Institut für Computertechnik

Mikrocomputer – UE 384.056

Skriptum zum Übungsteil

TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

Mikrocomputer Übungsteil
Skriptum

Autor: Dipl.-Ing. Martin Horauer
Anpassungen 2002/03: Dipl.-Ing. Martin Jandl

Dieses Übungsskriptum wurde unter Suse Linux 6.3 mit LATEX gesetzt.

Bezugsquelle: Institut für Computertechnik E384, Gußhausstraße 27-29, A-1040Wien
oder über http://mc.ict.tuwien.ac.at.

Kopieren, Verbreiten und Modifizieren des vorliegenden Skriptums ist erlaubt sofern das entsprechende Copyright/Copyleft beachtet wird. Auch wenn eine Menge Fehler beseitigt werden konnten, möchte ich den Eindruck vermeiden, es handle sich jetzt um ein fertiges Produkt. Die dicken Fehler sind nur besser versteckt ;-). Vielleicht habe ich ja die Chance es beim nächsten Mal besser zu machen.

Fehlerhinweise und Kommentare bitte an Dipl.-Ing. Martin Jandl (jandl@ict.tuwien.ac.at).

Bitte beachten Sie den Foliensatz sowie die Beiblätter, die ebenfalls von der MC-Webseite heruntergeladen werden können. Die Folien sind Grundlage der Übungsstunden. Die Beiblätter dienen als Nachschlagewerk für Beispiele (Übungstest!).

Vorwort zur zweiten, überarbeiteten Auflage

Die Vorlesung Mikrocomputer soll einen Einblick in heute übliche Systeme der Mikroprozessortechnik geben. Nach einem kurzen geschichtlichen Abriss wird der Aufbau von Mikroprozessoren. In weiterer Folge werden dann die verschiedensten Speichertechnologien behandelt bevor die Vorlesung mit einem Auszug über das Interfacing abgeschlossen wird. In der zugehörigen Übung soll dieser Einblick anhand einiger ausgewählter Themenbereiche vertieft und intensiviert werden. Im folgenden wird der C167, ein Vertreter der C166 Familie von Infineon, näher erläutert. Dieser Controller ist aufgrund seiner Realisierung für eine derartige Übung gut geeignet, da er viel Peripherie neben einer Recheneinheit mit einem Speicher integriert. Somit kann eine zum Teil oft recht aufwendige Betrachtung des Interfacings zu verschiedensten anderen Komponenten entfallen, da mit dem Controller alleine schon sehr viele praktische Aufgaben sehr elegant gelöst werden können.
In diesem Skriptum werden die einzelnen Komponenten dieses Controller kurz erläutert und jeweils im Anschluß mittels eines kurzen Programmier-Beispiels im praktischen Einsatz gezeigt. Für ein vollständiges Verständnis muß im Allgemeinen aber jeweils zusätzlich das entsprechende Datenbuch mit herangezogen werden. Die entsprechenden Dateien, Demosoftware mit Simulatoren, weiterführende Beispiele und diverse Hinweise sind über den WWW Server des Instituts unter

http://mc.ict.tuwien.ac.at

abrufbar.

Diese Übung begleitet somit den Vorlesungsinhalt nur sehr bedingt. Vielmehr soll das in der Vorlesung vermittelte Allgemeinwissen mit Detailkenntnissen zu konkreten Realisationen erweitert werden. Die in der Übung vermittelten Kenntnisse dienen als Vorbereitung für weitere Spezialvorlesungen, Labors, Praktika und Diplomarbeiten. Die folgenden Lehrveranstaltungen werden begleitend oder im Anschluß zur Übung empfohlen.

- Mikrocomputer UE Laborteil (findet direkt im Anschluß an den Übungsteil statt)
- Hochintegrierte Bauelemente - 384.000 - VO

- Datenkommunikation und Netzwerke - 384.560 - VO
- Datensicherung - 384.015 - VO
- Signalprozessoren - 382.646 – VO

Allgemeine Kenntnisse der Digitaltechnik werden vorausgesetzt. Weiters sind Kenntnisse der Programmiersprache C vorteilhaft. Hierfür wird der Besuch der Lehrveranstaltung

- Programmieren 1 - 360.016 - VU (Institut für Mikroelektronik)

parallel zur Übung empfohlen.

Wien, 2002

Martin Jandl

Notationen

xxx#: Das Signal *xxx* wird mit der nachgestellten Raute als low aktiv gekennzeichnet.

0x*1234*: Der Prefix “0x” gibt an, daß die Zahl “1234” in hexadezimaler Notation verstanden wird.

10**b**: Der Sufix “b” gibt an, daß die Zahl “10” in binärer Notation verstanden wird.

10**d**: Der Sufix “d” gibt an, daß die Zahl “10” in dezimaler Notation verstanden wird.

Abkürzungen

μC	**Micro C**ontroller
μP	**Micro P**rocessor
ADC	**A**nalog **D**igital **C**onverter
AGP	**A**ccelerated **G**raphics **P**ort
ASC	**A**synchronous **S**erial **C**hannel
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
ATM	**A**synchronous **T**ransfer **M**ode
CAN	**C**ontroller **A**rea **N**etwork
CAPCOM	**Cap**ture **Com**pare Unit
CISC	**C**omplex **I**nstruction **S**et **C**omputer
CPLD	**C**omplex **P**rogrammable **L**ogic **D**evice
CPU	**C**entral **P**rocessing **U**nit
CRC	**C**yclic **R**edundant **C**ode
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
DSP	**D**igital **S**ignal **P**rocessor
EBC	**E**xternal **B**us **C**ontroller
EIDE	**E**nhanced **I**ntegrated **D**rive **E**lectronics
EPP	**E**nhanced **P**arallel **P**ort
FDDI	**F**iber **D**istributed **D**ata **I**nterface
FIFO	**F**irst-**I**n **F**irst-**O**ut Memory
GPT	**G**eneral **P**urpose **T**imer
IDE	**I**ntegrated **D**rive **E**lectronics
ISA	**I**ndustry **S**tandard **A**rchitecture
ISR	**I**nterrupt **S**ervice **R**outine
IRQ	**I**nterrupt **Req**uest
PEC	**P**eripheral **E**vent **C**ontroller
PCI	**P**eripheral **C**omponent **I**nterconnect
PLL	**P**hase **L**ocked **L**oop
PWM	**P**ulse **W**idth **M**odulation
RISC	**R**educed **I**nstruction **S**et **C**omputer
RTOS	**R**eal **T**ime **O**perating **S**ystem

SCSI	**S**mall **C**omputer **S**ystem **I**nterface
SRAM	**S**tatic **R**andom **A**ccess **M**emory
SPI	**S**erial **P**eriheral **I**nterface
SSC	**S**ynchronous **S**erial **C**hannel
TCB	**T**ask **C**ontrol **B**lock
TS	**T**ri**s**tate
USART	**U**niversal **S**ynchronous **A**synchronous **R**eceiver **T**ransmitter
USB	**U**niversal **S**erial **B**us
VME	**V**ersa **M**odule **E**urocard

Inhaltsverzeichnis

1 Mikrocomputer Architektur / Mikroprozessortechnik

Der Themenbereich Mikrocomputer Architektur und Mikroprozessortechnik umfaßt im weitesten Sinne die digitale Verarbeitung von Informationen. Hierzu gehören u.a. Methoden zur Informationsdarstellung, zur Verarbeitung von Information mit Mikroprozessoren, deren Speicherung, diverse Programmiertechniken sowie Systeme zur Ein- und Ausgabe. Vereinfacht ausgedrückt sind Mikroprozessorsysteme universell programmierbare Digitalrechner. Ihre Vorteile liegen in der Miniaturisierung der Komponenten, in der Möglichkeit die Hardware modular an die Problemstellung anzupassen und in der großen Flexibilität bedingt durch die Programmierbarkeit. Bedingt durch die fortschreitenden Technologien kann die Integrationsdichte immer stärker erhöht werden. Gleichzeitig bewirkt dies eine starke Verkürzung der Schaltzeiten, sodaß die Bausteine mit immer höheren Taktfrequenzen arbeiten können. Integrationsdichten von 5 Millionen Transistoren auf $100mm^2$ Siliziumfläche sowie Taktfrequenzen von einigen 100MHz sind heutzutage bereits Stand der Technik. Zudem wird auch immer mehr an verschiedener Funktionalität auf einer Siliziumscheibe integriert. So vereint z.B. der net+arm Mikroprozessor eine herkömmliche CPU mit einem Netzwerkcontroller und diversen Interface Blöcken auf einem IC. Ein weiteres Beispiel ist die Microcontroller Familie TriCore, die DSP Funktionalität mit den Eigenschaften und Peripherie Modulen, die bis dato in erster Linie für Mikrocontroller kennzeichnend waren, integriert. Dahingehend wird der Begriff “System on Silicon” immer stärker zur Realität. Weiters besteht zum Teil auch ein starker Trend zur Spezialisierung indem für spezielle Anwendungen ein eigener ASIC entwickelt wird. Dies ist unter anderem dann zielführend wenn niedrigere Taktgeschwindigkeit und geringere Leistungsaufnahme erwünscht sind, oder eine hohe Stückzahl produziert wird.
Um die Marktrelevanz des Gebietes der Microcomputer ein wenig zu verdeutlichen sind in Tabelle 1, die Umsatzdaten der Top Drei Unternehmen aus dem Jahre 1997/8[a] in Millionen USD für vier unterschiedliche Bereiche wiedergegeben.

