1_in),.dout(sdo_buffer1_out));

  audio_buffer #(.WIDTH(20)) sdo_buffer2 (.reset(clr_buffer2),.clk(clk_buffer2),
    .din(sdo_buffer2_in),.dout(sdo_buffer2_out));

  audio_buffer #(.WIDTH(20)) sdo_buffer3 (.reset(clr_buffer3),.clk(clk_buffer3),
    .din(sdo_buffer3_in),.dout(sdo_buffer3_out));

  audio_buffer #(.WIDTH(20)) sdo_buffer4 (.reset(clr_buffer4),.clk(clk_buffer4),
    .din(sdo_buffer4_in),.dout(sdo_buffer4_out));

  mux4to1 sdo_output_mux (.sel(sdo_buffer_sel),
    .din0(!wso_dc_sync & sdo_buffer4_out),.din1(wso_dc_sync & sdo_buffer1_out),
    .din2(!wso_dc_sync & sdo_buffer2_out),.din3(wso_dc_sync & sdo_buffer3_out),
    .dout(sdo2output));

  always @(negedge dc_clk) begin
    if(reset)
      rpi_sdo <= 1'b0;
    else
      rpi_sdo <= sdo2output;
  end

endmodule
//////////////////////////// AUDIO_OUT_HANDLER END ////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////
//////////////////////////// VIDEO_IN_HANDLER START ////////////////////////////////
module video_in_handler(video_in_sel,
                rpi_0_pclk, rpi_1_pclk, rpi_2_pclk, rpi_3_pclk,
                rpi_0_vsync, rpi_1_vsync, rpi_2_vsync, rpi_3_vsync,
                rpi_0_hsync, rpi_1_hsync, rpi_2_hsync, rpi_3_hsync,
                rpi_0_d0, rpi_1_d0, rpi_2_d0, rpi_3_d0,
                rpi_0_d1, rpi_1_d1, rpi_2_d1, rpi_3_d1,
                rpi_0_d2, rpi_1_d2, rpi_2_d2, rpi_3_d2,
                rpi_0_d3, rpi_1_d3, rpi_2_d3, rpi_3_d3,
                rpi_0_d4, rpi_1_d4, rpi_2_d4, rpi_3_d4,
                rpi_0_d5, rpi_1_d5, rpi_2_d5, rpi_3_d5,
                rpi_0_d6, rpi_1_d6, rpi_2_d6, rpi_3_d6,
                rpi_0_d7, rpi_1_d7, rpi_2_d7, rpi_3_d7,
                csi_pclk, csi_vsync, csi_hsync, csi_d0, csi_d1, csi_d2,
                csi_d3, csi_d4, csi_d5, csi_d6, csi_d7);

  input [1:0] video_in_sel;
  input rpi_0_pclk, rpi_1_pclk, rpi_2_pclk, rpi_3_pclk;
  input rpi_0_vsync, rpi_1_vsync, rpi_2_vsync, rpi_3_vsync;
  input rpi_0_hsync, rpi_1_hsync, rpi_2_hsync, rpi_3_hsync;
  input rpi_0_d0, rpi_1_d0, rpi_2_d0, rpi_3_d0;
  input rpi_0_d1, rpi_1_d1, rpi_2_d1, rpi_3_d1;
  input rpi_0_d2, rpi_1_d2, rpi_2_d2, rpi_3_d2;
  input rpi_0_d3, rpi_1_d3, rpi_2_d3, rpi_3_d3;
  input rpi_0_d4, rpi_1_d4, rpi_2_d4, rpi_3_d4;
  input rpi_0_d5, rpi_1_d5, rpi_2_d5, rpi_3_d5;
  input rpi_0_d6, rpi_1_d6, rpi_2_d6, rpi_3_d6;
  input rpi_0_d7, rpi_1_d7, rpi_2_d7, rpi_3_d7;

  output csi_pclk, csi_vsync, csi_hsync, csi_d0, csi_d1, csi_d2, csi_d3, csi_d4;
  output csi_d5, csi_d6, csi_d7;

  mux4to1 csi_pclk_mux (.sel(video_in_sel),.din0(rpi_0_pclk),.din1(rpi_1_pclk),
    .din2(rpi_2_pclk),.din3(rpi_3_pclk),.dout(csi_pclk));

  mux4to1 csi_vsync_mux (.sel(video_in_sel),.din0(rpi_0_vsync),.din1(rpi_1_vsync),
    .din2(rpi_2_vsync),.din3(rpi_3_vsync),.dout(csi_vsync));

  mux4to1 csi_hsync_mux (.sel(video_in_sel),.din0(rpi_0_hsync),.din1(rpi_1_hsync),
    .din2(rpi_2_hsync),.din3(rpi_3_hsync),.dout(csi_hsync));

  mux4to1 csi_d0_mux (.sel(video_in_sel),.din0(rpi_0_d0),.din1(rpi_1_d0),
    .din2(rpi_2_d0),.din3(rpi_3_d0),.dout(csi_d0));

  mux4to1 csi_d1_mux (.sel(video_in_sel),.din0(rpi_0_d1),.din1(rpi_1_d1),
    .din2(rpi_2_d1),.din3(rpi_3_d1),.dout(csi_d1));

  mux4to1 csi_d2_mux (.sel(video_in_sel),.din0(rpi_0_d2),.din1(rpi_1_d2),
    .din2(rpi_2_d2),.din3(rpi_3_d2),.dout(csi_d2));

  mux4to1 csi_d3_mux (.sel(video_in_sel),.din0(rpi_0_d3),.din1(rpi_1_d3),
    .din2(rpi_2_d3),.din3(rpi_3_d3),.dout(csi_d3));

  mux4to1 csi_d4_mux (.sel(video_in_sel),.din0(rpi_0_d4),.din1(rpi_1_d4),
    .din2(rpi_2_d4),.din3(rpi_3_d4),.dout(csi_d4));

  mux4to1 csi_d5_mux (.sel(video_in_sel),.din0(rpi_0_d5),.din1(rpi_1_d5),
    .din2(rpi_2_d5),.din3(rpi_3_d5),.dout(csi_d5));

  mux4to1 csi_d6_mux (.sel(video_in_sel),.din0(rpi_0_d6),.din1(rpi_1_d6),
    .din2(rpi_2_d6),.din3(rpi_3_d6),.dout(csi_d6));

  mux4to1 csi_d7_mux (.sel(video_in_sel),.din0(rpi_0_d7),.din1(rpi_1_d7),
    .din2(rpi_2_d7),.din3(rpi_3_d7),.dout(csi_d7));

endmodule
//////////////////////////// VIDEO_IN_HANDLER END ////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////// DELAYLINE START /////////////////////////////////
module delayline(reset, clk, din, dout);

  parameter DELAY = 6;

  input reset, clk, din;

  output dout;

  reg [DELAY-1:0] delayline;

   assign dout = delayline[DELAY-1];

  always@(posedge clk) begin
    if(reset) begin
      delayline <= 0;
    end
    else begin
      delayline <= delayline << 1;
      delayline[0] <= din;
    end
  end


endmodule
////////////////////////////// DELAYLINE END ///////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////////////////////////// MOSI_SS_PROCESSING START ////////////////////////////
module mosi_ss_processing(clk, reset, init_cnt, sclk_cnt, ss, sclk, mosi,
  last_mosi, mosi_ss);

  input clk, reset, ss, sclk, mosi, last_mosi;
  input [2:0] sclk_cnt;
  input [3:0] init_cnt;

  output mosi_ss;

