B. Tryggve Eyser

Softwareportierung für das Radarsystem WERA

Studienarbeit

20. Januar 2002

Fachbereich Geowissenschaften
und
Fachbereich Informatik
Universität Hamburg

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Die im Institut für Meereskunde des Fachbereichs Geowissenschaften der Universität Hamburg beheimatete Arbeitsgruppe Fernerkundung [WERA] konnte 1996 die Entwicklung eines Radarsystems zur Seegang-Messung (WELLEN RADAR genannt WERA) zu ihrem vorläufigen Abschluss bringen. Das System gestattet flächenhaftes Messen von Seegang mit einer Reichweite von 60km und mehr in Abhängigkeit von der verwendeten Frequenz, wobei die maximale Reichweite mit zunehmender Frequenz nachlässt, während gleichzeitig das räumliche Ausfösungsvermögen besser wird. Die gemessenen Daten lassen Rückschlüsse auf die Strömung und den Seegang zu. In den folgenden Jahren kam das System in mehreren Messkampagnen zum Einsatz.

Die Beschaffung von Bauteilen für den Messrechner stellt sich jedoch mittlerweile ein Problem dar, da sich seit dem ursprünglichen Aufbau des Systems mehrere Entwicklungszyklen in der Computertechnik vollzogen haben. Die im Rahmen einer Kooperation der Universität Hawaii benötigten Radaranlagen können mit der alten Architektur bereits nicht mehr gebaut werden, da die Bauteile nicht mehr hergestellt werden. Daher kam es 2001 zu einem Hardwarewechsel des Messrechners. Er wurde auch Anlaß für diese Studienarbeit, da dadurch auch größere Änderungen in der Software des Messrechners erforderlich wurden. Randbedingung für die Umstellung der Software war, dass einerseits die Programme zur Steuerung des Systems und zur Datenauswertung unverändert bleiben[1], und dass zum anderen Erweiterungen des Funkionsumfangs möglichst auch in die alte Software integriert werden. Hintergrund dafür ist, dass die vorhandenen Messrechner aus kostengründen nicht einfach durch neue ersetzt werden können.

Kapitel 2

Hier soll zunächst gezeigt werden, wozu das WERA-System dient und wie die grundsätzliche Arbeitsweise ist.

Kapitel 3

Zunächst erfolgt ein Einblick in die alte Hardware des Messrechners.
Danach wird dessen neue Hardware beschrieben.

[1] diese Programme laufen auf anderen Rechner und sind unabhängig von der Umstellung des Messrechners

Abbildung 1.1: *Array aus 16 Antennen, linear angeordnet parallel zur Küste; links die Landseite, rechts die Seeseite.*

Kapitel 4
Dieses Kapitel ist dem Hauptthema dieser Arbeit gewidmet.
Hier geht es um die Software des Messrechners, Programmstrukturen und Abläufe sollen dokumentiert werden.

2 Das Radarsystem WERA

Im Jahr 1996 entwickelte die Arbeitsgruppe Fernerkundung im Institut für Meereskunde [IfM], Fachbereich Geowissenschaften der Universität Hamburg ein Radarsystem, um von der Küste aus Strömungen der Meeresoberfläche und Seegang zu messen. Hergeleitet von der Anwendung bekam es den Namen WERA (WEllen RAdar). Das System kam in den darauf folgenden Jahren in mehreren Experimenten zum Einsatz.

Bedingt durch die Entwicklung des Hardwaremarktes sind heute jedoch praktisch keine Ersatzteile für die anwendungsspezifische Hardware (s. Abs. 3.1) mehr zu beschaffen. Außerdem soll im Zuge einer Kooperation der Universität Hawaii ein Radarsystem aufgebaut werden, das aus dem gleichen Grund nicht auf den alten Hardwarekomponenten basieren kann. Daher musste der Messrechner samt Analog/Digital-Wandlersystem unter Verwendung aktueller Hardware neu entworfen und aufgebaut werden.

2.1 Überblick

Wie in Abbildung 2.1 gezeigt besteht ein WERA System aus einem zentralen Steuerrechner und zwei oder mehr (gleichen) Messstationen, die an der Küste installiert werden. Der Abstand der Messstationen voneinander ist dabei entscheidend für die Genauigkeit und die Größe (Anzahl Antennen) des Antennen-Arrays jeder Messstation für die mögliche Richtungsauflösung. Jede einzelne Station misst nur eine Radialkomponente der Strömung. Aus mindestens zwei Radialkomponenten werden in einem späteren Schritt an zentraler Stelle die tatsächlichen Richtungs- und Geschwindigkeitsinformationen errechnet.

Bei der momentan verwendeten Messfrequenz von etwa 30 MHz beträgt die Reichweite ca. 45 km und der Abstand der Stationen voneinander ungefähr 20 km. Der besondere Vorteil des Verfahrens besteht darin, dass das Radiosignal in diesem Frequenzbereich der Erdkrümmung folgt und somit Messungen bis hinter den Horizont gestattet. Diese Eigenschaft ist nicht nur in zivilen Anwendungen genutzt, sondern auch in militärischen Anwendungen.

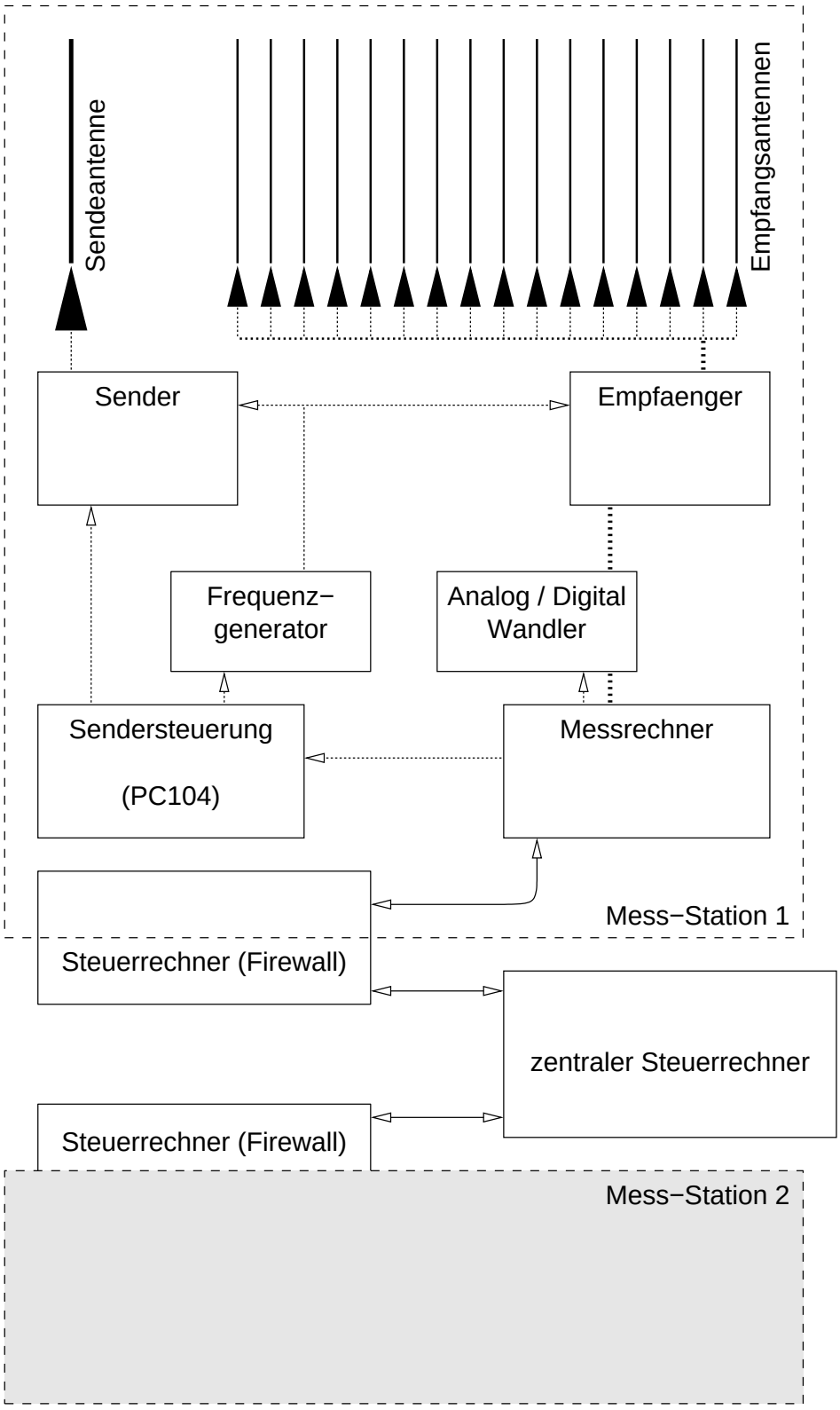

Abbildung 2.1: *Überblick über das WERA Gesamtsystem bestehend aus (mindestens) zwei Mess-Stationen und einem zentralen Steuerrechner*

2.2 Zentrale Steuerung

Das System besteht aus mindestens zwei Messstationen, die den gleichen Frequenzbereich verwenden. Würden die Stationen gleichzeitig senden, so käme es zu gegenseitigen Störungen. Um diese zu vermeiden, ist im derzeitigen System die einfachste denkbare Strategie implementiert: der zentrale Steuerrechner fordert die Stationen immer abwechselnd zu einer vollständigen Messung auf. Die Messung einer Station dauert knapp 9 Minuten.

In einer späteren Version sollen die Systeme so synchronisiert werden, dass ein gleichzeitiges Arbeiten möglich wird. Dazu ist eine Zeitsynchronisation mit einer Genauigkeit von 10 ms oder weniger erforderlich.

Der zentrale Steuerrechner dient aber nicht nur der zeitlichen Koordinierung. Außerdem werden dort die Daten von den Messstationen zusammengeführt und ausgewertet.

2.3 Die einzelne Mess-Station

Schnittstelle zur Außenwelt ist für eine Messstation nur der sogenannte Messrechner, der über TCP/IP von außen erreicht werden kann. Über diese Verbindung teilt der Steuerrechner dem Messrechner die Parameter für die nächste Messung mit. Die Parameter umfassen Werte für die Sendersteuerung (die den Frequenzgenerator programmiert) ebenso wie die Dauer der Messung, die Anzahl der verwendeten Antennen und den Dateinamen, unter dem die Messwerte gespeichert werden sollen. Nachdem das Analog/Digital-Wandlersystem (A/D-System) vorbereitet ist, bekommt die Sendersteuerung den Start-Befehl (und nach Ende der Messung den Stop-Befehl).

Der Frequenzgenerator erzeugt einen linear ansteigenden, periodisch wiederholten Frequenzverlauf, hier *Chirp* genannt. In der aktuellen Software ist der Frequenzgenerator auf 27,65 MHz $\pm$ 62,5 kHz eingestellt. Der Chirp wird mit 30 Watt Sendeleistung von einer Antenne über die Meeresoberfläche gesendet. Von dort zurück gestreute Strahlungsanteile werden von mehreren speziell angeordneten Antennen empfangen. Das Signal jeder Antenne wird vom Empfänger demoduliert[1], digitalisiert und vom Messrechner aufgezeichnet. Die Daten durchlaufen mehrere Verarbeitungsschritte, an deren Ende die entfernungsauflösende FFT[2] steht, die für jede Antenne ein Spektrum erzeugt. Ausgesuchte Entfernungsbereiche aus diesen Spektren werden schließlich an den zentralen Steuerrechner übermittelt, um dort als Rohdaten weiterverarbeitet zu werden.

Jede Messstation liefert eine Radialkomponente der Geschwindigkeiten des Strömungsfeldes, weshalb für vollständige Richtungs- und Geschwindigkeitsin-

[1] nach Frequenz 0 gemischt, in Phase als I-Anteil und um $\pi/2$ verschoben als Q-Anteil des komplexen Messwertes

[2] Fast Fourier Transformation

formationen (mindestens) zwei Messstationen benötigt werden.

2.4 Messverfahren

Das WERA System verwendet zur Entfernungsauflösung FMCW[3], was bedeutet, dass zyklisch eine Frequenzrampe (nachfolgend *Chirp* oder *Sweep* genannt) erzeugt wird. Hierfür wird ein Sender basierend auf einem digitalen Frequenzgenerator verwendet, der einen linearen Chirp erzeugt. In der aktuellen Konfiguration liegt der Chirp zwischen 27.5875 MHz und 27.7125 MHz, was sich aber dank des programmierbaren digitalen Frequenzgenerators jederzeit softwareseitig ändern lässt.

Abb. 2.2 zeigt den zeitlichen Verlauf des gesendeten und des empfangenen Signals *eines* reflektierenden Ziels. Dabei hängt die Frequenzverschiebung Δf, die das empfangene Signal gegenüber dem zur gleichen Zeit gesendeten Signal hat, von der Entfernung des Ziels ab und ist (anders als in der Abbildung) normalerweise die Überlagerung von Echos aller Ziele aus allen Entfernungen.

Die Dauer eines Chirps beträgt 0.26 Sekunden und in seinem Verlauf werden 1536 Abtastwerte aufgenommen. Das vom Frequenzgenerator erzeugte Signal wird mit 30 Watt ausgestrahlt. Von der Wasseroberfläche zurück gestreute Anteile werden über ein Antennen-Array empfangen. Pro Antenne mischt ein Empfängermodul das Signal mit der aktuellen Senderfrequenz nach 0 Hz. Das daraus resultierende Δf wird mit einem 800 Hz Tiefpaß gefiltert, in I- und Q-Anteil zerlegt und digitalisiert. Durch das Tiefpaß werden die sehr großen Δf-Werte zwischen 0 und Δt und zwischen T und $T + \Delta t$ eliminiert.

Die Abtast-Trigger müssen bei jedem Chirp mit hoher Genauigkeit im gleichen Zeitraster erzeugt werden. Andernfalls entstehen Schwebungen in Δf, die die Auswertung verfälschen, da zu dessen Berechnung die Differenzen der aufeinander folgenden Chirp-Spektren benutzt werden.

Vor der entfernungsauflösenden FFT werden die gewonnenen Daten mit einer Fensterfunktion überlagert, die die Anfangs- und Endwerte der Chirps zu Null werden lässt. So werden Sprünge an den Übergängen der Chirps verhindert, die ansonsten zu großen Energieanteilen in den hohen Frequenzbereichen der Spektren führen würden.

Die Frequenzverschiebung Δf des empfangenen Signals gegenüber dem gleichzeitig gesendeten Signal hängt folgendermaßen von der Entfernung r des reflektienden Ziels ab:

$$\Delta t = \frac{2r}{c} = \Delta f \frac{T}{B} \qquad \Delta f = \frac{B}{T}\frac{2r}{c} \qquad r = \Delta f \frac{T}{B}\frac{c}{2}$$

[3]Frequency Modulated Continuous Wave

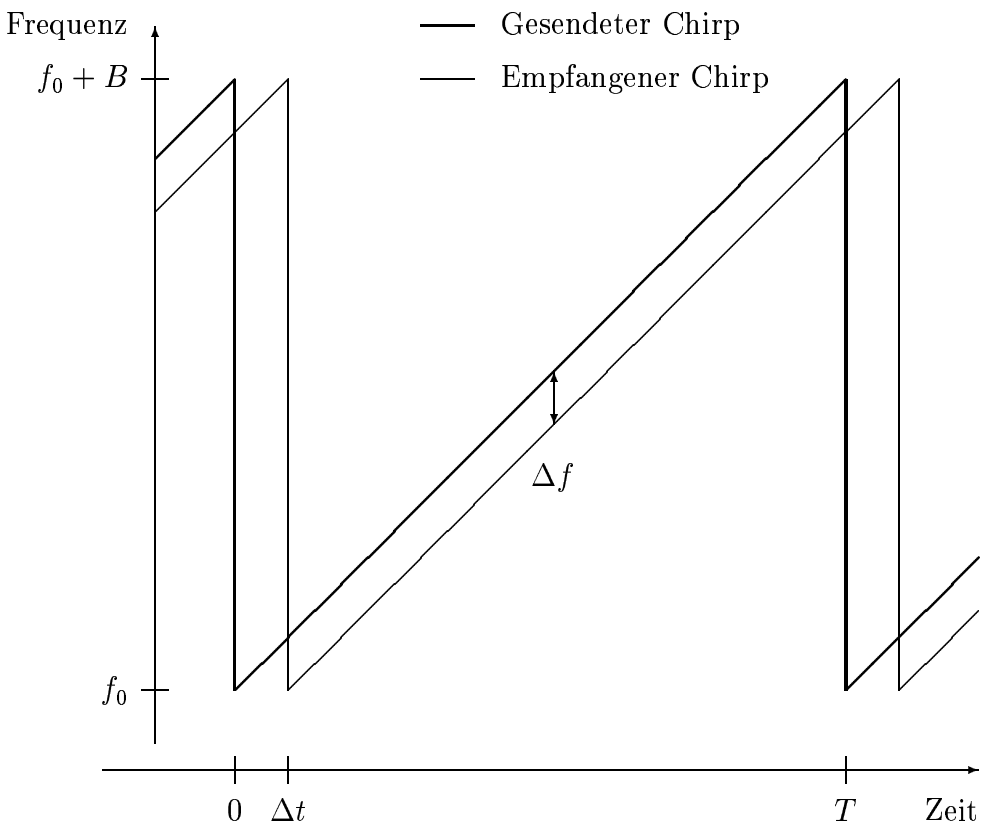

Abbildung 2.2: *Reichweitenauflösung unter Verwendung einer linearen Frequenzrampe (*Chirp*). Vom Frequenzgenerator erzeugtes Ausgangssignal und zeitversetztes Eingangssignal. Im Verlauf eines Chirps werden exakt 1536 Samples gelesen.*

Δf: Frequenzverschiebung
B : Bandbreite eines Chirps
T : Dauer eines Chirps
r : Entfernung reflektierender Ziele
c : Lichtgeschwindigkeit

- Eine FFT über einen einzelnen Chirp löst Entfernungsbereiche auf, die quantisiert sind entsprechend:

$$\frac{1}{T} = \Delta f_{quant} = \frac{B}{T}\frac{2r_{quant}}{c} \qquad \rightarrow \qquad r_{quant} = \frac{c}{2B}$$

- Eine Entfernungsauflösung von 1.2 km erfordert eine Bandbreite von 125 kHz, für 300 m Entfernungsauflösung wird eine Bandbreite von 500 kHz benötigt.