Nr	Firma	1997	1998	Nr	Firma	1997	1998
Mikroprozessoren				DRAM			
1	Intel	19.158	20.161	1	Samsung	3.700	3.350
2	AMD	801	1.291	2	Hyundai	1.930	1.480
3	Motorola	985	977	3	Micron	1.650	1.300
Mikrocontroller				Signalprozessoren			
1	Motorola	2.330	1.955	1	Texas Instruments	1.440	1.650
2	Texas Instruments	1.400	1.875	2	Lucent Techn.	885	983
3	Hitachi	1.763	1.535	3	Motorola	375	447

Tabelle 1: Umsatzdaten der Top Unternehmen

---

[a] Quelle: http://www.ebnews.com/topsemi99

1.1 Komponenten und Klassifizierungen

Der rasche technologische Fortschritt bewirkt, daß heutzutage eine unüberschaubar große Vielfalt an Komponenten auf dem freien Markt erhältlich ist. In Abbildung 1 ist beispielsweise ein sehr grober Überblick über einen kleinen Teil verfügbar Mikroprozessorfamilien gegeben. Die Klassifizierung ist hierbei nach freiem Ermessen basierend auf Erfahrungswerten vorgenommen worden und eher trendmäßig anzusehen. Weiterführende Übersichten und Hinweise zu verfügbaren Bauteilen sind über die WWW Seite zur Übung unter http://mc.ict.tuwien.ac.at abrufbar, bzw. aus der entsprechenden Sekundärliteratur [FL94], [HP94], [Tan90], [Sch98], [Sch99] und [Tab95] zu entnehmen.

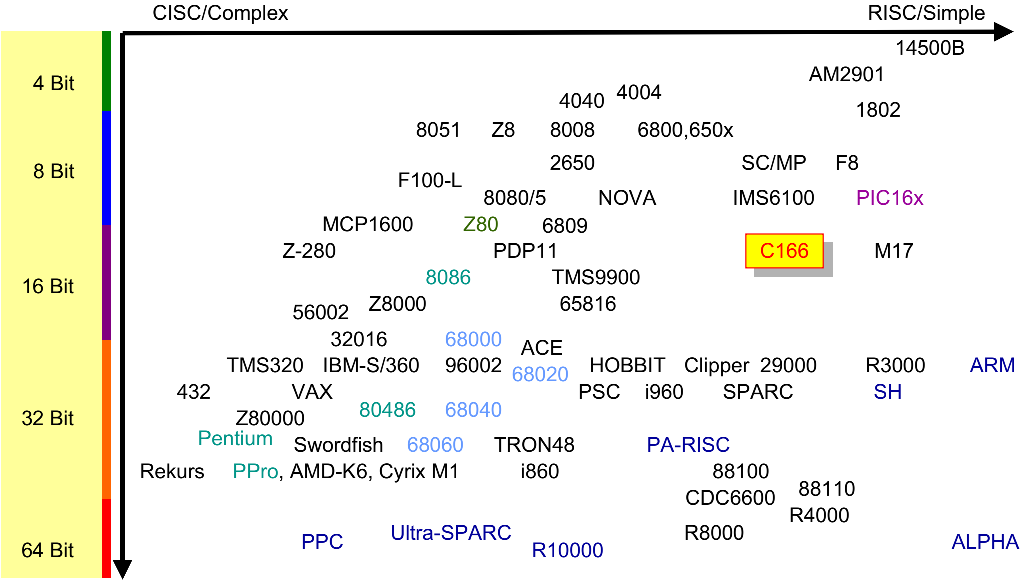


Abbildung 1: Überblick über Prozessorfamilien

Zu den Systemen für die Ein- und Ausgabe zählen u.a.:

- Displays, Monitore, LCD-Panels, Projektoren, Lautsprecher, Tastaturen, Zeigergeräte, etc.

Weiters gibt es zahlreiche Medien zur Speicherung von Daten und Informationen:

- Harddisks, Floppys, Speicher-IC´s (SRAM, DRAM, FIFOs, Cache Speicher, ...), Magneto-Optische Speichermedien, Compact Disks, Bandlaufwerke, etc.

Um diese Systeme ansteuern zu können ist zumeist ein entsprechendes Interface erforderlich:

- Serielle und parallele Controller (USART, USB, SPI, EPP, ...), Schnittstellen zu Massenspeichern (SCSI, EIDE, ...), Busanbindungen (AGP, PCI, ISA, VME, IDE, EIDE, S-Bus, ...), Netzwerkanbindungen (ATM, FDDI, Ethernet, Feldbusse, Infrarotschnittstellen,...), etc.

1.2 Kriterien zur Produktselektion

Für den Ingenieur stellt sich in diesem Zusammenhang meist das Problem der Selektion und Auswahl eines entsprechenden Produktes für die jeweilige Aufgabe die es zu lösen gilt. Hierfür ist es i.a. nötig die einzelnen Produkte entsprechend klassifizieren, einordnen und bewerten zu können. In vielen Fällen wird diese Aufgabe zumeist dadurch erschwert, daß viele Produkte verschiedener Kategorien miteinander verschmelzen. Die neue Prozessorentwicklung der Firma Infineon mit der Bezeichnung TriCore vereint, wie bereits erwähnt, in einem Prozessor die Funktionalität eines Controllers, eines Signalprozessors und eines herkömmlichen Mikroprozessors. Im Bereich der Mikroprozessoren von Personal Computern wird weiters zunehmend 3D Funktionalität integriert um Grafikoperationen zu beschleunigen (z.B. MMX Technologie der Fa. Intel, 3DNow! von AMD, etc.). Versucht man verschiedene Produkte zu klassifizieren oder zu vergleichen so sollte man stets die Zielanwendung in den Vordergrund stellen und versuchen für diese die geeigneteste Lösung zu finden. Ein einfacher Vergleich mit Benchmarks macht keinen Sinn wenn man diesen nicht für eine konkrete Anwendung in Betracht zieht. Untersucht man z.B. Mikroprozessoren, so können diese grob klassifiziert werden nach ihrer Datenbusbreite (16 Bit, 32 Bit, ...), ihrem Anwendungsbereich (Controller, Signalprozessor, reiner Mikroprozessor, ...) oder auch nach dem Umfang ihres Befehlssatzes (RISC, CISC, ...) und der zugrundeliegenden Architektur. Eine weitere, unvollständige Liste von zu berücksichtigenden Kriterien ist:

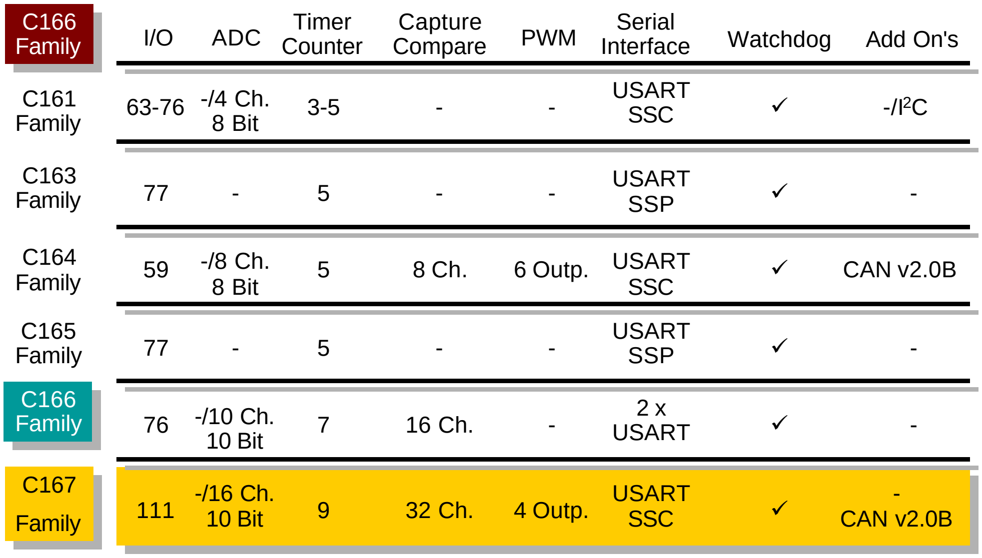
- Preis/Stück
- Stückzahl, Verfügbarkeit (2^{nd} Source, ...), Kompatibilität
- Platzbedarf, verfügbare Gehäusetypen, benötigte Peripherie, etc.
- Leistungsverbrauch (Stromversorgung, Wärmeabfuhr, ...)
- Entwicklungsumgebungen (Programmiersprachen, Simulatoren, Emulatoren, ...)
- Wartbarkeit, Modularität, Integrität, etc.

All diese Entscheidungskriterien sind von Fall zu Fall neu abzuwägen und entsprechend zu gewichten.

