

# Entwurf und Implementierung eines TTCAN-Treibers für OSEKtime/Tricore TC1796

Studienarbeit im Fach Informatik

vorgelegt von  
Stefan Rehm  
26.09.1981, Erlangen

Lehrstuhl für Verteilte Systeme und Betriebssysteme (Informatik 4)  
Institut für Informatik  
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Prof. Dr. Wolfgang Schröder-Preikschat  
Beginn der Arbeit: 01.01.2007  
Abgabe der Arbeit: 01.10.2007

## Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Richtlinien des Lehrstuhls für Examensarbeiten habe ich gelesen und anerkannt.

Erlangen, den .....

Unterschrift \_\_\_\_\_

## **Kurzzusammenfassung/Abstract**

Diese Studienarbeit beschreibt den Entwurf und die Implementierung eines Treibers für die TTCAN-Schnittstelle des Tricore TC1796. Dieser ist innerhalb des zeitgesteuerten Betriebssystems ProOSEK/Time verwendbar. Bei TTCAN handelt es sich um ein zeitgesteuertes Kommunikationsprotokoll. Der Austausch von Nachrichten erfolgt also strikt nach einem vorab statisch festgelegten Ablaufplan. Da das Erstellen eines solchen Ablaufplans im Allgemeinen ein nicht triviales Problem ist, wurde außerdem ein Konfigurationswerkzeug entwickelt, das diese Aufgabe übernimmt. Der Benutzer muss nur noch die Knoten im Netzwerk und deren Nachrichten in einer Konfigurationsdatei angeben.

This document describes the design and implementation of a driver for the TTCAN controller of the Tricore TC1796. The driver is designed to be used within the ProOSEK/Time operating system. TTCAN is a time triggered communication protocol. The transmission of messages over the communication medium is based on a precomputed static scheduling table. Because the generation of this scheduling table is in general a nontrivial issue, a tool has been developed, that automates this task. All the user has to do is supplying a configuration file containing the nodes and their corresponding messages.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>8</b>
1.1	Ziele der Arbeit . . . . .	8
1.2	Kapitelübersicht . . . . .	8
<b>2</b>	<b>Zeitgesteuerte Systeme und Kommunikation</b>	<b>10</b>
2.1	Kommunikation in zeitgesteuerten Systemen . . . . .	10
2.2	Ablaufplanung in zeitgesteuerten Systemen . . . . .	11
<b>3</b>	<b>Controller Area Network</b>	<b>13</b>
3.1	CAN-Bus . . . . .	13
3.1.1	Physikalische Randbedingungen . . . . .	13
3.1.2	Buszugriff . . . . .	13
3.2	CAN-Nachrichten . . . . .	14
3.2.1	Data Frames . . . . .	14
3.2.2	Remote Frames . . . . .	15
3.2.3	Error Frames . . . . .	16
3.3	Datensynchronisierung . . . . .	16
3.4	TC1796 CAN-Hardware . . . . .	18
3.4.1	Das CAN-Modul im Überblick . . . . .	18
3.4.2	Listenverwaltung . . . . .	18
3.4.3	Nachrichtenobjekt . . . . .	18
3.4.4	CAN-Kontrolllogik . . . . .	19
<b>4</b>	<b>TTCAN-Erweiterung</b>	<b>20</b>
4.1	TTCAN als zusätzliche CAN-Kommunikationsschicht . . . . .	20
4.2	Die globale Zeit . . . . .	21
4.2.1	Die Referenznachricht . . . . .	21
4.2.2	Globale Zeiteinheit . . . . .	21
4.2.3	Drift Korrektur . . . . .	22
4.3	Basiszyklus . . . . .	22
4.4	Matrixzyklus . . . . .	23
4.5	TC1796 TTCAN-Hardware . . . . .	23
4.5.1	Zeitkontrolle und Uhrensynchronisation . . . . .	24
4.5.2	Scheduler . . . . .	24
<b>5</b>	<b>ProOSEK/Time</b>	<b>26</b>
5.1	Ablauftabelle . . . . .	26
5.2	Anwendungsmodi . . . . .	26
5.3	Arbeitsaufträge . . . . .	27
5.4	Synchronisierung . . . . .	27
<b>6</b>	<b>Entwurf und Implementierung des Treibers</b>	<b>28</b>
6.1	Designüberblick . . . . .	28

6.2	CAN-Funktionalität . . . . .	29
6.2.1	Tricore CAN-Modul . . . . .	29
6.2.2	Nachrichten Verwaltung . . . . .	29
6.2.3	Nachrichtenobjekte . . . . .	29
6.2.4	CAN-Knoten . . . . .	30
6.2.5	Einbindung des CAN-Treibers in ProOSEK . . . . .	30
6.3	TTCAN-Funktionalität . . . . .	32
6.3.1	TTCAN-Erweiterungsmodul . . . . .	32
6.3.2	TTCAN-Scheduler Modul . . . . .	32
6.3.3	Einbindung des TTCAN-Treibers in ProOSEK/Time . . . . .	33
6.4	Evaluierung . . . . .	34
<b>7</b>	<b>Entwurf und Implementierung des Konfigurationswerkzeugs</b>	<b>36</b>
7.1	Designüberblick . . . . .	36
7.2	Programmablauf . . . . .	37
7.2.1	Einlesen der Nachrichten . . . . .	37
7.2.2	Ablaufplanung . . . . .	38
7.2.3	Generierung und Export des lokalen Matrixzyklus . . . . .	42
7.2.4	Ausgabe des Quelltextes . . . . .	42
7.3	Verwendung . . . . .	42
7.4	Evaluierung . . . . .	43
<b>8</b>	<b>Zusammenfassung</b>	<b>46</b>

# Abbildungsverzeichnis

3.1	CAN Data Frame . . . . .	15
3.2	CAN-Bitzeit . . . . .	17
3.3	MultiCAN Module . . . . .	19
4.1	TTCAN-Basiszyklus . . . . .	23
4.2	TTCAN-Matrixzyklus . . . . .	24
7.1	Nicht präemptiver EDF . . . . .	41
7.2	Präemptiver EDF . . . . .	42
7.3	Gesamtlaufzeiten des Konfigurationswerkzeugs . . . . .	45

# Tabellenverzeichnis

6.1	Laufzeiten der Funktion ttCanStart() . . . . .	34
6.2	Speicherverbrauch des Treibers . . . . .	35
7.1	Laufzeiten der Phasen des Konfigurationswerkzeugs . . . . .	44

# 1 Einleitung

Für die Realisierung fehlertoleranter Echtzeitsysteme kommen in der Praxis vornehmlich zeitgesteuerte Plattformen zum Einsatz [5]. In solchen System erfolgt die Kommunikation dementsprechend über zeitgesteuerte Kommunikationsmedien. Strikt einem vorab statisch bestimmten Ablaufplan folgend, werden bestimmte Nachrichten dem Kommunikationssystem zu bestimmten Zeiten übergeben bzw. vom Kommunikationsmedium entgegengenommen. Der Tricore TC1796 Microcontroller besitzt eine zeitgesteuerte Kommunikationsschnittstelle, die nach dem TTCAN-Protokoll arbeitet. TTCAN ist ein zeitgesteuertes Kommunikationsprotokoll, das auf dem vor allem in der Automobilbranche weit verbreiteten und ereignisgesteuerten CAN-Protokoll aufbaut.

## 1.1 Ziele der Arbeit

Die Ziele dieser Studienarbeit sind zum einen die Entwicklung eines Treibers für die TTCAN-Schnittstelle des Tricore TC1796 und zum anderen die Entwicklung eines Konfigurationswerkzeugs, das unter anderem die Erstellung des statischen Ablaufplans zum Nachrichtenaustausch in einem TTCAN-Netzwerk automatisiert. Der Treiber soll innerhalb des zeitgesteuerten Betriebssystems ProOSEK/Time verwendbar sein. Die Verwendung des Treibers soll sich jedoch für den Benutzer weitestgehend transparent gestalten. Der Abstraktionsgrad des Treibers muss also hoch genug sein, so dass kein Wissen über das darunterliegende Kommunikationsprotokoll bzw. die Hardware notwendig ist. Das Konfigurationswerkzeug besitzt zur Erstellung des Ablaufplans außerdem bereits alle Informationen, die notwendig sind, um zusätzlich die Generierung des Quelltextes zur Initialisierung und Konfiguration der TTCAN-Schnittstelle zu automatisieren. Weiterhin muss es dem Benutzer die Ergebnisse der Ablaufplanung zugänglich machen, damit dieser seine Echtzeitanwendung entsprechend der Sende- und Empfangszeiten von Nachrichten planen kann. Als Eingabe erwartet das Konfigurationswerkzeug lediglich die Informationen über die einzelnen Knoten im Netzwerk und die zu versendenden Nachrichten.

## 1.2 Kapitelübersicht

Dieses Dokument ist in sieben weitere Kapitel unterteilt. Das zweite Kapitel behandelt zunächst die Grundlagen der Kommunikation in zeitgesteuerten Systemen im Allgemeinen. Dazu zählen vor allem das Aufstellen und Aktualisieren der Echtzeitdaten, sowie die Ablaufplanung der Nachrichtenübermittlung. Danach folgt im dritten Kapitel eine Einführung in das CAN-Protokoll. Es beschreibt den CAN-Bus als Kommunikationsmedium, die Mechanismen der Datenübertragung, die übertragbaren Nachrichtenformate und die Hardware der CAN-Schnittstelle auf dem TC1796. Darauf aufbauend wird im

vierten Kapitel gezeigt, wie TTCAN das CAN-Protokoll um die Zeitsteuerung erweitert, indem es eine zusätzliche Kommunikationsschicht einführt. Hier werden die Mechanismen zur Definition und Synchronisierung der globalen Zeit und zur Abbildung des statischen Kommunikationsablaufplans erklärt. Das Kapitel endet mit einer Übersicht der TTCAN-Hardware Module des TC1796. Das fünfte Kapitel erklärt die wesentlichen Konzepte von ProOSEK/Time zur Ablaufplanung und Synchronisierung von verteilten Anwendungen, bevor sich das sechste Kapitel dem Entwurf und der Implementierung des TTCAN-Treibers widmet. Es beschreibt das grundlegende Design des Treibers sowie die implementierte Funktionalität sortiert nach der CAN- und TTCAN-Schicht. Analog dazu behandelt das siebte Kapitel den Entwurf und die Implementierung des Konfigurationswerkzeugs, indem es die einzelnen Verarbeitungsschritte des Programmablaufs und die Verwendung des Programms erläutert. Sowohl das sechste als auch das siebte Kapitel enden mit einer Evaluierung der jeweiligen Implementierung. Den Abschluß der Arbeit bildet schließlich eine Zusammenfassung der erreichten Ziele.

## 2 Zeitgesteuerte Systeme und Kommunikation

Dieses Kapitel behandelt zunächst die Kommunikation in zeitgesteuerten Systemen im Allgemeinen, bevor in späteren Kapiteln auf TTCAN im Besonderen eingegangen wird. Dazu wird im ersten Teil des Kapitels der Unterschied zwischen ereignis- und zustands-basierter Kommunikation und im zweiten die Ablaufplanung in zeitgesteuerten Systemen genauer betrachtet.

### 2.1 Kommunikation in zeitgesteuerten Systemen

Zeitgesteuerte Systeme finden häufig Verwendung in Sensor-Aktor Anwendungen, bei denen ein System aufgrund von Messdaten kontrolliert und gesteuert werden muss. Das dynamische Verhalten des zu kontrollierenden Systems lässt sich durch eine Menge von Zustandsvariablen beschreiben, die ihren Wert zur Laufzeit des Systems mit fortschreitender Zeit ändern. Einige relevante Zustandsvariablen für das System *Auto* wären zum Beispiel:

- Geschwindigkeit
- Benzinverbrauch
- Drehzahl des Motors
- Ölstand etc.

Die Anzahl der Zustandsvariablen wächst mit der Komplexität des betrachteten Systems. Um eine konkrete Aufgabe zu erfüllen, ist es für eine Anwendung jedoch meist nicht notwendig, den kompletten Zustand des Systems zu betrachten. Eine Anwendung, die in einem Auto den Luftdruck der Reifen überwacht, muss beispielsweise nicht den momentanen Benzinverbrauch kennen. Die Teilmenge der für eine Anwendung relevanten Zustandsvariablen bezeichnet man als Echtzeitinstanzen. Echtzeitinstanzen besitzen verschiedene dynamische und statische Attribute. Letztere sind unter anderem der Typ und der Wertebereich. Das wichtigste dynamische Attribut ist natürlich der zu einem bestimmten Zeitpunkt gesetzte Wert. Der Wert einer Instanz zu einem bestimmten Zeitpunkt kann durch Beobachtung festgestellt werden. Eine Beobachtung wird als Tupel definiert:

$$\text{Beobachtung} = \langle \text{Name}, \text{Wert}, \text{Zeitpunkt} \rangle$$

Sie besteht also aus dem Namen der Echtzeitinstanz, ihrem Wert und dem Zeitpunkt, zu dem dieser beobachtet wurde. Man unterscheidet zwei verschiedene Arten von Beobachtungen:

- Zustandsbasierte Beobachtung
- Ereignisbasierte Beobachtung

Ein Echtzeitabbild ist die Repräsentation der Beobachtung einer Echtzeitinstanz. Ein solches Abbild einer Instanz ist immer nur für einen bestimmten Zeitraum gültig. So kann die Gültigkeitsdauer zum Beispiel von der Abtastrate eines Sensor begrenzt werden. Die Menge der gültigen Echtzeitabbilder bildet eine Echtzeitdatenbasis. Die Aktualisierung der Datenbasis ist unmittelbar mit der Art der Beobachtung der Echtzeitabbilder verbunden. Sie kann entweder zeitgesteuert in periodischen Abständen erfolgen oder aber ereignisbasiert zu spontanen Zeitpunkten. Während bei der zeitgesteuerten Aktualisierung die Aktualisierungszeitpunkte fest vorgegeben sind, löst bei ereignisbasierter Aktualisierung jede Wertänderung einer Echtzeitinstanz eine Aktualisierungsoperation aus. Bei zustandsbasierter Beobachtung kann die Aktualisierung zeitgesteuert erfolgen. Die Aktualisierungsoperation ist dabei idempotent. Ereignisbasierte Beobachtungen erfordern jedoch eine ereignisgesteuerte Aktualisierung. Die Anforderungen an das Netzwerkprotokoll unterscheiden sich ebenfalls für die Art der Aktualisierung. Während eine zeitgesteuerte Aktualisierung aufgrund der Idempotenz lediglich eine *at-least-once* Aufrufsemantik voraussetzt, benötigt die ereignisgesteuerte Aktualisierung eine *exactly-once* Semantik. Die Mehrfachausführung einer Aktualisierungsoperation oder das Verpassen einer Wertänderung würde sonst dazu führen, dass der Zustand des Systems nicht mehr aus den Beobachtungen rekonstruierbar wäre.