- Reichweiten auflösende FFTs für aufeinander folgende Chirps messen die Phasenverschiebung innerhalb einer Reichweite und ergeben so die Zeitserie der See-Echos für einen Entfernungsbereich.

Eine genaue Beschreibung des Verfahrens findet sich in [GES01a].

2.5 Anwendung

Bei jedem Chirp werden für jede Antenne 1536 komplexe Messwerte immer synchron mit dem Frequenzgenerator genommen. Jeder dieser Messwerte ist die Summe aller Reflektionen aus allen Entfernungsbereichen, wobei die Frequenzverschiebung mit zunehmender Entfernung wächst.

Für jede Antenne und jeden Chirp wird eine FFT berechnet, die wieder 1536 Werte enthält. Da das Messverfahren bei einem Chirp hin zu höheren Frequenzen nur negative Frequenzverschiebungen erwartet (mit Ausnahme des Frequenzsprungs beim Chirp-Wechsel), ist der positive Teil des Spektrums fast[4] irrelevant. Der negative Teil des Spektrums enthält also noch 768 Linien, von denen jede einem Entfernungsbereich[5] entspricht. Daraus wird die gewünschte Anzahl an Entfernungsbereichen *geschnitten*, z. B. 128 Linien für genau so viele Entfernungsbereiche.

Jede entfernungsauflösende FFT jedes Chirp liefert einen Abtastwert für alle Entfernungsbereiche und Antennen. Die Abfolge von Chirps bildet die entsprechenden Zeitserien für die Entfernungsbereiche.

Hier endet die Verarbeitung in der Messstation. Die weiteren Schritte erfolgen im Zentralrechner.

Auf den Zeitserien der einzelnen Antennen und Entfernungsbereiche werden wieder FFTs gerechnet. In Abb. 2.3 ist das Spektrum für eine Entfernungszelle senkrecht vom Array in etwa 20km Entfernung zu sehen. Um nur Signale aus einer Richtung zu bekommen, werden die Daten aller Antennen zusammengefaßt[6]. Der zentrale Peak bei 0Hz resultiert von unbeweglichen reflektierenden Objekten. Links und rechts davon sind zwei weitere Peaks[7] sichtbar. Sie sind auf die

[4] nur fast, weil er noch zur Rauschunterdrückung um 3dB und teilweise zur Unterdrückung von Störungen genutzt werden kann

[5] kreisförmig um die Antenne

[6] siehe *Beam Forming* in [GA97]

[7] bezeichnet mit First Order Peak

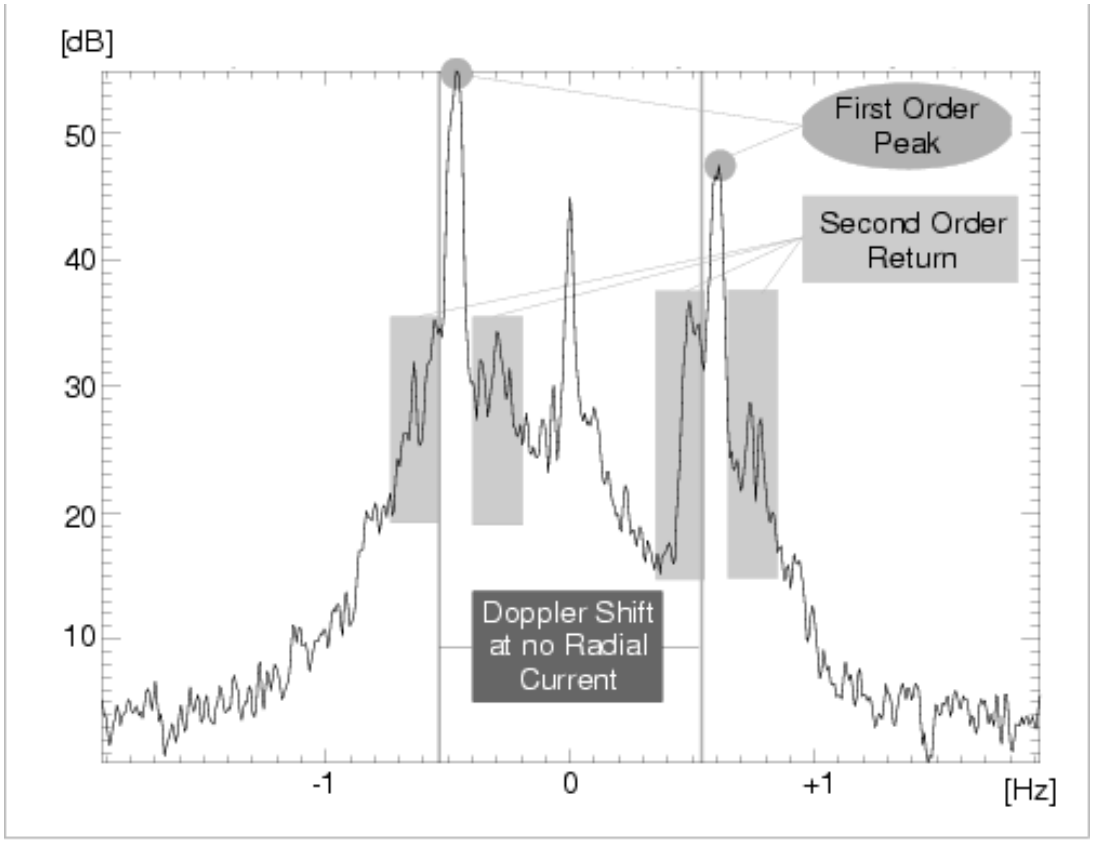

Abbildung 2.3: *Spektrum mit theoretisch zu erwartenden (*at no Radial Current*) und tatsächlich gemessenen (*First Order Peak*) Dopplerverschiebungen der Seegang-Echos senkrecht vom Array in ca. 20km Entfernung. (siehe [WERA])*

Dopplerverschiebung zurückzuführen, die die Bewegung des Seegangs verursacht. In stehendem Wasser kann die Geschwindigkeit der Wellen nach der Dispersionsrelation für Gravitationswellen[8] genau berechnet werden:

$$C = \sqrt{\frac{g\Lambda}{2\pi}}$$

Nach Bragg hängt die Wellenlänge der Wasserwelle über folgende Relation mit der Wellenlänge der zurückgestreuten elektromagnetischen Welle zusammen:

$$\lambda_{Wasser} = \frac{\lambda_{el}}{2}$$

Bei $\lambda_{el} = 10m$ ergibt sich $\lambda_{Wasser} = 5m$. Die hierfür zu erwartenden Frequenzen entsprechen den beiden Linien, die in Abb. 2.3 mit Doppler Shift at no Radial Current bezeichnet sind.

[8] mit g : Erdbeschleunigung, Λ : Wellenlänge

Aus der Verschiebung der tatsächlichen First Order Peaks gegenüber den theoretisch zu erwartenden Werten kann schließlich auf die Strömung geschlossen werden, die den Seegang zum Zeitpunkt der Messung transportiert hat.

Die Abbildungen 2.4 und 2.5 zeigen von den einzelnen Stationen gemessene Strömungsfelder. Das sind aber nur Radialkomponenten der tatsächlichen Strömung. Die Radialkomponenten mindestens zweier Messstationen lassen sich wie in Abbildung 2.6 zu einem Gesamtbild zusammenfassen.

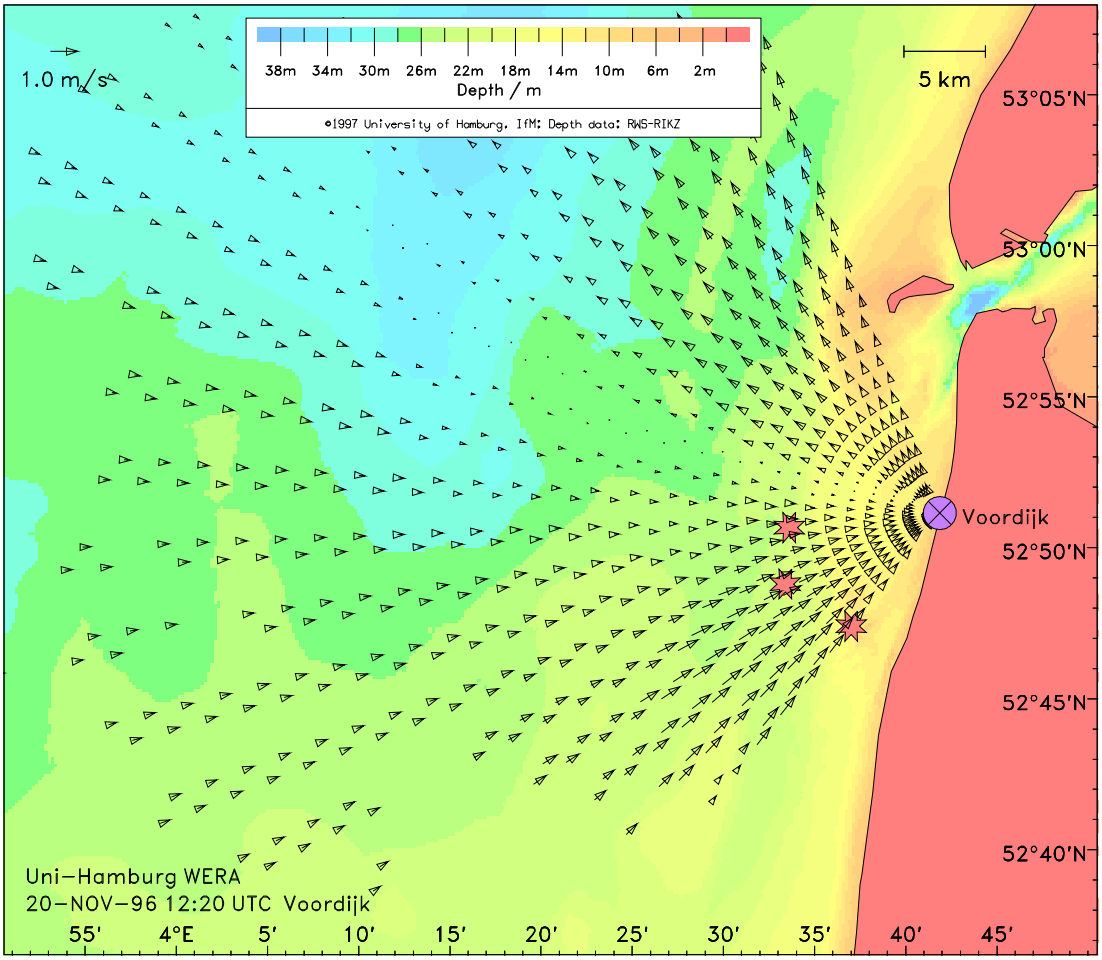

Abbildung 2.4: *Radialkomponente des Strömungsfeldes, die von einer der Messstationen (⊗ Voordijk) gemessen wurde. Offensichtlich werden von einer einzelnen Messstation nur die Geschwindigkeitsanteile gemessen, die auf die Station zu oder von ihr weg gerichtet sind. (siehe [WERA])*

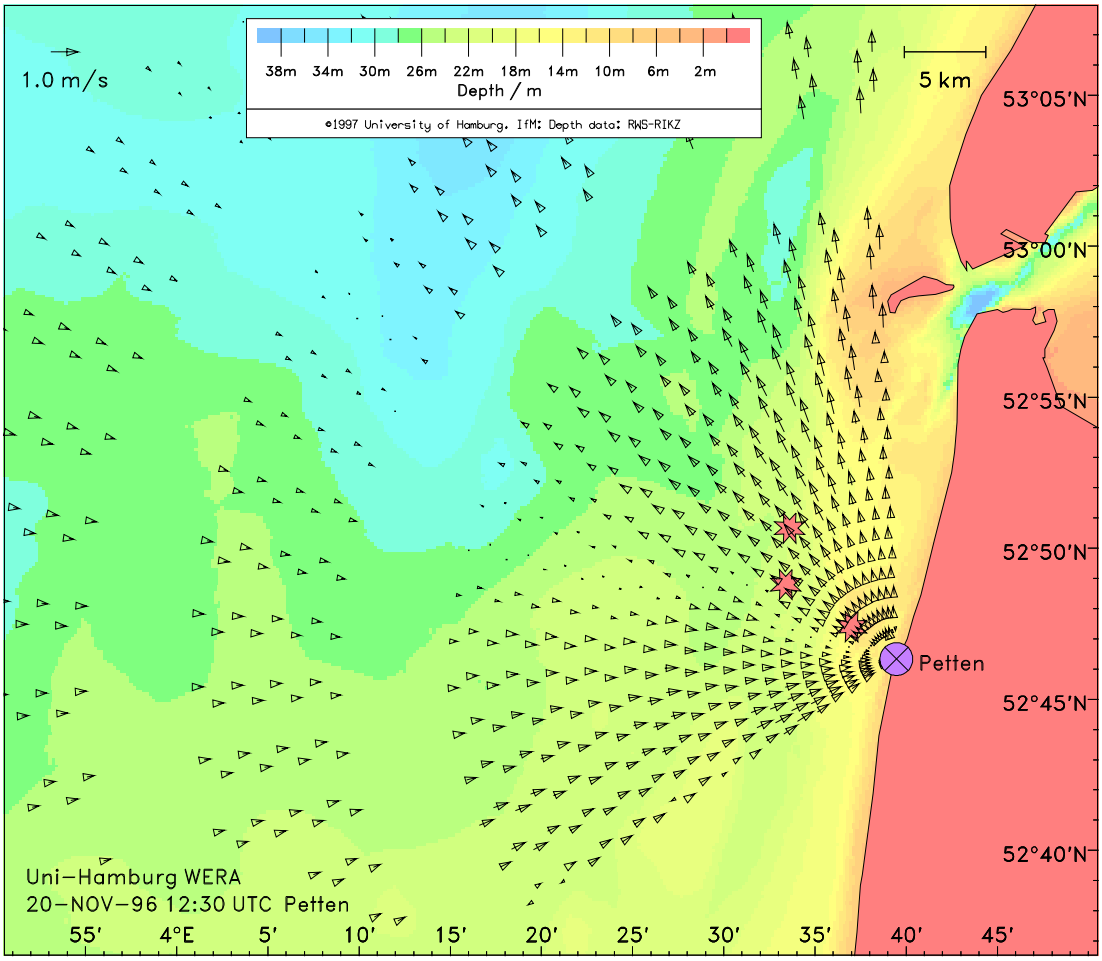

Abbildung 2.5: *Radialkomponente des Strömungsfeldes, die von der zweiten Messstation (⊗ Petten) zehn Minuten später gemessen wurde. In diesem Fall sind zwei Entfernungsbereiche durch Störungen unbrauchbar. (siehe [WERA])*

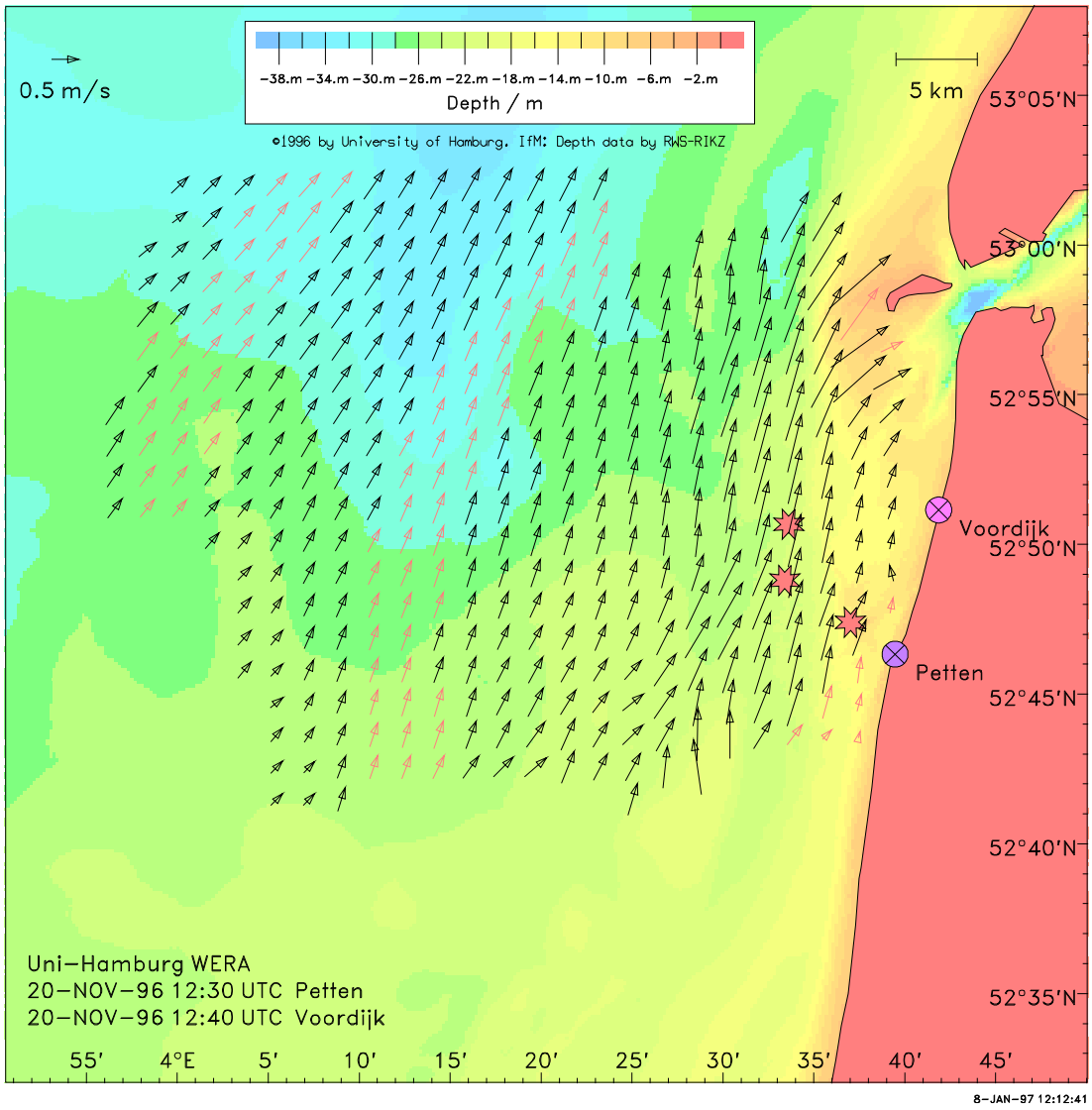

Abbildung 2.6: *Gesamtströmungsfeld berechnet aus den beiden von den Radarstationen (⊗) gemessenen Radialkomponenten. Die bereits in Abb. 2.5 sichtbaren Störungen führen dazu, dass die gleichen Entfernungsbereiche hier nicht direkt berechnet werden können sondern interpoliert werden müssen. (siehe [WERA])*

3 Hardware des Messrechners

Nachfolgend soll zunächst kurz die Hardware des alten Messrechners beschrieben werden, um daraus die Anforderungen für den Entwurf des neuen Systems herzuleiten.