2 Die Mikrocontroller Familie C166

Die C166er Familie wurde von der Fa. Infineon (vormals Siemens) in erster Linie für zeitkritische Steuerungs- und Regelungsaufgaben entwickelt. Ausgestattet mit einem 16-Bit breiten Datenbus und zahlreichen Peripherieeinheiten stellt sie dem Anwender eine sehr leistungsfähige, integrierte Lösung zur Verfügung. Wie für einen Controller typisch besitzt sie ein ausgeprägtes Interrupt-System, zahlreiche konfigurierbare I/O Leitungen und diverse Timer. In Abbildung 2 wird ein Überblick über die einzelnen Familien der C166er Mikrocontroller gegeben. Wie daraus ersichtlich, sind sehr viele Mitglieder dieser Controller Familie zusätzlich mit integrierten Analog Digital Wandlern (ADC), Capture Compare Einheiten, einem Pulsweiten Modul (PWM) und seriellen

Schnittstellen ausgestattet. Die ADCs dienen zum Erfassen von analogen Meßgrößen und sind zumeist als sukzessiv approximierende Typen ausgeführt. Typische Wandlungszeiten bewegen sich hierbei im Bereich von $\geq 10\mu s$ bei Auflösungen von bis zu 10-Bit und 16 Kanälen. Mit Hilfe der Capture Compare Einheiten lassen sich sehr elegant digitale Signalpegel erfassen bzw. digitale Pulsmuster erzeugen. Die PWM Module sind in erster Linie zur Ansteuerung von Leistungsteilen für die Motorensteuerung und für Umrichter gedacht. Die synchronen und asynchronen Schnittstellen erlauben weiters ein sehr einfaches Interfacing zu Terminals und Host Computern. Einige Derivate der Familie sind mit weiteren speziellen Interfaces ausgestattet. Das I^2C Interface erlaubt eine sehr einfache Verbindung zu anderen Bauteilen und Baugruppen. Bemerkenswert aber ist vorallem auch, daß einige Controller mit einem Controller Area Network (CAN) Interface ausgestattet sind. Diese Module erlauben einen sehr einfachen Anschluß an den gleichnamigen Feldbus CAN, der als Feldbus für die Automobilindustrie entwickelt wurde. Durch die große Verfügbarkeit von Modulen und Komponenten hat sich dieser Feldbus mittlerweile aber auch in zahlreichen anderen Bereichen etabliert, sodaß man diesen Feldbus nun auch zur Vernetzung von z.B. Liftanlagen einsetzt. Ein starkes Verbreitungsgebiet ist allerdings nach wie vor der Automobilbereich.



C166 Family	I/O	ADC	Timer Counter	Capture Compare	PWM	Serial Interface	Watchdog	Add On's
C161 Family	63-76	-/4 Ch. 8 Bit	3-5	-	-	USART SSC	✓	-/I^2C
C163 Family	77	-	5	-	-	USART SSP	✓	-
C164 Family	59	-/8 Ch. 8 Bit	5	8 Ch.	6 Outp.	USART SSC	✓	CAN v2.0B
C165 Family	77	-	5	-	-	USART SSP	✓	-
C166 Family	76	-/10 Ch. 10 Bit	7	16 Ch.	-	2 x USART	✓	-
C167 Family	111	-/16 Ch. 10 Bit	9	32 Ch.	4 Outp.	USART SSC	✓	- CAN v2.0B

Abbildung 2: Übersicht über die C166 Mikrocontroller Familie

Diese 16-Bit Mikrocontroller Familie kommt mit nur sehr wenigen Maschinenbefehlen aus. Der C166 besitzt z.B. nur 78 Assembleranweisungen die in den meisten Fällen in nur einem Maschinenzyklus abgearbeitet werden. Aufgrund der sehr niedrigen Anzahl an Befehlen kann dieser Controller als RISC Typ mit einigen Erweiterungen eingestuft werden.

Die Speicherorganisation ist in einer *von Neumann Architektur* ausgeführt, d.h. Programme und Daten teilen sich gemeinsam einen linearen Adreßraum. Die einzelnen Peripherie-Komponenten sind hingegen recht modular aufgebaut und mit einem sehr aufwendigen Bussystem untereinander vernetzt. Dies erlaubt intern die parallele Durchführung von Transfers zwischen den Modulen und dem Rechenwerk.

Sehr bemerkenswert ist auch, daß sowohl Chip-Selects und Adreßbereiche als auch das Bustiming für den Anschluß externer Komponenten wie z.B. Speicherbausteinen programmiert werden können. Dadurch wird z.T. zusätzliche Dekodier- und Anpassungslogik beim Systemdesign eingespart.

Die folgende Aufzählung soll einen Einblick in mögliche Anwendungsgebiete geben:

- Verkehrswesen (Eisenbahnen, LKWs, ...), Automobilelektronik (Zündelektronik, Boardcomputer, etc.)
- Industrielle Automatisierung (Druckereimaschinen, Transporteinrichtungen, Motorsteuerungen, etc.)
- Medizintechnik (Röntgeninstrumente, Computertomographen, Zahnarztstühle, etc.)
- Gebäudeautomatisierung (Aufzüge, Liftanlagen, Lüftungsanlagen, etc.)
- Büroautomatisierung (Kopiermaschinen, Harddisk Controller für Festplatten, etc.)
- sonstige Anwendungen (digitale Videorecorder, etc.)

Marktpreise für das High-End Produkt, den C167, betrugen Mitte 1998 ca: 450 öS für 1 Stück bzw. für Abnahmemengen von ca. 100 Stk. 350 ÖS[b].

Konkurrenzprodukte zur C166er Familie sind unter anderem:

- H8/300 Familie von Hitachi
- 68HC12 und 68HC16 Familie von Motorola
- ST9 und ST10 Familie von SGS Thomson (die ST10 Familie ist baugleich zur C166er Familie)
- $\mu PD78k$ Familie von NEC
- MCS96 Familie von Intel

Die meisten dieser aufgezählten Produktfamilien sind ähnlich wie die C166er Familie ausgestattet. Jede der Familien weist zumeist einen ADC, serielle Schnittstellen, einen Watchdog Timer, zahlreiche General Purpose Timer sowie Module zum Erfassen und Ausgeben von Pulsmustern (vgl. PWM, CAPCOM Einheit).

Sicherlich bemerkenswert im Zusammenhang mit der C166er Familie ist die Tatsache, daß es bereits sehr zahlreiche und sehr ausgefeilte Entwicklungsumgebungen von verschiedenen Herstellern gibt. Neben Assembler, C-Compiler, Hardware Emulatoren und Software Simulatoren sind auch zahlreiche Echtzeitbetriebsysteme erhältlich. Weiters erleichtern die verschiedensten Starterkits den Einstieg in die Entwicklung von Applikationen mit der C166 Familie erheblich. Auf der WWW Seite zu dieser Übung http://mc.ict.tuwien.ac.at finden Sie weitere Hinweise und genauere Informationen zu den verfügbaren Produkten.

Im folgenden sollen ausgewählte Funktionen der C167 Familie im Detail näher erläutert werden. Die Erklärungen erfolgen hierbei stets im engen Zusammenhang mit dem Datenbuch [Si96a].

---

[b] Diese Preisangaben stammen aus einem Versandkatalog eines Distributors.

2.1 C167 Mikrocontroller Familie

Die C167er Familie fußt auf der Basisarchitektur der C166 Familie die um 1990 von der Firma Siemens in München neu entwickelt wurde. Im Gegensatz zu anderen Mikrocontrollern und Mikroprozessoren wurde hierbei ein moderneres, neueres Konzept realisiert. Hauptaugenmerk wurde hierbei auf eine effizientere Interrupt Behandlung, die optimierte Fähigkeit mit einzelnen Bits schnell und effizient zu arbeiten und auf einen erhöhten Datendurchsatz zwischen der On-Chip Peripherie, dem Speicher und der CPU zu erlangen, gelegt. Weiters spielte die Erweiterbarkeit mit anderen Peripherie Modulen eine wichtige Rolle um für verschiedene Anwendungen einen optimierten Mikrocontroller zu erhalten. Abbildung 3 zeigt die interne Blockstruktur der C167er Familie und gibt einen Überblick wie die einzelnen Module miteinander gekoppelt sind.