  `define IDLE     4'b0000
  `define MOSI_SS_0  4'b0001
  `define MOSI_SS_2  4'b0011
  `define MOSI_SS_4  4'b0101
  `define MOSI_SDS_0 4'b0111
  `define MOSI_SDS_2 4'b1001
  `define MOSI_SDS_4 4'b1011
  `define MOSI_SDS_6 4'b1101
  `define WAIT     4'b1111

   reg [3:0] current_state;
   reg [3:0] next_state;

   reg select;
   reg mosi_ss_sel;

   //fsm synchrouneous process
   always @(posedge clk) begin
    if (reset)
      current_state <= `IDLE;
    else
      current_state <= next_state;
   end

   //fsm combinatorial process
   always @(current_state or ss or sclk_cnt or init_cnt or last_mosi) begin
    next_state <= `IDLE;
    case (current_state)
      `IDLE :
        begin
          if(!ss & init_cnt > 4'b0111)
            next_state <= `MOSI_SS_0;
```

```
        else
          next_state <= `IDLE;
      end
    `MOSI_SS_0 :
      begin
        next_state <= `MOSI_SS_2;
      end
    `MOSI_SS_2 :
      begin
        next_state <= `MOSI_SS_4;
      end
    `MOSI_SS_4 :
      begin
        next_state <= `WAIT;
      end
    `WAIT :
      begin
        if(!ss | (ss & sclk_cnt[2] & sclk_cnt[1] & sclk_cnt[0]))
          next_state <= `WAIT;
        else if(sclk_cnt == 3'b000 & last_mosi)
          next_state <= `MOSI_SDS_0;
        else if(sclk_cnt == 3'b000 & !last_mosi)
          next_state <= `MOSI_SDS_6;
        else
          next_state <= `IDLE;
      end
    `MOSI_SDS_0 :
      begin
        next_state <= `MOSI_SDS_2;
      end
    `MOSI_SDS_2 :
      begin
        next_state <= `MOSI_SDS_4;
      end
    `MOSI_SDS_4 :
      begin
        next_state <= `IDLE;
      end
    `MOSI_SDS_6 :
      begin
        next_state <= `MOSI_SDS_0;
      end
    default : next_state <= `IDLE;
  endcase
 end

//fsm ouput logic
always @(posedge clk) begin
  if(reset) begin
    select <= 1'b1;
    mosi_ss_sel <= 1'b0;
  end

  else begin
    case (current_state)
      `IDLE :
        begin
          select <= 1'b1;
          mosi_ss_sel <= 1'b0;
        end
      `MOSI_SS_0 :
        begin
          select <= 1'b0;
          mosi_ss_sel <= 1'b0;
        end
      `MOSI_SS_2 :
        begin
          select <= 1'b1;
          mosi_ss_sel <= 1'b0;
        end
      `MOSI_SS_4 :
        begin
```

```
            select <= 1'b0;
            mosi_ss_sel <= 1'b0;
          end
        `WAIT :
          begin
            select <= 1'b1;
            mosi_ss_sel <= 1'b1;
          end
        `MOSI_SDS_0 :
          begin
            select <= 1'b0;
            mosi_ss_sel <= 1'b0;
          end
        `MOSI_SDS_2 :
          begin
            select <= 1'b1;
            mosi_ss_sel <= 1'b0;
          end
        `MOSI_SDS_4 :
          begin
            select <= 1'b0;
            mosi_ss_sel <= 1'b0;
          end
        `MOSI_SDS_6 :
          begin
            select <= 1'b1;
            mosi_ss_sel <= 1'b0;
           end
        default :
          begin
            select <= 1'b1;
            mosi_ss_sel <= 1'b0;
          end
      endcase
    end
  end

  assign mosi_ss = mosi_ss_sel ? mosi : select;

endmodule
//////////////////////////// MOSI_SS_PROCESSING END ////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////
/////////////////////////// WSO_SCKO_SYNCHRONIZER START ////////////////////////////
module wso_scko_synchronizer (upchannel_clk, downchannel_clk, reset, scko_int,
                    wso_int, scko, wso, wso_uc_sync, wso_dc_sync,
                    enable_sdo);

  input upchannel_clk, downchannel_clk, reset, scko_int, wso_int;
  output scko, wso, wso_uc_sync, wso_dc_sync, enable_sdo;

  reg wso;
  wire [4:0] scko_cnt;

  synchronizer #(.DELAY(3)) scko_synchronizer (.clk(upchannel_clk),
    .reset(reset),.din(scko_int),.dout(scko));

  synchronizer #(.DELAY(2)) wso_uc_synchronizer (.clk(upchannel_clk),
    .reset(reset),.din(wso_int),.dout(wso_uc_sync));

  always @(negedge scko) begin
    if(wso_uc_sync)
      wso <= 1'b1;
    else
      wso <= 1'b0;
  end

  synchronizer #(.DELAY(2)) wso_dc_synchronizer (.clk(downchannel_clk),
    .reset(reset),.din(wso_int),.dout(wso_dc_sync));
```