## 2.2 Ablaufplanung in zeitgesteuerten Systemen

In sicherheitskritischen Systemen (z.B. X-by-Wire in der Automobil- und Luftfahrtindustrie) ist es zunehmend notwendig, deterministische Aussagen über das Laufzeitverhalten der Aufträge im System treffen zu können. Es muss garantiert werden, dass zeitkritische Aufträge termingerecht verarbeitet und beendet werden können. Zwar ist es beispielsweise auch mit Hilfe von prioritätsbasierter Ablaufplanung möglich, Garantien bezüglich einzelner Arbeitsaufträge zu geben, die Planung der Prioritätenvergabe ist jedoch sehr zeitaufwendig und komplex. So kann die Ausführung eines Auftrags von einem höher priorisierten Auftrag unterbrochen werden und selbst ein Auftrag mit der höchsten Priorität kann einem Jitter unterliegen, wenn es noch mindestens einen weiteren Auftrag mit derselben Priorität gibt, der sich gerade in Ausführung befindet.

Der Ablauf einer zeitgesteuerten Anwendung wird deshalb durch das Fortschreiten der physikalischen Zeit kontrolliert. Die Einlastung von Arbeitsaufträgen geschieht nur zu auf der Zeitachse genau definierten Zeitpunkten. Um für alle Aufträge eines Systems einen geeigneten Ablaufplan zu erstellen, müssen alle Parameter aller Aufträge a priori bekannt sein. Dazu zählen vor allem:

**Die maximale Ausführungszeit** (*WCET*), d.h. die benötigte Prozessorzeit um einen Auftrag auszuführen.

**Der Betriebsmittelbedarf** um Wartezeiten zu vermeiden oder wenigstens mit einplanen zu können.

**Die Auslösezeitpunkte**, also die Zeitpunkte zu denen ein Auftrag in den Zustand *bereit* übergeht und eingelastet werden kann. Sie sind entweder nicht periodisch oder

periodisch auf der Zeitachse verteilt. Nicht periodische Aufträge werden meist von externen Ereignissen ausgelöst. Sie sind insofern problematisch, als dass man für ihre Ankunftszeiten nur eine statistische Verteilungsfunktion angeben und somit lediglich mit einer maximalen Ankunftsrate in einem bestimmten Zeitintervall planen kann.

**Der Termin** d.h. der Zeitpunkt, an dem der Auftrag beendet sein muss.

Dieses Wissen ist nötig, um deterministische Aussagen über die Einlastungs- und Laufzeiten eines jeden Auftrags machen zu können. Die minimale Anforderung an einen gültigen Ablaufplan ist, dass alle Aufträge ihre Termine einhalten können. Das Finden eines solchen Ablaufplanes ist ein NP-vollständiges Problem. Für jeden zeitlich festgelegten Auftrag müssen im schlimmsten Fall alle Permutationen der übrigen Aufträge betrachtet werden, um einen Ablaufplan zu finden, der die minimalen Bedingungen erfüllt. Außerdem gibt es im Allgemeinen zu einer Anwendung mehrere gültige Ablaufpläne, so dass man zusätzliche Kriterien einführen kann, um den jeweils optimalen Ablaufplan auszuwählen. Durch das vorhandene a priori Wissen ist es jedoch möglich, den Ablaufplan im Voraus statisch zu erstellen, so dass die Komplexität zur Laufzeit keine Rolle mehr spielt. Es lässt sich außerdem zeigen, dass sich die Ausführungsreihenfolge periodischer Aufträge mit unterschiedlichen Perioden nach einer bestimmten Zeitspanne wiederholt. Sie wird als Hyperperiode bezeichnet und ihre Länge entspricht genau dem kleinsten gemeinsamen Vielfachen der einzelnen Perioden. Der zeitliche Rahmen des Ablaufplans ist somit ebenfalls begrenzt. Die verwendeten Algorithmen arbeiten im Allgemeinen auf Graphen, deren Knoten die Aufträge repräsentieren. Ein Pfad durch den Graphen beschreibt also eine Permutation der Ausführungsreihenfolge der Aufträge. Das Finden eines gültigen Pfades ist damit ein klassisches Wegfindungsproblem, das sich z.B. mit dem A\* Algorithmus effizient lösen lässt.

In verteilten, zeitgesteuerten Systemen ist es weiterhin sinnvoll, die Planung der Ablaufsteuerung in zwei Phasen mit festgelegter Reihenfolge zu unterteilen: zuerst die Planung globaler Belange und danach die Planung lokaler Belange. Die Planung des Nachrichtenaustauschs ist ein globaler Belang und muss gleichzeitig für alle Knoten des Systems geschehen. Ein Auftrag ist hier das Versenden oder Empfangen einer Nachricht. Nur wenn alle Teilnehmer nach dem gleichen Ablaufplan und einer global gültigen Zeit handeln, kann der Zugriff auf das Kommunikationsmedium eindeutig geregelt werden. Außerdem muss jeder Knoten Betriebsmittel wie z.B. Speicherplatz für die zu empfangenden Nachrichten bereitstellen. Dies ist nur möglich, wenn das globale Wissen über gesendete Nachrichten vorhanden ist. Die Einplanung der Arbeitsaufträge dagegen ist ein lokaler Belang. Ein Auftrag ist hier z.B. ein Task. Für die übrigen Knoten in einem Netzwerk ist die Reihenfolge der Tasks eines einzelnen Knoten uninteressant. Es muss lediglich gewährleistet sein, dass der lokale Ablaufplan den globalen Kommunikationsablaufplan nicht verletzt. Es ist also darauf zu achten, dass die Aktualisierungszeitpunkte der Echtzeitdaten eines Knotens mit den Sende- und Empfangszeiten der entsprechenden Nachrichten des Kommunikationsablaufplans übereinstimmen.

# 3 Controller Area Network

Der CAN-Standard wurde von Bosch in Zusammenarbeit mit Intel entwickelt und 1985 vorgestellt. Seit 2003 ist er als ISO 11898 Standard akzeptiert. CAN ist heutzutage vor allem in der Automobilbranche weit verbreitet. Da TTCAN eine Erweiterung des CAN-Standards ist, ist es wichtig, zumindest ein Basiswissen über CAN zu besitzen. Dieses Kapitel soll dieses Wissen vermitteln. Dazu wird auf den CAN-Bus und die möglichen Nachrichtenformate eingegangen. Anschließend folgt dann ein kürzer Überblick über die Hardwareimplementierung des CAN-Protokolls auf dem TC1796.

## 3.1 CAN-Bus

Zunächst wird die physikalische Struktur des CAN-Busses und die Art des Zugriffs eines Knotens auf den Bus beschrieben.

### 3.1.1 Physikalische Randbedingungen

Der CAN-Bus ist ein asynchrones, serielles Bussystem. Jeder CAN-Bus besteht aus mindestens zwei gleichberechtigten Busteilnehmern, die Knoten genannt werden. Die Topologie der Knoten ist meist eine Linearstruktur. Möglich sind aber auch ringförmige oder sternförmige Anordnungen, diese sind aber weniger zuverlässig. Im Ring müsste nur ein Knoten ausfallen, um den gesamten Bus lahmzulegen, ebenso genügt in einer Sternstruktur der Ausfall des Zentralknotens. Die maximale Übertragungsrate über den CAN-Bus ist auf 1 MB/s festgelegt. Soll diese Bandbreite erreicht werden, ist zu beachten, dass aufgrund physikalischer Einflüsse wie der Signalausbreitungsgeschwindigkeit, die in einem widerstandsbehafteten Medium nur einen Bruchteil der Lichtgeschwindigkeit beträgt, und der Zeit, die ein Knoten braucht, um eine Nachricht vom Bus entgegenzunehmen, die Leitungslänge auf ca. 40 Meter begrenzt ist. Niedrigere Bitraten erlauben entsprechend längere Busse, allerdings nimmt die Signalstärke natürlich mit wachsender Distanz ab.

### 3.1.2 Buszugriff

Da alle Knoten prinzipiell gleichberechtigt sind, aber immer nur einer Zugriff auf den Bus haben kann, müssen Vorkehrungen getroffen werden, um Kollisionen auf dem Bus zu vermeiden. Das CAN-Protokoll arbeitet deshalb nach dem *CSMA/CD with NDA* (*Carrier Sense Multiple Access/Collision Detection with Non-Destructive Arbitration*) Prinzip. Die Buslogik entspricht dabei einer *Und* Verknüpfung. 0-Bits sind dominant und 1-Bits

rezessiv. Dominante Bits überschreiben rezessive Bits, d.h. das Schreiben eines dominanten 0-Bits ändert den auf dem Bus anliegenden Pegel ebenfalls auf das 0-Niveau. Ein Knoten lauscht also zuerst, ob bereits eine Nachricht eines anderen Knoten übertragen wird (*Carrier Sense*). Dazu wird einerseits jede Nachricht NRZ (*Non return to zero*) codiert, so dass der Pegel auf dem Bus nicht in regelmäßigen Abständen auf den 0-Pegel zurückkehrt. Ferner nutzt CAN die Technik des *Bitstuffing*. Dabei wird definiert, dass der Bitstrom einer Nachricht maximal 5 aufeinanderfolgende Bits gleichen Pegels enthält. An Stellen innerhalb der Nachricht an denen diese Bedingung nicht zutrifft, wird vom sendenden Knoten einfach ein Bit mit dem inversen Pegel eingefügt. Diese Operation ist bijektiv, kann also von den Empfängern eindeutig wieder rückgängig gemacht werden (*Destuffing*). Empfängt ein sendebereiter Knoten 6 rezessive Bits in Folge, weiß er, dass der Bus gerade untätig ist. Wenn nun mehrere Knoten gleichzeitig zu senden beginnen (*Multiple Access*), d.h. alle zur gleichen Zeit zu dem Ergebnis gekommen sind, dass der Bus untätig ist, muss dies von jedem Knoten als Kollision erkannt (*Collision Detection*) und gemeinsam eine Entscheidung getroffen werden, wer zuerst senden darf (*Non-Destructive Arbitration*). Um Kollisionen zu erkennen, liest ein Knoten nach dem Senden eines Bits den Bus Pegel nochmals ein und kontrolliert, ob der gesendete Pegel mit dem gelesenen Pegel übereinstimmt. Sendet der Knoten also ein rezessives Bit und liest ein dominantes, liegt eine Kollision vor. Im CAN-Protokoll ist definiert, dass eine Nachricht mit einem dominanten Bit eine höhere Priorität gegenüber einer Nachricht mit einem rezessiven Bit an gleicher Bitposition im Datenstrom besitzt. Damit ist die Sendereihenfolge festgelegt und der Knoten, der die Kollision erkannt hat, stoppt seine Sendeveruche solange, bis die höher priore Nachricht auf dem Bus übertragen wurde. Ein gerade sendender Knoten wird *Busmaster* genannt, alle anderen folglich *Buslaves*.

## 3.2 CAN-Nachrichten

Die Repräsentation einer CAN-Nachricht auf dem Bus wird als *Frame* bezeichnet. Es gibt insgesamt drei Arten von Frames: den *Data Frame* zur Übermittlung von Daten, den *Remote Frame* um Daten anzufordern und den *Error Frame* um einen erkannten Fehlerstatus mitzuteilen.

### 3.2.1 Data Frames

Ein Data Frame ist in fünf Segmente unterteilt:

1. Arbitrationssegment
2. Kontrollsegment
3. Datensegment
4. CRC Segment
5. Bestätigungssegment

Abbildung 3.1 zeigt eine genaue Aufschlüsselung dieser Felder. Sie gibt die exakte Bezeichnung und die Position der einzelnen Bits eines Frames sowie den zugehörigen Bus Pegel an, sollte dieser für ein Bit definiert sein. Das Arbitrationssegment beinhaltet unter

anderem die global eindeutige Identifikationsnummer der Nachricht und ist damit der für die in 3.1.2 beschriebene Busarbitrierung wichtige Teil einer Nachricht. Die Identifikationsnummer kann entweder 11 Bit breit sein, dann spricht man von einem *Standard Frame*, oder aber 29 Bit breit für einen *Extended Frame*. Das wichtigste Datum im Kontrollsegment ist die Länge der Datenbytes des Datensegments. Ein CAN Data Frame kann bis zu 8 Bytes an Nutzdaten speichern. Das CRC Feld enthält die Prüfsumme der Nachricht, um es zu ermöglichen, Übertragungsfehler zu erkennen. Wurde von den empfangenden Knoten ein Übertragungsfehler erkannt, wird dieser umgehend durch Senden eines dominanten Bits während des Bestätigungssegments signalisiert. Aufgrund dieser unmittelbaren Rückmeldung zählt das CAN-Protokoll zur Familie der *in-bit-response* Protokolle. Ein Data Frame wird von einem dominantem Bit zu Beginn und von 7 rezessiven Bits am Ende eingerahmt. Diese Bits signalisieren die Transition des Busses vom Zustand untätig nach belegt bzw. von belegt nach untätig unter Benutzung des in 3.1.2 erklärten *Carrier Sense* Mechanismus.