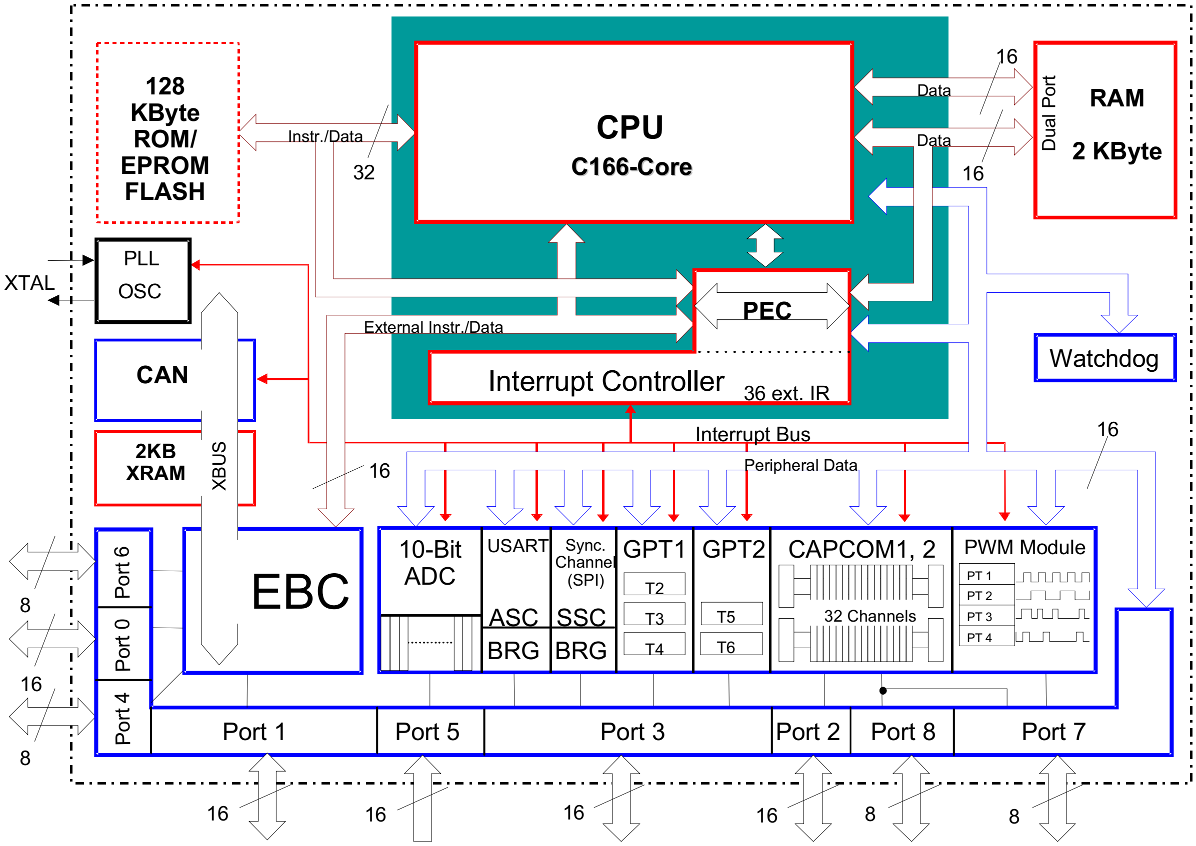


Abbildung 3: Blockschaltbild der C167er Familie

Die wichtigsten Strukturelemente sind der *CPU-Kern*, der eine geringfügige Erweiterung gegenüber der C166er CPU erfahren hat, der *Interrupt Controller* und die *Peripherie-Einheit*. Diese drei Kernblöcke bestimmen die Performance und die Reaktionsfähigkeit des Controllers auf externe Ereignisse und werden in den folgenden Abschnitten etwas detaillierter betrachtet. Desweiteren wurde ein Oszillator integriert, der bei einigen Derivaten über eine zusätzliche PLL verfügt, die den externen Takt mit einem einstellbaren Faktor multipliziert bevor er an die CPU und die interne

Peripherie weitergeleitet wird. Neben einem internen RAM Bereich verfügen einige Derivate auch über ein Masken-ROM[c], ein OTP-ROM[d] oder ein FLASH[e] Modul.

2.2 Die Central Processing Unit des C167

Die Hauptaufgabe der CPU[f] ist das Holen und Dekodieren von Befehlen, die Steuerung und Verarbeitung von Operanden in der Arithmetic Logic Unit (ALU) sowie das Retourspeichern der Ergebnisse. Weiters werden von der CPU die internen Busse zwischen den einzelnen Modulen gesteuert. Die Kommunikation mit dem externen Bus wird jedoch nicht von der CPU gehandhabt, sondern erfolgt über einen eigenen Externen Bus Controller (EBC). Die Abbildung 4 gibt einen groben Überblick über die Interna der CPU.

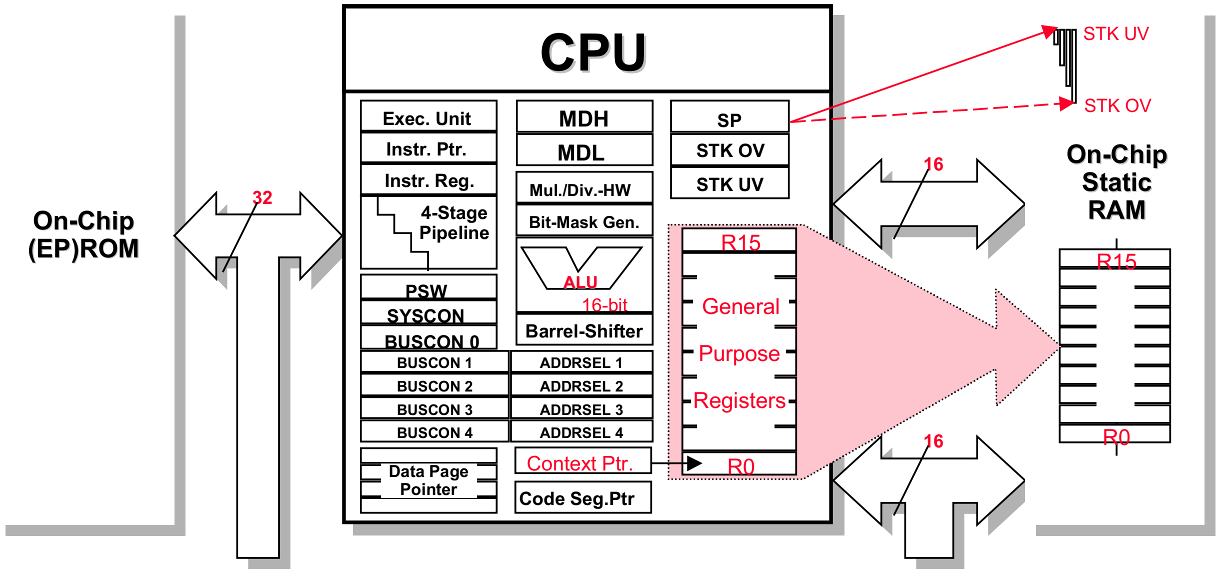


Abbildung 4: CPU Blockschaltbild des C167

Die Bestandteile der CPU sind eine 16-Bit ALU, die vor allem Additionen und Subtraktionen durchführt. Für Rotate- und Shift-Operationen ist ein eigener Barrel Shifter implementiert. Ein Bit Masken Generator erlaubt ein maskieren von einzelnen Bits und für Multiplikationen und Divisionen ist ebenfalls eine eigene Hardware vorhanden. Mit Hilfe dieser zahlreichen Funktionen ist es möglich, daß die meisten Befehle in einem einzigen Maschinenzyklus abgearbeitet werden können. Einzige Ausnahmen sind Befehle zur Multiplikation und Division sowie die Verzweigungs-

---

[c] Der Programmcode wird bei einem Masken-ROM vom Chiphersteller bei der Fertigung in das On-Chip ROM gebrannt. Dies ist vorallem bei großen Stückzahlen relevant.

[d] Ein OTP-ROM kann mit Hilfe eines Programmiergerätes einmal vom Entwickler direkt programmiert werden und eignet sich daher besonders für Kleinserien.

[e] Ein Flash Modul ist mehrmals elektrisch lösch- und programmierbar.

[f] Der Begriff *CPU* ist die englische Abkürzung für den Begriff *Central Processing Unit* der im allgemeinen mit dem Ausdruck *Zentraleinheit* übersetzt wird. Der Begriff Zentraleinheit selbst ist in der DIN Norm 44300 festgeschrieben, siehe [D78].

(Branch-) Befehle. Entsprechend dem Resultat werden auch zahlreiche Flags im Programm-Status-Wort gesetzt, detailliertere Informationen finden Sie in [Si96a] auf den Seiten 4-14ff. Folgende Abbildung 5 gibt dieses Register kurz wieder.

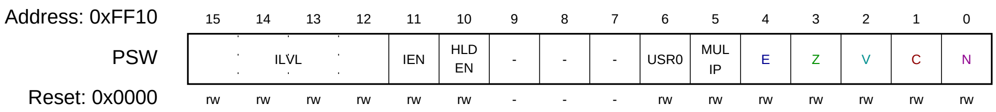


Abbildung 5: Das Programm Status Wort (PSW)

Das Negative Flag (N) wird gesetzt, wenn die ALU Operation ein negatives Ergebnis liefert. Negative Zahlen werden hierbei immer als Zweierkomplement einer positiven Zahl dargestellt. Das Carry Flag (C) kennzeichnet ein Carry/Borrow bei einer Addition/Subtraktion. Bei Shift und Rotate Operationen nimmt dieses Bit den Wert des jeweils hinausgeschobenen bzw. rotierenden Bits an. Wird bei einer arithmetischen Operation der Zahlenbereich überschritten so wird das Overflow Flag (V) gesetzt. Für Zweierkomplement 16-Bit Operationen beträgt der Zahlenbereich (-0x8000 bis +0x7FFF) und entsprechend für 8-Bit (-0x80 bis +0x7F). Ist das Ergebnis einer Operation Null, so wird das Zero Flag (Z) gesetzt. Eine Besonderheit stellt hier das End-of-Table Flag dar, das gesetzt wird wenn der Quelloperand einer Instruktion den Wert 0x8000 oder 0x80 annimmt, sprich wenn das Ende einer Tabellensuche erreicht wurde.