```
  counter_sync_clr #(.WIDTH(5)) scko_counter (.clk(!scko),
    .reset(wso_int & !scko_cnt[4]),.enable(!wso_int & !scko_cnt[4]),
    .counter(scko_cnt));

  assign enable_sdo = scko_cnt[4];

endmodule
//////////////////////////// WSO_SCKO_SYNCHRONIZER END ////////////////////////////
////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////
/////////////////////////// SDO_BUFFER_CONTROLLER START ////////////////////////////
module sdo_buffer_controller (scko, downchannel_clk, reset, wso_int, wso_uc_sync,
                    wso_dc_sync, enable_sdo, clear_buffer1,
                    clear_buffer2, clear_buffer3, clear_buffer4,
                    clk_buffer1, clk_buffer2, clk_buffer3, clk_buffer4,
                    sel);

  input scko, downchannel_clk, reset, wso_int, wso_uc_sync, wso_dc_sync;
  input enable_sdo;

  output clear_buffer1, clear_buffer2, clear_buffer3, clear_buffer4;
  output clk_buffer1, clk_buffer2, clk_buffer3, clk_buffer4;
  output reg [1:0] sel;

  wire [4:0] dc_clk_cnt1, dc_clk_cnt2, dc_clk_cnt3, dc_clk_cnt4;
  wire dc_clk_buffer1, dc_clk_buffer2, dc_clk_buffer3, dc_clk_buffer4;
  wire scko_clk_buffer1, scko_clk_buffer2, scko_clk_buffer3, scko_clk_buffer4;

  reg disable_dc_cnt_buffer1, disable_dc_cnt_buffer2;
  reg disable_dc_cnt_buffer3, disable_dc_cnt_buffer4;
  reg delayed_disable_dc_cnt_buffer1, delayed_disable_dc_cnt_buffer2;
  reg delayed_disable_dc_cnt_buffer3, delayed_disable_dc_cnt_buffer4;

  always @(posedge wso_uc_sync or negedge wso_uc_sync or posedge reset) begin
    if(reset)
      sel <= 2'b00;
    else begin
      if(enable_sdo)
        sel <= sel + 1'b1;
      else
        sel <= sel;
    end
  end

  assign clear_buffer1 = wso_int & sel[1] & sel[0];
  assign clear_buffer2 = !wso_int & !sel[1] & !sel[0];
  assign clear_buffer3 = wso_int & !sel[1] & sel[0];
  assign clear_buffer4 = !wso_int & sel[1] & !sel[0];


  counter_sync_clr #(.WIDTH(5)) dc_cnt_buffer1 (.clk(downchannel_clk),
    .reset(!wso_dc_sync & sel[1] & !sel[0]),
    .enable(wso_dc_sync & !sel[1] & sel[0] & disable_dc_cnt_buffer1),
    .counter(dc_clk_cnt1));

  counter_sync_clr #(.WIDTH(5)) dc_cnt_buffer2 (.clk(downchannel_clk),
    .reset(wso_dc_sync & sel[1] & sel[0]),
    .enable(!wso_dc_sync & sel[1] & !sel[0] & disable_dc_cnt_buffer2),
    .counter(dc_clk_cnt2));

  counter_sync_clr #(.WIDTH(5)) dc_cnt_buffer3 (.clk(downchannel_clk),
    .reset(!wso_dc_sync & !sel[1] & !sel[0]),
    .enable(wso_dc_sync & sel[1] & sel[0] & disable_dc_cnt_buffer3),
    .counter(dc_clk_cnt3));

  counter_sync_clr #(.WIDTH(5)) dc_cnt_buffer4 (.clk(downchannel_clk),
    .reset(wso_dc_sync & !sel[1] & sel[0]),
    .enable(!wso_dc_sync & !sel[1] & !sel[0] & disable_dc_cnt_buffer4),
    .counter(dc_clk_cnt4));
```

```
//disable counter after 20 clk cykles
always @(negedge downchannel_clk) begin
  if(reset)
    disable_dc_cnt_buffer1 <= 1'b1;
  else
    disable_dc_cnt_buffer1 <= !(dc_clk_cnt1[4] & !dc_clk_cnt1[3] &
    !dc_clk_cnt1[2] & dc_clk_cnt1[1] & dc_clk_cnt1[0]);
end

//delay disable to next posedge clk to get proper clk signal for buffers
always @(posedge downchannel_clk) begin
  if(reset)
    delayed_disable_dc_cnt_buffer1 <= 1'b1;
  else
    delayed_disable_dc_cnt_buffer1 <= disable_dc_cnt_buffer1;
end

//disable counter after 20 clk cykles
always @(negedge downchannel_clk) begin
  if(reset)
    disable_dc_cnt_buffer2 <= 1'b1;
  else
    disable_dc_cnt_buffer2<= !(dc_clk_cnt2[4] & !dc_clk_cnt2[3] &
    !dc_clk_cnt2[2] & dc_clk_cnt2[1] & dc_clk_cnt2[0]);
end

//delay disable to next posedge clk to get proper clk signal for buffers
always @(posedge downchannel_clk) begin
  if(reset)
    delayed_disable_dc_cnt_buffer2 <= 1'b1;
  else
    delayed_disable_dc_cnt_buffer2 <= disable_dc_cnt_buffer2;
end

//disable counter after 20 clk cykles
always @(negedge downchannel_clk) begin
  if(reset)
    disable_dc_cnt_buffer3 <= 1'b1;
  else
    disable_dc_cnt_buffer3 <= !(dc_clk_cnt3[4] & !dc_clk_cnt3[3] &
    !dc_clk_cnt3[2] & dc_clk_cnt3[1] & dc_clk_cnt3[0]);
end

//delay disable to next posedge clk to get proper clk signal for buffers
always @(posedge downchannel_clk) begin
  if(reset)
    delayed_disable_dc_cnt_buffer3 <= 1'b1;
  else
    delayed_disable_dc_cnt_buffer3 <= disable_dc_cnt_buffer3;
end

//disable counter after 20 clk cykles
always @(negedge downchannel_clk) begin
  if(reset)
    disable_dc_cnt_buffer4 <= 1'b1;
  else
    disable_dc_cnt_buffer4 <= !(dc_clk_cnt4[4] & !dc_clk_cnt4[3] &
    !dc_clk_cnt4[2] & dc_clk_cnt4[1] & dc_clk_cnt4[0]);
end

//delay disable to next posedge clk to get proper clk signal for buffers
always @(posedge downchannel_clk) begin
  if(reset)
    delayed_disable_dc_cnt_buffer4 <= 1'b1;
  else
    delayed_disable_dc_cnt_buffer4 <= disable_dc_cnt_buffer4;
end

assign dc_clk_buffer1 = !downchannel_clk & wso_dc_sync &
                delayed_disable_dc_cnt_buffer1 & !sel[1] & sel[0];
assign dc_clk_buffer2 = !downchannel_clk & !wso_dc_sync &
                delayed_disable_dc_cnt_buffer2 & sel[1] & !sel[0];
```