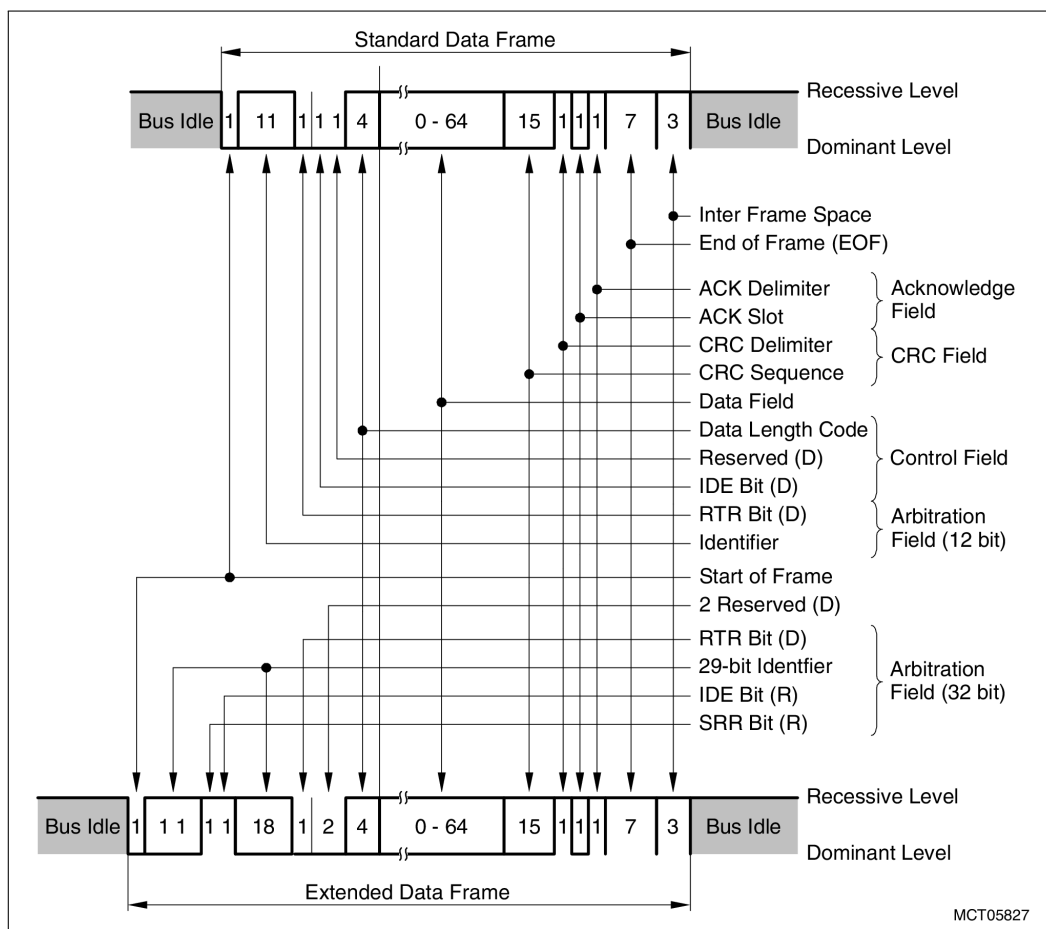


Abbildung 3.1: CAN Data Frame

### 3.2.2 Remote Frames

Remote Frames geben einem Knoten die Möglichkeit, Daten aus dem Netzwerk anzufordern. Der Knoten sendet dabei einen Remote Frame, dessen Identifikationsnummer der Nummer der Nachricht entspricht, die er benötigt. Der Knoten, der diese Nachricht bereitstellen kann, empfängt diesen Frame und sendet daraufhin den entsprechenden Data Frame. Ein Remote Frame ist in seinem Aufbau identisch mit dem Data Frame. Er besitzt

lediglich keine Nutzdaten. Auch hier unterscheidet man zwischen Standard und Extended Frames.

### 3.2.3 Error Frames

Ein Error Frame wird generiert, wenn ein Knoten einen Fehler auf dem Bus bemerkt. Ein solcher Busfehler ist z.B. die Verletzung der **Bitstuffing** Regeln. Es gibt zwei Arten von Error Frames: den aktiven und den passiven Error Frame. Ein Error Frame besteht im Gegensatz zu den Data und Remote Frames aus nur zwei Segmenten. Das erste Segment kennzeichnet den Frame als aktiv oder passiv. Ein aktiver Error Frame verletzt im Unterschied zum passiven Error Frame die Regeln des **Bitstuffings** und erzeugt so weitere von den übrigen Knoten gesendete Error Frames. Auf diese Weise kann sichergestellt werden, dass alle Knoten des Netzwerks von einer Fehlersituation ausgehen. Die Zeitspanne, in der das zweite Segment übertragen wird, dient den Knoten zur Vorbereitung der gemeinsamen Wiederaufnahme der Kommunikation nach einer Fehlersituation.

## 3.3 Datensynchronisierung

Bis hierhin wurde gezeigt, wie die einzelnen Knoten in einem CAN-Netzwerk auf den Bus zugreifen und welche Nachrichten sie verschicken können. Sowohl Arbitrierung als auch Nachrichtenaustausch können jedoch nur funktionieren, wenn jeder Knoten weiß, zu welchem Zeitpunkt er den Pegel auf dem Bus für jedes Bit messen muss. Mit anderen Worten, der Messzeitpunkt muss global eindeutig definiert sein, damit jeder Knoten für jedes Bit den gleichen Pegel misst. Diese Definition wird als *Bittiming* bezeichnet. Im CAN-Protokoll wird dazu eine *Bitzeit* (die Zeit, in der ein Bit auf dem Bus repräsentiert wird) in vier Abschnitte unterteilt, wie in Abbildung 3.2 dargestellt. Jeder Abschnitt nimmt dabei ein ganzzahliges Vielfaches eines Zeitquants  $t_q$  in Anspruch. Dieses Zeitquant begrenzt die zeitliche Auflösung in einem CAN-Netz. Für die Bitzeit gilt:

$$\text{Bitzeit} = \frac{1}{\text{Baudrate}}$$

Werden also z.B. 1Mbit Bandbreite benötigt, bleibt für jedes Bit logischerweise genau  $1\mu\text{s}$ . Die Länge des Zeitquants wird normalerweise von der Taktfrequenz des CAN-Knoten ( $f_{can}$ ) bestimmt. Um unterschiedliche Taktraten einzelner Knoten auszugleichen, lässt sich das Zeitquant mit einem programmierbaren Teiler (meist als *BRP* bezeichnet) skalieren:

$$t_q = \frac{BRP}{f_{can}}$$

Ein gültiges Bit-Timing muss eine Bitzeit in 8-25 Zeitquanten unterteilen

Der erste Abschnitt ist das Synchronisationssegment (*SYNC\_SEG*). Ein evtl. Pegelwechsel zwischen 2 Bits findet hier statt. Seine Länge ist immer  $1 t_q$ . Der zweite Abschnitt

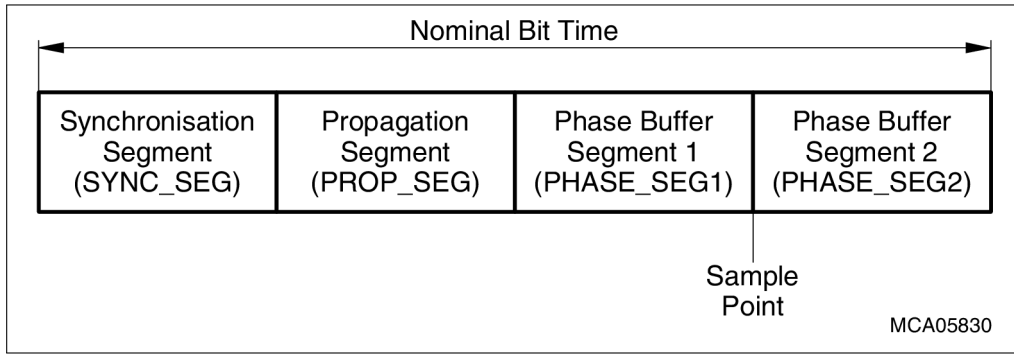


Abbildung 3.2: CAN-Bitzeit

- das Propagationssegment (*PROP\_SEG*) - soll die Ausbreitungsverzögerung im Netzwerk kompensieren. Damit die Kollisionserkennung korrekt funktioniert, muss die Länge von *PROP\_SEG* mindestens der doppelten Summe aller Verzögerungen im Netzwerk entsprechen (gerundet auf das nächste Vielfache eines Zeitquants). Die wichtigsten Verzögerungsursachen sind die endliche Signalausbreitungsgeschwindigkeit auf dem Busmedium und die Zeit, die ein CAN-Knoten braucht um ein Bit zu verarbeiten, d.h. es auf den Bus zu schreiben oder vom Bus zu lesen. Die Anzahl an Zeitquanten für *PROP\_SEG* hängt also im wesentlichen von der benötigten Kabellänge ab, da die Verarbeitungszeiten vergleichsweise klein sind. Die beiden letzten Abschnitte sind die sogenannten Phasensegmente (*PHASE\_SEG1*, *PHASE\_SEG2*). Sie sind dazu da, Phasenverschiebungen beim Flankenwechsel auszugleichen. Solche Phasenverschiebungen resultieren aus den Drifts der einzelnen Knoten. Der Messzeitpunkt am Ende von *PHASE\_SEG1* ist deshalb innerhalb dieser zwei Segmente um die *Synchronisation Jump Width* (SJW) verschiebbar. Genauer gesagt kann *PHASE\_SEG1* verlängert und *PHASE\_SEG2* entsprechend verkürzt werden. Beide Segmente sind zwischen 1 und 8  $t_q$  und SJW zwischen 1 und 4  $t_q$  lang, letzteres jedoch nicht länger als das Minimum der beiden Phasensegmente. Ein gültiges Bittiming für eine maximale Kabellänge von 40 Metern, einer Baudrate von 1Mbit und einer Frequenz von 75 MHz für  $f_{can}$  wäre zum Beispiel:

- $Bitzeit = 1\mu s$
- $BRP = 5$
- $t_q = 66.6\eta s$
- $\#t_q = 15$
- $t_{sync} = 1t_q$
- $t_{prop} = 9t_q$
- $t_{phase1} = 2t_q$
- $t_{phase2} = 3t_q$
- $t_{sjw} = 2t_q$

Es sei nochmals darauf hingewiesen, dass alle Knoten des CAN-Netzes auf das gleiche Bittiming programmiert sein müssen! Für eine detaillierte Beschreibung des Bittimings siehe [1].

## 3.4 TC1796 CAN-Hardware

Nachdem im Vorfeld die Grundlagen erläutert wurden, folgt nun ein Überblick, wie das CAN-Protokoll auf dem TC1796 implementiert wurde. Diese Informationen sind vor allem wichtig, um einige Begriffe, die in den nächsten Kapiteln verwendet werden, kennenzulernen. Dieser Abschnitt nimmt nicht für sich in Anspruch, eine vollständige Dokumentation des CAN-Knoten auf dem TC1796 zu sein. Dies würde bei Weitem den Rahmen dieser Arbeit sprengen. Die komplette Dokumentation findet sich unter [2].

### 3.4.1 Das CAN-Modul im Überblick

Bei den verschiedenen CAN-Implementierungen müssen zwei Sorten unterschieden werden: zum einen *Basic-CAN* Implementierungen und zum anderen *Full-CAN* Implementierungen. Während erstere nur die Basisfunktionalitäten des CAN-Protokolls (meistens nur die Kontrolle des Bitstroms auf dem Bus) implementieren und den Rest der CPU überlassen, implementiert die *Full-CAN* Variante das komplette Protokoll in Hardware, wodurch die CPU wesentlich weniger belastet wird. Bei der CAN-Hardware des TC1796 handelt es sich um eine *Full-CAN* Implementierung. Abbildung 3.3 zeigt eine schematische Darstellung der Hardware. Sie besteht im Wesentlichen aus vier autonomen CAN-Knoten, einem Vorrat von Nachrichten (*Nachrichtenobjekte*) und einer Listenverwaltung, die die Zuordnung der Nachrichten zu den Knoten regelt. Zusammen mit der CAN-Kontrolllogik sind diese Komponenten in der Lage, das gesamte CAN-Protokoll selbständig abzuarbeiten. Die CPU ist nur noch für die Initialisierung zuständig. Außerdem ist der erste Knoten noch mit der TTCAN-Steuerung verbunden. Diese wird später noch separat behandelt. Hier folgt nun eine kurze Beschreibung der einzelnen Komponenten der CAN-Schnittstelle.

### 3.4.2 Listenverwaltung

Die Allokation eines Nachrichtenobjekts an einen der vier Knoten wird über doppelt verkettete Listen geregelt. Jedem Knoten wird dabei eine Liste zugeordnet. Die erste Liste ist keinem Knoten zugeordnet und die Objekte darin gelten als nicht allokiert. Die Einfügeoperationen der Listen werden von der CAN-Steuerung ebenfalls bereits in Hardware implementiert und können direkt aufgerufen werden. Ein manuelles Schreiben der entsprechenden Register der Nachrichtenobjekte per Software ist nicht möglich.

### 3.4.3 Nachrichtenobjekt

Nachrichtenobjekte sind die Hardwarerepräsentation der Data und Remote Frames. Von welchem Typ ein Objekt ist, ist frei konfigurierbar. Jedes Objekt kann deswegen auch auf 8 Byte Datenspeicher zugreifen. Der Vorrat an Nachrichtenobjekten des CAN-Moduls umfasst 128 Objekte. Mehr Nachrichten können gleichzeitig nicht verwaltet werden. Neben den framespezifischen Einstellungen besitzt jedes Objekt noch Verkettungsinformationen wie *NEXT* bzw. *PREV* Zeiger, die Listenzugehörigkeit und Informationen über

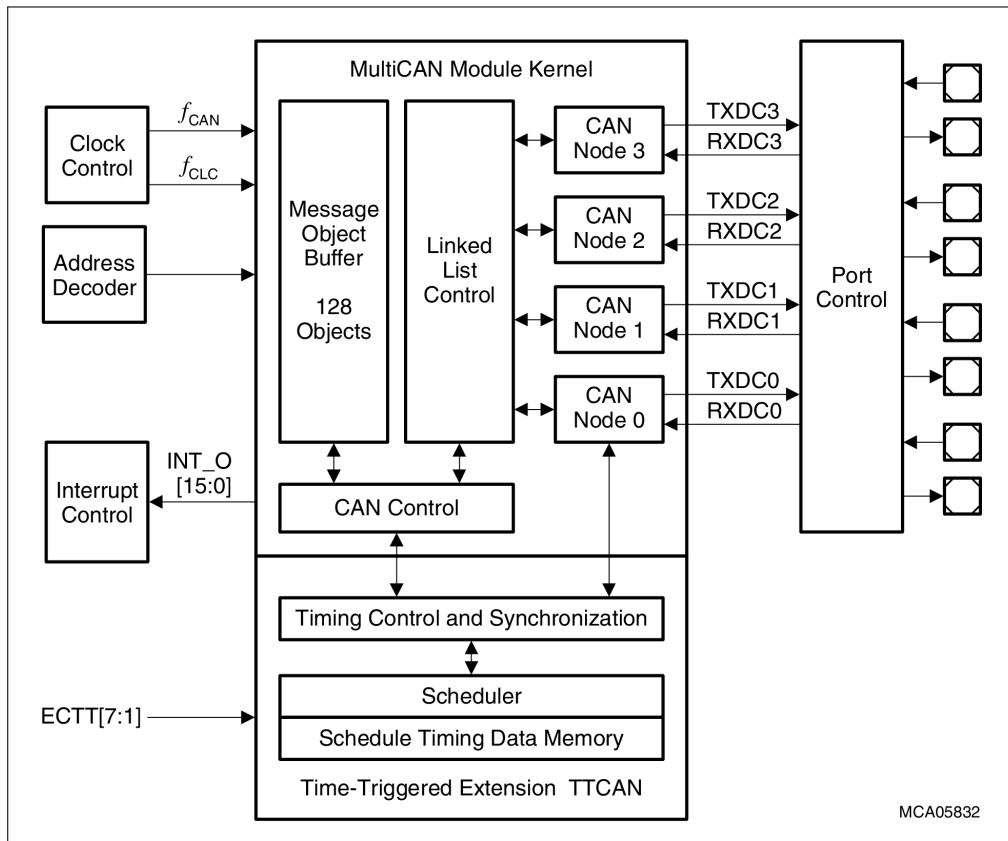


Abbildung 3.3: MultiCAN Module

Sende- bzw. Empfangsbereitschaft. Diese werden von der Listenverwaltung und der CAN-Kontrolllogik benötigt.