3.1 Altes System

Das System wurde Mitte der 1990er Jahre gebaut. Es basiert auf einer VME-Bus[1] Hauptplatine und ist mit einem DEC Alpha Prozessor und 64MB RAM bestückt. Es gibt keine lokalen Massenspeicher, der Bootvorgang erfolgt über lokale Netzwerk, das den Messrechner mit dem Steuerrechner verbindet. Die Analog-Digital Wandlung erfolgt mit zwei 16bit A/D Wandlerkarten, deren Eingang jeweils mit einer 16 Kanal Analog Multiplexer Karte verbunden ist. Diese vier Karten sind per VME-Bus in das System integriert.

Die Verwendung der analogen Multiplexer hat zur Folge, dass nicht alle Kanäle mit der gleichen Genauigkeit gemessen werden können. Der Grund hierfür ist, dass alle Eingangswerte zur gleichen Zeit speichert werden, um sie dann aber nacheinander mit dem Analog/Digital Wandler zu digitalisieren. Die dielektrischen Absorbtion der analogen Speicher verursacht messbare Verschlechterungen der später digitalisierten Werte im Vergleich zu den ersten (siehe Datenblatt [DVX2601]). Dieser Umstand wurde in der ursprünglichen WERA-Software besonders berücksichtigt und führte zu einer komplizierten, über mehrere Ebenen indizierte Datenstruktur.

Zunächst die einzelnen Komponenten des alten Systems (auch Abb. 3.1):

Hauptplatine
digital EBV 12-AD
LAN (AUI→10baseT), 2×Seriell on board

CPU
DEC 21066 Alpha AXP 160MHz

Speicher
64MB RAM

[1] Standard für eine Busarchitektur

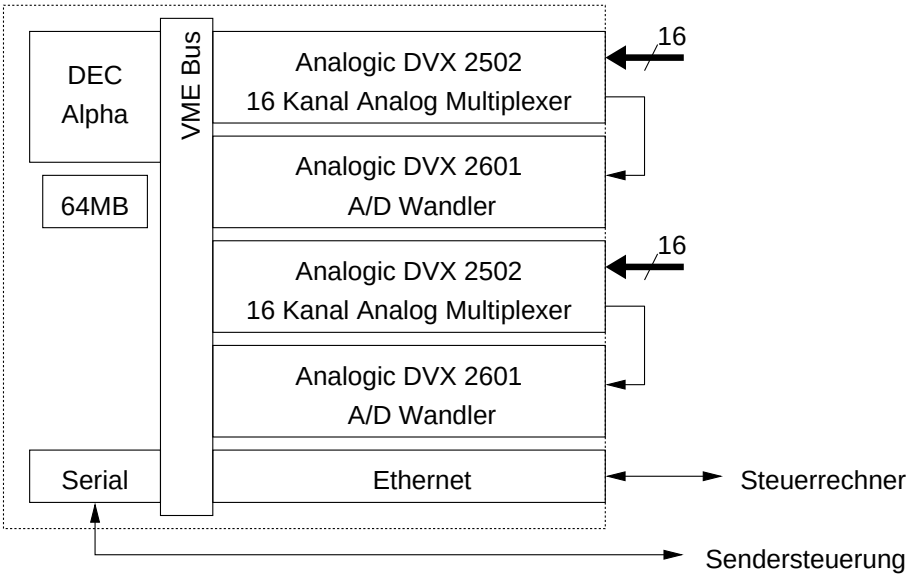

Abbildung 3.1: *Alter Messrechner mit analogen Multiplexern und externer Sendersteuerung durch seriell verbundenen PC104*

Analog/Digital - Wandlung
Analogic DVX2601 + Analogic DVX2502
2× 16bit A/D Wandlerkarte für VME-Bus (DVX2601) verbunden mit 16 Kanal Analog-Multiplexer (DVX2502), zur Messung von (bis zu) 16 Antennen (pro Antenne 2 Kanäle: I+Q je 16bit).

3.2 Anforderungen

Aus den Leistungsdaten des alten Systems ergeben sich Mindestanforderungen für das neue Systems, um damit wenigstens die gleiche Leistung wie bisher zu erhalten. Es wird aber darüber hinaus die Möglichkeit angestrebt, später weitere Funktionen in das System zu integrieren, die zum Beispiel bisher vom Zentralrechner erledigt werden oder noch gar nicht existieren. Daher ist ein Vielfaches der Rechenleistung des neuen Systems gegenüber dem alten wünschenswert, der durch die stetige Fortentwicklung der Hardware leicht zu erreichen ist.

Um einen Anhaltspunkt für die Rechenleistung des alten Alphaprozessors zu bekommen, wurde eine kleine Recherche bei *www.spec.org* durchgeführt. Diese ergab zunächst, dass die DEC 21066 Alpha AXP 160MHz dort gar nicht aufgeführt war. Als Ausweg wurden die vorhandenen Leistungswerte der beiden nächsten Verwandten benutzt, der eine mit einer etwas geringeren Taktrate und der andere mit der nächsten höheren Taktrate. Die Vergleichbarkeit leidet neben den unterschiedlichen Taktraten auch unter unterschiedlichen Cachegrößen und Hauptplatinen, RAM-Größen etc. Die Hoffnung bestand jedoch darin, we-

Tabelle 3.2: SPEC-Werte

Prozessor	SPECint95	SPECfp95
DEC 21064 150MHz	2.15	3.65
DEC 21064A 225MHz	3.66	5.71
Intel Celeron 433MHz	15.80	10.40

Quelle: www.spec.org

nigstens eine grobe Obergrenze für die Rechenleistung es alten Prozessors zu bekommen, um später nicht womöglich einen unterlegenen Prozessor im neuen System zu verwenden. Das Ergebnis ist in Tabelle 3.2 zu sehen. SPECint ist eine Maßzahl für die Rechenleistung mit Integer-Werten und SPECfp entsprechend für Gleitkomma-Operationen.

3.3 Neues System

Nachdem der Aufbau der ursprünglichen Hardware beschrieben wurde, kommen wir nun zum neuen System. Abbildung 3.2 zeigt schematisch die Komponenten in der ersten neuen Version. Abbildung 3.8 schematisiert die zukünftige Konfiguration, in der der zusätzliche Rechner (PC104) (siehe Abb. 2.1) zur Steuerung des Senders durch eine zweite Parallel IO Karte ersetzt werden soll.

3.3.1 CPU

Der alte Messrechner hatte gerade genügend Rechenleistung, um die erforderlichen FFTs in der verfügbaren Zeit durchführen zu können. Die Auslegung des neuen Systems für bis zu 64 Antennen bedingte eine CPU, die entweder gleich den Leistungsanforderungen der höchsten geplanten Ausbaustufe gewachsen ist, oder die wenigstens ohne weitere Hardware-Modifikationen durch eine genügend schnelle ersetzt werden kann. Außerdem sollte es ein Prozessor sein, der bei einem Ausfall ohne Probleme neu beschafft werden kann. Schließlich fiel die Wahl auf einen aktuellen Intel Celeron, dessen Gleitkomma-Leistung einige Reserven gegenüber dem alten Alpha-Prozessor erwarten läßt. In Tabelle 3.2 sind die SPEC-Werte des neuen Prozessors mit denen des alten zu vergleichen.

3.3.2 A/D-Wandlersystem

Das alte System besteht aus zwei A/D-Wandlern, die jeweils mit einem 16 Kanal Multiplexer mit Sample & Hold Funktionalität verbunden sind. Wie sich im Betrieb gezeigt hatte, verursachen Netzwerktransfers während der Messung Störungen in den Antennensignalen. Deshalb wird Kommunikation über das Netzwerk während der Messung soweit wie möglich vermieden, Ergebnisdateien werden erst nach Beendigung der Messung an den Zentralrechner übertragen.

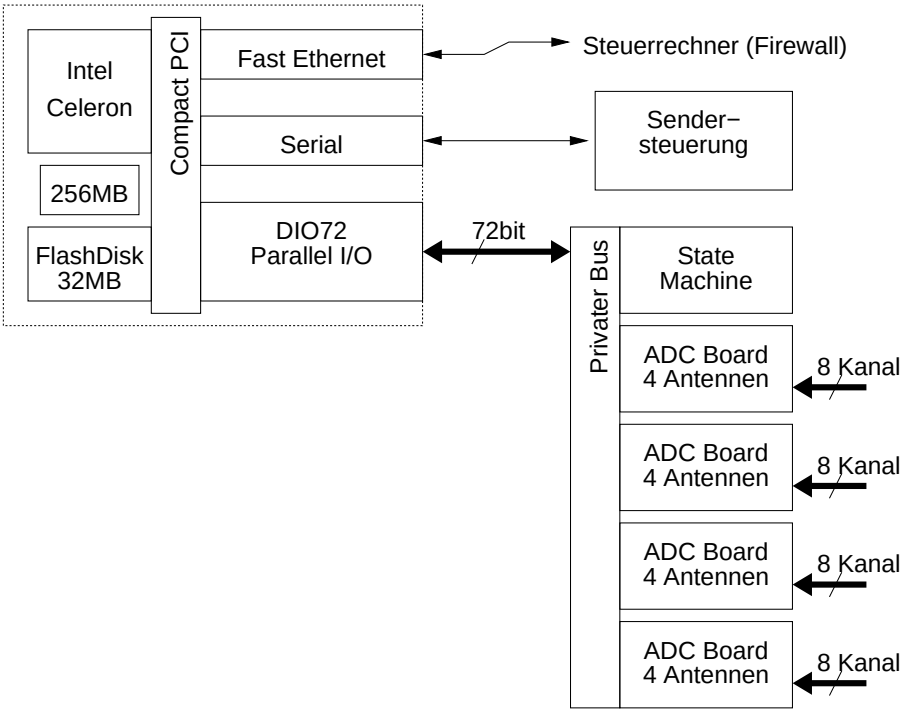

Abbildung 3.2: *Neuer Messrechner mit digitalem Multiplexing und externer Sendersteuerung wie im alten System*

Die Analog/Digital-Wandler sollten möglichst gut von Störeinflüssen abgeschirmt werden. Deshalb kam eine direkte Verbindung mit dem PCI-Bus nicht in Frage. Es wurden Wandlerkassetten entworfen, die jeweils 8 A/D-Wandler (für 4 Antennen) enthalten. Aus der Kassette wird ein privater paralleler Bus herausgeführt, über den bis zu 16 Kassetten mit dem Messrechner verbunden werden. Der Messrechner selbst greift mittels einer Parallel I/O Karte auf den Bus zu.

Außerdem sollte die Datenerfassung mit möglichst wenig Rechenaufwand des Hauptprozessors ablaufen, um diesen die Fast-Fourier-Transformationen rechnen lassen zu können. Dazu wird eine Parallel I/O Karte eingesetzt, die komplexe DMA[2]-Fähigkeiten besitzt. Zu Beginn der Messung wird der DMA mit mehreren Puffern initialisiert, die danach automatisch gefüllt werden. Der Hauptprozessor erhält lediglich nach dem Füllen eines Puffers einen Interrupt, während der Transfer schon mit dem nächsten Puffer fortgesetzt wird. Das zyklische Adressieren der Antennenkanäle erledigt eine spezielle State Machine auf dem Parallelbus ebenso wie das Triggern des DMA-Vorgangs in der Parallel I/O Karte. Selbige verfügt über 72 Bit Ein-/Ausgänge, von denen 32 Bit für das Lesen der A/D-Daten benutzt werden. So ist es möglich, mit jedem Transfer eine Antenne vollständig auszulesen (je 16 Bit pro I- und Q-Kanal) und trotzdem die Frequenz auf dem

[2]Direct Memory Access, Direkte Übertragung von Daten vom und zum Arbeitsspeicher, ohne den Hauptprozessor zu beanspruchen

Parallel-Bus so niedrig zu halten, dass Störstrahlungen für die A/D-Wandler nicht befürchtet werden müssen. Die verbleibenden 40 Bit der I/O Karte werden für die unterschiedlichen Steuerleitungen des Busses verwendet, einige sind jedoch unbenutzt. Abbildung 3.3 zeigt die Ein- und Ausgänge der verschiedenen Geräte auf dem privaten Bus. Realisiert wird der private Bus durch eine Platine (*Backplane*), auf die die Geräte gesteckt werden. Aus der Anzahl der Addressbits für die Wandlerkassetten geht hervor, dass bis zu 16 der Kassetten installiert werden können. Dazu müssen dann vier Backplanes aneinander gereiht werden, da jede vier Steckplätze für Wandlerkarten besitzt.

3.3.3 Hardware Spezifikation

Hauptplatine
SBS CL7 CompactPCI Motherboard
VGA, LAN (10/100baseTX), 2×Seriell, Parallel, 2×USB

CPU
Intel Celeron 433MHz

Speicher
256MB RAM + 32MB Flashdisk

A/D-Wandlung
Externe Backplane bestückbar mit State Machine und bis zu 16 Wandlerkarten für jeweils 4 Antennen (2 Kanäle pro Antenne: I+Q je 16bit);
Anbindung an den Messrechner über proprietären Parallelbus;
AD-Wandler (16Bit, 100kHz max. Samplerate):

Bezeichnung	THD	SFDR
Analog Devices AD 976	-96dB	96dB
Linear Technology LTC1604 (second source)	-100dB	96dB

Details zu den THD[3] und SFDR[4] Werten finden sich in [LTC1604].

Parallelbus-Schnittstelle
esd CPCI-DIO72
Busmaster-DMA-fähige digital I/O-Karte;
72 Bit einzeln als Ein-/Ausgang konfigurierbar;
Softwareunterstützung für VxWorks/Intel Pentium.

3.3.4 Betriebssystem

Ursprünglich kam das Echtzeit-Betriebssystem VxWorks 5.2 von WindRiver zum Einsatz. Die nächstliegende Möglichkeit war natürlich, die aktuelle Version 5.4

[3] THD = Total Harmonic Distortion
[4] SFDR = Spurious Free Distortion Range

des gleichen Betriebssystems zu verwenden. Allerdings wurden aus Kostengründen auch Alternativen untersucht. Der zeitkritische Character der Anwendung ließ hierfür nur wirkliche Echtzeitbetriebssysteme in Frage kommen. In Verbindung mit dem Kostenfaktor entwickelte sich RealTime Linux [Linux] zu einem Kandidaten, der als Ersatz dienen konnte. Gleichzeitig gab es aber noch ein weiteres Kriterium, das sich aus der Wahl der Schnittstelle zu den Analog/Digital Wandlern ergab: Treiber für die Parallel-IO-Karte mussten auch verfügbar sein. Diese existieren für VxWorks, aber leider nicht für RealTime Linux. Der Hersteller der Parallel-IO-Karte bot auf Anfrage die Programmierung von RealTime Linux Treibern an, allerdings zu einem Preis, der über dem für das VxWorks lag, verbunden mit zusätzlicher Programmierarbeit, die die Arbeitsgruppe hätte leisten sollen. Damit war der Kostenvorteil des Linux hinfällig oder vielmehr ins Gegenteil umgeschlagen. Unter Berücksichtigung eines weiteren Kriteriums (möglichst wenig Aufwand für die Portierung der Software) fiel die Wahl letztlich auf VxWorks 5.4.

3.4 Realer Aufbau

In Abbildung 3.4 ist der reale Aufbau zu sehen, wie er in der Entwicklungsphase im Labor benutzt wurde. Im oberen Bildteil befindet sich links der Frequenzgenerator, der vom PC104 rechts davon programmiert und gesteuert wird. Unter dem PC104 findet sich der Messrechner (CL7), der per RS232 Kommandos an diesen schickt und der hier mit einer DIO72 parallel IO Karte bestückt ist.

Die linke Hälfte dieser Ebene enthält die ADC-Boards (hier vier) und ganz links die State Machine. Die State Machine ist durch zwei Koaxialkabel mit dem Frequenzgenerator darüber verbunden, durch die sie den Clock (45 MHz) und das davon abgeleitete Start ADC vom Frequenzgenerator bekommt. Die Backplane des ADC-Systems befindet sich im Inneren des ganzen und ist deshalb hier nicht sichtbar. Um mehr als vier ADC-Boards an das System anzuschließen, müssten weitere Backplanes kaskadiert werden, bis zu vier verkettete Backplanes sind vorgesehen.

Im unteren Teil des Photos sind außerdem zwei Ebenen mit Empfängermodulen erkennbar, von denen jedes mit einem ADC-Kanal eines ADC-Boards verbunden ist. Die Empfänger werden auf der Rückseite mit den Antennen verbunden. Die auf der Vorderseite freien BNC Anschlüsse dienen als Testpunkte für ein Oszilloskop; im Bild ist der linke obere Empfänger entsprechend verkabelt.

3.4.1 State Machine (STM)

Die State Machine ist ebenso wie die Backplane und die Wandlerkassetten eine Entwicklung der Arbeitsgruppe Fernerkundung in Kooperation mit der Firma Helzel.