```
  assign dc_clk_buffer3 = !downchannel_clk & wso_dc_sync &
                  delayed_disable_dc_cnt_buffer3 & sel[1] & sel[0];
  assign dc_clk_buffer4 = !downchannel_clk & !wso_dc_sync &
                  delayed_disable_dc_cnt_buffer4 & !sel[1] & !sel[0];

  assign scko_clk_buffer1 = scko & !wso_int& !sel[1] & !sel[0];
  assign scko_clk_buffer2 = scko & wso_int & !sel[1] & sel[0];
  assign scko_clk_buffer3 = scko & !wso_int & sel[1] & !sel[0];
  assign scko_clk_buffer4 = scko & wso_int & sel[1] & sel[0];

  mux2to1 clk_sdobuffer1_mux (.sel(wso_uc_sync),.din0(scko_clk_buffer1),
                    .din1(dc_clk_buffer1),.dout(clk_buffer1));
  mux2to1 clk_sdobuffer2_mux (.sel(!wso_uc_sync),.din0(scko_clk_buffer2),
                    .din1(dc_clk_buffer2),.dout(clk_buffer2));
  mux2to1 clk_sdobuffer3_mux (.sel(wso_uc_sync),.din0(scko_clk_buffer3),
                    .din1(dc_clk_buffer3),.dout(clk_buffer3));
  mux2to1 clk_sdobuffer4_mux (.sel(!wso_uc_sync),.din0(scko_clk_buffer4),
                    .din1(dc_clk_buffer4),.dout(clk_buffer4));

endmodule
////////////////////////// SDO_BUFFER_CONTROLLER END //////////////////////////////
//////////////////////////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////////////////////////
////////////////////////////////// MUX2TO1 START //////////////////////////////////
module mux2to1(sel, din0, din1, dout);

  input sel, din0, din1;

  output reg dout;

  always @(din0 or din1 or sel) begin
    case(sel)
      1'b0 : dout = din0;
      1'b1 : dout = din1;
      default : dout = din0;
    endcase
  end

endmodule
////////////////////////////////// MUX2TO1 END ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////////////////////////
////////////////////////////// AUDIO_BUFFER START //////////////////////////////////
module audio_buffer(reset, clk, din, dout);

  parameter WIDTH = 16;

  input reset, clk, din;

  output dout;

  reg [WIDTH-1:0] buffer;

   assign dout = buffer[WIDTH-1];

  always@(posedge clk or posedge reset) begin
    if(reset) begin
      buffer <= 20'b00000000000000001111;
    end
    else begin
      buffer <= buffer << 1;
      buffer[0] <= din;
    end
  end

endmodule
////////////////////////////// AUDIO_BUFFER END ////////////////////////////////////
//////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////// TOPMODULE START /////////////////////////////////
module topmodule(reset, dc_clk, uc_clk, rpi_sclk, rpi_mosi_ss, miso,
          lrclki, bclki, sdin, lrclko, bclko, rpi_sdo, sclk, ss, mosi,
          rpi_miso, rpi_wsi, rpi_scki, rpi_sdi, rpi_wso, rpi_scko, sdout);

  input reset, dc_clk, uc_clk;

  //SPI In- and Outputs
  input rpi_sclk, rpi_mosi_ss, miso;
  output sclk, ss, mosi, rpi_miso;

  //Audio-In In- and Outputs
  input lrclki, bclki, sdin;
  output rpi_wsi, rpi_scki, rpi_sdi;

  //Audio-Out In- and Outputs
  input lrclko, bclko, rpi_sdo;
  output rpi_wso, rpi_scko, sdout;

  spi_recovery spi_recovery (.clk(uc_clk),.reset(reset),.mosi_ss(rpi_mosi_ss),
    .miso_i(miso),.sclk_i(rpi_sclk),.sclk_o(sclk),.ss(ss),.miso_o(rpi_miso),
    .mosi(mosi));

  audio_in_synchronizer audio_in_synchronizer (.clk(uc_clk),.reset(reset),
    .lrclki(lrclki),.bclki(bclki),.sdin(sdin),.wsi(rpi_wsi),.scki(rpi_scki),
    .sdi(rpi_sdi));

  audio_out_synchronizer audio_out_synchonizer (.dc_clk(dc_clk),.uc_clk(uc_clk),
    .reset(reset),.lrclko(lrclko),.bclko(bclko),.sdo(rpi_sdo),
    .wso(rpi_wso),.scko(rpi_scko),.sdout(sdout));

endmodule
////////////////////////////// TOPMODULE END ///////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////////////////////////// SPI_RECOVERY START //////////////////////////////////
module spi_recovery(clk,reset, mosi_ss, miso_i, sclk_i, sclk_o, ss, miso_o, mosi);

  input clk, reset, sclk_i, mosi_ss, miso_i;
  output sclk_o, ss, miso_o;
  output reg mosi;

  wire en_sclk_counter;
  wire [2:0] sclk_cnt;

  synchronizer_with_DETFF #(.DELAY(2)) miso_synchronizer (.clk(clk),
    .reset(reset),.din(miso_i),.dout(miso_o));

  counter_async_clr #(.WIDTH(3)) sclk_counter (.clk(sclk_i),.reset(reset),
    .enable(en_sclk_counter),.counter(sclk_cnt));

  ss_mosi_reprocessing ss_mosi_reprocessing (.sclk(sclk_i),.reset(reset),
    .mosi_ss(mosi_ss),.sclk_cnt(sclk_cnt),.ss(ss),.select(en_sclk_counter));

  always @(negedge sclk_i) begin
    if(reset)
      mosi <= 1'b0;
    else begin
      if(!ss)
        mosi <= mosi_ss;
    end
  end

  assign sclk_o = sclk_i;

endmodule
////////////////////////////// SPI_RECOVERY END ////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////// AUDIO_IN_SYNCHRONIZER START /////////////////////////////
module audio_in_synchronizer(clk, reset, lrclki, bclki, sdin, wsi, scki, sdi);

  input clk, reset, lrclki, bclki, sdin;
  output wsi, scki, sdi;

  synchronizer #(.DELAY(2)) bclki_synchronizer(.clk(clk),.reset(reset),
    .din(bclki),.dout(scki));

  synchronizer #(.DELAY(2)) lrclki_synchronizer(.clk(clk),.reset(reset),
    .din(lrclki),.dout(wsi));

  synchronizer #(.DELAY(2)) sdin_synchronizer(.clk(clk),.reset(reset),
    .din(sdin),.dout(sdi));

endmodule
////////////////////////// AUDIO_IN_SYNCHRONIZER END ///////////////////////////////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////////////////////////// AUDIO_OUT_SYNCHRONIZER START ////////////////////////////
module audio_out_synchronizer(dc_clk, uc_clk, reset, lrclko, bclko, sdo,
                    wso, scko, sdout);

  input dc_clk, uc_clk, reset, lrclko, bclko, sdo;
  output wso, scko;
  output reg sdout;

  wire enable_tx_start_cnt, clear_tx_start_cnt;
  wire enable_tx_cnt, clear_tx_cnt;
  wire enable_dc_clk_cnt, clear_dc_clk_cnt;
  wire [1:0] tx_start_cnt;
  wire [4:0] tx_cnt, dc_cnt;
  wire [1:0] sdo_buffer_in_sel, sdo_buffer_out_sel;
  wire buffer1_clk, buffer2_clk, buffer3_clk, buffer4_clk;
  wire sdo_buffer1_in, sdo_buffer2_in, sdo_buffer3_in, sdo_buffer4_in;
  wire sdo_buffer1_out, sdo_buffer2_out, sdo_buffer3_out, sdo_buffer4_out;
  wire sdout2delay, temp_sdout;