### 3.4.4 CAN-Kontrolllogik

Neben der Kontrolle des Bitstroms auf dem Bus (Schreiben und Lesen der Frames) und der Einhaltung des Bittimings ist diese Komponente vor allem für das Versenden und Empfangen von Nachrichten zuständig. Sie muss bestimmen, welches der sendebereiten Nachrichtenobjekte als Nächstes versendet und ob eine empfangene Nachricht in einem empfangsbereiten Nachrichtenobjekt gespeichert werden soll. Die Entscheidung über das Versenden von Nachrichtenobjekten wird *Transmit Acceptance Filtering* und die Empfangsentscheidung wird *Receive Acceptance Filtering* genannt.

# 4 TTCAN-Erweiterung

In diesem Kapitel wird zunächst der Zusammenhang zwischen CAN und TTCAN erklärt, dann wird darauf eingegangen, wie TTCAN ein TDMA-Verfahren realisiert. Anschließend folgt genau wie im vorhergehenden Kapitel ein Überblick über die Hardwareimplementierung auf dem TC1796.

## 4.1 TTCAN als zusätzliche CAN-Kommunikationsschicht

TTCAN ist eine Erweiterung des CAN-Standards und als solche unter ISO11898-4 definiert. Die Idee hinter TTCAN ist, ein zeitgesteuertes Kommunikationsprotokoll bereitzustellen, das komplett auf dem ereignisbasierten CAN aufsetzt. TTCAN nutzt ausschließlich die von CAN bereitgestellten Mittel oder führt echte Erweiterungen ein, so dass keine Änderungen am CAN-Protokoll gemacht werden mussten. TTCAN arbeitet nach dem TDMA-Verfahren. Den Knoten werden zum Versenden von Nachrichten exklusive Zeitfenster zugeordnet. Die Grenzen dieser Zeitfenster sind durch das Voranschreiten der globalen Zeit eindeutig bestimmt. Kollisionen lassen sich so von vorne herein ausschließen. TTCAN bietet ein vorhersehbares Verhalten, während es trotzdem vollständig kompatibel zum standard CAN-Protokoll bleibt. TTCAN kann auf zwei Ebenen implementiert werden:

**TTCAN-Ebene 1** (benutzt standard Frames) unterstützt im Betrieb keine Synchronisierung der globalen Zeit unter den Knoten. Das bedeutet, dass die Sicht der einzelnen Knoten auf die globale Zeit mit dem Voranschreiten der Zeit auseinander driften kann.

**TTCAN-Ebene 2** (benutzt erweiterte Frames) bietet einen Mechanismus zur Uhrensynchronisation.

Um eine zeitgesteuerte Kommunikationsschicht mithilfe von CAN zu implementieren, sind im Grunde nur vier Erweiterungen notwendig:

- Das Transmit Acceptance Filtering darf eine zeitgesteuerte Nachricht nur dann als Gewinner ermitteln, wenn deren Sendezeitpunkt erreicht wurde. Das ist eine zusätzliche Bedingung, die nicht mit den CAN-Regeln des Transmit Acceptance Filterings kollidiert.
- Es muss sichergestellt werden, dass zeitgesteuerte Nachrichten beim Transmit Acceptance Filtering immer bevorzugt werden. Die konkrete Implementierung dieser Priorisierung hängt davon ab, wie Nachrichten von der Hardware verwaltet werden.

- Die Sendefenster aller Nachrichten aller Knoten müssen kollisionsfrei auf der globalen Zeitachse angeordnet werden. Dazu benutzt TTCAN das Konzept des Matrixzyklus (siehe 4.4.)
- Es muss eine globale Zeitbasis bereitgestellt werden. Dazu dient im TTCAN-Protokoll die Referenznachricht (siehe 4.2.1).

## 4.2 Die globale Zeit

Die globale Zeitbasis wird in TTCAN von einem sog. *Timemaster* bereitgestellt. Ein Timemaster ist ein normaler Knoten, dessen lokale Systemzeit als die globale gültige Zeit definiert und den übrigen Knoten bekannt gemacht wird. In einem CAN-Netzwerk kann es aus Gründen der Redundanz mehrere Timemaster geben, jedoch ist immer nur einer der aktuelle Timemaster. Alle anderen Master Knoten werden als *potenzielle Timemaster* bezeichnet. Ein potentieller Timemaster kann bei einem Ausfall des aktuellen Timemasters zum neuen Timemaster werden. Knoten, die keine potenziellen Timemaster sind, heißen *Slaves*.

### 4.2.1 Die Referenznachricht

Die Referenznachricht wird vom aktuellen Timemaster in regelmäßigem Abstand gesendet. Die Nutzdaten dieser Nachricht enthalten Informationen über die globale Zeitbasis, mit deren Hilfe die übrigen Knoten die lokale in globale Zeit umrechnen können. Die Rangfolge unter den potentiellen Timemaster Knoten wird explizit mit Hilfe der drei ersten Identifikationsbits der Referenznachricht festgelegt. Die übrigen Identifikationsbits sind für jede Referenznachricht eines potentiellen Timemasters gleich. Da jeder Master-Knoten seine Referenznachricht zur selben Zeit verschicken will, entscheidet die CAN-Busarbitrierung, welcher Knoten die Nachricht verschicken darf. Der aktuelle Timemaster ist also immer der Knoten, der seine Referenznachricht versenden konnte.

### 4.2.2 Globale Zeiteinheit

Da in einem TTCAN-Netz nicht alle Knoten identisch sein müssen, kann nicht garantiert werden, dass die Uhren aller Knoten mit der gleichen Geschwindigkeit ticken. Deswegen wird die globale Zeit nicht in Zeitgebertaktschritten gezählt, sondern in einer abstrakten Größe: der *Network Time Unit* (NTU). Die Länge einer NTU wird beim Anwendungsentwurf festgelegt und entspricht der kleinsten Zeitspanne, die im Netzwerk auflösbar sein soll.

Im TTCAN-Ebene 1 ist eine NTU immer gleichbedeutend mit einer Bitzeit:

$$1 \text{ NTU} = 1 \text{ Bitzeit}$$

Im TTCAN-Ebene 2 muss die lokale Zeit synchronisiert werden. Die lokale Definition einer NTU, also die entsprechende Anzahl an Taktschritten, ist deshalb variabel. Die Umrechnung der lokalen Zeit in NTUs erfolgt hier nach folgender Formel:

$$1 \text{ NTU} = \frac{1}{TUR} \text{ Taktschritte}$$

TUR steht dabei für *Time Unit Ratio* und bezeichnet den lokalen Teiler, der das Verhältnis von lokaler zu globaler Zeit bestimmt.

### 4.2.3 Drift Korrektur

Die lokalen Uhren der Knoten unterliegen aufgrund physikalischer Einflüsse wie z.B. Temperaturschwankungen unweigerlich einer Drift. Diese Drift führt mit der Zeit dazu, dass ein Knoten nicht mehr synchron zum aktuellen Timemaster läuft. Aus Sicht eines Slave Knoten können innerhalb eines bestimmten Intervalls also mehr oder weniger NTUs vergangen sein, als für den Timemaster. Um die Synchronität wieder herzustellen, muss die lokale Definition einer NTU in regelmässigen Abständen mit der des Timemasters verglichen und korrigiert werden. Dies geschieht, indem man die lokal gemessene Zeitdifferenz zwischen zwei Synchronisierungszeitpunkten mit der entsprechenden globalen Differenz vergleicht. Daraus ergibt sich der prozentuale Laufzeitunterschied.

$$TURADJ = \frac{LREFM - LLREFM}{GREFM - LGREFM}$$

LREFM und LLREFM sind dabei die aktuelle lokale Zeit (LREFM) und die lokale Zeit zum Zeitpunkt der letzten Synchronisierung (LLREFM) gemessen in NTUs. Entsprechend sind GREFM und LGREFM die Werte des Timemasters, die mit der Referenznachricht übertragen wurden. Der TUR Wert wird dann mit Hilfe von TURADJ korrigiert, und somit die Länge einer NTU lokal verändert:

$$TUR_{n+1} = TUR_n * TURADJ$$

Diese Art der Synchronisierung ist nur im TTCAN-Ebene 2 verfügbar.

## 4.3 Basiszyklus

Die Periode zwischen zwei Referenznachrichten des Timemasters wird Basiszyklus genannt. Ein Basiszyklus besteht aus mehreren Zeitfenstern, denen Nachrichten zugeordnet werden können. Die Grenzen der Fenster sind durch Zeitmarken eindeutig gekennzeichnet. Die Anzahl an NTU, die seit dem Beginn des Basiszyklus vergangen sind nennt man Zykluszeit. Mit Hilfe der Zykluszeit kann das Erreichen einer Zeitmarke erkannt werden. Die Zykluszeit beginnt per Definition zu Beginn eines jeden Basiszyklus bei 0. Alle Zeitmarken werden somit relativ zum Zyklusbeginn angegeben.

Es gibt drei Arten von Zeitfenstern:

- **Exklusive Fenster** sind für die Übertragung von TTCAN-Nachrichten bestimmt.
- **Arbitrierungsfenster** sind Zeitschlitze, die nicht von TTCAN-Nachrichten belegt sind. Hier können normale CAN-Nachrichten verschickt werden.
- **Freie Fenster** werden weder von CAN noch von TTCAN-Nachrichten belegt.

Wie man sieht, ist es also durchaus möglich ein Mischsystem aus CAN und TTCAN zu betreiben ohne die zeitliche Determiniertheit zu verlieren.

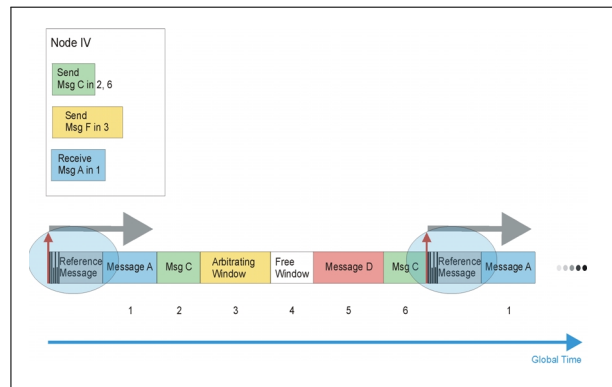


Abbildung 4.1: TTCAN-Basiszyklus

Abbildung 4.1 zeigt einen einfachen Basiszyklus. Neben einem Arbitrierungs- und einem freien Fenster belegen die Nachrichten A, C und D exklusive Zeitfenster. Die Zeitfenster für die Referenznachricht wurden zur Verdeutlichung eingekreist. In TTCAN-Ebene 2 erfolgt die Zeitsynchronisierung am Ende bzw. Anfang eines jeden Basiszyklus, nämlich dann, wenn von den Slave Knoten eine neue Referenznachricht empfangen wurde. Diese enthält die entsprechenden globalen Informationen aus der Sicht des Timemasters.

## 4.4 Matrixzyklus

Ein einzelner Basiszyklus ist in der Praxis jedoch meist zu unflexibel. Es kann z.B. Nachrichten geben, deren Periode länger als ein Basiszyklus ist, so dass ein Senden in jedem Basiszyklus unnötig ist. Deshalb kombiniert TTCAN mehrere Basiszyklen zum sog. Matrixzyklus.

Wie in Abbildung 4.2 zu sehen, ist ein Matrixzyklus also eine lineare Abfolge von mehreren Basiszyklen. Jeder Basiszyklus kann dabei prinzipiell eine komplett andere Abfolge von Nachrichten beinhalten. Um die Matrixform zu wahren, stellt TTCAN allerdings die Bedingung, dass sich die Fensterbreiten in einer Spalte von einem zum anderem Basiszyklus nicht ändern dürfen.

## 4.5 TC1796 TTCAN-Hardware

Die TTCAN-Hardware des TC1796 besteht im Wesentlichen aus 2 Komponenten:

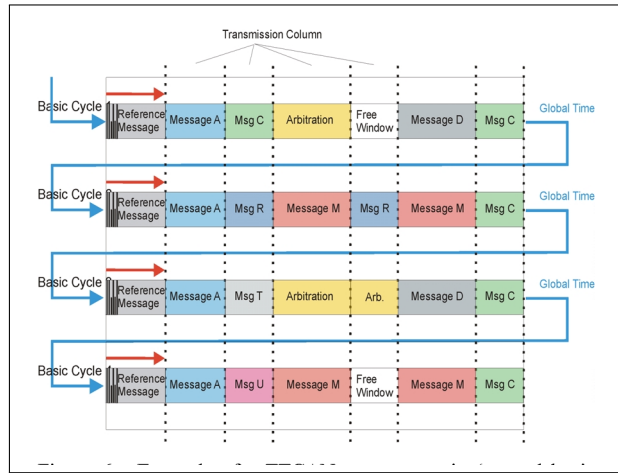


Abbildung 4.2: TTCAN-Matrixzyklus

- Zeitkontrolle und Uhrensynchronisation
- Scheduler

Im Folgenden werden diese beiden Komponenten kurz vorgestellt. Für eine Komplette Beschreibung siehe [2].

### 4.5.1 Zeitkontrolle und Uhrensynchronisation

Diese Einheit ist dafür verantwortlich die Zykluszeit in der richtigen Frequenz zu inkrementieren. Dazu benötigt es als Eingabe die Frequenz des lokalen Zeitgebers und die zu verwendende TTCAN-Ebene. Darüber hinaus speichert es die TUR und TURADJ Werte und stellt noch die Logik zur Verfügung, um die lokale Zeit automatisch mit der globalen Zeit zu synchronisieren.

### 4.5.2 Scheduler

Der Scheduler ist die Hardwarerepräsentation des Matrixzyklus. Er besteht neben der Kontrolllogik nur aus Speicher für insgesamt 128 32-Bit Einträge. Jeder dieser Einträge bestimmt einen Zeitpunkt oder eine Aktion. Die wichtigsten Typen von Einträgen sind:

- **Zeitmarkeneintrag:** definiert eine Zeitmarke und damit den Beginn eines neuen Zeitfensters
- **Sendekontrolleintrag:** bestimmt eine Nachricht, die in einem Zeitfenster verschickt werden soll
- **Empfangskontrolleintrag:** gibt an, dass der Empfang einer Nachricht, die im letzten Zeitfenster versendet wurde, geprüft werden soll.
- **Referenznachrichteintrag:** nur gültig für potentielle Timemaster. Sendet die Referenznachricht.
- **Basiszyklus-Ende Eintrag:** markiert das Ende des Basiszyklus.