Die State Machine steuert während der Messung den Datentransfer von den

Tabelle 3.4.1: Register der State Machine

	Register	Inhalt	
0	Status	Bit0→Go, Bit1→Error	read/write
1	ADC-DIV-LOW	Clock Divisor (Low Byte)	read/write
2	ADC-DIV-HIGH	Clock Divisor (High Byte)	read/write
3	ADC-ADDR	ADC Board Count	read/write
4	ERROR-BYTE-LOW	Error Bits Card 0-7	read
5	ERROR-BYTE-HIGH	Error Bits Card 8-16	read
6	-		
7	Version	STM Firmware Revision	read

A/D Wandlern zur Parallel IO Karte. Vor der Messung muss sie vom Messrechner mit der gewünschten Anzahl Antennen konfiguriert werden. Nachdem dieser den Sender und das externe Triggersignal aktiviert hat, läuft die gesamte Datenerfassung praktisch ohne weiteres Zutun des Hauptprozessors, da die DIO72 Karte die Daten per DMA in den Arbeitsspeicher befördert, ohne CPU-Rechenzeit zu beanspruchen.

Ein programmierbarer Logikbaustein von ALTERA bildet den Kern der State Machine. Dieser wird beim Hochfahren des Systems mit der aktuellen Firmware initialisiert und verliert die Information beim Abschalten wieder. Dadurch ist ein hohes Maß an Flexibilität gegeben, das sich bereits bei der Behebung einiger kleiner logischer Fehler bezahlt gemacht hat. Abbildung 3.7 zeigt die Programmlogik der State Machine. Es sind bis zu 8 Register (á 8 Bit) vorgesehen, die teilweise nur lesbar und teilweise schreibbar sind. Sie sind in Tabelle 3.4.1 aufgeführt.

3.4.2 Erkennung installierter ADC Boards

Sobald die State Machine mit den geeigneten Werten für die Register ADC-DIV und ADC-ADDR initialisiert ist, kann sie die Parallel IO Karte automatisch zum Auslesen der AD-Wandlerdaten veranlassen. Das Initialisieren mit den geeigneten Werten ist ihr jedoch nicht möglich, so dass dies vom Messrechner erledigt werden muss. Dabei liegt der Wert für den Clock Divisor praktisch im Programm schon fest; die Anzahl installierter ADCs muss ermittelt werden.

Abbildung 3.5 zeigt die relevanten Busleitungen während der Erkennung der installieren ADC-Boards. Zusätzlich ist der Wert des STM-Go Bits aufgezeichnet, das Low sein muss, wenn die Parallel IO Karte ADC-Adressen auf dem Bus anlegen soll. Im angehaltenen Zustand sind die Ausgangsleitungen der State Machine im TriState Zustand. Dann werden die ADC-Addr und ADC-Board-Addr Leitungen der Parallel IO Karte von Eingang (gleich TriState) auf Ausgang umkonfiguriert.

Nachdem die State Machine vom Messrechner angehalten wurde, setzt dieser die beiden ADC Adressbits auf 0, um hier einen definierten Zustand zu haben. Dann werden alle möglichen ADC-Board-Adressen (4 Bit) der Reihe nach auf

dem Bus angelegt. Das jeweils adressierte ADC-Board setzt die Leitung ADC Ident automatisch auf High und je nach internem Zustand die ADC Error Leitung auf High oder Low. Die Erkennungsroutine prüft also die Leitungen ADC Ident und ADC Error, um festzustellen, ob die gerade angelegte ADC Board Adresse belegt ist oder nicht. ADC Ident wird vom Bus auf Low gezogen, sodass bei einem High Pegel auf dieser Leitung davon ausgegangen werden kann, dass auch ein Board vorhanden ist. Dessen Fehlerfreiheit kann über die Error Leitung geprüft werden.

Nachdem alle Adressen abgefragt sind, werden die vorübergehend als Ausgänge geschalteten Leitungen der Parallel IO Karte wieder als Eingänge (wieder Tri-State) konfiguriert, damit wieder die State Machine Adressen anlegen kann.

Die Anzahl der erkannten ADC-Boards wird in das Register ADC-ADDR der State Machine geschrieben, bevor diese wieder durch setzen des STM-Go Bits gestartet wird. Da die State Machine keine ADC-Adressen auslassen kann und immer bei Null zu zählen anfängt, müssen ADC-Boards bei Adresse Null beginnend installiert werden.

3.4.3 State Machine im Messbetrieb

Die Aufnahme von Samples läuft folgendermaßen ab (s. Abb. 3.6):
Bei gesetztem Go-Bit wartet die State Machine auf eine fallende Flanke des externen ADC Trigger. Diese Flanke veranlasst gleichzeitig die ADCs zum Aufnehmen der aktuellen analogen Signale, wofür max. 10 μs benötigt werden. Nach deren Verstreichen kann die State Machine mit dem Adressieren der ADCs beginnen, um die soeben festgehaltenen Daten in den Messrechner zu transferieren.

Es wird also (beginnend bei Null) die Adresse des Analog Digital Wandlers auf den Bus gelegt. Dann wird DMA Read auf Low gesetzt, um dem ADC zu signalisieren, dass er seine Daten auf den Bus legen soll. Schließlich wird auch auf DMA Trig eine fallende Flanke erzeugt, die der Parallel IO Karte das Vorliegen neuer Daten anzeigt und einen DMA Transfer anstößt. Nach einer gebührenden Pause, während der die Parallel IO Karte die Daten abholt, wird noch das Error Bit des aktuellen ADC Boards geprüft und ggf. in das entsprechende ERROR-BYTE Register der STM übertragen. Dann kann die interne ADC Adresse der State Machine inkrementiert bzw. bei Überschreiten von ADC-Count auf Null zurückgesetzt werden. Wurde ADC-Count noch nicht erreicht, setzt sich der Vorgang mit der nächsten ADC-Adresse fort, andernfalls muss erst wieder auf eine fallende Flanke des externen ADC Trigger gewartet werden.

Dieser Vorgang läuft, so lange das STM-Go Bit gesetzt (und kein Fehler in der State Machine aufgetreten) ist. Die Software des Messrechners muss also nur das STM-Go Bit setzen und den DMA-Transfer entsprechend vorbereitet haben, um Messwerte in beliebiger Menge einzulesen. Sobald das Bit zurückgesetzt wird, bricht die State Machine den Vorgang ab.

Die ADC Ident Leitung sollte bei gültiger ADC Adresse immer High sein, aber in

der Praxis kann ein Gerät im Betrieb auch ausfallen. Für ADC Error gilt prinzipiell das gleiche, bei gültiger ADC Adresse sollte die Leitung Low sein, im Fehlerfall (für den sie letztendlich gedacht ist) erscheint hier aber ein High.

3.5 Ausblick

Obwohl Realtime Linux in dieser Phase des Projekts nicht mehr zum Einsatz kommen wird, soll dessen Verwendung zu einem späteren Zeitpunkt erneut überdacht werden.

Die nächste Hardware-Version soll eine zweite DIO72 Karte enthalten (Abb. 3.8), über die die Senderprogrammierung und Steuerung erfolgen soll. Damit kann der zur Zeit noch benötigte (seriell angeschlossene) PC104 entfallen, wodurch die Messstation insgesamt kleiner und einfacher wird.

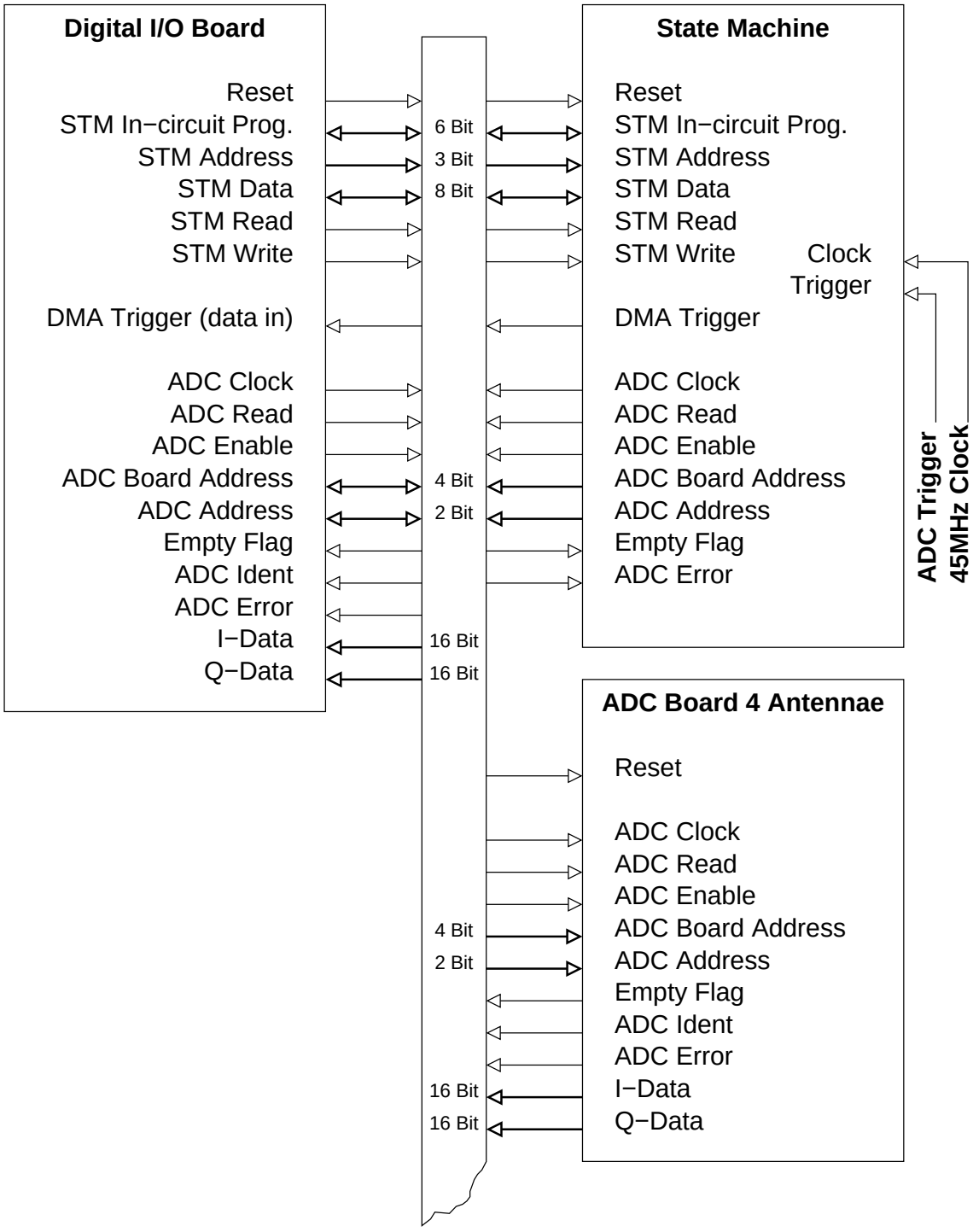

Abbildung 3.3: *Leitungen auf dem privaten Bus.*
Es können max. 16 ADC Boards für bis zu 64 Antennen angeschlossen werden.

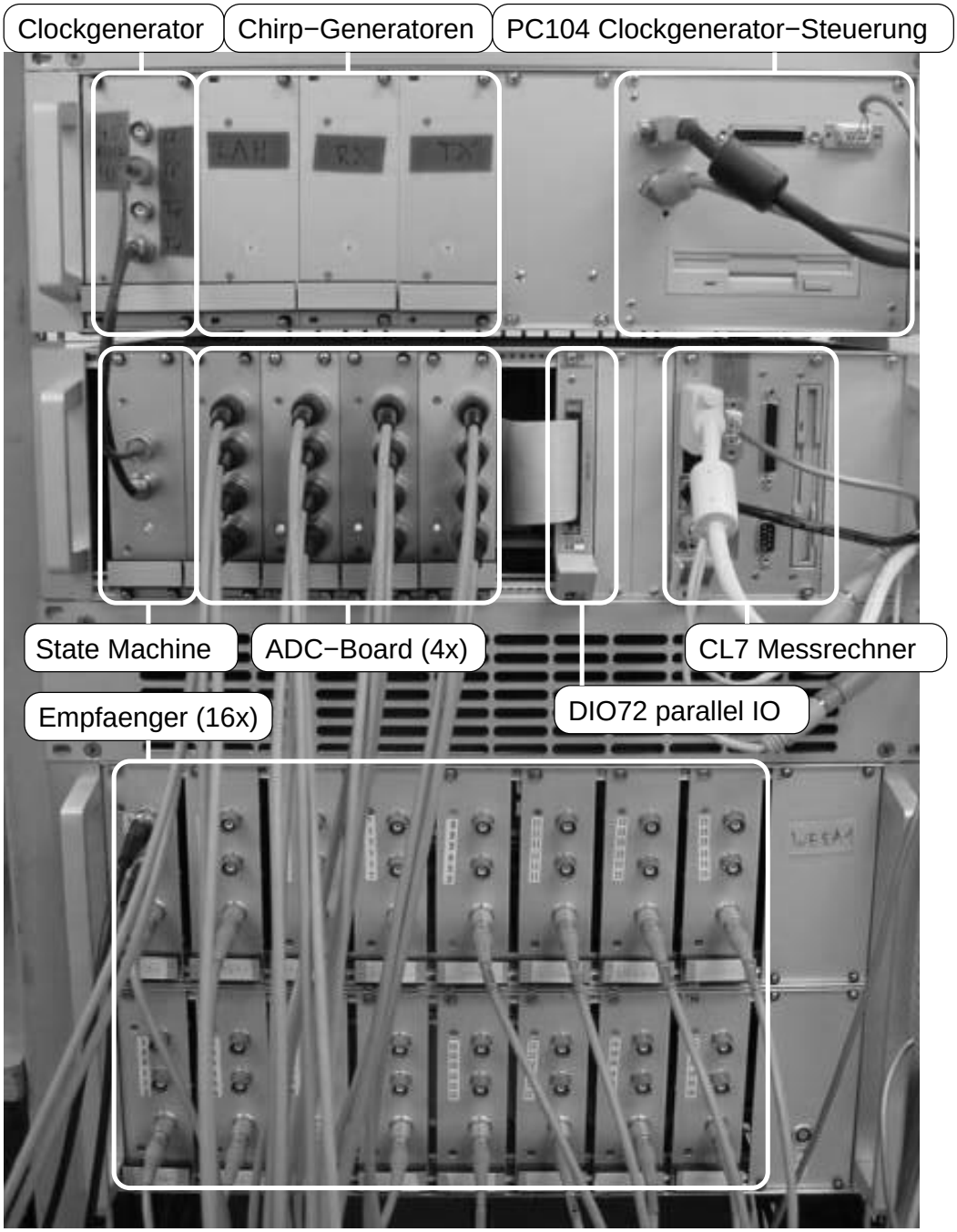

Abbildung 3.4: *Reales System im Labor während der Entwicklung; Bestückung mit 16 Antennenkanälen*

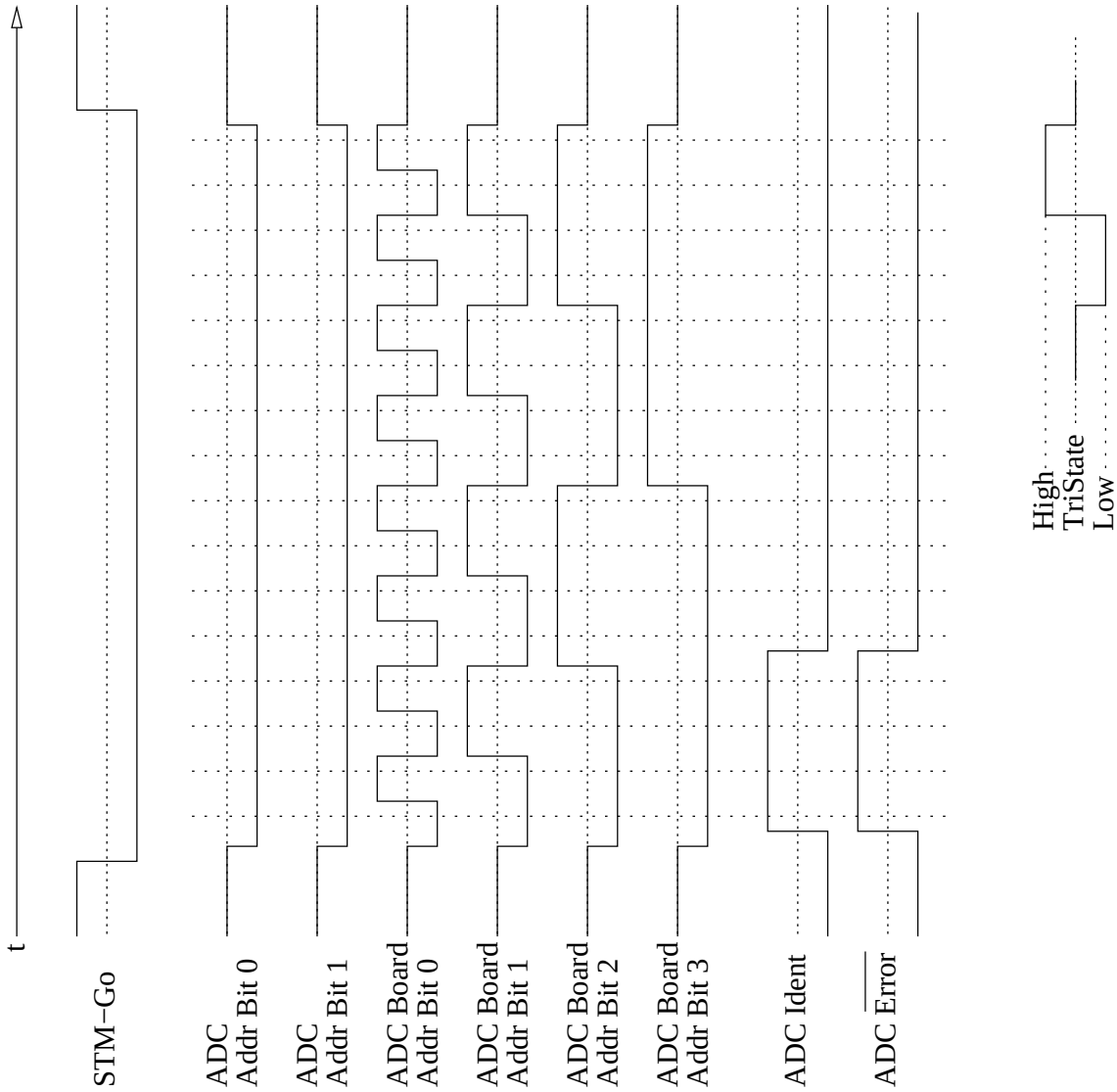

Abbildung 3.5: *Zeitlicher Verlauf der Bussignale beim Erkennen der vorhandenen ADC-Boards. (*STM-Go *ist keine Busleitung, hier aber trotzdem aufgeführt.) Im Beispiel sind 4 Boards installiert, die mit* ADC-Ident *High und* ¬ADC-Error *High antworten; diese Leitungen werden während der Abfrage der übrigen Adressen nicht mehr High und von den Pull-Down Widerständen des Busses auf Low gezogen.*