  synchronizer #(.DELAY(2)) bclko_synchronizer(.clk(uc_clk),.reset(reset),
    .din(bclko),.dout(scko));

  synchronizer #(.DELAY(2)) lrclko_synchronizer(.clk(uc_clk),.reset(reset),
    .din(lrclko),.dout(wso));

  sdo_buffer_controller sdo_buffer_controller (.dc_clk(dc_clk),
    .bclko(bclko),.lrclko(lrclko),.reset(reset),.tx_start_cnt(tx_start_cnt),
    .enable_tx_cnt(enable_tx_cnt),.buffer1_clk(buffer1_clk),
    .buffer2_clk(buffer2_clk),.buffer3_clk(buffer3_clk),
    .buffer4_clk(buffer4_clk),.sel_demux(sdo_buffer_in_sel),
    .sel_mux(sdo_buffer_out_sel));

  counter_controller counter_controller (.clk(dc_clk),.reset(reset),
    .tx_start_cnt(tx_start_cnt),.tx_cnt(tx_cnt),.dc_cnt(dc_cnt),
    .en_tx_start_cnt(enable_tx_start_cnt),.clr_tx_start_cnt(clear_tx_start_cnt),
    .en_tx_cnt(enable_tx_cnt),.clr_tx_cnt(clear_tx_cnt),
    .en_dc_clk_cnt(enable_dc_clk_cnt),.clr_dc_clk_cnt(clear_dc_clk_cnt));

  counter_sync_clr #(.WIDTH(5)) dc_clk_counter (.clk(dc_clk),
    .reset(clear_dc_clk_cnt),.enable(enable_dc_clk_cnt),
    .counter(dc_cnt));

  counter_sync_clr #(.WIDTH(2)) tx_start_counter (.clk(dc_clk),
    .reset(clear_tx_start_cnt),.enable(sdo & enable_tx_start_cnt),
    .counter(tx_start_cnt));

  counter_sync_clr #(.WIDTH(5)) tx_counter (.clk(dc_clk),.
    reset(clear_tx_cnt),.enable(enable_tx_cnt),.counter(tx_cnt));

  demux1to4 sdo_buffer_demux (.sel(sdo_buffer_in_sel),.din(sdo),
    .dout0(sdo_buffer1_in),.dout1(sdo_buffer2_in),
```

```
    .dout2(sdo_buffer3_in),.dout3(sdo_buffer4_in));

  audio_buffer #(.WIDTH(16)) sdo_buffer1 (.reset(reset),.clk(buffer1_clk),
    .din(sdo_buffer1_in),.dout(sdo_buffer1_out));

  audio_buffer #(.WIDTH(16)) sdo_buffer2 (.reset(reset),.clk(buffer2_clk),
    .din(sdo_buffer2_in),.dout(sdo_buffer2_out));

  audio_buffer #(.WIDTH(16)) sdo_buffer3 (.reset(reset),.clk(buffer3_clk),
    .din(sdo_buffer3_in),.dout(sdo_buffer3_out));

  audio_buffer #(.WIDTH(16)) sdo_buffer4 (.reset(reset),.clk(buffer4_clk),
    .din(sdo_buffer4_in),.dout(sdo_buffer4_out));

  mux4to1 sdo_out_mux (.sel(sdo_buffer_out_sel),
    .din0(sdo_buffer1_out),.din1(sdo_buffer2_out),
    .din2(sdo_buffer3_out),.din3(sdo_buffer4_out),
    .dout(sdout2delay));

  audio_buffer #(.WIDTH(16)) sdo_delay (.reset(reset),.clk(bclko),
    .din(sdout2delay),.dout(temp_sdout));

  always @(negedge bclko) begin
    if(reset)
      sdout <= 1'b0;
    else
      sdout <= temp_sdout;
  end
endmodule
////////////////////////// AUDIO_OUT_SYNCHRONIZER END ////////////////////////////
//////////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////////////////
/////////////////////////// SS_MOSI_REPROCESSING START ///////////////////////////
module ss_mosi_reprocessing(sclk, reset, mosi_ss, sclk_cnt, ss, select);

  input sclk, reset, mosi_ss;
  input [2:0] sclk_cnt;

  output reg ss;

  output reg select;

  reg [1:0] current_state;
  reg [1:0] next_state;

  `define IDLE       2'b00
  `define SELECT       2'b01
  `define WAIT_4_SDS   2'b11
  `define DESELECT    2'b10


  always @(negedge mosi_ss or posedge reset) begin
    if (reset)
      current_state = `IDLE;
    else
      current_state = next_state;
  end

  always @(current_state or sclk or sclk_cnt) begin
    next_state = `IDLE;

    case(current_state)
      `IDLE :
        begin
          if(sclk)
            next_state = `SELECT;
          else
            next_state = `IDLE;
        end
      `SELECT :
        begin
```

```
          if(sclk)
            next_state = `WAIT_4_SDS;
          else
            next_state = `IDLE;
        end
      `WAIT_4_SDS :
        begin
          if(sclk_cnt == 3'b000)
            next_state = `DESELECT;
          else
            next_state = `WAIT_4_SDS;
        end
      `DESELECT :
        begin
          if(sclk)
            next_state = `IDLE;
          else
            next_state = `DESELECT;
        end
      default : next_state = `IDLE;
    endcase
  end

  always @(negedge mosi_ss or posedge reset) begin
    if(reset) begin
      ss = 1'b1;
      select = 1'b0;
    end

    else begin
      case (current_state)
        `IDLE :
          begin
            ss = 1'b1;
            select = 1'b0;
          end
        `SELECT :
          begin
            ss = 1'b0;
            select = 1'b1;
          end
        `WAIT_4_SDS :
          begin
            ss = 1'b0;
            select = 1'b1;
          end
        `DESELECT :
          begin
            ss = 1'b1;
            select = 1'b0;
          end
        default :
          begin
            ss = 1'b1;
            select = 1'b0;
          end
      endcase
    end
  end

endmodule
//////////////////////////// SS_MOSI_REPROCESSING END ////////////////////////////
////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////
/////////////////////////// SDO_BUFFER_CONTROLLER START ////////////////////////////
module sdo_buffer_controller (dc_clk, bclko, lrclko, reset, tx_start_cnt,
  enable_tx_cnt, buffer1_clk, buffer2_clk, buffer3_clk, buffer4_clk,
  sel_demux, sel_mux);

  input dc_clk, bclko, lrclko, reset, enable_tx_cnt;
  input [1:0] tx_start_cnt;
```