Die Einträge werden nach Zeitmarken geordnet in den Speicher geschrieben. Um die Anzahl der Einträge zu reduzieren, können einem Zeitmarkeneintrag mehrere Aktionseinträge folgen. Weiterhin dürfen nicht einfach alle Basiszyklen nacheinander in den Speicher geschrieben werden. Da die Zeitfenster innerhalb einer Spalte für alle Basiszyklen identisch sind, genügt es die Aktionseinträge einer Spalte des Matrixzyklus nach dem entsprechenden Zeitmarkeneintrag zu definieren und sie für Basiszyklen auszumaskieren, in denen sie nicht benötigt werden. Als Konsequenz darf es auch nur einen Basiszyklus-Ende Eintrag im Scheduler Speicher geben. Dieser ist gleichzeitig immer der letzte Eintrag. Wenn er gelesen wird, beginnt die Abarbeitung der Scheduler Einträge für den nächsten Basiszyklus von vorne. Ein Timemaster muss unmittelbar vor dem Basiszyklus-Ende Eintrag den Referenznachrichteintrag definieren.

# 5 ProOSEK/Time

Dieses Kapitel widmet sich dem Betriebssystem, für das der TTCAN-Treiber entwickelt wird. ProOSEK/Time ist eine Implementierung des OSEKtime 1.0 Standards (siehe [4]). OSEKtime definiert ein zeitgesteuertes Echtzeitbetriebssystem für eingebettete Systeme. Das gesamte System mit allen Ressourcen und zeitlichen Randbedingungen muss deshalb im Voraus vom Anwendungsentwickler definiert werden. Zur Systemkonfiguration dient ein graphisches Konfigurationswerkzeug mit dessen Hilfe der Entwickler das System beschreibt. Aus dieser Beschreibung kann dann automatisch der Code des Betriebssystemkerns generiert werden. Im Folgenden werden nun die wichtigsten Konzepte und Funktionalitäten des ProOSEK/Time Kerns beschrieben.

## 5.1 Ablauftabelle

ProOSEK/Time ist ein Multitasking fähiges Betriebssystem und benötigt als solches einen Ablaufplan, der den einzelnen Tasks Prozessorzeit zuweist. Dieser muss vom Entwickler aus dem Vorwissen über die Eigenschaften der einzelnen Arbeitsaufträge (WCET, Periode, Termin...) entweder manuell oder mit Hilfe eines separaten Werkzeugs im voraus erstellt werden. Das Konfigurationswerkzeug von ProOSEK/Time bietet hierfür keine Unterstützung. Die sogenannte Ablauftabelle dient in ProOSEK/Time zur Speicherung des statischen Ablaufplans. Sie definiert ein Zeitintervall, in dem man einzelnen Zeitpunkten Aktionen zuordnen kann. Mögliche Aktionen sind Aktivierung von Arbeitsaufträgen, Interruptbehandlung und Terminüberprüfung. Diese Aktionen werden dann zur Laufzeit mit Voranschreiten der physikalischen Zeit durch den Systemzeitgebers ausgelöst. Die Dispatcher Tabelle ist zyklisch. Nach einem Durchlauf wird wieder am Anfang begonnen.

## 5.2 Anwendungsmodi

ProOSEK/Time kennt verschiedene Anwendungsmodi. Jeder Modus definiert dabei eine neue Dispatcher Tabelle. Dies erhöht die Flexibilität für den Fall, dass ein bestimmter Ablaufplan mit einer periodischen Dispatcher Tabelle allein nicht darstellbar ist, oder wenn das System mehrere Anwendungen implementiert. Einzige Einschränkung ist, dass alle Tabellen gleich lang sein müssen. Die Umschaltung erfolgt zur Laufzeit mit den Systemaufruf `ttSwitchAppMode()` und wird immer an Ende der aktuellen Dispatcher Tabelle ausgeführt. Der anfängliche Modus wird beim Systemstart der Funktion `ttStartOS()` übergeben.

## 5.3 Arbeitsaufträge

Die Aktivierung eines Arbeitsauftrags erfolgt anhand der Ablaufabelle. In ProOSEK/Time besitzt jeder Arbeitsauftrag zu einem bestimmten Zeitpunkt einen von drei Zuständen:

- pausiert (suspended)
- aktiv (running)
- verdrängt (preempted)

Jeder Arbeitsauftrag befindet sich zu Anfang im Zustand *pausiert*. Der Auftrag, der gerade im Besitz des Prozessors ist, befindet sich im Zustand *aktiv*. Ein aktiver Task wechselt in den Zustand *verdrängt*, wenn ein anderer in der Ablaufabelle definierter Auftrag ausgeführt werden soll. Ein verdrängter Auftrag bleibt solange verdrängt, bis der Auftrag, der ihn verdrängt hat, seine Arbeit beendet hat und wieder in den Zustand *pausiert* übergeht. Weitere Zustände sind in einem statischen System nicht notwendig, da keine neuen Arbeitsaufträge erzeugt werden können und alle Aufträge die ganze Laufzeit über im System vorhanden bleiben. Ist kein Auftrag aktiv, wird der Leerlauf task gestartet. Anstelle des Leerlauf tasks kann auch ein komplettes OSEK/VDX System treten (siehe [4]). Dieses ist nur dann aktiv, wenn kein zeitgesteuerter Arbeitsauftrag aktiv ist, so dass auch hier keine Echtzeitanforderungen verletzt werden. In ProOSEK/Time laufen alle Arbeitsaufträge im selben Adressraum.

## 5.4 Synchronisierung

Zur Unterstützung von verteilten Anwendungen bietet ProOSEK/Time verschiedene Möglichkeiten, die lokale Zeit mit der globalen Zeit zu synchronisieren:

**Harte Synchronisation** findet am Ende des aktuellen Durchlaufs der Dispatcher Tabelle statt. Der aktuelle Durchlauf wird bei dieser Methode entweder verkürzt oder der nächste Durchlauf verzögert.

**Weiche Synchronisation** ähnelt der harten Synchronisation mit dem Unterschied, dass die Synchronisierung über mehrere Dispatcher Durchläufe hinweg geschieht.

Es ist auch möglich, auf Synchronisation zu verzichten. Das System kann nur synchronisiert werden, wenn es am Ende der Dispatcher Tabelle eine ungenutzte Zeitspanne gibt, da nur so die Verkürzung bzw. Verzögerung der Dispatcher Durchläufe möglich ist. Die Art der Synchronisation wird im bei der Konfiguration festgelegt. Der Systemaufruf `ttSyncTimes()` führt dann zur Laufzeit automatisch die korrekte Synchronisierung durch.

# 6 Entwurf und Implementierung des Treibers

Dieses Kapitel widmet sich nun der Entwicklung des TTCAN-Treibers für den TC1796. Es ist logisch in vier Bereiche aufgeteilt. Der Erste gibt einen allgemeinen Einblick in das Design des Treibers. Dem schließt sich eine Beschreibung der implementierten CAN- und TTCAN-Funktionalität an. Das Kapitel endet mit der Evaluierung der Implementierung hinsichtlich Speicherbedarf und Laufzeitverhalten.

## 6.1 Designüberblick

Wie bereits erwähnt handelt es sich beim CAN-Controller des TC1796 um einen Full-Featured Controller mit TTCAN-Erweiterung. Die Aufgabe des Treibers ist es also nicht, das CAN bzw. TTCAN-Protokoll zu implementieren, sondern viel mehr eine Schnittstelle zwischen Anwendung und Hardware zu schaffen, die die Programmierung der Hardware kapselt und so das vom Benutzer benötigte Hintergrundwissen minimiert.

Die CAN-Schnittstelle ist wie die gesamte Hardware des TC1796 modular aufgebaut. Die verschiedenen funktionalen Einheiten werden dabei durch separate Registergruppen nach außen repräsentiert. Diese logische Trennung spiegelt sich auch in der physikalischen Einblendung der Register in den CPU Adressraum wider. Funktional zusammengehörige Register belegen immer einen zusammenhängenden Speicherbereich. Das Hardwaredesign legt also bereits ein sinnvolles Design für den Treiber nahe. Die Trennung der Module erfolgt analog zur Hardware. Register und Registergruppen können leicht über eigene Datentypen angesprochen werden. Beides sind Konzepte, die von der verwendeten Programmiersprache C unterstützt werden.

Obwohl es in dieser Arbeit darum geht, einen Treiber zu schreiben, der die TTCAN-Funktionalität des TC1796 nutzbar macht, wurden zunächst die wichtigsten Module für den normalen CAN-Betrieb implementiert. Dies ist deswegen sinnvoll, da TTCAN eine weitere Kommunikationsschicht über CAN darstellt und der TTCAN-Treiber somit auf CAN-Funktionalität zurückgreifen muss. Eine saubere Schnittstelle zu den CAN-Modulen macht den Code des TTCAN-Treibers leichter les- und wartbar. Der Treiber kann folglich auch ohne TTCAN-Unterstützung kompiliert und genutzt werden. Es wurde allerdings keine Ereignis- bzw. Interruptbehandlung implementiert, so dass der Empfang von Nachrichten aktiv abgefragt werden muss.

## 6.2 CAN-Funktionalität

Im Folgenden werden nun die implementierten CAN-Module vorgestellt. Sie orientieren sich strikt an den CAN-Controller Komponenten des TC1796. Obwohl nicht die gesamte Funktionalität des CAN-Controllers unterstützt wird, ist mit den implementierten Modulen doch ein einfacher Nachrichtenaustausch über den CAN-Bus möglich. Deshalb wird anschließend gezeigt, wie man den CAN-Treiber in ProOSEK nutzen kann.

### 6.2.1 Tricore CAN-Modul

Dieses Modul implementiert Funktionen, die nötig sind, um das CAN-Modul auf dem TC1796 Board nach dem Einschalten bzw. einem Hardware Reset zu initialisieren. Dazu zählen vor allem:

- Abbildung der Speicheradressen für Register auf globale Datenstrukturen
- Aktivieren des CAN-Moduls durch Konfiguration der Taktgeber für  $f_{can}$
- Konfiguration des Bit-Timings und der Ein- und Ausgabelleitungen für die CAN-Knoten
- Deallokierung aller Nachrichtenobjekte

Da die Reihenfolge der Operationen eine Rolle spielt und der Code zum Teil auf geschützte Register zugreift, existiert eine Schnittstellenfunktion, die diese Funktionen kapselt:  
`void CanStart()`.

### 6.2.2 Nachrichten Verwaltung

Dieses Modul stellt Funktionen zur Verfügung, um Nachrichtenobjekte zu allokiieren, d.h. sie einem bestimmten CAN-Knoten zuzuordnen. Die Hardware des TC1796 stellt dazu mehrere doppelt verkettete Listen bereit. Jeder Knoten hat Zugriff auf eine Liste. Ein Nachrichtenobjekt kann von einem Knoten benutzt werden, um Nachrichten zu versenden oder zu empfangen, wenn das Nachrichtenobjekt in die Liste des entsprechenden Knotens eingefügt wurde.

### 6.2.3 Nachrichtenobjekte

Dieses Modul implementiert die Schnittstelle zur Hardwarerepräsentation eines Nachrichtenobjekts des TC1796. Es bietet Funktionen zur Konfiguration und Auslesen aller Parameter, die für einen korrekten Nachrichtenaustausch über das CAN-Protokoll notwendig sind. Dazu zählen vor allem:

- Lesen und Schreiben der Datenbytes
- Lesen und Schreiben der ID Nummer

- Setzen und löschen der Flags, die darüber entscheiden, ob in einer Nachricht Daten empfangen oder gesendet werden sollen.

## 6.2.4 CAN-Knoten

Dieses Modul implementiert Funktionen zur Steuerung des Verhaltens der insgesamt 4 CAN-Knoten des TC1796. Da die Konfiguration der Knoten zum größten Teil statisch während der Initialisierung des CAN-Moduls erfolgen kann, beschränkt sich der Funktionsumfang dieses Moduls auf das Nötigste:

- Aktivieren und Deaktivieren eines Knotens, d.h. die Verbindung mit dem Bus herstellen oder trennen und der Wechsel in bzw. aus dem Konfigurationsmodus.
- Auswahl des Busses. Es stehen ein interner Loopback Bus und ein externer Bus zur Verfügung.

## 6.2.5 Einbindung des CAN-Treibers in ProOSEK

Um den CAN-Treiber in ProOSEK nutzen zu können, muss das CAN-Modul während des Bootvorgangs initialisiert werden, also noch bevor die Abarbeitung der Anwendung beginnt. Dazu ist lediglich der Aufruf der Funktion `CanStart()` an geeigneter Stelle notwendig. ProOSEK bietet für die Initialisierung von Modulen der Hardware auf der Tricore Architektur deshalb eine vom Benutzer implementierbare Funktion an. Diese wird dann während des Bootvorgangs aufgerufen.

```
1 #include "can.h"
2 ...
3 void initTRICOREModules()
4 {
5     CanStart();
6 }
7 ...
```

Nachrichtenojekte können unter Verwendung der folgenden Funktionen an beliebiger Stelle allokiert und konfiguriert werden. Bezeichner in spitzen Klammern sind vom Benutzer anzugeben.

```

1 #include "can.h"
2 ...
3 /* Allokation und Konfiguration eines zu sendenden
4  * Nachrichtenobjekts
5  */
6 unsigned char tx;
7 tx = CanNewTXObject(<LIST>, <ID>, <EXTENDED>);
8
9 /* Allokation und Konfiguration eines zu empfangenden
10  * Nachrichtenobjekts
11  */
12 unsigned char rx;
13 rx = CanNewRXObject(<LIST>, <ID>, <MASK>, <EXTENDED>);

```

<LIST> ist die Liste, in die das Objekt eingefügt werden soll. Die Listen [1, 4] entsprechen jeweils einem der vier CAN-Knoten. <ID> ist die eindeutige Identifikationsnummer der Nachricht. <MASK> ermöglicht es festzulegen, welche Bits der Identifikationsnummer bei der Empfangsentscheidung übereinstimmen müssen, damit eine Nachricht akzeptiert wird. Der boolesche Wert <EXTENDED> gibt an, ob es sich um eine standard Identifikationsnummer mit 11 Bits handelt, oder um eine erweiterte mit 29 Bits. Alle übrigen Attribute des Nachrichtenobjekts werden mit standard Werten belegt. Dies sollte in den meisten Fällen vollkommen ausreichend sein. Spezielle Konfigurationen müssen ansonsten mit Hilfe von grundlegenden Treiberfunktionen vorgenommen werden, die jedoch hier nicht diskutiert werden. Der Rückgabewert beider Funktionen ist die Nummer des allokierten Nachrichtenobjekts.

Das Lesen und Schreiben von Daten aus bzw. in Nachrichtenobjekte ist mit den folgenden Funktionen möglich:

```

1 #include "can.h"
2 ...
3 /* Schreibe Daten von <DATA> in eine zu
4  * sendende Nachricht.
5  */
6 CanMoWrite(<MESSAGE>, <DATA>, <BYTES>);
7 ...
8 /* Lese Daten aus einer empfangenen Nachricht
9  * und speichere sie in <DATA>
10  */
11 CanMoRead(<MESSAGE>, <DATA>, <BYTES>);
12 ...