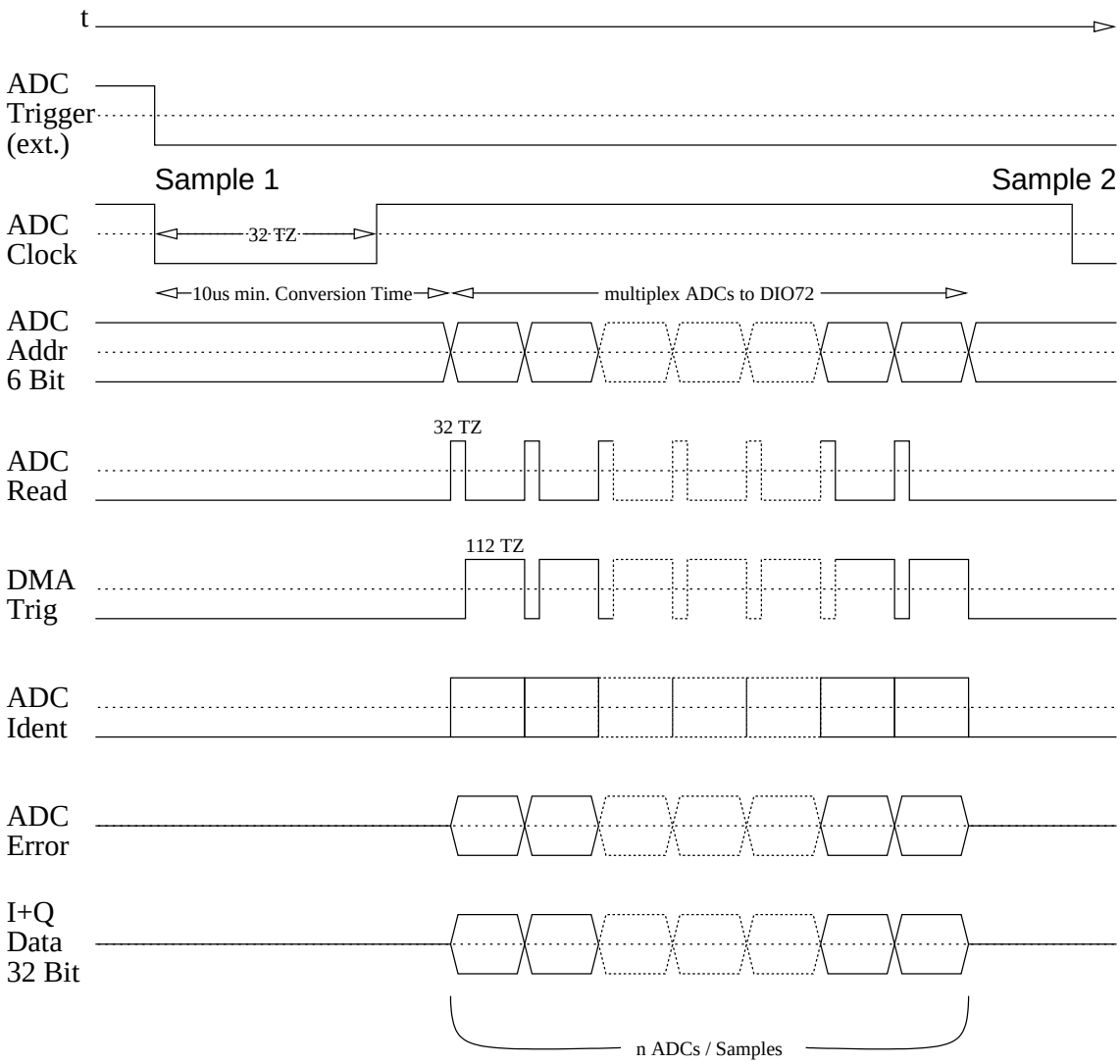

Abbildung 3.6: *Zeitdiagramm der Bus-Signale während der Messung. Bei jedem externen* DMA Trigger *konvertieren alle ADCs gleichzeitig ihren Messwert und werden dann der Reihe nach adressiert und ausgelesen.* ADC Trigger *ist auch Startsignal für den Frequenzgenerator, der auch das gleiche Taktsignal benutzt. Dadurch wird* ADC Clock *synchron mit dem Chirp erzeugt.*

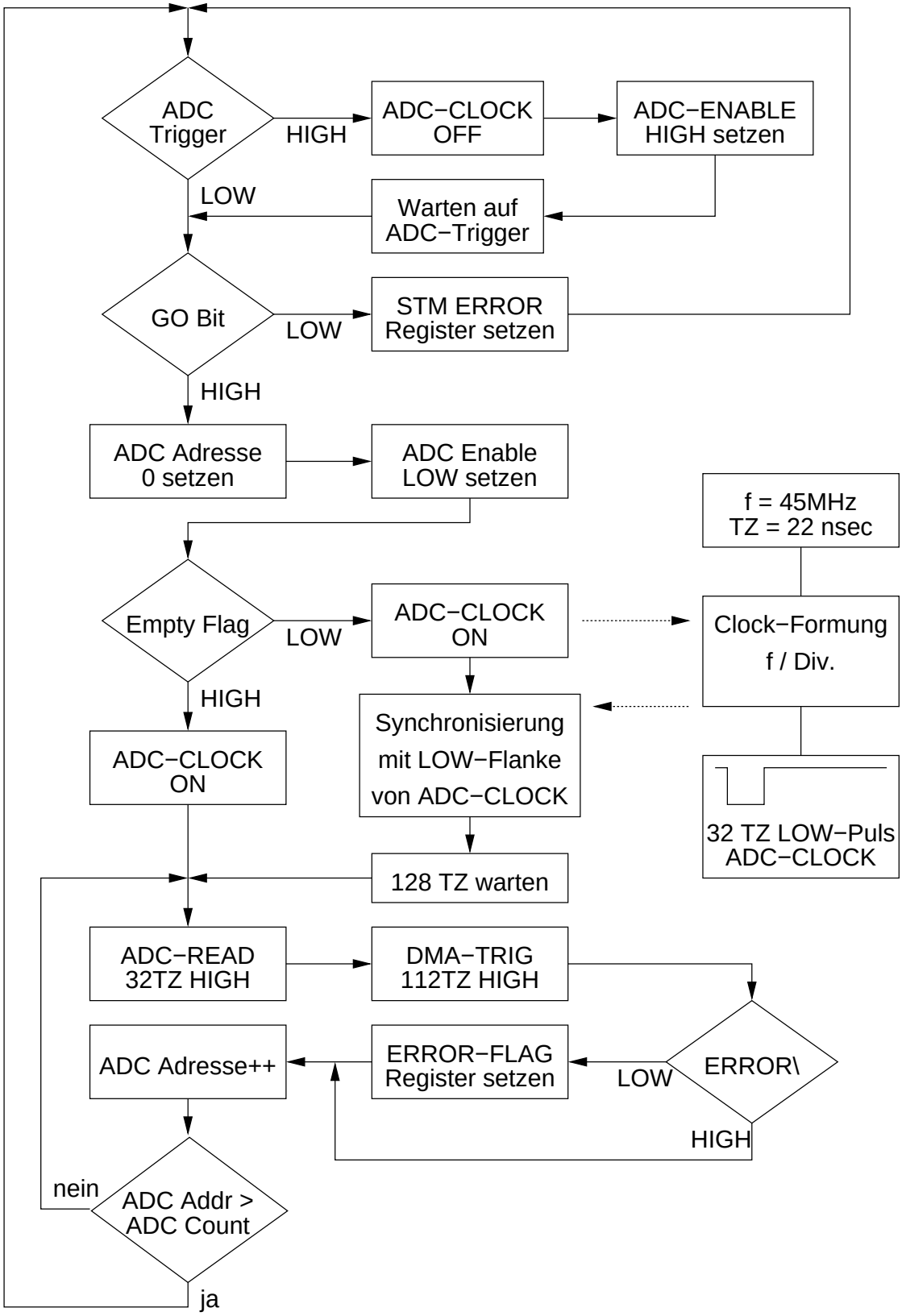

Abbildung 3.7: *Flussdiagramm der State Machine Firmware*

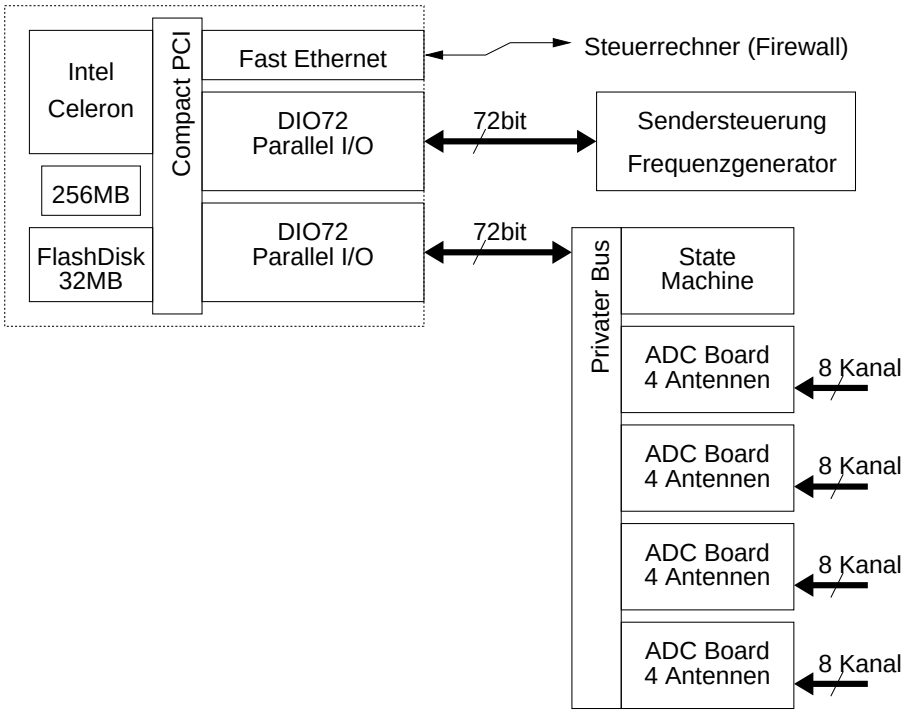

Abbildung 3.8: *Neuer Messrechner mit integrierter Sendersteuerung über zweite Digital IO Karte anstatt durch serielle Verbindung zum PC104*

4 Software

Die Software des Systems läßt sich grob in zwei Hauptfunktionsbereiche unterteilen:

- Netzwerk-Kommunikation
- Datenerfassung und Auswertung.

Beide Funktionsbereiche haben jeweils eine zentrale Task, die nur einmal im System existiert.

4.1 Netzwerk-Kommunikation

Beim Hochfahren des Systems wird die zentrale Kommunikationstask gestartet. Sie wartet auf ankommende Socketverbindungen vom Netzwerk und erzeugt für jede eine eigene Task, die dann die tatsächliche Kommunikation mit der Gegenstelle durchführt. Über die Verbindung werden zunächst die Konfigurationsdaten für die durchzuführende Messung empfangen. Da aber nur eine Messung zur Zeit laufen kann, geht dies nur, wenn das System gerade frei ist. Ansonsten wird ein *BUSY* zurückgegeben und die Verbindung beendet. In dem Fall muss es die Gegenstelle zu einem späteren Zeitpunkt noch einmal versuchen. Ist das System aber verfügbar, so kann nach Übertragung der Konfiguration die Messungen gestartet werden. Das Ende der Messung wird der Gegenstelle dann wieder mitgeteilt.

Obwohl mehrere Socketverbindungen gleichzeitig bestehen können, kann immer nur eine Messung zur Zeit stattfinden. Die Messung kann auch nur über die Verbindung gesteuert werden, die sie initiiert hat. In der aktuellen Software ist keine Möglichkeit zum Abbruch einer laufenden Messung vorgesehen, ebensowenig wie die Möglichkeit, die Steuerung zwischendurch an eine andere Verbindung zu übergeben. Vorgesehen ist jedoch schon jetzt, Funktionen die keinen exklusiven Zugriff benötigen, mehreren Gegenstellen auch gleichzeitig zur Verfügung zu stellen. Ein Beispiel hierfür ist die Abfrage der Zeit oder auch des Ortes der Station, die das System per GPS[1] ermitteln könnte (hierfür existiert allerdings derzeit keine Hardware).

[1] Global Positioning System, satellitenbasiertes weltweites Positionsbestimmungssystem errichtet und kontrolliert von den USA

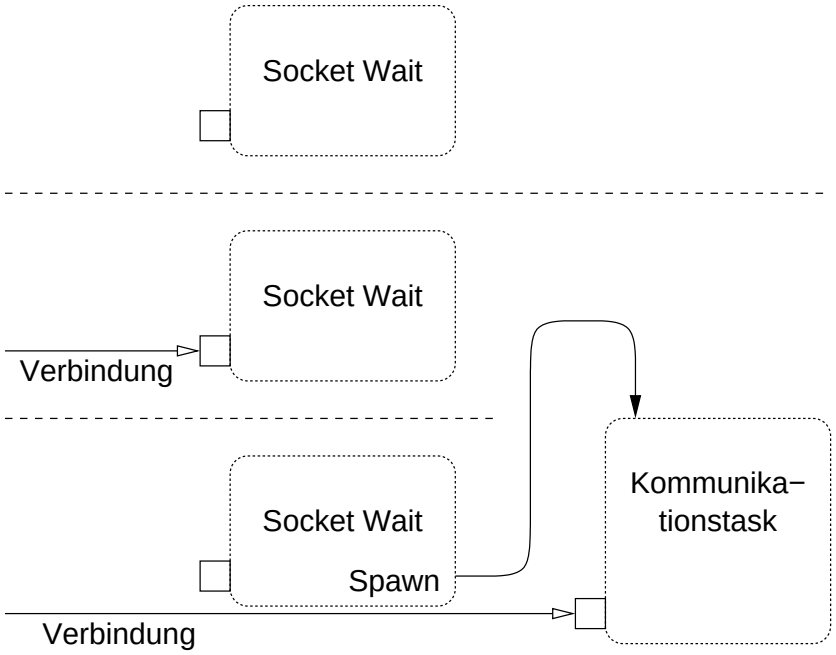

Abbildung 4.1: *Kommunikationstasks, um von außen Verbindung zum System aufzunehmen und es zu steuern*

Verbesserungsmöglichkeiten für zukünftige Versionen:

- Abbruch einer bereits laufenden Messung.
- Übernahme der Steuerung durch eine neue Verbindung, wenn die ursprüngliche Verbindung unterbrochen wurde.
- Fortschrittsmeldungen während der laufenden Messung.
- Fortschrittsabfragen auch von anderen Verbindungen als der Steuerung.
- Protokoll zur Einstellung / Übertragung der Sendeleistung

4.2 Datenerfassung und Auswertung

Abbildung 4.2 zeigt die für die tatsächlichen Messwerte-Erfassung und Auswertung benutzten Tasks mit den wichtigsten Datenstrukturen. Zu Beginn einer Messung müssen als erstes die erforderlichen Datenstrukturen angelegt und die bereits erwähnten Tasks erzeugt werden. Dies soll in der Abbildung durch den von Spawn ausgehenden verzweigenden Pfeil symbolisiert werden. Die vertikale Anordnung der Kästchen für die einzelnen Tasks verdeutlicht den Ablauf des Programms bzw. den Fluss der Daten. Nach dem Enable, welches nur den Semaphor A setzt, wartet die Hauptroutine einfach, bis Semaphor H das Ende der

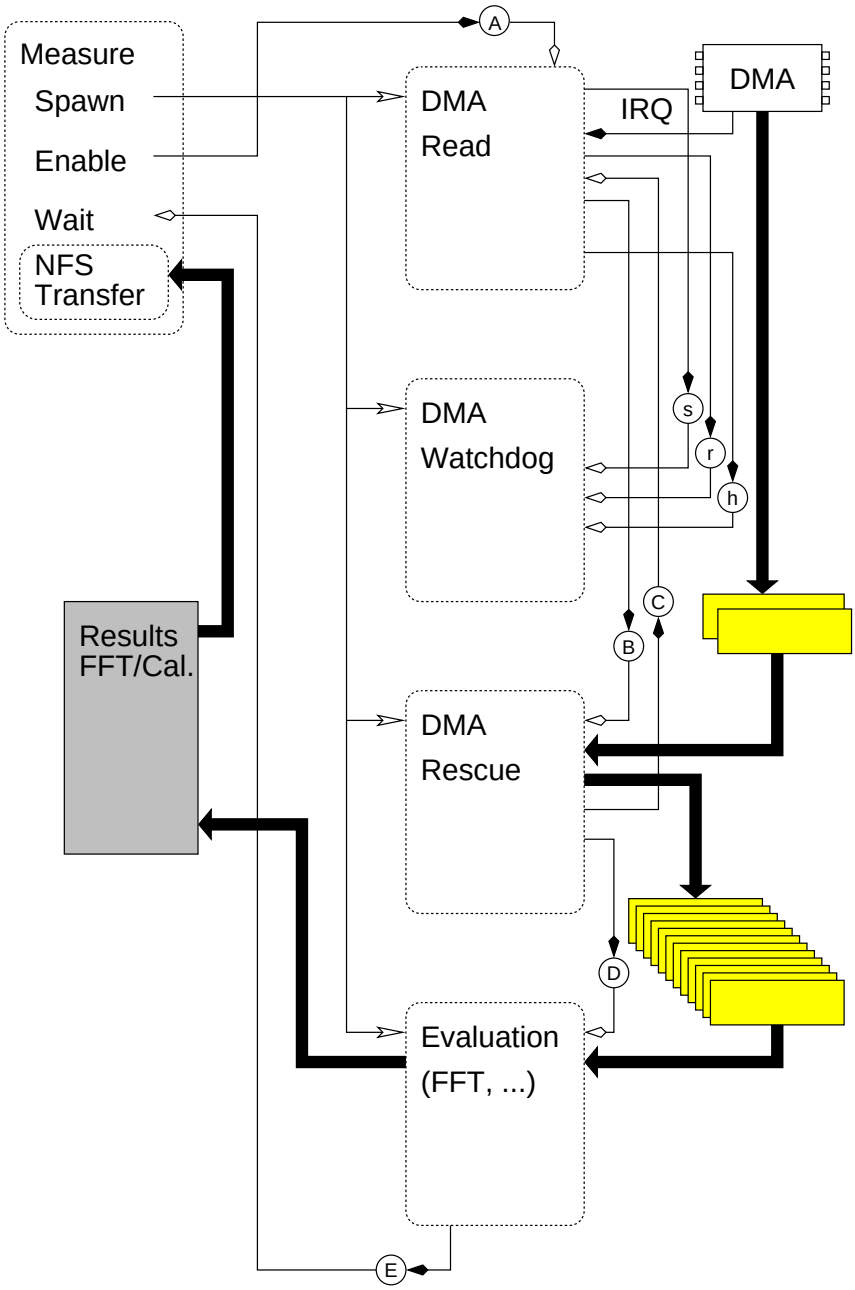

Abbildung 4.2: *Tasks der Datenerfassung und Auswertung mit ihren wichtigsten Datenstrukturen*

Messung signalisiert, um danach die Ergebnisse all der jetzt aber noch in der Zukunft liegenden Mühen zum Steuerrechner zu übertragen. Das geschieht mittels NFS. (Der Versuch, die Daten z.B. per FTP zu übertragen, scheiterte daran, dass das Betriebssystem bei FTP-Transfers Dateien erst vollständig im RAM puffert, bevor sie zum Zielrechner übertragen werden.)