```
output buffer1_clk, buffer2_clk, buffer3_clk, buffer4_clk;
output reg [1:0] sel_demux, sel_mux;

reg [2:0] current_input_state, next_input_state;
reg [2:0] current_output_state, next_output_state;

reg enable_dc_clk_buffer1, enable_dc_clk_buffer2;
reg enable_dc_clk_buffer3, enable_dc_clk_buffer4;
reg enable_bclko_buffer1, enable_bclko_buffer2;
reg enable_bclko_buffer3, enable_bclko_buffer4;
reg ready2read_buffer1, ready2read_buffer2;
reg ready2read_buffer3, ready2read_buffer4;

`define INPUT_IDLE  3'b000
`define LOAD_BUFFER1  3'b001
`define LOAD_BUFFER2  3'b010
`define LOAD_BUFFER3  3'b011
`define LOAD_BUFFER4  3'b100

always @(negedge dc_clk) begin
  if(reset)
    current_input_state <= `INPUT_IDLE;
  else
    current_input_state <= next_input_state;
end

always @(current_input_state or tx_start_cnt) begin
  next_input_state <= `INPUT_IDLE;
  case(current_input_state)
    `INPUT_IDLE : begin
      if(tx_start_cnt == 2'b11)
        next_input_state <= `LOAD_BUFFER1;
      else
        next_input_state <= `INPUT_IDLE;
    end
    `LOAD_BUFFER1 : begin
      if(tx_start_cnt == 2'b11)
        next_input_state <= `LOAD_BUFFER2;
      else
        next_input_state <= `LOAD_BUFFER1;
    end
    `LOAD_BUFFER2 : begin
      if(tx_start_cnt == 2'b11)
        next_input_state <= `LOAD_BUFFER3;
      else
        next_input_state <= `LOAD_BUFFER2;
    end
    `LOAD_BUFFER3 : begin
      if(tx_start_cnt == 2'b11)
        next_input_state <= `LOAD_BUFFER4;
      else
        next_input_state <= `LOAD_BUFFER3;
    end
    `LOAD_BUFFER4 : begin
      if(tx_start_cnt == 2'b11)
        next_input_state <= `LOAD_BUFFER1;
      else
        next_input_state <= `LOAD_BUFFER4;
    end
  endcase
end

always @(negedge dc_clk) begin
  if(reset) begin
    sel_demux <= 2'b00;
    enable_dc_clk_buffer1 <= 1'b0;
    enable_dc_clk_buffer2 <= 1'b0;
    enable_dc_clk_buffer3 <= 1'b0;
    enable_dc_clk_buffer4 <= 1'b0;
    ready2read_buffer1 <= 1'b0;
    ready2read_buffer2 <= 1'b0;
```

```
      ready2read_buffer3 <= 1'b0;
      ready2read_buffer4 <= 1'b0;
    end

    else begin
      case(current_input_state)
      `INPUT_IDLE : begin
        sel_demux <= 2'b00;
        enable_dc_clk_buffer1 <= 1'b0;
        enable_dc_clk_buffer2 <= 1'b0;
        enable_dc_clk_buffer3 <= 1'b0;
        enable_dc_clk_buffer4 <= 1'b0;
        ready2read_buffer1 <= 1'b0;
        ready2read_buffer2 <= 1'b0;
        ready2read_buffer3 <= 1'b0;
        ready2read_buffer4 <= 1'b0;
      end
      `LOAD_BUFFER1 : begin
        sel_demux <= 2'b00;
        enable_dc_clk_buffer1 <= 1'b1;
        enable_dc_clk_buffer2 <= 1'b0;
        enable_dc_clk_buffer3 <= 1'b0;
        enable_dc_clk_buffer4 <= 1'b0;
        ready2read_buffer1 <= 1'b0;
        ready2read_buffer2 <= 1'b0;
        ready2read_buffer3 <= 1'b0;
        ready2read_buffer4 <= 1'b1;
      end
      `LOAD_BUFFER2 : begin
        sel_demux <= 2'b01;
        enable_dc_clk_buffer1 <= 1'b0;
        enable_dc_clk_buffer2 <= 1'b1;
        enable_dc_clk_buffer3 <= 1'b0;
        enable_dc_clk_buffer4 <= 1'b0;
        ready2read_buffer1 <= 1'b1;
        ready2read_buffer2 <= 1'b0;
        ready2read_buffer3 <= 1'b0;
        ready2read_buffer4 <= 1'b0;
      end
      `LOAD_BUFFER3 : begin
        sel_demux <= 2'b10;
        enable_dc_clk_buffer1 <= 1'b0;
        enable_dc_clk_buffer2 <= 1'b0;
        enable_dc_clk_buffer3 <= 1'b1;
        enable_dc_clk_buffer4 <= 1'b0;
        ready2read_buffer1 <= 1'b0;
        ready2read_buffer2 <= 1'b1;
        ready2read_buffer3 <= 1'b0;
        ready2read_buffer4 <= 1'b0;
      end
      `LOAD_BUFFER4 : begin
        sel_demux <= 2'b11;
        enable_dc_clk_buffer1 <= 1'b0;
        enable_dc_clk_buffer2 <= 1'b0;
        enable_dc_clk_buffer3 <= 1'b0;
        enable_dc_clk_buffer4 <= 1'b1;
        ready2read_buffer1 <= 1'b0;
        ready2read_buffer2 <= 1'b0;
        ready2read_buffer3 <= 1'b1;
        ready2read_buffer4 <= 1'b0;
      end
      endcase
    end
  end

  `define OUTPUT_IDLE   3'b000
  `define OUTPUT_BUFFER1  3'b001
  `define OUTPUT_BUFFER2  3'b010
  `define OUTPUT_BUFFER3  3'b011
  `define OUTPUT_BUFFER4  3'b100
```