```

<MESSAGE> ist die Nummer des betreffenden Nachrichtenobjekts, <DATA> ist ein Zeiger auf den Speicher, der die zu lesenden bzw. zu schreibenden Daten enthält. <BYTES> ist die Anzahl der zu lesenden oder zu schreibenden Datenbytes. Zu beachten ist, dass nach jedem Schreibvorgang in ein Nachrichtenobjekt der CAN-Schnittstelle automatisch die Sendebereitschaft der Nachricht signalisiert wird. Die Nachricht wird damit bei nächster Gelegenheit gesendet.

Der Treiber kann leicht in den Erstellungsprozess von ProOSEK eingebunden werden. Das Makefile ist dazu lediglich in zwei Punkten zu ergänzen:

- Hinzufügen des Pfades zu den Treiber Quellen zur `VPATH` Variable
- Hinzufügen des Pfades zu den Header Dateien zur `CC_INCLUDE_USER` Variable

## 6.3 TTCAN-Funktionalität

Die Implementierung der TTCAN-Unterstützung im Treiber beruht im Wesentlichen auf zwei zusätzlichen Modulen, die hier kurz beschrieben werden. Anschließend wird noch auf die Einbindung des TTCAN-Treibers in ProOSEK/Time eingegangen. Dies geschieht zwar weitestgehend automatisch durch das Konfigurationsprogramm, die wichtigsten Punkte, die dazu nötig sind, sollen aber dennoch nicht undokumentiert bleiben.

### 6.3.1 TTCAN-Erweiterungsmodul

Dieses Modul ist eine Erweiterung des CAN-Moduls. Es kapselt dessen Funktionalität und ergänzt diese um weitere Adressabbildungen und Funktionen zur Uhrensynchronisation der Anwendung und zur Signalisierung der Sendebereitschaft einer Nachricht. Die wichtigsten Schnittstellenfunktionen sind:

**ttCanStart()** ist das Äquivalent zu `CanStart()` und kapselt die gesamte Initialisierung. Es wird jedoch nur der erste der vier CAN-Knoten des TC1796 initialisiert, da nur dieser mit der TTCAN-Hardware verbunden ist.

**ttCanSyncTimes(ttTickType)** synchronisiert den Systemzeitgeber mit der globalen Zeit. Diese Funktion muss einmal während eines Durchlaufs der ProOSEK/Time Dispatcher Tabelle aufgerufen werden.

**ttCanSetTransmitTrigger(volatile void\*)** muss nach jeder Aktualisierung der Daten einer Nachricht aufgerufen werden.

**ttCanSyncStart()** ermöglicht das synchrone Starten der ProOSEK/Time Ablaufabelle zum Beginn eines neuen Matrixzyklus.

### 6.3.2 TTCAN-Scheduler Modul

Dieses Module stellt die Funktionalität bereit, um den TTCAN-Scheduler auf dem TC1796 zu programmieren. Es implementiert Funktionen, um die verschiedenen Typen von Einträgen zu erstellen und diese in den Scheduler Speicher zu schreiben.

### 6.3.3 Einbindung des TTCAN-Treibers in ProOSEK/Time

Um den TTCAN-Treiber in ProOSEK/Time zu nutzen muss das CAN-Modul und der TTCAN-Knoten initialisiert werden. Dazu wird analog zu CAN-Treiber die Funktion `ttCanStart()` beim Systemstart aufgerufen.

```
1 #include "ttcan.h"
2 ...
3 void initTRICOREModules()
4 {
5     ttCanStart();
6 }
7 ...
```

Die Allokation und Konfiguration der Nachrichten sowie das Einfügen der TTCAN-Scheduler Einträge erfolgt automatisch durch vom TTCAN-Konfigurationswerkzeug erstellten Quelltext. Um die lokale mit der globalen Systemzeit zu synchronisieren, muss die Funktion `ttCanSyncTimes(ttTickType)` einmal während des Durchlaufs einer jeden Dispatcher Tabelle aufgerufen werden. Sie erwartet als Argument die momentane Zeit relativ zum Anfang der Dispatcher Tabelle in Ticks. Dies geschieht deswegen am Einfachsten in einem separaten Task, dessen Aktivierungszeitpunkt genau definiert ist.

```
1 #include "ttcan.h"
2 ...
3 ttTASK(SyncTask)
4 {
5     /* OSEKOSTTNS2TICKS ist ein von ProOSEK/Time
6      * definiertes Makro
7      */
8     ttCanSyncTimes(OSEKOSTTNS2TICKS(dispatcher_time));
9 }
10 ...
```

Der ProOSEK/Time Startmodus sollte in der Betriebssystemkonfiguration fest auf SYNC gesetzt werden, um Phasenverschiebungen zwischen der ProOSEK/Time Ablaufabelle und dem Matrixzyklus zu vermeiden. Im Leerlauf task muss in diesem Fall der Beginn eines neuen Matrixzyklus abgefragt werden, damit die Ausführung der Ablaufabelle entsprechend verzögert wird. Dazu wird die Schnittstellenfunktion `ttCanSyncStart()` verwendet.

```
1 #include "ttcan.h"
2 ...
3 ttTASK(ttIdleTask)
4 {
5     while (!ttCanSyncStart());
6
7     while (1)
8     {
9         ...
10    }
11 }
```

## 6.4 Evaluierung

Zur Bestimmung des Laufzeitverhaltens und des Speicherbedarfs der Schnittstellenfunktionen des Treibers werden eine Reihe von ProOSEK/Time Betriebssystemkonfigurationen erstellt, die sich nur in der Anzahl der aktiven Nachrichtenobjekte unterscheiden. Zum Kompilieren der Betriebssysteme werden folgende Compileroptionen benutzt: `-O2 -falign-functions=4`. Alle Laufzeiten werden durch Auslesen und Differenzbildung der Systemzeitgeberwerte vor und nach der Ausführung der jeweiligen Funktion gemessen. Zu beachten ist bei allen Laufzeiten außerdem, dass diese in nicht unerheblichem Maße dadurch beeinflusst werden, dass die auszuführenden Instruktionen im externen RAM des TC1796 liegen. Beim Zugriff darauf kommt es zu Verzögerungen ausgelöst durch Wartezyklen (*Waitstates*) des Prozessors. Tabelle 6.1 zeigt die Laufzeiten der Funktion `ttCanStart()` in Abhängigkeit zur Anzahl der Verwendeten Nachrichtenobjekte. Die Messung der maximalen Ausführungszeit für eine feste Zahl von Nachrichtenobjekten ist bei dieser Funktion nicht möglich, da die Anzahl der enthaltenen Instruktionen nicht unmittelbar mit der Anzahl der Nachrichten zusammenhängt. Das liegt daran, dass eine Nachricht je nach Ablaufplan eine unbestimmte Anzahl an Einträgen in den Scheduler Speicher bedingt. Die Laufzeit ist damit zwar für jede Konfiguration von Nachrichten konstant, muss jedoch bei Bedarf für jede Konfiguration neu gemessen werden.

Nachrichten	Taktschritte
1	11925
2	13935
4	18699
8	25358
12	32378
16	40832
24	56196

Tabelle 6.1: Laufzeiten der Funktion `ttCanStart()`

Wie man erkennen kann, wächst die Ausführungszeit der Funktion in etwa linear mit der Anzahl der Nachrichtenobjekte. Die Konfiguration eines zusätzlichen Objekts benötigt ca. 2000 Taktschritte. Bei einem synchronen Betriebssystemstart muss zu den gemessenen Zeiten zusätzlich noch die Hyperperiode aller Nachrichten addiert werden, um die Dauer bis zum Beginn der Abarbeitung der ProOSEK/Time Ablaufabelle abschätzen zu können. Die Ablaufabelle kann in diesem Fall nämlich erst zum Start eines neuen Matrixzyklus ausgeführt werden.

Die Ausführungszeit der Funktion `ttCanSetTransmitTrigger(volatile void*)` ist für alle Nachrichten konstant und beträgt 491 Taktschritte. Die Ausführungszeit der Funktion `ttCanSyncStart()` ist nicht von Interesse, da diese nur im Leerlauf task ausgeführt werden kann.

Tabelle 6.2 enthält den Speicherverbrauch in Abhängigkeit von der Nachrichtenanzahl aufgeschlüsselt nach Segmenten.

Der Tabelleneintrag mit keiner Nachricht repräsentiert ein Betriebssystem, das den TTCAN-Treiber nicht mit einbindet. Aus der Differenz zum Eintrag mit einer Nachricht lassen sich

Nachrichten	.text	.data	.bss	Gesamt
0	2592	1024	78	3694
1	6035	1024	648	7708
2	6192	1024	652	7868
4	6496	1024	660	8180
8	7120	1024	676	8820
12	7800	1024	692	9516
16	8492	1024	708	10224
24	9868	1024	740	11632

Tabelle 6.2: Speicherverbrauch des Treibers

so die minimalen Mehrkosten, die durch die Verwendung des Treibers entstehen ablesen. Aus der Tabelle geht hervor, dass die Größe des `.text` Segments in etwa linear mit der Anzahl der Nachrichten wächst. Die Instruktionen zur Konfiguration einer Nachricht benötigen also in etwa 160 Byte. Aber auch hier gilt wie bei den Ausführungszeiten der Funktion `ttCanStart()`, dass dies aufgrund der variablen Anzahl an Einträgen in den Scheduler Speicher keine festen Werte sein können. Eine Optimierung der relativ hohen Fixkosten von über 4 KByte für die Treiberinstruktionen könnte in begrenztem Umfang durch Ersetzen von Treiberfunktionen mit statische Initialisierungsanweisungen erfolgen. Viele Treiberfunktionen lösen jedoch lediglich in Hardware implementierte Operationen aus, so dass sie nur zur Laufzeit ausgeführt werden können. Der Speicherzuwachs im `.bss` Segment entspricht neben den Fixkosten für globale Datenstrukturen genau den 4 Byte für den exportierten Zeiger auf die Nutzdaten einer Nachricht.

# 7 Entwurf und Implementierung des Konfigurationswerkzeugs

Dieses Kapitel beschäftigt sich mit der Entwicklung des Konfigurationswerkzeugs für den TTCAN-Treiber. Es gliedert sich in vier Abschnitte. Nach einer Beschreibung des grundsätzlichen Designs der Anwendung folgt ein genauerer Blick auf die einzelnen Phasen des Programmablaufs. Der dritte Abschnitt erklärt die Benutzung des Programms. Das Kapitel endet schließlich mit der Evaluierung der Implementierung hinsichtlich des Laufzeitverhaltens.

## 7.1 Designüberblick

Die Aufgabe des Programms besteht zunächst darin, die Erstellung der Ablaufpläne für das Versenden und Empfangen von Nachrichten im TTCAN-Netzwerk zu automatisieren. Weiterhin muss es Quelltext generieren, um jeden Knoten mit Hilfe der Treiberfunktionen entsprechend des erstellten Ablaufplans zu konfigurieren. Die Konfiguration beinhaltet vor allem die Allokation der benötigten Nachrichtenobjekte und die Programmierung des TTCAN-Schedulers. Der Treiber erwartet die Implementation dieser Aktionen in einer Funktion mit der Signatur `void ttCanLoadConfig()`. Als Eingabe benötigt das Konfigurationswerkzeug eine Beschreibung der zu verschickenden Nachrichten und deren Zuordnung zu den Knoten. Aus dieser Beschreibung muss zunächst ein globaler Matrixzyklus erstellt werden, aus dem dann für jeden Knoten ein lokaler Matrixzyklus erstellt werden kann, der nur die lokal relevanten Informationen enthält. Das Überprüfen des lokalen Matrixzyklus und des lokalen Task Ablaufplans von ProOSEK/Time auf Konflikte gehört nicht zum Funktionsumfang der Anwendung. Sie ist nur für den globalen Belang der Planung des Nachrichtenaustauschs zuständig. Der Benutzer ist selbst für die zeitlich korrekte Einplanung der Lese- und Schreibvorgänge auf Nachrichten verantwortlich. So müssen die Daten einer zu versendenden Nachricht vor dem im Kommunikationsablaufplan bestimmten Sendezeitpunkten aktualisiert werden. Genauso können empfangene Nachrichten erst nach den entsprechenden Empfangszeitpunkten neue Daten enthalten. Damit der Benutzer dazu in der Lage ist, muss das Konfigurationswerkzeug für jeden Knoten neben der Ausgabe des Quelltexts noch die Sende- und Empfangszeiten der Nachrichten des lokalen Matrixzyklus exportieren. Der Programmablauf gliedert sich also in vier Phasen:

1. Einlesen der Nachrichtenbeschreibung
2. Ablaufplanung und Erstellung des globalen Matrixzyklus
3. Für jeden Knoten: Generierung und Export des lokalen Matrixzyklus
4. Für jeden Knoten: Generierung des Quelltextes zur Initialisierung

## 7.2 Programmablauf

In diesem Abschnitt werden nun die einzelnen Phasen des Programmablaufs genauer betrachtet.

### 7.2.1 Einlesen der Nachrichten

In dieser Phase liest ein Parser die Nachrichtendefinitionen aus einer Datei ein. Das Dateiformat ist an OIL angelegt und hat folgenden Aufbau:

```
1 TUR      1-1023
2
3 Node <NAME>
4 {
5
6     Timemaster yes|no|true|false
7
8     Priority    0-7
9
10    Message <NAME>
11    {
12        Period      x
13        Deadline    y
14        Receiver    <NAME>,<NAME>,...
15        Length      1-8
16    }
17
18    Message <NAME>
19    {
20        ...
21    }
22
23    ...
24 }
25
26 Node <NAME>
27 {
28     ...
29 }
30
31 ...
```

*TUR* ist das initiale Verhältnis von lokaler zu globaler Zeit. Die globale Zeit wird dabei vom Timemaster mit der höchsten Priorität vorgegeben. Der Wert bestimmt damit, die Länge einer *NTU* (Network Time Unit). Bedingt durch die Hardware darf *TUR* nur diskrete Werte im Bereich  $[\frac{1}{1023}, \frac{1}{1}]$  annehmen. Deswegen muss hier der reziproke Wert im Intervall  $[1, 1023]$  angegeben werden. Da die Zeitgeber aller Knoten vom Treiber gleich initialisiert

werden, kann dieser Wert global angegeben werden. Die Langer einer NTU berechnet sich somit aus:

$$NTU = t_q * TUR \text{ ns}$$

Die Lange des Zeitquants  $t_q$  ist auf Treiberseite fest auch 66.6 ns eingestellt. Ein hoherer NTU Wert ermoglicht langere Basiszykluszeiten, verringert jedoch das zeitliche Auflosungsvermogen. Die Angabe des TUR Wertes ist optional.