Die Execution Unit besteht aus dem Instruction Pointer (IP) der auf den jeweils gültigen Befehl zeigt. Der Instruction Pointer ist vom User aus nicht direkt manipulierbar sondern nur indirekt über Beeinflussung des Programmflusses (Jump, Call, etc.). Zugeordnet zur Execution Unit sind weiters das Instruction Register und die CPU Befehlspipeline. Der C167 besitzt eine 4-stufige Befehlspipeline, die es erlaubt, daß gleichzeitig mehrere Befehle “aktiv” sind. Die Pipeline des C167 ist in vier Phasen untergliedert. In der FETCH Phase wird ein Befehl aus dem Programmspeicher geholt. Dieser wird in der nächsten Phase, der DECODE Phase, dekodiert. In der Execution Phase wird der Befehl dann ausgeführt und in der WRITE BACK Phase werden dann die daraus resultierenden Werte retourgesichert. In Abbildung 6 ist dies illustriert.

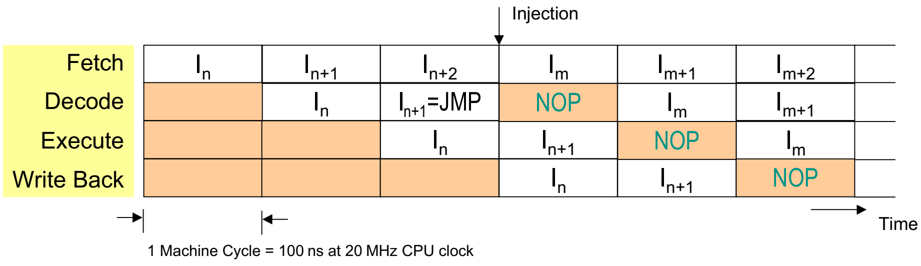


Abbildung 6: Die Befehlspipeline des C167

Neben dem Vorteil, daß durch Verwendung einer Pipeline der Durchsatz erhöht wird, sind auch einige Nebeneffekte zu beachten. Zu Beginn eines Programms muß die Befehlspipeline erst gefüllt werden. Dadurch kommt diese zu Beginn nicht sofort voll zum Tragen. Weitere Nebeneffekte sind z.B. die Behandlung von Sprüngen und Programmverzweigungen. Ein Standardsprung, siehe

Abbildung 6, wird erst in der DECODE Phase (In+1) erkannt. Zu diesem Zeitpunkt ist jedoch bereits der nächste Befehl (In+2) in der FETCH Phase. Der Befehl des Sprungzieles (Im) wird in die nächste FETCH Phase eingefügt und zusätzlich muß gleichzeitig in der DECODE Phase ein NOP Befehl anstelle des Befehls (In+2) plaziert werden. Dadurch wird der Durchsatz der Pipeline vermindert. Dies würde sich gravierend bei wiederholten Sprüngen auswirken. Um dies zu vermeiden wird der Befehl des Sprungzieles (Im) zusätzlich in einen Jump Cache kopiert. Wird der Befehl ein weiteres mal dekodiert, so wird er nicht im nächsten Maschinenzyklus in der FETCH Phase sondern sofort in der DECODE Phase eingefügt, siehe [Si96a] Seite 4-3ff. Eine weitere Problematik einer Befehlspipeline besteht darin, daß oft zahlreiche aufeinanderfolgende Befehle zueinander eine sequentielle Abhängigkeit aufweisen. Programmiert man in Assembler so ist hierauf bei der Manipulation des Context Pointers, der Data Page Pointer und des Stack Pointers Rücksicht zu nehmen. Bei Programmierung in einer höheren Programmiersprache wird dies vom Compiler berücksichtigt.

Weiters sind in der CPU enthalten:

- Code Segment Pointer (CSP) und Instruction Pointer (IP) zur Programmcode Adressierung
- Data Page Pointer (DPP0-3) zur Adressierung der Daten
- Context Pointer (CP) zur Adressierung der General Purpose Register
- Stack Pointer (SP), Stack Underflow (STKUN) und Stack Overflow (STKOV) zur Stackverwaltung
- Die Register MDL, MDH und MDC werden für Multiplikation und Divisions Operationen benötigt.
- Das Programm Status Wort (PSW) enthält Informationen über den CPU Status.
- Die Register SYSCON, BUSCON0-4 und ADDRSEL 1-4 dienen zur System- und Bussteuerung.

Nähere Informationen zu diesen Registern sind der begleitenden Literatur bzw. aus den folgenden Kapiteln zu entnehmen.

2.3 Die Speicherorganisation des C167

Wie bereits zuvor erwähnt besitzt der C167 einen nach der *von Neumann Architektur* organisierten Speicher. Dies ist durch einen gemeinsamen linearen Porgramm- und Datenspeicher gekennzeichnet. Der C167 kann bis zu maximal 16 MByte adressieren. Dieser Bereich ist unterteilt in 256 Code Segmente die jeweils 64 KByte groß sind. Weiters ist dieser Bereich in 1024 Data Pages zu je 16 KByte organisiert. Die Code Segmente werden über den Code Segment Pointer (CSP), und die Datenbereiche über 4 Data Page Pointer (DPP0-3) angesprochen. Diese Aufteilung ist auch aus der Abbildung 7 ersichtlich.

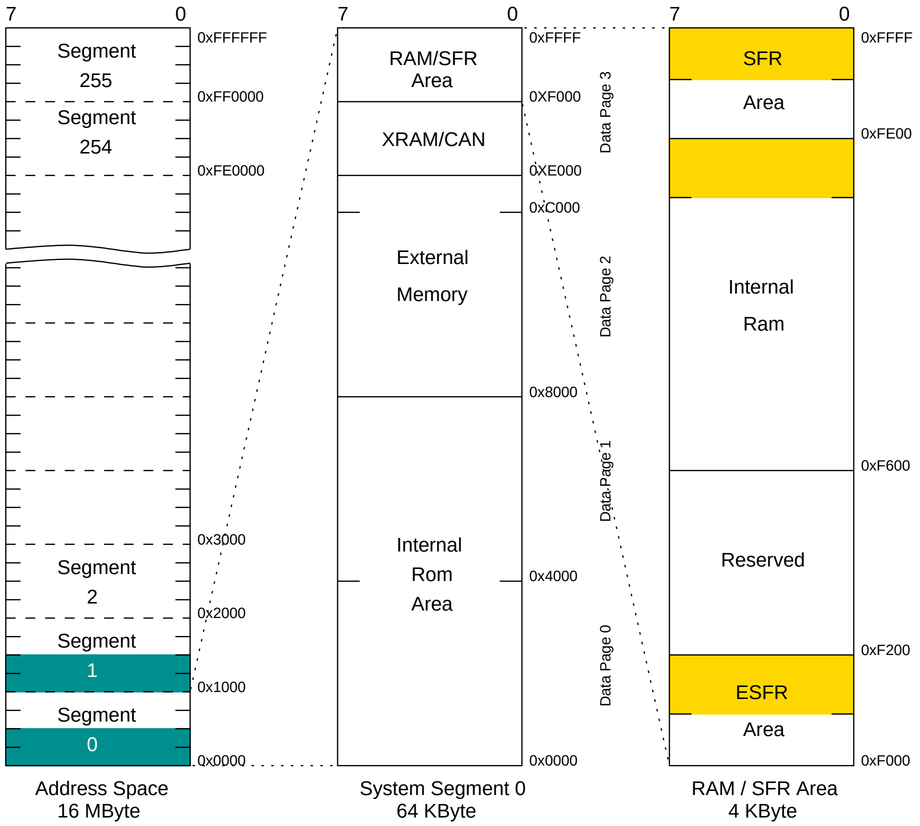


Abbildung 7: Speicherorganisation des C167

Das Segment 0 stellt hierbei einen Spezialfall dar, da es sämtliche Register der Peripheriemodule, die General Purpose Register (GPR), den Stack, das interne ROM –sofern vorhanden– und das interne RAM enthält. Ab Adresse 0x0000 im Segment 0 befindet sich das interne ROM[g], das auch in das Segment 1 eingeblendet werden kann. Dies wird während des Reset Vorganges über das CPU Register SYSCON festgelegt. Ab Adresse 0xE000 befindet sich der Bereich des sogenannten XRAMs (On-chip extension RAM) bzw. die Register des CAN Moduls. Das XRAM verhält sich wie ein normaler externer Speicher, der keine Wait-States benötigt, wobei jedoch hierfür keine externen Buszyklen generiert werden. Im Adreßbereich zwischen 0xF000 und 0xFFFF befindet sich das interne RAM und die Special Function Register (SFR). Dieser Bereich enthält sämtliche Peripherieregister des C167. Hierbei ist bemerkenswert, daß jene Register, die in den grau unterlegten Adreßbereichen liegen (0xF100-0xF1FF, 0xFD00-0xFDFF und 0xFF00-0xFFFF), bitweise adressierbar sind. D.h. zur Veränderung dieser Bits ist kein sogenannter Read-Modify-Write Zyklus nötig, wie er üblicherweise beim Manipulieren von Registern mittels einer Bitmaske verwendet wird. Das interne RAM enthält weiters den System Stack, der in seiner Größe im Bereich von 32 -1024 Worten programmierbar ist. Der Stack wächst hierbei von höheren Adressen zu niedrigeren Adressen hin. Mit Hilfe der Stack Underflow (STKUN) bzw. Stack Overflow (STKOV) Register kann der Stack auch ausgelagert und somit dynamisch erweitert werden. Man spricht in diesem Fall von einem zirkulärem Stack. Weiters sind in diesem Bereich die General Purpose Register an-

---

[g] Der in der Mikrocomputer Laborübung verwendete Baustein C167CR-LM verfügt über kein internes ROM.

gelegt, die über den Context Pointer (CP) adressiert werden. Wird ein neuer Registersatz benötigt (z.B. bei einem Unterprogrammaufruf oder der Verzweigung in eine Interrupt-Service-Routine) so kann durch Umsetzen des CP einfach auf eine neue Registerbank umgeschaltet werden. Dies hat den Vorteil, daß bei einer Programmverzweigung anstelle sämtlicher verwendeter Register nur der CP auf den Stack gesichert werden muß. Weiters befinden sich in diesem Adreßbereich die Source und Destination Pointer für den Peripheral Event Controller (PEC), siehe Abschnitt 2.4. Das restliche interne RAM dient entweder als Speicher für Variablen und Daten oder für Programmcode.