```
always @(posedge lrclko or negedge lrclko) begin
  if(reset)
    current_output_state <= `OUTPUT_IDLE;
  else
    current_output_state <= next_output_state;
end

always @(current_output_state or ready2read_buffer1 or ready2read_buffer2
      or ready2read_buffer3 or ready2read_buffer4) begin
  next_output_state <= `OUTPUT_IDLE;
  case(current_output_state)
    `OUTPUT_IDLE : begin
      if(ready2read_buffer1)
        next_output_state <= `OUTPUT_BUFFER1;
      else
        next_output_state <= `OUTPUT_IDLE;
    end
    `OUTPUT_BUFFER1 : begin
      if(ready2read_buffer2)
        next_output_state <= `OUTPUT_BUFFER2;
      else
        next_output_state <= `OUTPUT_IDLE;
    end
    `OUTPUT_BUFFER2 : begin
      if(ready2read_buffer3)
        next_output_state <= `OUTPUT_BUFFER3;
      else
        next_output_state <= `OUTPUT_IDLE;
    end
    `OUTPUT_BUFFER3 : begin
      if(ready2read_buffer4)
        next_output_state <= `OUTPUT_BUFFER4;
      else
        next_output_state <= `OUTPUT_IDLE;
    end
    `OUTPUT_BUFFER4 : begin
      if(ready2read_buffer1)
        next_output_state <= `OUTPUT_BUFFER1;
      else
        next_output_state <= `OUTPUT_IDLE;
    end
  endcase
end

always @(posedge lrclko or negedge lrclko or posedge reset) begin
  if(reset) begin
    sel_mux <= 2'b11;
    enable_bclko_buffer1 <= 1'b0;
    enable_bclko_buffer2 <= 1'b0;
    enable_bclko_buffer3 <= 1'b0;
    enable_bclko_buffer4 <= 1'b0;
  end

  else begin
    case(current_output_state)
    `OUTPUT_IDLE : begin
      sel_mux <= 2'b11;
      enable_bclko_buffer1 <= 1'b0;
      enable_bclko_buffer2 <= 1'b0;
      enable_bclko_buffer3 <= 1'b0;
      enable_bclko_buffer4 <= 1'b0;
    end
    `OUTPUT_BUFFER1 : begin
      sel_mux <= 2'b00;
      enable_bclko_buffer1 <= 1'b1;
      enable_bclko_buffer2 <= 1'b0;
      enable_bclko_buffer3 <= 1'b0;
      enable_bclko_buffer4 <= 1'b0;
    end
    `OUTPUT_BUFFER2 : begin
      sel_mux <= 2'b01;
      enable_bclko_buffer1 <= 1'b0;
```

```
        enable_bclko_buffer2 <= 1'b1;
        enable_bclko_buffer3 <= 1'b0;
        enable_bclko_buffer4 <= 1'b0;
      end
      `OUTPUT_BUFFER3 : begin
        sel_mux <= 2'b10;
        enable_bclko_buffer1 <= 1'b0;
        enable_bclko_buffer2 <= 1'b0;
        enable_bclko_buffer3 <= 1'b1;
        enable_bclko_buffer4 <= 1'b0;
      end
      `OUTPUT_BUFFER4 : begin
        sel_mux <= 2'b11;
        enable_bclko_buffer1 <= 1'b0;
        enable_bclko_buffer2 <= 1'b0;
        enable_bclko_buffer3 <= 1'b0;
        enable_bclko_buffer4 <= 1'b1;
      end
      endcase
    end
  end

  assign buffer1_clk = (dc_clk & enable_tx_cnt & enable_dc_clk_buffer1) |
                (bclko & enable_bclko_buffer1);
  assign buffer2_clk = (dc_clk & enable_tx_cnt & enable_dc_clk_buffer2) |
                (bclko & enable_bclko_buffer2);
  assign buffer3_clk = (dc_clk & enable_tx_cnt & enable_dc_clk_buffer3) |
                (bclko & enable_bclko_buffer3);
  assign buffer4_clk = (dc_clk & enable_tx_cnt & enable_dc_clk_buffer4) |
                (bclko & enable_bclko_buffer4);

endmodule
//////////////////////////// SDO_BUFFER_CONTROLLER END ////////////////////////////
////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////
//////////////////////////// COUNTER_CONTROLLER START ///////////////////////////////
module counter_controller(clk, reset, tx_start_cnt, tx_cnt, dc_cnt,
                en_tx_start_cnt, clr_tx_start_cnt, en_tx_cnt, clr_tx_cnt,
                en_dc_clk_cnt, clr_dc_clk_cnt);

  input clk, reset;
  input [1:0] tx_start_cnt;
  input [4:0] tx_cnt, dc_cnt;
  output reg en_tx_start_cnt, clr_tx_start_cnt;
  output reg en_tx_cnt, clr_tx_cnt;
  output reg en_dc_clk_cnt, clr_dc_clk_cnt;

  reg [2:0] current_state;
  reg [2:0] next_state;

  `define IDLE_          3'b000
  `define INITBIT1_DETECTED 3'b001
  `define INITBIT2_DETECTED 3'b010
  `define INITBIT3_DETECTED 3'b011
  `define INITBIT4_DETECTED 3'b100
  `define READ_AUDIOSAMPLE  3'b101
  `define ERROR          3'b110

  always @(negedge clk or posedge reset) begin
    if(reset)
      current_state <= `IDLE_;
    else
      current_state <= next_state;
  end



  always @(current_state or tx_start_cnt or tx_cnt or dc_cnt) begin
    next_state <= `IDLE_;
    case (current_state)
```

```
      `IDLE_ : begin
        if(tx_start_cnt == 2'b01)
          next_state <= `INITBIT1_DETECTED;
        else
          next_state <= `IDLE_;
      end
      `INITBIT1_DETECTED : begin
        if(tx_start_cnt == 2'b10)
          next_state <= `INITBIT2_DETECTED;
        else
          next_state <= `ERROR;
      end
      `INITBIT2_DETECTED : begin
        if(tx_start_cnt == 2'b11)
          next_state <= `INITBIT3_DETECTED;
        else
          next_state <= `ERROR;
      end
      `INITBIT3_DETECTED : begin
        if(tx_start_cnt == 2'b00)
          next_state <= `INITBIT4_DETECTED;
        else
          next_state <= `ERROR;
      end
      `INITBIT4_DETECTED : begin
        next_state <= `READ_AUDIOSAMPLE;
      end
      `READ_AUDIOSAMPLE : begin
        if(tx_cnt == 5'b01111)
          next_state <= `IDLE_;
        else
          next_state <= `READ_AUDIOSAMPLE;
      end
      `ERROR : begin
        if(dc_cnt == 5'b10011)
          next_state <= `IDLE_;
        else
          next_state <= `ERROR;
      end
    endcase
  end

  always @(negedge clk or posedge reset) begin
    if(reset) begin
      en_tx_start_cnt <= 1'b1;
      en_tx_cnt <= 1'b0;
      clr_tx_start_cnt <= 1'b0;
      clr_tx_cnt <= 1'b0;
      en_dc_clk_cnt <= 1'b0;
      clr_dc_clk_cnt <= 1'b0;
    end
    else begin
      case (current_state)
        `IDLE_ : begin
          en_tx_start_cnt <= 1'b1;
          en_tx_cnt <= 1'b0;
          clr_tx_start_cnt <= 1'b0;
          clr_tx_cnt <= 1'b0;
          en_dc_clk_cnt <= 1'b0;
          clr_dc_clk_cnt <= 1'b1;
        end
        `INITBIT1_DETECTED : begin        //tx_start_cnt == 2'b01
          en_tx_start_cnt <= 1'b1;
          en_tx_cnt <= 1'b0;
          clr_tx_start_cnt <= 1'b0;
          clr_tx_cnt <= 1'b1;
          en_dc_clk_cnt <= 1'b1;
          clr_dc_clk_cnt <= 1'b0;
        end
        `INITBIT2_DETECTED : begin        //tx_start_cnt == 2'b10
          en_tx_start_cnt <= 1'b1;
          en_tx_cnt <= 1'b0;
```