<NAME> sind immer eindeutige Bezeichner fur Knoten und Nachrichten. Die Prioritat eines Knotens darf nur angegeben werden, wenn der Knoten ein potenzieller Timemaster ist. Je kleiner der Wert umso hoher ist die Prioritat eines Timemasters. Die Werte fur Periode und Termin einer Nachricht sind in Nanosekunden anzugeben. Zu beachten ist hier, dass diese Werte vom Konfigurationswerkzeug in NTU Einheiten umgerechnet werden mussen. Um Rundungsfehler bei dieser Umrechnung zu vermeiden, muss sowohl die Periode als auch der Termin ein Vielfaches einer NTU sein. Ist dies nicht der Fall, gibt das Programm eine entsprechende Warnung aus. Ein Rundungsfehler bei dieser Umrechnung hatte zur Folge, dass die Hyperperiode nicht exakt berechnet werden kann.

Der Parser erstellt aus diesen Informationen eine Liste aller Nachrichten.

## 7.2.2 Ablaufplanung

In dieser Phase wird versucht, einen globalen Matrixzyklus zu erstellen, in dem alle eingelesenen Nachrichten termingerecht gesendet werden. Hier mussen also zwei Probleme behandelt werden:

- die Festlegung der Reihenfolge der Nachrichten
- die Reihenfolge darf die Bedingungen an einen gultigen Matrixzyklus hinsichtlich der Spaltenbreiten und Anzahl der Basiszyklen nicht verletzen.

Der Algorithmus, der diese zwei ineinander greifenden Probleme versucht zu losen hat vereinfacht dargestellt folgenden Aufbau:

Listing 7.1: Algorithmus der Ablaufplanung

```
1 foreach(Basic cycle)
2 {
3     while (time left in current cycle)
4     {
5         GetArrivedMessages();
6
7         while (current time window is empty)
8         {
9             if (there are messages available)
10            {
11                Message = Scheduler.NextMessage();
12
13                if (Message fits in current time window)
14                    Insert(Message);
15                else
16                    Reject(Message);
17            }
18            else
19                break;
20        }
21        TimeStep();
22        Unreject();
23        CheckDeadLines();
24
25        FreeTime = unused time after current window;
26        while (FreeTime)
27        {
28            GetArrivedMessages();
29
30            while (there are messages available)
31            {
32                Message = Scheduler.NextMessage();
33
34                if (Message fits in FreeTime)
35                {
36                    InsertColumnBetween(current and next
37                                         window);
38                    Insert(Message);
39                    break;
40                }
41                else
42                    Reject(Message);
43            }
44            TimeStep();
45            Unreject();
46            CheckDeadLines();
47        }
48    }
```

Es werden also alle Basiszyklen durchlaufen (Zeilen 1-3). Die Anzahl der Basiszyklen lässt sich errechnen aus:

$$\text{Anzahl der Basiszyklen} = \frac{\text{Hyperperiode}}{\text{maximale Zykluszeit}}$$

Die maximale Zykluszeit ist von der Hardware vorgegeben. Die Anzahl der Basiszyklen muss auf die nächste zweier Potenz aufgerundet werden, um dem TTCAN-Standard zu genügen. Die reale Zykluszeit ist damit:

$$\text{Zykluszeit} = \frac{\text{Hyperperiode}}{\text{gerundete Anzahl der Basiszyklen}}$$

Auch hier kann es wieder zu einem Rundungsfehler bei der Berechnung der Zykluszeit kommen, wenn die Hyperperiode nicht durch die benötigte zweier Potenz teilbar ist. Eine gerundete Zykluszeit würde jedoch bedeuten, dass sich die Phasen der Nachrichten beim wiederholten Durchlaufen des Matrixzyklus verschieben, da die Gesamtdauer eines Matrixzyklus dann nicht mehr der Hyperperiode entspricht. Das Konfigurationswerkzeug gibt in diesem Fall eine entsprechende Warnung aus.

Für jedes Fenster eines Basiszyklus wird zunächst versucht, in der Liste der sendebereiten Nachrichten (Zeile 5) eine Nachricht zu finden, die in diesem Fenster gesendet werden kann (Zeilen 6-20). Eine Nachricht ist dann sendebereit, wenn seit ihrer letzten Aktivierung mindestens ihre Periode abgelaufen ist. Ob eine Nachricht im aktuellen Fenster gesendet werden kann, hängt davon ab, ob das Zeitintervall, das durch das Fenster beschrieben wird groß genug ist, um alle Daten der Nachricht zu übertragen. Da der TTCAN-Standard verlangt, dass die Fenster einer Spalte des Matrixzyklus alle gleich breit sind, wird die Breite einer Spalte durch den ersten Eintrag einer Nachricht in ein Zeitfenster dieser Spalte festgelegt. Wenn eine Nachricht gesendet werden kann, wird sie in das aktuelle Zeitfenster eingefügt (Zeilen 13+14). Ansonsten muss sie bis zum Ende des Fensters zurückgestellt werden (Zeilen 15+16). Die Reihenfolge, in der die sendebereiten Nachrichten auf ihre Eignung hin untersucht werden, hängt von einer separat implementierten Auswahlstrategie ab (Zeile 11). Sollte keine Nachricht im aktuellen Zeitfenster gesendet werden können, muss es übersprungen werden (Zeilen 18+19). Unabhängig von der Zuweisung einer Nachricht wird anschließend an das Ende des aktuellen Zeitfensters gesprungen (Zeile 21) und alle zurückgestellten Nachrichten werden wieder verfügbar gemacht (Zeile 22). Nach jedem Zeitschritt müssen die Termine aller sendebereiten Nachrichten anhand des neu errechneten Zeitpunkts überprüft werden (Zeile 23). Sollte vor dem Ende des Matrixzyklus ein Termin überschritten worden sein, bricht der Algorithmus ab. Ein Termin ist spätestens dann überschritten, wenn die Zeitspanne zwischen dem aktuellen Zeitpunkt und dem Termin nicht mehr ausreicht, die Nachricht komplett zu übertragen. Im Gegensatz zur Anzahl der Basiszyklen, lässt sich die Anzahl der Spalten im Matrixzyklus nicht im voraus bestimmen. Zu Beginn besitzt der Matrixzyklus deswegen keine Spalten und somit sind auch noch keine Zeitfenster innerhalb der Basiszyklen definiert. Eine neue Spalte wird erst dann dynamisch eingefügt, wenn eine Nachricht zu einem Zeitpunkt versendet werden soll, der außerhalb der bereits definierten Zeitfenster liegt. Im Anschluss an das aktuelle Fenster muss daher die Existenz eines solchen ungenutzten Zeitintervalls überprüft werden (Zeile 25). Die Ausdehnung dieses Intervalls ist entweder durch den Beginn des nächsten Zeitfensters oder durch das Ende des Basiszyklus begrenzt. Zu jedem Zeitpunkt innerhalb des Intervalls (Zeile 27) muss nun überprüft werden, ob es in

der Liste der sendebereiten Nachrichten eine Nachricht gibt, die innerhalb der verbleibenden ungenutzten Zeit komplett übertragen werden kann (Zeilen 28-42). Ist eine passende Nachricht gefunden, wird eine neue Spalte in den Matrixzyklus eingefügt (Zeile 36) und das so entstandene neue Zeitfenster im aktuellen Basiszyklus mit der Nachricht belegt (Zeile 37). Alle anderen Nachrichten werden dagegen zurückgestellt (Zeilen 40+41). Auch hier werden nach jedem Zeitschritt (Zeile 43) die zurückgestellten Nachrichten wieder verfügbar gemacht (Zeile 44) und die Termine aller sendebereiten Nachrichten überprüft (Zeile 45). Ein alternativer Ansatz zur Behandlung der ungenutzten Zeitintervalle zwischen den Zeitfenstern wurde zugunsten des beschriebenen Vorgehens verworfen. Er sah vor, diese Zeitintervalle bereits im ersten Basiszyklus mit leeren Spalten der Breite 1 zu füllen und diese in den weiteren Basiszyklen bei Bedarf entsprechend der Länge der Nachricht zusammenzufügen. Dies hätte den Quelltext zum Teil stark vereinfacht, die Komplexität der Einfüge- und Zusammenfügeoperationen ist jedoch so hoch, dass aufgrund der dann stark ansteigenden Zahl der Aufrufe dieser Operationen die Performance des Algorithmus überdurchschnittlich gelitten hätte.

Während die Einfügeoperation in den Matrixzyklus immer nach diesem Muster abläuft, ist die Auswahlstrategie der nächsten Nachricht im Prinzip beliebig. Entsprechend wurde diese Phase so implementiert, dass der Auswahlalgorithmus austauschbar ist. Es können also leicht neue Strategien hinzugefügt werden. Die implementierte Standardstrategie verfolgt den *Earliest Deadline First* Ansatz. Es wird also immer die Nachricht ausgewählt, deren Termin am nächsten am Entscheidungszeitpunkt liegt. Die Einreihung der Nachrichten erfolgt dabei nicht präemptiv, wodurch der Algorithmus eventuell eigentlich lösbare Ablaufpläne zurückweist. Abbildung 7.1 zeigt so einen Fall anhand eines Beispiels mit drei Nachrichten N1 - N3. Zu sehen ist eine Zeitachse, auf der die Aktivitätsphasen der Nachrichten farblich dargestellt sind und darüber die Bereitzeiten und Termine sowie die Sendezeiten der Nachrichten in Balkenform. In Abbildung 7.2 ist das gleiche Beispiel mit Verdrängung skizziert. Hier kann Nachricht 2 gesendet werden, sobald sie bereit ist. Nachricht 3 hat danach immer noch genügend Zeit übrig, um den Termin einhalten zu können.

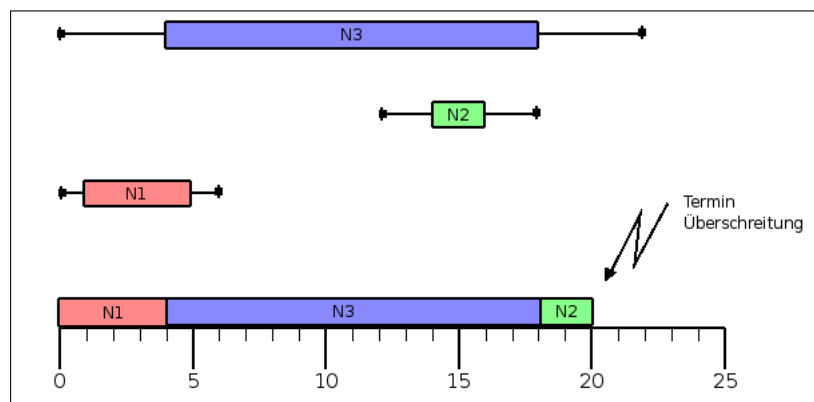


Abbildung 7.1: Nicht präemptiver EDF

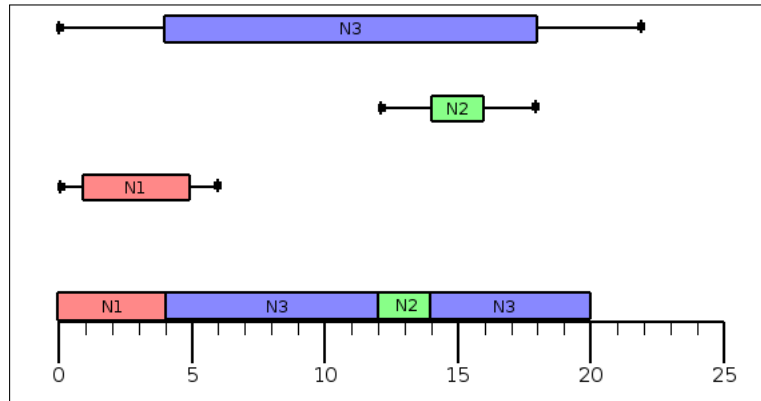


Abbildung 7.2: Präemptiver EDF

### 7.2.3 Generierung und Export des lokalen Matrixzyklus

Die Aufgabe dieser Phase ist es, für jeden Knoten aus dem globalen Matrixzyklus die Zeitfenster herauszusuchen, in denen der Knoten Nachrichten senden oder empfangen soll. Das Ergebnis ist ein Matrixzyklus, der nur die lokal notwendigen Informationen enthält. Die Sende- und Empfangszeiten der Nachrichten im lokalen Matrixzyklus werden anschließend für jeden Knoten in der Datei `local.txt` abgelegt.

### 7.2.4 Ausgabe des Quelltextes

Danach kann mit der Generierung des Quelltextes begonnen werden. Es müssen für jeden Knoten die Nachrichtenobjekte für jede Nachricht allokiert und konfiguriert und der TTCAN-Scheduler mit den Informationen aus dem lokalen Matrixzyklus programmiert werden. Ferner müssen die Speicheradressen der Nutzdaten der einzelnen Nachrichten der Anwendung zur Verfügung gestellt werden, damit neue Daten leicht gelesen oder geschrieben werden können. Die Ausgabe dieser Phase sind schließlich zwei Dateien für jeden Knoten:

**node\_config.c** enthält die Implementierung der vom Treiber erwarteten Funktion `ttCanLoadConfig()`

**ttcan\_messages.h** enthält die Zeiger auf die Speicheradressen der Nutzdaten

## 7.3 Verwendung

Das Programm arbeitet nur auf einer definierten Verzeichnisstruktur. Für jeden Knoten muss es ein Unterverzeichnis geben, das den in der Nachrichtendefinition angegebenen Knotennamen trägt. Jedes dieser Verzeichnisse besitzt wiederum zwei Unterverzeichnisse `src` und `include`. Diese Struktur ist nötig, damit das Programm entscheiden kann, wohin es die generierten Dateien für jeden Knoten schreiben muss.

Um das Erstellen dieser Arbeitsumgebung zu vereinfachen, existiert ein Hilfsprogramm, das diese Aufgabe übernimmt. Es erstellt außerdem noch für jeden Knoten das benötigte

Makefile, die Datei `main.c` mit dem Grundgerüst einer jeden ProOSEK/Time Anwendung, eine Header Datei `ttcan.h`, die die Einbindung wichtiger Treiber Dateien kapselt, und eine globale, leere Datei `message_config.txt` für die Nachrichtendefinitionen.

Angenommen, das Wurzelverzeichnis einer neuen Anwendung ist `ttcan/app` und die Quelltexte des Konfigurationswerkzeugs und des Treibers befinden sich in `ttcan/tool` respektive `ttcan/driver`, dann umfasst das Erstellen der Anwendung folgende Schritte:

1. Erstellen der Arbeitsumgebung:

```
$ ttcan/app> ../tool/init.sh NAME1 NAME2 NAME3
```

Wobei `NAME*` eine beliebig lange Liste von Knotennamen ist.