Word Zugriffe müssen immer an geraden Adressen erfolgen, wobei das lower Byte an der geraden Adresse (z.B 0xF612) und das higher Byte an der darauffolgenden ungeraden Adresse zu liegen kommt. Die einzelnen Bits werden aufsteigend von der niedrigeren (Bitnummer 0) zur höheren Adresse hin durchnumeriert.

Bei der Programmierung in Assembler ist zu beachten, daß zwar aufeinanderfolgende Datenbereiche die zu verschiedenen Speicherbereichen gehören problemlos angesprochen werden können, beim Ausführen von Programmcode über eine Segmentgrenze hinweg ist jedoch unbedingt ein expliziter Segmentwechsel mittels des Code Segment Pointers (CSP) nötig.

2.4 Das Interrupt System des C167

Als Microcontroller besitzt der C167 ein ausgeprägtes Interrupt System, das es ihm ermöglicht sehr effizient auf externe und interne Ereignisse zu reagieren. Grundsätzlich ist hierbei zu unterscheiden zwischen:

- normalen Interrupts
- Peripheral Event Controller (PEC) Interrupts
- Trap Funktionen (Hardware und Software Traps)
- externen Interrupts

Die Vorgehensweise des Ablaufs bei Auftreten eines Interrupts soll nun näher erläutert werden. Zunächst ist eine Interrupt Service Routine (ISR) zu erstellen, die i.a. einem Unterprogramm entspricht, jedoch mit der Ausnahme, daß für die entsprechende Interrupt Quelle eine definierte Interrupt Trap Nummer, siehe Tabellen 2 und 3, angegeben werden muß. Bei Auftreten eines Interrupts wird dann über diese Trap Nummer in der Interrupt Vektor Tabelle, die sich ab der Adresse 0x0000 im Adreßraum befindet, die Interrupt Service Routine aufgerufen.

Die entsprechende Interrupt Quelle ist mit einem geeigneten Interrupt Prioritäts Level (ILVL) und Group Prioritäts Level (GLVL) zu versehen. Der C167 besitzt 16 ILVL Level und jeweils vier Group Level. Es können somit maximal 64 verschiedene Interrupt-Prioritäten vergeben werden. Treten mehrere Interrupts gleichzeitig auf, so gewinnt jener mit höherem ILVL und GLVL Wert. Tritt nur ein Interrupt auf, so wird nur der ILVL Wert mit dem CPU-ILVL Wert im Programm Status Wort (PSW) verglichen. Nur wenn der Wert höher ist wird die CPU in ihrem aktuellen Programmfluß unterbrochen. Nach Festlegung der Signifikanz der einzelnen Interrupt Quelleb

Source of Interrupt PEC Service	**Request Flag**	**Enable Flag**	**Interrupt Vector**	**Vector Location**	**Trap Number**
CAPCOM Register 0	CC0IR	CC0IE	CC0INT	0x0040	0x10
CAPCOM Register 1	CC1IR	CC1IE	CC1INT	0x0044	0x11
CAPCOM Register 2	CC2IR	CC2IE	CC2INT	0x0048	0x12
CAPCOM Register 3	CC3IR	CC3IE	CC3INT	0x004C	0x13
CAPCOM Register 4	CC4IR	CC4IE	CC4INT	0x0050	0x14
CAPCOM Register 5	CC5IR	CC5IE	CC5INT	0x0054	0x15
CAPCOM Register 6	CC6IR	CC6IE	CC6INT	0x0058	0x16
CAPCOM Register 7	CC7IR	CC7IE	CC7INT	0x005C	0x17
CAPCOM Register 8	CC8IR	CC8IE	CC8INT	0x0060	0x18
CAPCOM Register 9	CC9IR	CC9IE	CC9INT	0x0064	0x19
CAPCOM Register 10	CC10IR	CC10IE	CC10INT	0x0068	0x1A
CAPCOM Register 11	CC11IR	CC11IE	CC11INT	0x006C	0x1B
CAPCOM Register 12	CC12IR	CC12IE	CC12INT	0x0070	0x1C
CAPCOM Register 13	CC13IR	CC13IE	CC13INT	0x0074	0x1D
CAPCOM Register 14	CC14IR	CC14IE	CC14INT	0x0078	0x1E
CAPCOM Register 15	CC15IR	CC15IE	CC15INT	0x007C	0x1F
CAPCOM Register 16	CC16IR	CC16IE	CC16INT	0x00C0	0x30
CAPCOM Register 17	CC17IR	CC17IE	CC17INT	0x00C4	0x31
CAPCOM Register 18	CC18IR	CC18IE	CC18INT	0x00C8	0x32
CAPCOM Register 19	CC19IR	CC19IE	CC19INT	0x00CC	0x33
CAPCOM Register 20	CC20IR	CC20IE	CC20INT	0x00D0	0x34
CAPCOM Register 21	CC21IR	CC21IE	CC21INT	0x00D4	0x35
CAPCOM Register 22	CC22IR	CC22IE	CC22INT	0x00D8	0x36
CAPCOM Register 23	CC23IR	CC23IE	CC23INT	0x00DC	0x37
CAPCOM Register 24	CC24IR	CC24IE	CC24INT	0x00E0	0x38
CAPCOM Register 25	CC25IR	CC25IE	CC25INT	0x00E4	0x39
CAPCOM Register 26	CC26IR	CC26IE	CC26INT	0x00E8	0x3A
CAPCOM Register 27	CC27IR	CC27IE	CC27INT	0x00EC	0x3B
CAPCOM Register 28	CC28IR	CC28IE	CC28INT	0x00F0	0x3C
CAPCOM Register 29	CC29IR	CC29IE	CC29INT	0x0110	0x44
CAPCOM Register 30	CC30IR	CC30IE	CC30INT	0x0114	0x45
CAPCOM Register 31	CC31IR	CC31IE	CC31INT	0x0118	0x46

Tabelle 2: Interrupt Ressourcen des C167 (Teil 1)

durch den Programmierer wird dieser Prioritätslevel (ILVL+GLVL) in das zugehörige Interrupt Control Register geschrieben. (z.B. T0IC = 0x0027; Weist dem Timer 0 Interrupt Control Register einen ILVL Wert von 9 und einen Group Level von 3 zu.) Das allgemeine Layout eines Interrupt Control Registers ist in Abbildung 8 wiedergegeben.

Source of Interrupt **PEC Service**	**Request Flag**	**Enable Flag**	**Interrupt Vector**	**Vector Location**	**Trap Number**
CAPCOM Timer 0	T0IR	T0IE	T0INT	0x0080	0x20
CAPCOM Timer 1	T1IR	T1IE	T1INT	0x0084	0x21
CAPCOM Timer 7	T7IR	T7IE	T7INT	0x00F4	0x3D
CAPCOM Timer 8	T8IR	T8IE	T8INT	0x00F8	0x3E
GPT1 Timer 2	T2IR	T2IE	T2INT	0x0088	0x22
GPT1 Timer 3	T3IR	T3IE	T3INT	0x008C	0x23
GPT1 Timer 4	T4IR	T4IE	T4INT	0x0090	0x24
GPT2 Timer 5	T5IR	T5IE	T5INT	0x0094	0x25
GPT2 Timer 6	T6IR	T6IE	T6INT	0x0098	0x26
GPT2 CAPREL Reg.	CRIR	CRIE	CRINT	0x009C	0x27
A/D Conversion Comp.	ADCIR	ADCIE	ADCINT	0x00A0	0x28
A/D Overrun Error	ADEIR	ADEIE	ADEINT	0x00A4	0x29
ASC0 Transmit	S0TIR	S0TIE	S0TINT	0x00A8	0x2A
ASC0 Transmit Buffer	S0TBIR	S0TBIE	S0TBINT	0x011C	0x47
ASC0 Receive	S0RIR	S0RIE	S0RINT	0x00AC	0x2B
ASC0 Error	S0EIR	S0EIE	S0EINT	0x00B0	0x2C
SSC Transmit	SSCTIR	SSCTIE	SSCTINT	0x00B4	0x2D
SSC Receive	SSCRIR	SSCRIE	SSCRINT	0x00B8	0x2E
SSC Error	SSCEIR	SSCEIE	SSCEINT	0x00BC	0x2F
PWM Channel 0...3	PWMIR	PWMIE	PWMINT	0x00FC	0x3F
CAN Interface	XP0IR	XP0IE	XP0INT	0x0100	0x40
X-Peripheral Node 1	XP1IR	XP1IE	XP1INT	0x0104	0x41
X-Peripheral Node 2	XP2IR	XP2IE	XP2INT	0x0108	0x42
PLL Unlock	XP3IR	XP3IE	XP3INT	0x010C	0x43

Tabelle 3: Interrupt Ressourcen des C167 (Teil 2)

Es gilt hierbei, je größer der ILVL und GLVL Wert desto höher ist die entsprechende Priorität. Nach der Festlegung der Priorität, ist die entsprechende Interrupt Quelle zu enablen (durch setzen des zugehörigen *yy*IE Bits). Soll ein PEC Transfer aktiviert werden, so ist an dieser Stelle das PECC Register, sowie die Source und Destination Adresse einzustellen. Nach der lokalen Interrupt Freigabe muß global ein Interrupt zugelassen werden. Dafür ist im CPU Programm Status Wort Register PSW das Bit 'Interrupt Enable' (IEN) zu setzten. Ab nun können auftretende Interrupts zugelassen und bearbeitet werden.