DMA Read

Sobald die DMA Read Task von der Hauptroutine mittels einem Semaphor A signalisiert bekommt, dass die Messung gestartet wurde, wartet sie auf Interrupts von der DMA Hardware. Diese treten immer dann auf, wenn die Digital I/O Karte einen Puffer mit Rohdaten gefüllt hat und zum nächsten DMA-Puffer gewechselt hat. Jeder DMA-Interrupt veranlasst DMA Read dazu, den Semaphor C zu nehmen und den Semaphor B zu setzen. Die Verfügbarkeit des Semaphors C signalisiert, dass der Pufferinhalt des vorhergehenden Durchgangs in Sicherheit gebracht wurde. Es wird hier auch das Ende der Messung erkannt, indem die DMA-Interrupts mitgezählt werden. Ist die geforderte Anzahl Messungen erreicht, deaktiviert diese Task den DMA und gibt an DMA Rescue anstatt einer normalen Pufferadresse ein NULL.

DMA Rescue

Auf das Setzen des Semaphors B wartet bereits die Task DMA Rescue. Hiermit wird ihr nämlich das Vorhandesein von Daten im DMA-Puffer angezeigt, die vor dem Überschreiben gesichert werden müssen. Nachdem der Puffer in einen neu allozierten Speicherbereich kopiert wurde, können die Semaphoren C und D gesetzt werden. C bedeutet, dass der DMA-Puffer gesichert wurde. D bedeutet, dass die Evaluation Task Eingabedaten zur Verfügung hat.

Evaluation

Einen Sonderfall stellt den Semaphor D dar. Alle anderen Semaphoren sind von binärer Natur, das heißt, dass sie auch nach mehrfachem Setzen (ohne zwischendurch geleert zu werden) nur ein Mal genommen werden können. Wäre dies auch bei D der Fall, so würde das Verwenden eines mehrfachen Puffers keinen Sinn machen. Dann könnte die Evaluation Task das Vorhandensein von mehr als einer gefüllten Pufferseite nicht mehr erkennen. Deshalb ist D ein zählender Semaphor.

In Abhängigkeit von der Art der Messung wird eine Kalibrationstask oder eine FFT-Task zur Auswertung erzeugt. Beide haben aber äusserlich das selbe Verhalten. Sie warten auf das Setzen von D, verarbeiten die im Puffer verfügbaren Rohdaten und schreiben ihre Ergebnisse nach Results. Das Ende der Messung wird wieder durch einen NULL Puffer erkannt und mit dem Setzen von E quittiert.

Alle drei Tasks terminieren, nachdem sie das Ende der Messung erkannt haben und abschließende Aufräumarbeiten durchgeführt sind.

DMA Watchdog

Die soweit beschriebenen Tasks gehen davon aus, dass der DMA-Transfer ohne Schwierigkeiten abläuft. Da dies aber keine vollständig realistische Annahme ist, sollten Fehler beim Transfer der Daten in den Messrechner zumindest erkannt und nach Möglichkeit auch die jeweilige Ursache analysiert werden. Eine Ursachenerkennung ist zum jetzigen Zeitpunkt nicht implementiert. Aber es existiert eine Fehlererkennung, die darauf beruht, dass jeder Sweep gemäß dem Messverfahren etwa 0.26 Sekunden dauert und demzufolge auch in diesen Zeitabständen DMA-Transfers abgeschlossen sein sollten. Es wurden also drei weitere Semaphoren `s`, `r`, `h` und die `DMA Watchdog` Task erzeugt. Die Semaphoren `s` und `h` teilen der Task den Beginn und das Ende der Messung mit. In der laufenden Messung prüft die Task nun mit einem Timeout von 5 Sekunden ab, ob der Semaphor `r` gesetzt ist. Ist das nicht der Fall, so kann bzw muss von einem Fehler bei der Messwerterfassung ausgegangen werden. Als Ursachen sind z.B. Softwarefehler im Treiber der Digital I/O Karte oder im Messprogramm oder im Steuerprogramm der State Machine aber auch Hardwarefehler denkbar. Tritt ein DMA-Fehler auf, dann kann die DMA-Task den Semaphor `r` nicht setzen. Daraufhin gibt die Watchdog-Task anstelle der DMA-Task ein NULL an die Folgetasks weiter, damit die Messung beendet wird. Ausserdem wird ein internes Flag gesetzt, das den DMA-Abbruch anzeigt. So kann die Hauptmesstask erkennen, ob der private Bus und die State Machine neu initialisiert werden sollten.

Auswertungen

Es gibt zwei Methoden der Auswertung. Je nachdem, ob das System kalibriert werden soll oder eine normale Messung erfolgen soll, wird eine der beiden Methoden verwendet.

Unabhänging von der aktuellen Messmethode ist aber die Datenstruktur, die als Eingabe für die Auswertungsroutine dient. In Abbildung 4.3 ist die Anordnung der Daten im DMA-Puffer illustriert. Bei jedem Triggersignal müssen die Werte aller Antennenkanäle eingelesen werden. Daher werden die Werte vom DMA hintereinander in den Puffer geschrieben. Beim nächsten Takt werden wieder alle Kanäle hintereinander in den Puffer geschrieben. Das heißt, dass die Zeitserien der einzelnen Kanäle nicht fortlaufend im Puffer existiert, sondern regelmäßig miteinander vermischt sind. Es wäre vielleicht wünschenswert, wenn der DMA-Baustein der Digital IO Karte die Kanäle gleich in eigene Puffer schreiben könnte. Im Nachhinein stellt sich dieser Wunsch aber als gar nicht so dringend heraus, denn die Routinen der FFTW-Bibliothek[2] sind auf genau so zerstückelte Eingaben bereits ausgelegt. Im Fall der Kalibration ist es auch nicht so tragisch, dass die Daten nicht schon vorsortiert sind, da in diesem Fall ja durch die nicht benötigte FFT genügend Rechenzeit zum Sortieren verfügbar ist.

[2] siehe [FFTW]

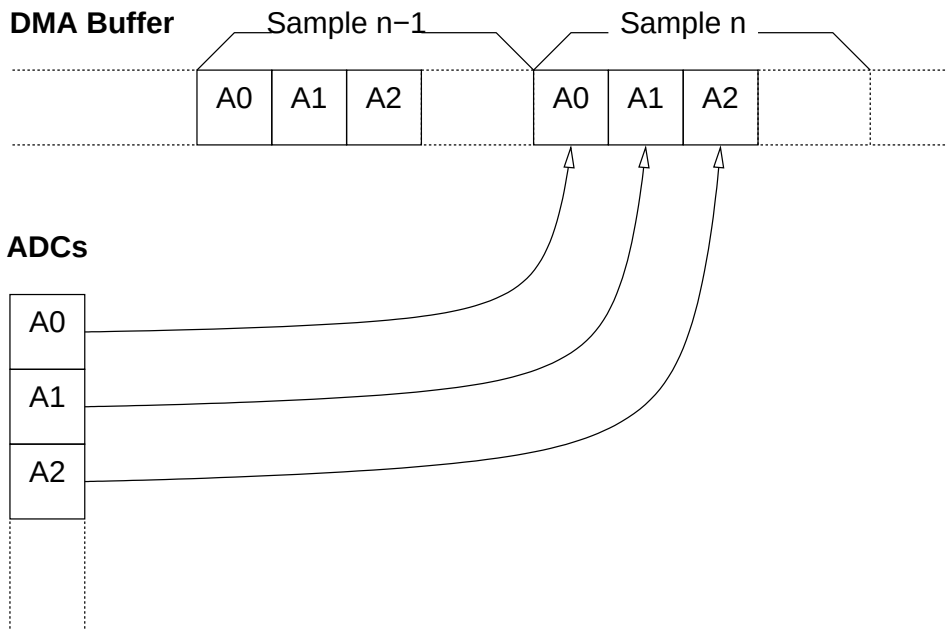

Abbildung 4.3: *Datenstruktur, die vom DMA in den Arbeitsspeicher geschrieben wird. Jeder Messwert besteht aus I- und Q-Teil (je 16 Bit).*

4.2.1 Fast Fourier Transformation (normale Messung)

Bei dieser Art der Messung wird für jede Antenne und jeden Chirp separat eine FFT gerechnet. Die Routinen zur Berechnung waren in der alten Software von einem Fortran to C Converter generiert worden und daher leider nicht direkt verständlich. Viel schwerwiegender war aber, dass die Daten in Feldern gehalten wurden, deren komplexe Indizierung zu durchschauen einfach nicht gelingen wollte. Die Messwerte waren nicht einfach nach aufsteigender Kanalnummer sortiert, sondern als erstes kamen die Kanäle 1, 2, 3 und 4, danach 16 und 15 und danach die dazwischen liegenden Kanäle in auf den ersten Blick eher willkürlich wirkender Folge. Hierfür gab es zwei Gründe. Erstens die analoge Multiplexer Hardware, durch die die Genauigkeit der Kanäle mit zunehmender Haltezeit der Messwerte abnimmt. Zweitens wurde ursprünglich eine besondere Anordnung der Antennen benutzt, bei der zwei Antennen in der Mitte der Reihe voreinander standen. Diese beiden Antennen sollten mit besonders hoher Genauigkeit gemessen werden, danach aber die äusseren Enden der Reihe.

Glücklicherweise gibt es diese mit der neuen Hardware nicht mehr, da die Kanäle alle gleichzeitig digitalisiert werden und es keinen Unterschied macht, wann der einzelne Messwert tatsächlich eingelesen wird. Es galt also nur noch, eine Bibliothek für die FFTs zu finden und einzubinden. Zum Einsatz kommt nun eine Bibliothek namens FFTW (siehe [FFTW]). Sie kann mit unterschiedlich strukturierten Ein- und Ausgabefeldern rechnen. Die Rohdaten sind in dieser An-

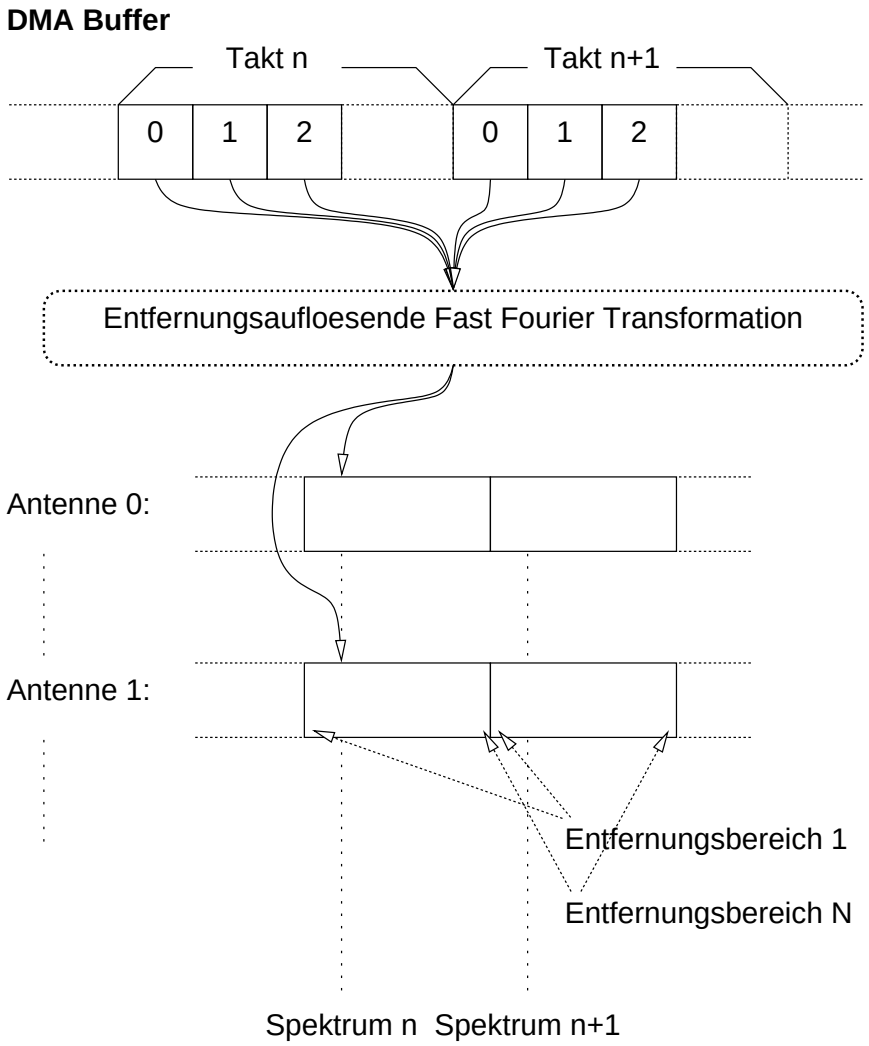

Abbildung 4.4: *Von der FFT erzeugte Datenstruktur. Die Eingabewerte wurden bereits mit den Kalibrationsdaten verknüpft. Pro Antenne existiert ein Ausgabepuffer, zur Aufnahme eines Frequenzspektrums pro Chirp.*

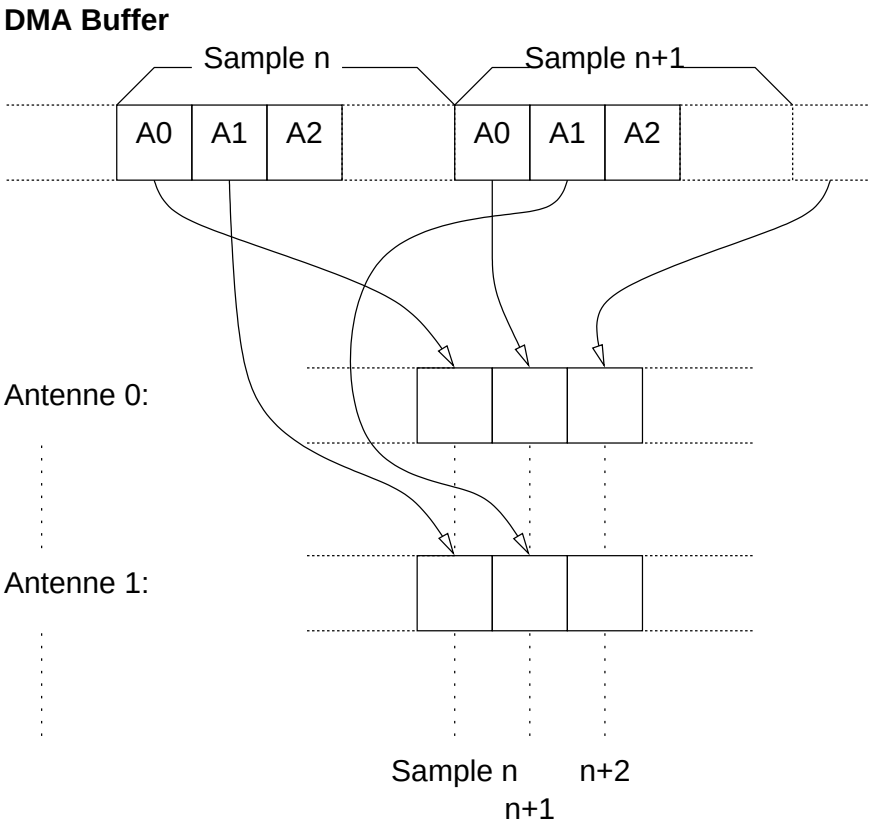

Abbildung 4.5: *Datenstruktur, die von der Kalibrationsmessung erzeugt wird. Die in den Eingangsdaten noch hintereinander liegenden Messwerte der verschiedenen Antennen werden in je einen eigenen Puffer pro Antenne kopiert und ggf. gemittelt.*

wendung nämlich nicht so geordnet wie die gewünschten Ergebnisse. Als Ergebnis wird für jede Antenne ein eigenes Spektrum benötigt, während die Eingabe aber Messwerte unterschiedlichen Antennen aufeinander folgen. Die FFTW-Bibliothek kann aus Eingaben in der hier gegebenen Anordnung die geforderten Ausgaben direkt berechnen (s. Abb. 4.4).