2. Anpassung der Variable `CAN_DRIVER_DIR` im Makefile und Erstellen der Nachrichtendefinitionen in `message_config.txt`. Zu beachten ist dabei, dass die Knotennamen in der Nachrichtendefinition identisch mit den beim Aufruf des Hilfsprogramms angegebenen sind.
3. Aufruf des Konfigurationswerkzeugs:

```
$ ttcan/app> ../tool/ttcan_config message_config.txt .
```

4. Programmieren der eigentlichen Anwendung und Erstellung der zugehörigen ProOSEK/Time Konfiguration mit Hilfe des ProOSEK/Time Konfigurationswerkzeugs. Auf die Daten der Nachrichten kann in der Anwendung zugegriffen werden, indem man `ttcan.h` einbindet. Es existiert zu jeder Nachricht ein Zeiger vom Typ `volatile void*`, der den in der Nachrichtendefinition angegebenen Namen trägt. Zu beachten ist, dass nach dem Schreiben an eine solche Adresse die Funktion `ttCanSetTransmitTrigger(volatile void*)` aufgerufen werden muss.

## 7.4 Evaluierung

Um das Laufzeitverhalten des Konfigurationswerkzeugs zu bestimmen, werden in einer Reihe von Testdurchläufen für jede der vier Ausführungsphasen die Laufzeiten gemessen. Die Anzahl der Nachrichten wird nach jedem Durchlauf erhöht. Da jeder Knoten nur eine durch die Hardware begrenzte Anzahl von Nachrichten verwalten kann, geschieht dies indirekt durch Erhöhung der Knotenanzahl. Die Anzahl der Nachrichten je Knoten bleibt jedoch konstant. Damit die Ergebnisse der Testdurchläufe vergleichbar sind, müssen die Attribute der Nachrichten festgelegten Regeln folgen. Nur so lässt sich sicherstellen, dass die Komplexität nur von der Anzahl der Nachrichten und Knoten abhängt. Für diesen Test wurden folgende Regeln festgelegt:

1. Alle Nachrichten haben die gleiche Periode
2. Alle Nachrichten haben die gleiche Länge
3. Der Termin von Nachricht  $n + 1$  ist um einen festen Wert höher als der Termin von Nachricht  $n$
4. Der Knoten  $n$  sendet seine Nachrichten an den Knoten  $n + 1$

Um Schwankungen der Messung bedingt durch die übrige Systemlast zu kompensieren, wird jeder Durchlauf mehrmals wiederholt und die gemessenen Zeiten gemittelt. Für die Ausführung der Testdurchläufe wurde das Konfigurationswerkzeug außerdem ohne Compileroptimierungen erstellt. Tabelle 7.1 zeigt die Ergebnisse der Messungen. Die Laufzeiten sind dabei in Millisekunden angegeben.

Nachrichten	Phase 1	Phase 2	Phase 3	Phase 4	Gesamt
40	1	12043	5	28	12078
80	2	19678	37	52	19770
160	4	30295	279	103	30683
240	7	36099	907	155	37169
320	9	31141	2100	207	33459
480	15	4751	6862	311	11941
640	19	5231	9173	419	14843
960	32	6373	13756	626	20789
1280	50	8324	18406	843	27624

Tabelle 7.1: Laufzeiten der Phasen des Konfigurationswerkzeugs

Wie man erkennen kann, werden die Gesamtzeiten vor allem durch die zweite Phase beeinflusst. In dieser Phase wird der globale Matrixzyklus erstellt. Interessant zu beobachten ist hier die Unstetigkeit der Ausführungszeiten. Bei etwa 480 Nachrichten fallen diese sprunghaft ab. Dies ist leicht anhand des Quelltextausschnitts 7.1 zu erklären. Ab diesem Punkt ist der erste Basiszyklus komplett gefüllt. Für die folgenden Zyklen entfällt daher die Ablaufplanung innerhalb der freien Zeiteinheiten in den Zeilen 25 bis 46. Die Breite und Anzahl der Zeitschritte ist dann nur noch von den Zeitfenster im ersten Basiszyklus abhängig, so dass der weitere Aufwand im Wesentlichen von der Komplexität des verwendeten Algorithmus zur Auswahl der nächsten Nachricht beeinflusst wird. Die Ausführungszeiten der dritten Phase steigen stetig. In dieser Phase werden die lokalen Matrixzyklen erstellt. In jedem Testdurchlauf erhöht sich die Zahl der Knoten, es müssen also auch entsprechend mehr Matrixzyklen erstellt werden. Die Laufzeiten der ersten und vierten Phase werden im wesentlichen durch die Ein-/Ausgabeoperationen bestimmt und sind vergleichsweise gering.

Wie Abbildung 7.3 zeigt, verhalten sich die Gesamtlaufzeiten des Konfigurationswerkzeugs in den Intervallen  $]0, 240]$  und  $[480, 1280]$  annähernd linear. Eine Ausnahme bildet lediglich der Übergang bei 320 Nachrichten. Dieses zeitliche Verhalten bei der Lösung eines eigentlich NP-Vollständigen Problems wird dadurch erkauft, dass der Algorithmus der Ablaufplanung nicht garantieren kann, dass eine Lösung gefunden wird, sollte sie existieren. Das liegt zum einen an der nicht präemptiven Auswahlstrategie. Zum anderen an der Unveränderbarkeit der Spaltenbreiten im Matrixzyklus. So kann es Fälle geben, in denen eine Nachricht nicht innerhalb eines leeren Zeitfensters versendet werden kann, obwohl die Zeit bis zum nächsten Zeitfenster ausreichen würde. Eine Optimierung der Spaltenbreiten könnte dies verhindern. Hierzu müsste jedoch der gesamte Ablaufplan betrachtet werden, um durch die Vergrößerung eines Zeitfensters im späteren Verlauf des Matrixzyklus keine weiteren Terminüberschreitungen zu provozieren. Zwei weitere, weniger kritische Einschränkungen betreffen die übertragbaren Nachrichten. So können nur Nachrichten mit einer Länge zwischen einem und acht Bytes versendet werden. Um größere Nachrichten zu versenden, müsste das Konfigurationswerkzeug in der Lage sein, diese auf mehrere Nachrichtenobjekte und Zeitfenster abzubilden. Der Nachrichtenaustausch könnte außer-

dem optimiert werden, wenn Nachrichten, die weniger als acht Byte beanspruchen und deren Auslösezeiten und Termine sich überlappen, in einem Nachrichtenobjekt zusammengefasst würden anstatt ein Nachrichtenobjekt pro Nachricht zu verwenden. Sowohl die Unterstützung größerer Nachrichten als auch eine präemptive Ablaufplanung hätten auch Änderungen im Treiber zur Folge. Hier müsste dann vom Lesen und Schreiben der Daten abstrahiert werden. Ein einfaches Exportieren der Zeiger auf die Nutzdaten der Nachrichtenobjekte ist dann nicht mehr ausreichend, da eine Nachricht auf mehrere Nachrichtenobjekte verteilt sein kann. Der Speicher der verschiedenen Nachrichtenobjekte hängt jedoch nicht linear zusammen.

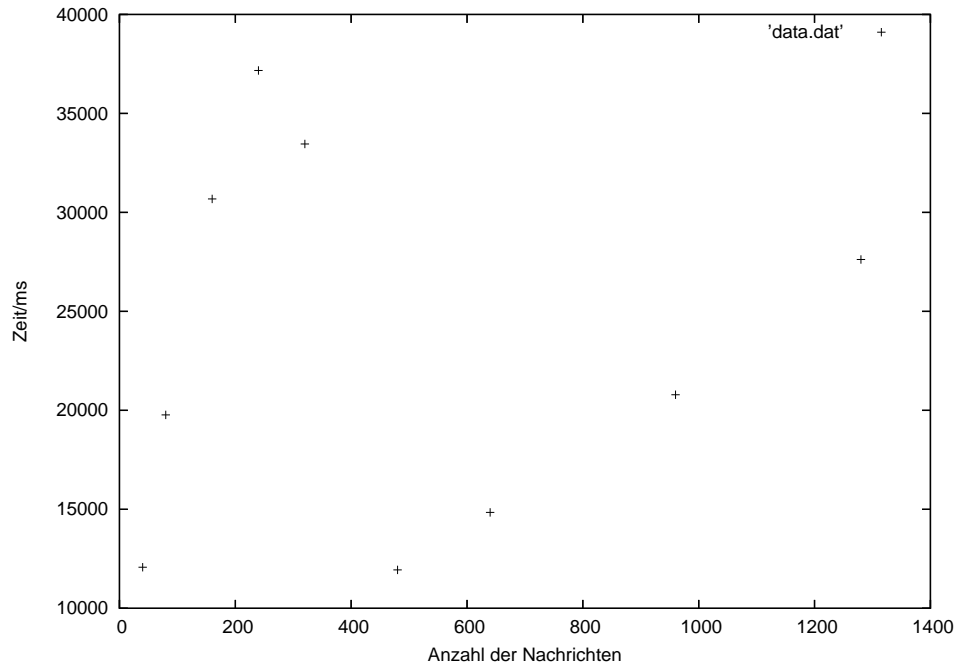


Abbildung 7.3: Gesamtlaufzeiten des Konfigurationswerkzeugs

## 8 Zusammenfassung

In dieser Arbeit wurden zunächst die zur Entwicklung des Treibers und des Konfigurationswerkzeugs notwendigen Grundlagen behandelt. Aufbauend auf einer allgemeinen Beschreibung zeitgesteuerter Kommunikation sowie Techniken zur statischen Ablaufplanung in zeitgesteuerten Systemen und einer Einführung in das CAN-Protokoll, wurde das TTCAN-Protokoll und dessen konkrete Implementierung auf dem Tricore TC1796 erläutert. Ein Überblick über das zu verwendende Betriebssystem ProOSEK/Time beendete schließlich die Grundlagenkapitel. Die beiden darauf folgenden Kapitel widmeten sich dem Entwurf und der Implementierung des TTCAN-Treibers und des dazugehörigen Konfigurationswerkzeugs. Diese beiden Kapitel stellen den praktischen Teil dieser Arbeit da und ihre Ergebnisse sollen deswegen hier noch einmal zusammenfassend bewertet werden. Obwohl sich die Benutzung des Treibers und des Konfigurationswerkzeugs aufgrund der in den vorangegangenen Kapiteln bereits diskutierten Einschränkungen in einigen Punkten noch als etwas unflexibel erweist, wurden doch im Wesentlichen alle in der Einleitung formulierten Ziele erreicht. Die Einschränkungen betreffen in erster Linie die nicht präemptive Ablaufplanung und die festen Spaltenbreiten im Matrixzyklus, aber auch die Begrenzung der Länge einer Nachricht auf acht Byte sowie die ineffiziente Nutzung der Nachrichtenobjekte bei vielen kleineren Nachrichten. Abgesehen davon unterstützt der TTCAN-Treiber jedoch alle Funktionen der TTCAN-Schnittstelle des TC1796, die für den Nachrichtenaustausch innerhalb eines TTCAN-Netzwerks notwendig sind. Dabei besteht die Schnittstelle des Treibers auf Benutzerseite lediglich aus vier Funktionen: `ttCanStart()` zur Hardwareinitialisierung, `ttCanSyncStart()` um den Start der ProOSEK/Time Ablaufabelle mit dem Matrixzyklus zu synchronisieren, `ttCanSyncTimes()` zur Synchronisierung der globalen mit lokalen Systemzeit in ProOSEK/Time und `ttCanSetTransmitTrigger()` nach jeder Aktualisierung eines Echtzeitdatums. Das Konfigurationswerkzeug erzeugt aus einer Konfigurationsdatei mit den Definitionen der Knoten und Nachrichten zunächst einen globalen Ablaufplan zur Nachrichtenübertragung, um daraus im Anschluss für jeden Knoten einen lokalen Ablaufplan zu extrahieren. Aus den Informationen des lokalen Ablaufplans erzeugt es ferner den Quelltext zur Initialisierung der TTCAN-Hardware und zur Allokation der Nachrichtenobjekte, der innerhalb der Treiberfunktion `ttCanStart()` ausgeführt wird.

# Literaturverzeichnis

- [1] [http://www.port.de/pdf/CAN\\_Bit\\_Timing.pdf](http://www.port.de/pdf/CAN_Bit_Timing.pdf) Florian Hartwich, Armin Bassemir (Robert Bosch GmbH, Abt. K8/EIS, Tübinger Straße 123, 72762 Reutlingen): The Configuration of the CAN Bit Timing (6th International CAN Conference 2nd to 4th November, Turin (Italy) )
- [2] Infineon Technologies AG, St.-Martin-Strasse 53, 81669 München, Germany: TC1796 Users Manual (V1.0, Juni 2005)
- [3] Thomas Führer, Bernd Müller, Werner Dieterle, Florian Hartwich, Robert Hugel, Michael Walther (Robert Bosch GmbH): Time Triggered Communication on CAN (Time Triggered CAN - TTCAN)
- [4] 3SOFT GmbH Frauenweiherstr. 14, 91058 Erlangen, Germany: ProOSEK/Time User Guide (Rev. 1.8)
- [5] Hermann Kopetz, Technical University of Vienna, Gunter Grunsteidl, Alcatel Austria Research Center: TTP - A Protocol for Fault-Tolerant Real-Time Systems
- [6] Radu Dobrin and Gerhard Fohler, Department of Computer Engineering, Malardalen University, Sweden: Implementing Off-line Message Scheduling on Controller Area Network (CAN)
- [7] Khawar M. Zuberi, Member, IEEE Computer Society, Kang G. Shin, Fellow, IEEE: Design and Implementation of Efficient Message Scheduling for Controller Area Network
- [8] Antonio Meschi, Marco Di Natale (Scuola Superiore S.Anna, Via Carducci, 40 - 56100 Pisa - Italy), Marco Spuri (Projet REFLECS - INMA, 78153 Le Chesnay Cedex - France): Earliest Deadline Message Scheduling with Limited Priority Inversion
- [9] Margaret Naughton and Donal Heffernan, Centre for Telecommunication & Value-chain Driven Research (CTVR), Engineering Research Building, University of Limerick, Limerick: SMART-Plan: A New Real-time Message Scheduling Tool for Control Networks
- [10] Jose Fonseca (DET / IEETA - Universidade de Aveiro, Portugal), Fernanda Coutinho, Jorge Barreiros (Instituto Superior de Engenharia de Coimbra, Portugal): Scheduling for a TTCAN network with a stochastic optimization algorithm
- [11] Hermann Kopetz, Günther Bauer: The Time-Triggered Architecture
- [12] Hermann Kopetz, Roman Nossal, Institut für Technische Informatik, Technical University of Vienna, Treitlstr. 3/182/1, A-1040 Vienna, Austria: The Cluster Compiler - A Tool for the Design of Time-Triggered Real-Time Systems
- [13] Paul Pop, Petru Eles, and Zebo Peng, Dept. of Computer and Information Science, Linköping University, Sweden: An Improved Scheduling Technique for Time-Triggered Embedded Systems

- [14] Wolfgang Schröder-Preikschat, Friedrich-Alexander-Universität Erlangen-Nürnberg, Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme): Grundlagen von Echtzeitbetriebssystemen
- [15] [http://de.wikipedia.org/wiki/Controller\\_Area\\_Network](http://de.wikipedia.org/wiki/Controller_Area_Network)