Wird ein Interrupt aktiviert —das *yy*IR Bit wird gesetzt— so wird zunächst eine Prioritätsüberprüfung durchgeführt. Hierfür ist im höherwertigen Nibble des PSW Information über den jeweils momentan gültigen Interrupt Status des Controllers enthalten, siehe Abbildung 5. Das Bit Interrupt Enable (IEN) —wenn gesetzt— gibt an, daß Interrupts des C167 global zugelassen sind. Durch Setzen bzw. Löschen dieses Bits können auf einfache Art und Weise Interrupts zugelassen bzw.

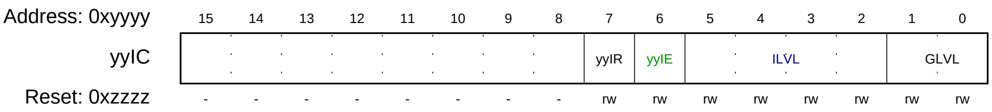


Abbildung 8: Layout eines Interrupt Control Registers des C167

gesperrt werden. Das Bit Feld ILVL im Program Status Word spiegelt den momentanen CPU Prioritätslevel wieder. Ist keine Interrupt Routine aktiv so entspricht dies einem ILVL Wert von 0x0000. Befindet sich die CPU jedoch in einer Interrupt Service Routine (ISR) so enthält dieses Feld den Prioritätswert der momentan gültigen Routine. Ist der ILVL Wert im yyIC Register größer als der ILVL Wert im PSW, so wird der Interrupt akzeptiert. Ist der ILVL Wert weiters ≥ 14, das zugehörige PECC Register programmiert und eine PEC Source und Destination Adresse aufgesetzt, so wird ein PEC Transfer durchgeführt. Wenn dies nicht der Fall ist wird eine normale Interrupt Service Routine durchgeführt. Hierzu wird zunächst das zugehörige Interrupt Request Flag (yyIR) gelöscht, die Werte des IP, CP, CSP und des PSW auf den Stack kopiert (push) und anschließend wird der ILVL Wert der auslösenden Interrupt Quelle aus dem Interrupt Control Register in das PSW kopiert. Danach wird in die Interrupt Vektor Tabelle verzweigt. Jede Interrupt Quelle hat hier einen festgelegten Eintrag mit einer zugehörigen Trap Nummer. Über diese Trap Nummer wird in die entsprechende Interrupt Service Routine verzweigt. Der Code der ISR wird nun abgearbeitet. Beim Verlassen der ISR werden die zuvor auf den Stack gesicherten Daten in umgekehrter Reihenfolge wieder retour kopiert (pop).

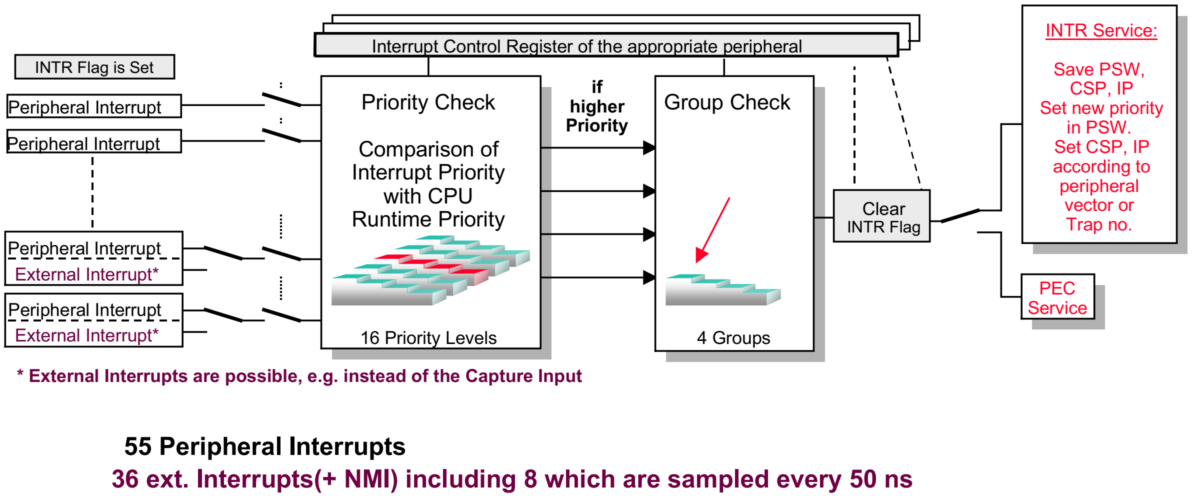


Abbildung 9: Interrupthandling im C167

Das Aufsetzen und die Abarbeitung eines Interrupts ist in der Abbildung 9 nocheinmal zusammenfassend veranschaulicht.

Wird hingegen ein *Peripheral Event Controller (PEC)* Service arbitriert, so wird ein einziger Byte/Word Transfer von der Source zur Destination Adresse innerhalb des Segments 0 in die Befehlspipeline eingefügt, siehe Abbildung 10.

Pipeline Stage	Cycle 1	Cycle 2	Cycle 3	Cycle 4
FETCH	N	N+1	N+2	N+2
DECODE	N-1	N	PEC	N+1
EXECUTE	N-2	N-1	N	PEC
WRITEBACK	N-3	N-2	N-1	N

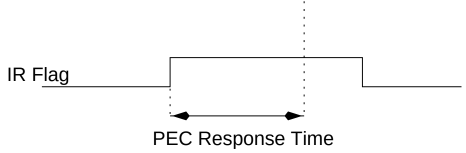


Abbildung 10: PEC Transfer injiziert in die Befehlspipeline

Der Vorteil dieser Variante liegt in der schnellen Reaktionszeit auf einen Interrupt Request. Es muß in diesem Fall kein Register auf den Stack gesichert werden. Die Konfiguration des PEC Transfers erfolgt über das zugehörige PECC Register. Dieses wird automatisch bei der Festlegung von ILVL und GLVL Wert entsprechend Abbildung 11 zugeordnet. Für einen PEC Transfer müssen Prioritätslevel $ILVL \geq 14$ programmiert werden. Dementsprechend können acht verschiedene PEC Transfers programmiert werden. Je nachdem ob konfiguriert wird im Anschluß an den PEC Transfer der COUNT Wert im PECCx Register dekrementiert und die zugehörige Source oder Destinationadresse inkrementiert. Dadurch wird eine fixe Anzahl an PEC Transfers ermöglicht. Es kann aber auch ein kontinuierlicher PEC Transfer programmiert werden indem im PECCx Register der COUNT Wert auf 0xFF programmiert wird, siehe Abbildung 12 und [Si96a] Seite 5-11ff.

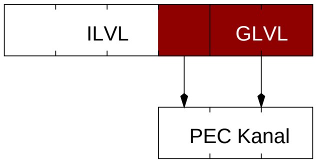


Abbildung 11: PECCx Register Festlegung

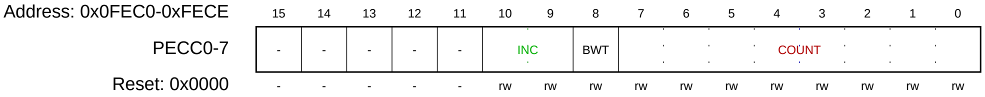


Abbildung 12: PECCx Register Layout

War der COUNT Wert vor dem Dekrementieren bereits 1 so wird das Interrupt-Request-Flag nicht mehr gelöscht und im Anschluß an diesen letzten PEC Transfer wird somit automatisch eine normale ISR aktiviert. Wird also ein PEC Transfer mit einem COUNT Wert $\leq 0xFF$ aufgesetzt, so ist für diese Interrupt Quelle auch eine normale Interrupt Service Routine zu erstellen.

Die meisten Peripherie Einheiten des C167 haben mehrere Interrupt Quellen denen jeweils ein eigenes Interrupt Control Register zugeordnet ist. Eine Ausnahme bilden hierbei das PWM und das CAN Modul, die zwar zahlreiche Interrupt Quellen haben aber nur mit einem Interrupt Control Register ausgestattet sind. In der CAN ISR müssen beispielsweise sämtliche Interrupt Quellen des CAN Moduls abgefragt werden. Erst wenn alle aufgetretenen Ursachen, die einen Request ausgelöst haben, bearbeitet wurden, kann die ISR wieder verlassen werden.