Aus dem Ergebnis der FFT wird nur ein kleiner Teil als tatsächliches Ergebnis verwendet, nämlich die niedrigsten 64 Frequenzen (bei 64 Entfernungsbereichen) im negativen Wertebereich. Diese Anzahl und der Offset im Frequenzbereich sind programmseitig variabel gehalten.

4.2.2 Kalibration

Ziel dieser Messart ist die Erzeugung von Kalibrationsdaten für normale Messungen. Derartige Kalibrationsdaten werden dann in der normalen Messung noch vor der FFT auf die Rohdaten angewendet, um z.B. Phasenverschiebungen oder

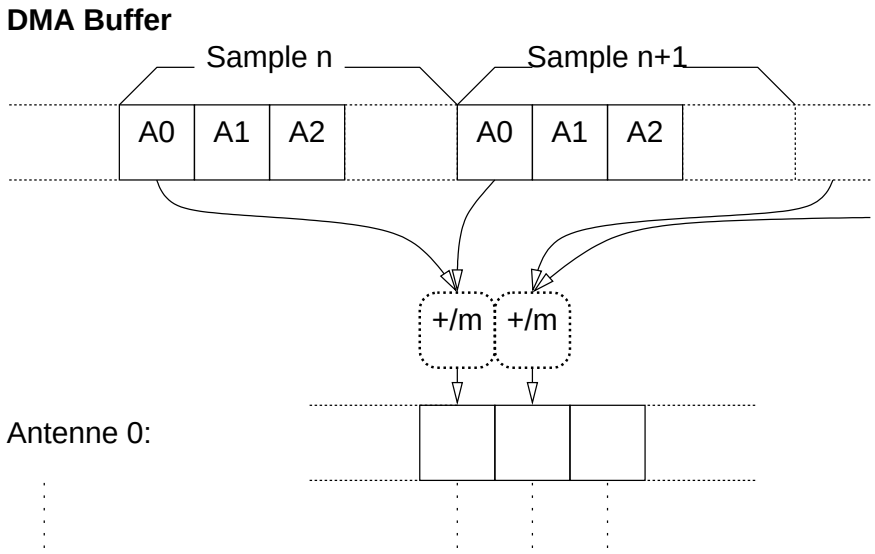

Abbildung 4.6: *Gemittelt werden immer direkt auseinander folgende Messwerte einer Antenne.*

unterschiedliche Verstärkungen der Kanäle auszugleichen.

Hierbei wird keine FFT gerechnet. Anstatt dessen werden die Daten nur so umsortiert, dass für jede Antenne der Chirp in einem Stück vorliegt (s. Abb. 4.5). Optional können auch Mittelwerte aufeinander folgender Messwerte gebildet werden (Abb. 4.6), um Rauschen zu reduzieren und um das später zu übertragende Datenvolumen zu reduzieren. Wie in den Abbildungen zu sehen entspricht jeder Messwert einem Takt, wenn keine Mittelung erfolgt; anderenfalls werden mehrere Takte zu einem Wert zusammengefasst. Dabei sind nur solche Mittelungsfaktoren sinnvoll, die auch echte Teiler der Länge eines Chirps (also von 1536) sind, da sonst in jedem Chirp die letzte Mittelung aus weniger Werten berechnet werden müsste.

5 Zusammenfassung

Das 1996 von der Arbeitsgruppe Fernerkundung begonnene entwickelte Radarsystem WERA wurde 2001 erfolgreich einer Verjüngungskur unterzogen, die primär die Hardware des Messrechners betraf, sekundär aber auch Anpassungen an der zugehörigen Software erforderte. Inzwischen wurden mehrere dieser Systeme zusammengebaut und an die Universität Hawaii ausgeliefert.

5.1 Hardware

Vor der Auslieferung der gebauten Systeme wurden diese getestet. Neben einem Dauertest des Systems war die Qualitätskontrolle der ADCs ein wesentlicher Punkt. Sie wurde durch Einspeisen verschiedener Eingangssignale auch über längere Zeiträume bewerkstelligt. Dabei stellte sich heraus, dass das System nach unterschiedlich langen Betriebszeiten in einen Zustand geriet, in dem keine Daten mehr von der DIO72 Karte geliefert wurden. Diese war dann in einen Verklemmungszustand geraten. Die Fehlersuche zog sich über mehere Wochen hin. Ohne Mithilfe des Kartenherstellers esd wäre sie nicht möglich gewesen, da für den Treiber der Karte kein Quelltext zur Verfügung stand. Wir erhielten verschiedene geänderte Versionen des Treibers, die aber zunächst alle keine Verbesserung brachten, sondern im Gegenteil zu einem sofortige Verklemmen des Systems führten. Letztendlich stellte sich heraus, dass von der DIO72 Karte unterschiedliche Versionen existierten. In unserer war ausgerechnet die Ressource defekt, die die neue Treiberversion zur Umgehung des ursprünglichen Fehlers (der in der PCI-Bridge der Karte liegt) benutzt.

5.2 Software

Die Testphase diente natürlich auch der Erprobung der Software. In ihrem Verlauf wurden noch verschiedene Details verbessert. Zum Beispiel erwies es sich als sinnvoll, dass im Fehlerfall keine Ergebnisdatei erzeugt wird, da diese ohnehin keine brauchbaren Daten enthalten würde. So ist zumindest sichergestellt, dass die Folgeverarbeitung den Fehlerzustand erkennen kann.

5.3 Ausblick

Auf der Software-Seite werden zunächst noch Anpassungen für die alten Messrechner auf VME-Basis notwendig sein, um die Kommunikationsschnittstelle anzugleichen. Die neue Mess-Software besitzt ja einen geringfügig erweiterten Funktionsumfang, der auch für die alten Systeme wünschenswert ist.

Aus Kostengründen (Lizenzkosten für VxWorks sind nicht unerheblich) wird mittelfristig die Umstellung auf RealTime Linux noch einmal untersucht werden und gegebenenfalls durchgeführt. Damit verbunden wäre ein erneutes Überarbeiten der Software. Voraussetzung sind DIO72 Treiber für das Betriebssystem.

Die Hardware des neuen Systems wird um eine zweite DIO72 Karte erweitert, die dann direkt zur Steuerung des digitalen Frequenzgenerators genutzt wird und den bisher notwendigen PC104 ersetzt. Dadurch wird insgesamt eine Vereinfachung des Aufbaus erreicht. Allerdings macht auch diese Modifikation wieder Anpassungen an der Software notwendig.

Anhang

A Software

Das alte System war in C programmiert. In Anbetracht der für das Projekt verfügbaren Zeit musste es bei C bleiben, ich hatte eine Umstellung auf eine objektorientierte Sprache (also C++, da es von der VxWorks Entwicklungsumgebung unterstützt wird) erwogen. Das hat leider zur Folge, dass der Quelltext relativ viele globale Variablen und Konstanten enthält. Die wichtigsten sind die folgenden:

```
#define STM_BOOTIMAGE "./vxworks/STM_BootImage.rbf"
#define STM_BOOTIMGSIZE 55100 // byte
```

Pfad und Größe des Bootimages für den ALTERA Chip auf der State Machine.

```
#define ADC_CLOCKDIV_LO 0x00
#define ADC_CLOCKDIV_HI 0x1E
#define ADC_STM_CLOCK 45371076.9
```

Teilerfaktor für die State Machine zur Erzeugung des ADC-Clock Signals aus dem externen 45 MHz Clock. `ADC_STM_CLOCK` ist der genaue Wert des Clock in Hz.

```
typedef struct {
    short int
        i,
        q;
} raw;
typedef struct {
    float
        im,
        re;
} complx;
```

Die Datenstruktur `raw` definiert das Eingabeformat der digitalisierten Daten, eingelesen per DMA. Die so vorliegenden Daten werden vor der FFT in das komplexwertige Format `complx` umgewandelt.

```
int antennae = 16;
```

Anzahl der im System vorhandenen Antennen; dieser Wert wird von `countADCs()` gesetzt. Das ADC Addr Register der State Machine muss den Wert `antennae/4` (oder weniger[1]) enthalten.

```
int sweepCount = SWEEPCOUNT;
```

Anzahl der Sweeps, die pro Messzyklus erzeugt werden sollen. Tatsächlich wird ein Sweep mehr generiert und der erste verworfen, damit sich das System einschwingen kann.

```
int sweepReduction = 2;
```

Nur für die Kalibration, gibt an, wie viele aufeinander folgende Werte gemittelt werden sollen. Sinnvoll sind nur echte Teiler von 1536 (Anzahl der Samples pro Sweep). Bei `(sweepReduction <= 1)` findet keine Mittelung statt.

```
SEM_ID semMeasureDone;
```

Dieser Semaphor wird gesetzt, sobald die Messung vollständig abgeschlossen ist.

```
SEM_ID semDMAReadAllow__;
SEM_ID semDMAReadFinished__;
SEM_ID semRawDMADataAvail__;
SEM_ID semRawDMADataRescued__;  //
raw *rawDMAData;
```

Semaphoren zur Steuerung und Erkennung des Status von `taskDMARead`. Sobald `semDMAReadAllow__` gesetzt ist, läuft die Task. Für jeden gefüllten DMA Buffer wird `semRawDMADataAvail__` gesetzt. Ist `semRawDMADataRescued__` bei Abschluss eines DMA Buffer nicht verfügbar, so liegt ein Überlauf vor, da dann `taskRescueRawData` den vorherigen Buffer nicht rechtzeitig retten konnte. Sobald `semRawDMADataAvail__` von `taskDMARead` gegeben wird, kopiert `taskRescueRawData` den mit `*rawDMAData` referenzierten Bufferinhalt in einen Ring Buffer für die Folgeschritte und gibt `semRawDMADataRescued__`, wenn der Buffer vollständig kopiert wurde.

```
SEM_ID semDMAWatchdogStart__;
SEM_ID semDMAWatchdogReset__;
SEM_ID semDMAWatchdogStop__;
#define WATCHDOG_TIMEOUT 5.0
int DMAAbort;
```

[1] Es wurde aber beschlossen, dass immer alle verfügbaren Antennen eingelesen werden und vom Benutzer keine Eingriffe an dieser Stelle erlaubt sind.

Semaphoren zur Steuerung von `taskWatchdog`, die das Blockieren von `taskDMARead` im Falle eines externen Fehlers erkennt.

Sobald `semDMAWatchdogStart__` gesetzt wurde, muss mindestens alle `WATCHDOG_TIMEOUT` Sekunden `semDMAWatchdogReset__` von `taskDMARead` gegeben werden. Andernfalls wird von hier aus `taskDMARead` beendet. Durch geben des `semDMAWatchdogStop__` Semaphors kann der Watchdog wieder beendet werden. Tritt tatsächlich einmal ein Timeout auf, dann setzt der Watchdog `DMAAbort`, um der Datenverarbeitungskette den Fehlerzustand anzuzeigen. Dort kann dann geeignet verfahren werden.

```
#define DATABUFFCOUNT 32
raw *rawDataBuffer[DATABUFFCOUNT];

SEM_ID semRawDataBufferAvail__;
SEM_ID semEvaluationFinished__;
```

Der Semaphor `semRawDataBufferAvail__` ist im Gegensatz zu den meisten anderen hier verwendeten Semaphoren zählend und nicht binär. So können auch mehrere Buffer in `rawDataBuffer[]` belegt werden, ohne dass die Evaluation Task diese sofort verarbeiten muss. Sie wird von `taskRescueRawData` für jeden neuen Eintrag in `rawDataBuffer[]` gegeben. Die jeweilige Evaluation Task gibt `semEvaluationFinished__`, sobald sie ihre Auswertung vollständig abgeschlossen hat und signalisiert der Hauptmessroutine damit das Ende der Messung.

```
char headerstring[513];
struct Header header;
char datafilename[80];
```

Konfigurationsdaten, die der Steuerrechner übermittelt hat und die an den Anfang jeder Ergebnis-Datei geschrieben werden.
`header` enthält die gleichen Informationen wie `headerstring`, allerdings als Struktur anstatt als Text.
`datafilename` ist der Name der nächsten zu erzeugenden Ergebnis-Datei.

```
raw *calResults;
```

Buffer für die Ergebnisse einer Kalibrationsmessung. Die Größe ist umgekehrt proportional zu `sweepReduction`.

```
fftw_complex corAmp[MAX_ANTENNAS],
             corAng[MAX_ANTENNAS];
fftw_complex *fftWindow;
```

Korrekturwerte, die vor der FFT auf die Messwerte angeendet werden. `*fftWindow` enthält die Kalibrationswerte, die mit den per DMA eingelesenen Rohdaten verknüpft werden müssen, bevor die FFT gerechnet wird.

```
#define FFTW_PLAN_METHOD  FFTW_ESTIMATE
fftw_plan fftwPlan;

#define RESULTWINDOW 128
#define RESULTOFFSET       1
int resultWindow = RESULTWINDOW;
int resultOffset = RESULTOFFSET;
```

Mit `FFTW_PLAN_METHOD` wird die Methode zum Erzeugen von `fftwPlan` festgelegt. `FFTW_ESTIMATE` ist schnell beim Erzeugen des Planes, aber `FFTW_MEASURE` erzeugt die Pläne, mit denen die FFTs schneller berechnet werden. `resultWindow` ist die Anzahl Frequenzen, die vom Ergebnis der FFT als tatsächliches Ergebnis übernommen werden. Entsprechend ist `resultOffset` der Offset im Ergebnisspektrum der FFT, ab dem `resultWindow` Frequenzen für die Weiterverarbeitung übernommen werden. Offset vom rechten (negativen) Teil des Spektrums, das aus der FFT heraus kommt. 0 bedeutet tiefste negative Frequenzen.

```
unsigned    sweepNumber;
```

Index (in `rawDataBuffer[]`) des nächsten Sweep, der von der FFT verarbeitet werden soll.

```
complx *fftResults;
// sweepCount * resultWindow * ANTENNAE * sizeof(complx);
complx *fftSortResults;
// dito.
```

`*fftResults` nimmt den relevanten Teil der von der FFT erzeugten Daten auf (s. `resultWindow` und `resultOffset`). Diese werden nach Ende der Messung derart in `fftSortResults` einsortiert, dass die Entfernungsbereiche zu Blöcken zusammengefasst werden.

A.1 Schnittstelle zum PC104

Die Kommunikation mit dem PC104 und damit die Steuerung des Frequenzgenerators erfolgt über diese Menge von Functions.

```
int auxport;            // FileID
int auxAutoFlag; // 0 = manual / 1 = auto
int taskAuxRead ();
long tidAuxPortRead;
```

In `auxAutoFlag` wird vermerkt, ob die Schnittstelle initialisiert ist oder nicht. Der Name ist historischen Ursprungs.

```
int auxInit ();
int auxFree ();
```

Initialisierung und Freigabe der seriellen Schnittstelle. Dabei wird auch eine Task erzeugt, um evtl. dort eintreffende Daten zu lesen und so einen Überlauf zu verhindern.

```
int auxTest ();
int auxSweep ();
int auxStop ();
```

Mit `auxTest()` kann eine Testsequenz übertragen werden. `auxSweep()` veranlasst den PC104 zum Einschalten des Trigger-Signals und `auxStop()` bewirkt das Gegenteil.