```
          clr_tx_start_cnt <= 1'b0;
          clr_tx_cnt <= 1'b0;
          en_dc_clk_cnt <= 1'b1;
          clr_dc_clk_cnt <= 1'b0;
        end
        `INITBIT3_DETECTED : begin        //tx_start_cnt == 2'b11
          en_tx_start_cnt <= 1'b0;
          en_tx_cnt <= 1'b1;
          clr_tx_start_cnt <= 1'b0;
          clr_tx_cnt <= 1'b0;
          en_dc_clk_cnt <= 1'b1;
          clr_dc_clk_cnt <= 1'b0;
        end
        `INITBIT4_DETECTED : begin        //tx_start_cnt == 2'b00
          en_tx_start_cnt <= 1'b0;
          en_tx_cnt <= 1'b1;
          clr_tx_start_cnt <= 1'b1;
          clr_tx_cnt <= 1'b0;
          en_dc_clk_cnt <= 1'b1;
          clr_dc_clk_cnt <= 1'b0;
        end
        `READ_AUDIOSAMPLE : begin
          en_tx_start_cnt <= 1'b0;
          en_tx_cnt <= 1'b1;
          clr_tx_start_cnt <= 1'b0;
          clr_tx_cnt <= 1'b0;
          en_dc_clk_cnt <= 1'b1;
          clr_dc_clk_cnt <= 1'b0;
        end
        `ERROR : begin
          en_tx_start_cnt <= 1'b0;
          en_tx_cnt <= 1'b0;
          clr_tx_start_cnt <= 1'b1;
          clr_tx_cnt <= 1'b0;
          en_dc_clk_cnt <= 1'b1;
          clr_dc_clk_cnt <= 1'b0;
        end
      endcase
    end
  end

endmodule
//////////////////////////// COUNTER_CONTROLLER END ////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////
////////////////////////////// AUDIO_BUFFER START ///////////////////////////////////
module audio_buffer(reset, clk, din, dout);

  parameter WIDTH = 6;

  input reset, clk, din;

  output dout;

  reg [WIDTH-1:0] buffer;

   assign dout = buffer[WIDTH-1];

  always@(posedge clk) begin
    if(reset) begin
      buffer <= 0;
    end
    else begin
      buffer <= buffer << 1;
      buffer[0] <= din;
    end
  end

endmodule
//////////////////////////////// AUDIO_BUFFER END ///////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////// MUX4TO1 START ///////////////////////////////////
module mux4to1(sel, din0, din1, din2, din3, dout);

    input [1:0] sel;
    input din0, din1, din2, din3;

    output reg dout;

   always @(sel or din0 or din1 or din2 or din3) begin
    case(sel)
      2'b00 : dout = din0;
      2'b01 : dout = din1;
      2'b10 : dout = din2;
      2'b11 : dout = din3;
      default : dout = 1'b1;
    endcase
  end

endmodule
//////////////////////////////// MUX4TO1 END ///////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////////////////////////////// DEMUX1TO4 START //////////////////////////////////
module demux1to4(sel, din, dout0, dout1, dout2, dout3);

  input [1:0] sel;
  input din;

  output reg dout0, dout1, dout2, dout3;

  always @(sel or din) begin
    case (sel)
      2'b00 : begin
        dout0 <= din;
        dout1 <= 1'b0;
        dout2 <= 1'b0;
        dout3 <= 1'b0;
      end
      2'b01 : begin
        dout0 <= 1'b0;
        dout1 <= din;
        dout2 <= 1'b0;
        dout3 <= 1'b0;
      end
      2'b10 : begin
        dout0 <= 1'b0;
        dout1 <= 1'b0;
        dout2 <= din;
        dout3 <= 1'b0;
      end
      2'b11 : begin
        dout0 <= 1'b0;
        dout1 <= 1'b0;
        dout2 <= 1'b0;
        dout3 <= din;
      end
      default : begin
        dout0 <= 1'b0;
        dout1 <= 1'b0;
        dout2 <= 1'b0;
        dout3 <= 1'b0;
      end
    endcase
  end

endmodule
//////////////////////////////// DEMUX1TO4 END //////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////// COUNTER_ASYNC_CLR START //////////////////////////////
module counter_async_clr(clk, reset, enable, counter);

  parameter WIDTH = 3;

  input clk, reset, enable;

  output reg [WIDTH-1:0] counter;

  always@(negedge clk or posedge reset) begin
    if(reset) begin
      counter <= 0;
    end
    else begin
      if(enable)
        counter <= counter + 1'b1;
    end
  end

endmodule
////////////////////////// COUNTER_ASYNC_CLR END ////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////////////////////// SYNCHRONIZER_WITH_DETFF START ////////////////////////////
module synchronizer_with_DETFF(clk, reset, din, dout);

  parameter DELAY = 6;

  input clk, reset, din;
  output dout;

  reg [DELAY-1:0] delaychain;

  assign dout = delaychain[DELAY-1];

  always@(posedge clk or negedge clk) begin
    if(reset) begin
      delaychain <= 13'b1111111111111;
    end
    else begin
      delaychain <= delaychain << 1;
      delaychain[0] <= din;
    end
  end

endmodule
////////////////////// SYNCHRONIZER_WITH_DETFF END //////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////// COUNTER_SYNC_CLR START ///////////////////////////////
module counter_sync_clr(clk, reset, enable, counter);

  parameter WIDTH = 3;

  input clk, reset, enable;

  output reg [WIDTH-1:0] counter;

  always@(posedge clk) begin
    if(reset) begin
      counter <= 0;
    end
    else begin
      if(enable)
        counter <= counter + 1'b1;
    end
  end

endmodule
////////////////////////////// COUNTER_SYNC_CLR END ///////////////////////////////
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
////////////////////////////// SYNCHRONIZER START ///////////////////////////////
module synchronizer(clk, reset, din, dout);

  parameter DELAY = 6;

  input clk, reset, din;
  output dout;

  reg [DELAY-1:0] delaychain;

  assign dout = delaychain[DELAY-1];

  always@(negedge clk) begin
    if(reset) begin
      delaychain <= 0;
    end
    else begin
      delaychain <= delaychain << 1;
      delaychain[0] <= din;
    end
  end

endmodule
////////////////////////////// SYNCHRONIZER END ///////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```