A.2 Bus & STM

```
typedef struct
{
    int  low;  // index of lowest bit
    int  num;  // number of bits from lowest
    long mask; // masking that in a long.
} BitInfo;
```

Die `struct BitInfo` dient zur Zuweisung der verschiedenen Leitungen des privaten parallelen Busses zu den Ein-/Ausgängen der DIO72 Karte. Für einige Funktionsaufrufe werden jedoch Bitmasken benötigt. Daher gibt es zu jeder `BitInfo` mit dem Namen `XXX` auch ein `#define XXX_`, das die Bits noch einmal als Bitmaske beschreibt. Diese wird jeweils dem `mask` Feld der Strukur zugewiesen. Deren Deklaration soll an dieser Stelle jedoch entfallen.
Die nachfolgend deklarierten Bits werden im normalen Betrieb des Systems benutzt. Sie liegen auf den untersten 32 Bit der DIO72 Karte.

```
static BitInfo All =
               {low:  0, num: 32, mask: All_};
static BitInfo ADC_Addr =
               {low:  0, num:  2, mask: ADC_Addr_};
static BitInfo ADC_BaseAddr =
               {low:  2, num:  4, mask: ADC_BaseAddr_};
static BitInfo ADC_Ident =
               {low:  6, num:  1, mask: ADC_Ident_};
```

```
static BitInfo ADC_NotError =
               {low:  7, num:  1, mask: ADC_NotError_};
static BitInfo ADC_Empty =
               {low:  8, num:  1, mask: ADC_Empty_};
static BitInfo ADC_Read =
               {low: 12, num:  1, mask: ADC_Read_};
static BitInfo ADC_Enable =
               {low: 13, num:  1, mask: ADC_Enable_};
static BitInfo ADC_Clock =
               {low: 14, num:  1, mask: ADC_Clock_};
static BitInfo IO_Ready =
               {low: 15, num:  1, mask: IO_Ready_};
static BitInfo STM_Addr =
               {low: 16, num:  3, mask: STM_Addr_};
static BitInfo STM_Data =
               {low: 19, num:  8, mask: STM_Data_};
static BitInfo STM_Read =
               {low: 27, num:  1, mask: STM_Read_};
static BitInfo STM_Write =
               {low: 28, num:  1, mask: STM_Write_};
static BitInfo STM_Select =
               {low: 29, num:  1, mask: STM_Select_};
static BitInfo Reset =
               {low: 31, num:  1, mask: Reset_};
```

Der auf der State Machine verwendete ALTERA muss nach der Hochfahren des Systems programmiert werden. Für diese Aufgabe werden einige spezielle Leitungen auf dem privaten Bus verwendet, die nachfolgend deklariert sind. In diesem Fall ist die `low` Information *nicht* absolut, da die Leitungen auf den Pins 64 bis 71 der DIO72 liegen.

```
static BitInfo bootSTM_ConfDone =
               {low: 5, num:  1, mask: bootSTM_ConfDone_};
static BitInfo bootSTM_Data =
               {low: 4, num:  1, mask: bootSTM_Data_};
static BitInfo bootSTM_NStatus =
               {low: 3, num:  1, mask: bootSTM_NStatus_};
static BitInfo bootSTM_DCLK =
               {low: 2, num:  1, mask: bootSTM_DCLK_};
static BitInfo bootSTM_nConfig  =
               {low: 1, num:  1, mask: bootSTM_nConfig_};
static BitInfo bootSTM_NCE =
               {low: 0, num:  1, mask: bootSTM_NCE_};
```

```
#define RX 0
#define TX 1
```

Die Pins der DIO72 können einzeln als Ein- oder Ausgang benutzt werden. Zur Konfiguration der Leitungen wird dem DIO72 Treiber eine Bitmaske übergeben, in der alle auf `1` gesetzten Bits Ausgang und alle auf `0` gesetzten Bits Eingang (= TriState) sind.

```
HANDLE hnd;
```

Globales Handle für Zugriffe auf die DIO72.

```
long busModeBitsLow  = 0x00000000;
long busModeBitsHigh =
     STM_Addr_ | STM_Data_ | STM_Read_ | STM_Write_ | Reset_;
```

Die unteren 32 Bit sind Eingänge für den DMA Transfer. Bits 32 bis 63 sind Steuerleitungen des privaten Busses.

```
long busModeBitsCfg =
     bootSTM_nConfig_ | bootSTM_Data_ | bootSTM_DCLK_;
```

Dies sind die obersten 8 Bit (64 bis 71), die der Konfiguration des ALTERA dienen.
Alle drei Variablen werden von `busCfg` und `busCfgInfo` verwendet, um den bisherigen Status der Pins zu speichern. So können einzelne Leitungen umkonfiguriert werden, ohne dabei die Konfiguration der übrigen Leitungen zu zerstören.

```
unsigned long busRead ();
unsigned long busMaskRead (BitInfo info);
unsigned long busWrite (unsigned long data);
unsigned long busMaskWrite (unsigned long data, BitInfo info);
unsigned long busReadCfg ();
unsigned long busMaskReadCfg (BitInfo info);
unsigned long busWriteCfg (unsigned long data);
unsigned long busMaskWriteCfg (unsigned long data, BitInfo info);
```

Verschiedene Funktionen zum Lesen und Schreiben auf dem Bus und zur Konfiguration. Beim Schreiben werden nur diejenigen Bits tatsächlich gesetzt, die auch als Ausgang konfiguriert sind. Alle anderen bleiben unverändert. Beim unmaskierten Einlesen werden die Werte der Ausgänge mit eingelesen. Das maskierte Lesen und Schreiben verschiebt die Daten entsprechend Info.

```
void busCfgInfo (int output, BitInfo bits);
void busCfg (int output, long mask);
void busConfig (unsigned long highbits, unsigned long lowbits);
```

Verschiedene Funktionen zur Konfiguration des Busses. `int output` sollte `TX` oder `RX` sein.

```
void busInit ();
void busFree ();
```

Öffnen und Schließen des DIO72 Treibers für die Karte. das `busInit()` initialisiert alle Leitungen des Busses mit den Standardwerten. Die STM wird jedoch nicht automatisch gebootet. Mehrfache Aufrufe der Funktionen sind nicht schädlich, da der aktuelle Status überprüft wird.

```
void busReset ();
int  getIOReady ();
void setIOReady (int data);
```

Grundfunktionen zum Steuern des Busses. `busReset` führt einen Reset auf dem Bus durch, für den Reset auf `0` und nach kurzer Zeit wieder auf `1` gesetzt werden muss.

```
int  countADCs ();
int  STMBoot ();
int  STMGetReg (int addr);
void STMInfo ();
void STMSetReg (int addr, int data);
void STMInit ();
void STMGo ();
void STMStop ();
unsigned int getErrorBytes ();
unsigned getSTMFirmwareRevision ();
```

Mit `countADCs()` kann die Anzahl der vorhandenen ADC Boards ermittelt werden. Das Ergebnis wird für die Konfiguration der STM benötigt. Die übrigen Funktionen dienen zur Konfiguration und Kommunikation mit der STM. Dabei führt `STMInit()` schon eine vollständige Initialisierung einschließlich aller Registerwerte durch. Nach diesem Aufruf braucht die State Machine nur noch mit `STMGo()` gestartet zu werden.

A.3 Netzwerk-Kommunikation

```
int    haveBasics = 0;
long   tidWeraServer;
SEM_ID semWeraAvail;
```

```
void initBasics ();
// init all basic global structs, e.g. semWeraAvail
void freeBasics ();
// free anything allocated by initBasics

int weraAvail (int *haveIt);
void weraReturn();
// gives semWeraAvail back.

void runWera ();
void stopWera ();
STATUS taskWeraSocketServer();
```

`taskWeraSocketServer()` wird beim Systemstart gespawnt. Hiermit wartet das System auf Socketanfragen aus dem Netz und startet für jede eine eigene Kommunikationstask `serverWorkTask`. Mit `runWera()` und `stopWera()` kann `taskWeraSocketServer()` erzeugt bzw. wieder beendet werden (was im normalen Betrieb aber nicht nötig ist). Zu jedem Zeitpunkt gibt es maximal eine Task von diesem Typ, deren ID in `tidWeraServer` gespeichert ist.

Mehrere Instanzen gleichzeitig kann es prinzipiell von `serverWorkTask` geben. Da aber nur eine Messung zur Zeit stattfinden kann, wird über den Semaphor `semWeraAvail` mit Hilfe der Funktionen `weraAvail` und `weraReturn` gesteuert, welche serverWorkTask Zugriff darauf hat.

Ein Aufruf von `weraAvail` ermittelt, ob die aufrufende Funktion einen Messvorgang durchführen lassen darf. Dies wird von der `serverWorkTask` immer dann gemacht, wenn für die Messung relevante Ressourcen des Systems in Anspruch genommen werden. Im Parameter `*haveIt` übergibt die `serverWorkTask` eine lokale Variable; so kann erkannt werden, ob schon ein Aufruf von `weraAvail` erfolgreich (`haveIt == 1`) oder erfolglos (`haveIt == -1`) war und der Zustand kann beibehalten werden. Erst nach Aufruf von `weraReturn()` kann wieder eine andere `serverWorkTask` Messungen veranlassen.

A.4 Mess-Software Routinen

```
int  initFFTWPlan ()
int  initFFTWPlanWisdom ()
void freeFFTWPlan ()
```

Zur Initialisierung des FFT Planes stehen zwei Funktionen zur Verfügung, entweder nach der schnellen Methode oder für schnelle (besonders auf Rechenzeit optimierte) FFTs. Bei Änderung der Antennenanzahl muss ein neuer Plan erzeugt werden. Da diese Änderung aber nicht im Betrieb vorgesehen ist, reicht es aus, den Plan einmal beim Bootvorgang zu erzeugen. Die erzeugte Struktur ist

relativ komplex mit dynamischen Unterstukturen. Sie kann mit `freeFFTWPlan` wieder freigegeben werden.

```
int loadCalibration ()
```

Zum Lesen der Kalibrationsdaten kann diese Funktion aufgerufen werden.
Die Daten werden in die beiden globalen Feldern `corAmp[MAX_ANTENNAS]` und `corAng[MAX_ANTENNAS]` gelesen.

```
int  initFFTWindow ()
void freeFFTWindow ()
```

Auch das globale `*fftWindow` muss initialisiert und eventuell wieder freigegeben werden. Es wird im Messbetrieb benötigt, um Anfang und Ende der Sweeps auszublenden und so die Übergänge zwischen den Sweeps anzugleichen; sonst enthalten die berechneten Spektren unter Umständen ungünstig hohe Energieanteile in hohen Frequenzbereichen.

```
int initDIO72SGDMA (void *bufferlist[DMA_BUFFCOUNT],
                    int bufsize, int bufcount)
```

Der sogenannte Scatter Gather DMA Betrieb der DIO72 Karte benötigt eine Pufferliste und die Größe und Anzahl der in der Liste enthaltenen Puffer. Die Konfiguration der Trigger-Quelle etc. geschieht in `initDIO72SGDMA()`. Dann kann die Treiberfunktion `io72MultiDmaStart()` aufgerufen werden, um danach durch Aufruf von `io72MultiDmaRead (hnd, &actBuff)` auf den ersten Interrupt des DMA Transfers zu warten. Der erfolgt, sobald ein Puffer vollständig gefüllt ist. Es muss also für jeden zu füllenden Puffer einmal `io72MultiDmaRead` aufgerufen und der Inhalt des dann gefüllten Puffers verarbeitet werden. Einen Zeiger auf den Puffer liefert die Funktion in `actBuff`.

```
int taskDMAWatchdog ()
```

Task zur Überwachung des DMA-Transfers bzw. von `taskDMAReadWait`. Der von dieser Task bei jedem vollständigen DMA-Puffer gesetzte Semaphor `semDMAWatchdogReset__` wird von `taskDMAWatchdog` mit einem Timeout von derzeit 5 Sekunden erwartet. Tritt beim Nehmen des Semaphors ein Timeout auf, so wird die DMARead-Task terminiert und der nachfolgenden Verarbeitung über die entsprechenden Semaphoren das Ende der Messung signalisiert. Zusätzlich wird der Fehlerzustand in der globalen Variablen `DMAAbort` mit dem Wert `1` angezeigt. Die Semaphoren `semDMAWatchdogStart__` und `semDMAWatchdogStop__` starten und beenden den Watchdog-Betrieb.

```
int taskDMAReadWait (int actSweep)
```

Innerhalb dieser Task läuft eine Schleife, die die gefüllten Puffer des DMA erwartet und weiterleitet.

Die Schleife beginnt, sobald der Semaphor `semDMAReadAllow__` gegeben wurde.

Das Warten auf den nächsten vollen Puffer geschieht mit der DIO72 Treiberfunktion `io72MultiDmaRead`, deren Aufruf erst mit dem Puffer-Wechsel des DMA-Transfers zurückkehrt. Darauf wird die Adresse des Puffers in `rawDMAData` geschrieben und der Semaphor `semRawDMADataAvail__` gegeben, um den zu sichernden Pufferinhalt zu signalisieren. Über `semRawDMADataRescued__` wird sichergestellt, dass der vorhergehende Puffer erfolgreich gesichert wurde. Andernfalls ist ein Fehler aufgetreten. Nachdem `semDMAWatchdogReset__` gegeben wurde, beginnt die Schleife von vorne.

Abbruchkriterium ist das Erreichen der gewünschten Pufferanzahl, die gleich der Anzahl gemessener Sweeps ist. Den weiteren Verarbeitungsschritten wird das Ende der Messung durch `rawDMAData = NULL` bedeutet.

```
int taskRescueRawData ()
```

Die Task wartet in einer Endlosschleide unbegrenzt auf den Semaphor `semRawDMADataAvail__`.
Sobald er verfügbar ist, wird der Inhalt des mit `rawDMAData` referenzierten Puffers in den Ringpuffer `rawDataBuffer[dataBufferIndex]` kopiert, danach wird `dataBufferIndex` inkrementiert.

Durch das geben von `semRawDMADataRescued__` wird an die DMARead-Task signalisiert, dass der DMA-Puffer wieder verfügbar ist. Der nachfolgenden Verarbeitungstask wird das Vorhandensein neuer Daten über `semRawDataBufferAvail__` mitgeteilt. Auch hier zeigt der Pufferinhalt `NULL` das Ende der Messung an und ist Abbruchkriterium der Schleife.

`dataBufferIndex` ist eine lokale Variable der Task.

```
int saveFFTResults ()
int sortFFTResults ()
int calculateFFT (raw *rawBuffer, complx *fftResult)
int taskEvaluateDataBufferFFT ()
```

Im Falle einer normalen Messung wird `taskEvaluateDataBufferFFT` als Auswertungstask gespawnt. Die Verarbeitungsschleife wartet auf `semRawDataBufferAvail__` und ruft `calculateFFT` mit dem aktuellen Puffer `rawDataBuffer[dataBufferIndex]` auf. `dataBufferIndex` ist auch hier eine lokale Variable der Task. Anschließend wird `rawDataBuffer[dataBufferIndex]` auf `NULL` gesetzt und somit wieder für neue Rohdaten verfügbar. Sobald das Ende der Messung erreicht ist, werden die Ergebnisdaten durch `sortFFTResults` umsortiert und mit Hilfe von `saveFFTResults` zum Steuerrechner übertragen.

```
int copyCalibrationData (raw *rawBuffer, raw *calResult)
int saveCalibrationResults ()
int taskEvaluateDataBufferCalibration ()
```

Bis auf die unterschiedliche Berechnung der Ergebnisse ist der Ablauf identisch mit dem FFT-Fall. Die Schleife ruft zur Ergebnisberechnung `copyCalibrationData` auf und am Ende der Messung werden die Daten von `saveCalibrationResults` zum Steuerrechner übertragen.

```
int createDummyData ()
int freeDummyData ()
```

Falls der Compilerschalter `USE_DUMMY_DATA` definiert ist, werden anstatt realer Messwerte Pseudodaten an die Verarbeitungskette übergeben. Um das Zeitverhalten der Verarbeitungskette nicht zu stark durch permanentes Berechnen von Pseudodaten zu verändern, werden diese vor der Messung erzeugt.

```
int doDMARead (HANDLE hnd, void *bufferlist[DMA_BUFFCOUNT],
               int calibrate, int sweeps)
```

Innerhalb dieser Funktion werden alle Tasks der Verarbeitungskette für die Messung erzeugt. Mit `calibrate` wird gesteuert, ob `taskEvaluateDataBufferFFT` gespawnt wird oder `taskEvaluateDataBufferCalibration`. Dann wird der Scatter Gather DMA Betrieb initialisiert und die Verarbeitungskette durch geben von `semDMAReadAllow__` aktiviert. Nachdem der externe Trigger aktiviert wurde, muss auf das Setzen von `semDMAReadFinished__` gewartet werden. Anschließend wird die Datenerfassungshardware wieder deaktiviert und nachdem auch die Auswertung abgeschlossen ist, terminiert die Funktion.

```
int DMARead (HANDLE hnd, int calibrate, int sweeps)
```

In dieser Funktion werden Speicherverwaltungsaufgaben der Messung zusammengefasst. Nachdem alle Puffer und Semaphoren initialisiert sind, wird `doDMARead` aufgerufen und anschließend werden die Puffer und Semaphoren bereits wieder freigegeben.

```
int measure (int calibrate)
```

Diese Funktion wird aufgerufen, um eine Messung ausführen zu lassen. Über den Parameter kann gesteuert werden, ob eine Kalibrationsmessung oder eine auswertende Messung (mit FFT) erfolgen soll. Nach der Initialisierung einiger globaler Variablen wird `DMARead` aufgerufen.

```
int serverWorkTask (sFd, pClntSockAddr)
```

Für jede Netzwerkverbindung wird eine Task gespawnt. Darin erfolgt dann die tatsächliche Kommunikation, deren Ziel im Normalfall die Durchführung einer Messung ist. Daher wird von hier aus auch die Funktion `measure` aufgerufen.

`STATUS taskWeraSocketServer ()`

Permanente Task, die an einem Socket auf ankommende Verbindungen wartet. Für jede Verbindung wird eine `serverWorkTask` gespawnt. Es gibt immer höchstens eine solche Task.

Literaturverzeichnis

[IfM] Universität Hamburg, Institut für Meereskunde. *http://www.ifm.uni-hamburg.de*

[WERA] Universität Hamburg, Institut für Meereskunde, Arbeitsgruppe Fernerkundung.*http://wera.ifm.uni-hamburg.de*

[GES01] K.-W. Gurgel, H.-H. Essen, T. Schlick. *Applications of WERA within EuroROSE.* First international Radiowave Oceanography Workshop, 2001.

[GES01a] K.-W. Gurgel, H.-H. Essen, T. Schlick. *The University of Hamburg WERA HF Radar - Theory and Solutions.* First international Radiowave Oceanography Workshop, 2001.

[GEK99] K.-W. Gurgel, H.-H. Essen, S.P. Kingsley. *HF Radars: Physical limitations and recent developments.* Coastal Engineering, Vol. 37, Nos. 3-4, 201–218, 1999.

[Gur99] K.-W. Gurgel. *Applications of Coastal Radars for Monitoring the Coastal Zone.* Proceedings EUROMAR Workshop '99, 21–30, 1999.

[Gur01] K.-W. Gurgel et al. *Wellen Radar (WERA), a new ground-wave based HF Radar for ocean remote sensing* Coastal Engineering, Vol. 37, Nos. 3–4, 219–234, 1999.

[Win99] WindRiver. *VxWorks Programmer's Guide 5.4*, Edition 1, 1999.

[GA97] K.-W. Gurgel, G. Antonischki. *Remote Sensing of Surface Currents and Waves by the HF Radar WERA.* Seventh IEE Conference on Electronic Engineering in Oceanography, Proceedings, ISBN 0 85296 689 X, 211–217, 1997

[DVX2601] Analogic. *Analogic DVX 2601 Manual*, http://www.analogic.com

[LTC1604] Linear Technology. *Linear Technology LTC 1604*, http://www.linear-tech.com

[Linux] Realtime Linux. *Realtime Linux Community*, http://www.rtlinux.org

[FFTW] FFTW Bibliothek. http://www.fftw.org

Danksagung

Ich möchte mich für die Unterstützung und die Motivation bedanken, die mir verschiedentlich gegeben wurde von den Betreuern dieser Arbeit Klaus-Werner Gurgel und Norman Hendrich und der Arbeitsgruppe Fernerkundung...

...und von meinem kleinen Bruder.