Der C167 besitzt nur einen einzigen *dedicated* Interrupt Pin – den Non-maskable Interrupt (NMI) – es können jedoch zahlreiche IO Pins mit der zugeordneten Logik so konfiguriert werden, daß ein externer Signalwechsel einen Interrupt Request auslösen kann. Diese Interrupts werden typischerweise alle 400ns bei einer Taktrate von 20MHz abgetastet. Die oberen acht Pins von Port 2 können jedoch zusätzlich als ***fast external Interrupts*** konfiguriert werden, siehe [Si96a] Seite 5-23. Hierbei kann auch der Flankenwechsel, der einen Request auslösen soll, programmiert werden. In diesem speziellen Fall erfolgt die Abtastung der Eingänge alle 50ns bei einer CPU Taktrate von 20MHz.

Neben den bisher besprochenen Interrupt Mechanismen gibt es noch die sogenannten Trap Funktionen, wobei zwei Arten unterschieden werden. Bei den ***Software Traps*** wird im Programmfluß über eine Trap Funktion (z.B: _trap_(0x10)) ein Interrupt ausgelöst. Hierbei erfolgt die Bearbeitung wie bei einer normalen ISR mit den Ausnahmen, daß das Interrupt Request (IR) Flag nicht gesetzt wird und der ILVL Prioritätswert nicht in das Programm Status Wort kopiert wird. Dies bedeutet, daß Software Traps von Interrupts mit niedrigerer Priorität stets unterbrochen werden können. Unter den ***Hardware Traps*** sind Fehlzustände der CPU zusammengefaßt (z.B: mis-aligned Zugriffe, Opcode Violations, etc.). Diese sind nicht maskierbar und haben Priorität über jede andere CPU Aktivität. Innerhalb dieser Hardware Traps gibt es eine Priorisierung in verschiedene Klassen, je nach Schwere des auftretenden Fehlers, siehe [Si96a] Seite 5-5. Die Bearbeitung dieser Fehler erfolgt wie bei einer normalen ISR mit der Ausnahme, daß stets ein ILVL Wert von 15 in das Programm Status Wort kopiert wird.

Das Aufsetzen und die Abarbeitung eines PECs sind in der Abbildung 13 nocheinmal zusammenfassend veranschaulicht.

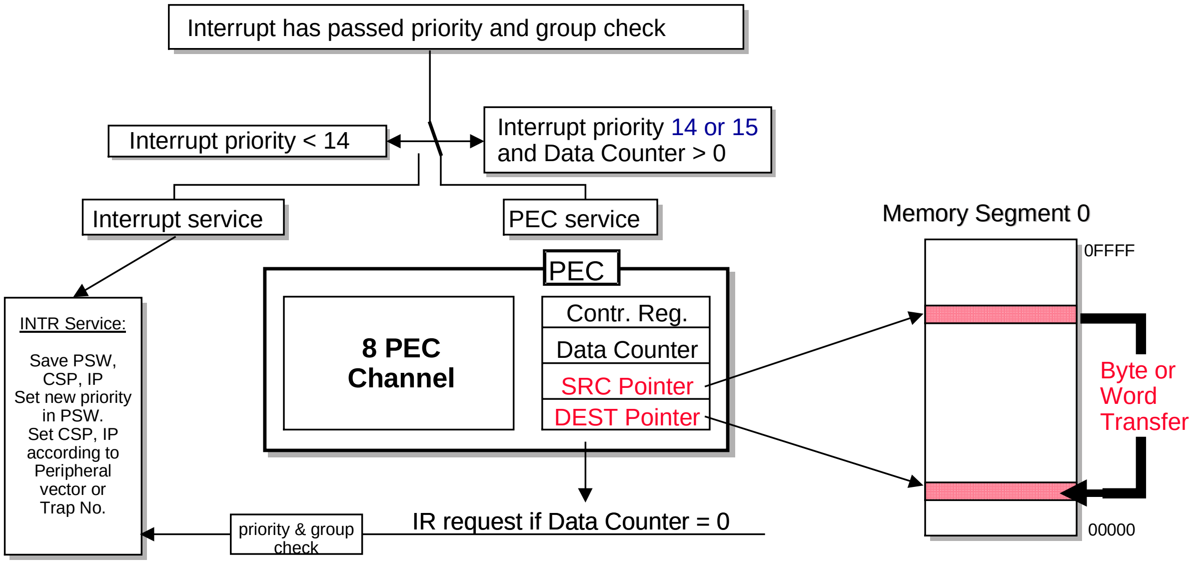


Abbildung 13: Funktionsweise des Peripheral Event Controllers

2.4.1 Beispiele zum Interrupthandling

Das folgende Beispiel[h] soll das Grundgerüst der normalen Interrupt Handhabung illustrieren.

```
#include <reg167.h>       /* in dieser Header Datei sind die Register des C167
                             definiert */

void timer0_isr (void) interrupt 0x20 {

   ......    /* Code der ISR */

}

void main (void) {

  .....            /* Timer0 initialisieren und starten */
  T0IC = 0x0052;   /* ILVL = 4; GLVL = 2; IE = 1; */
  IEN = 1;         /* globale Interruptfreigabe im PSW */
  while (1){       /* Endlosschleife */
    ;
  }
}
```

Ein weiteres Beispiel soll das Grundgerüst einer PEC Routine demonstrieren. Es soll 20 mal ein PEC Transfer durchgeführt werden, wobei die Destination Adresse inkrementiert werden soll. Nach den 20 Transfers soll eine normale ISR das Szenario abrunden.

```
#include <reg167.h>      /* in dieser Header Datei sind die Register des C167
                            definiert */

void adc_isr (void) interrupt 0x28 {

   ......    /* Code der ISR wird nach den 20 PEC Transfers aktiviert */

}

void main (void) {

  int feld[19];

  .....             /* ADC initialisieren und starten */
  ADCIC = 0x0079;   /* ILVL = 14; GLVL = 1; IE = 1;
                      =>> daraus resultiert PEC Kanal 1 */
  SRCP1 = _sof_(&ADDAT);   /* Quell-Adresse ist das Ergebnis Register
                              des AD Converters */
  DSTP1 = _sof_(feld[0]);  /* Ziel-Adresse ist das Array feld[];
                              diese Adresse wird nach jedem
```

---

[h] Die Beispiele zum C167 wurden mit der Entwicklungsumgebung der Fa. Keil erstellt. Diese sollen in erster Linie zur Illustration der einzelnen Module dienen und sind somit auch unter diesem Aspekt zu betrachten. Es obliegt dem Leser sich Gedanken über effizientere Implementierungen zu machen.

```
                             Transfer inkrementiert */
    PECC1 = 0x0314;    /* INC = 01b =>> inkrementiert DSTP1 um 1;
                          BWT = 1b =>> transferiert 1 Byte;
                          COUNT = 20d = 0x14 */
    IEN = 1;           /* globale Interrupt Freigabe */
    while (1) {        /* Endlosschleife */
      ;
    }
  }
```

2.5 Die Ports des C167

Der C167 besitzt 111 Ein- und Ausgänge (I/O), die in 8, einen 15-Bit und 16-Bit Ports organisiert sind. Zusätzlich verfügt er über einen 16-Bit Input Port zum Interfacing mit analogen Signalen. Ob ein Port als Eingang oder Ausgang verwendet wird, ist über die jeweils zugehörigen Direction Port Register (DPx) einstellbar (DPx.y=1: Ausgangsport, DPx.y=0: Eingangsport). Bei einigen Ports kann zusätzlich über ein Open Drain Port (OPDx) Register festgelegt werden, ob ein Pin als Push/Pull (Standard Konfiguration bei ’0’) oder als Open Drain(’1’) Ausgang verwendet wird. Die Open Drain Konfiguration erlaubt es, direkt mehrere Ausgangsports über einen Pull-Up Widerstand miteinander zu verbinden. In Abbildung 14 ist die Standardschaltung eines I/O Ports des C167 dargestellt.

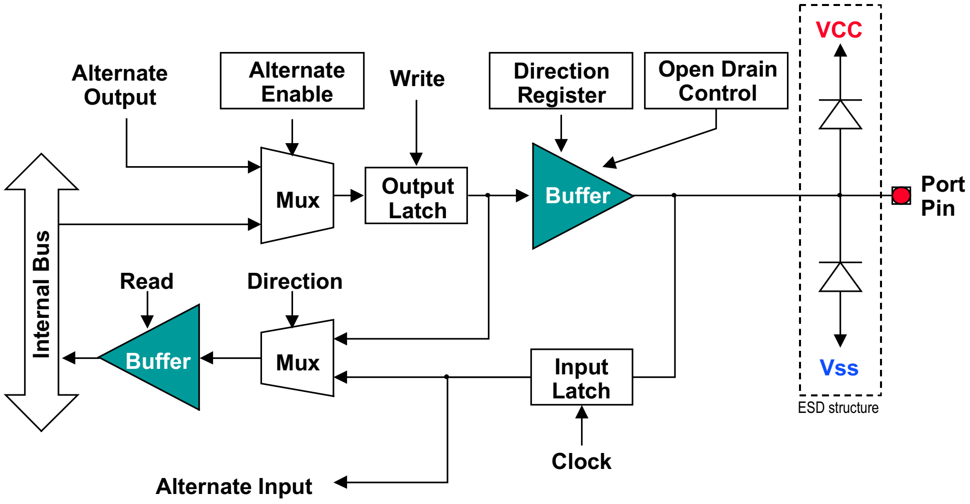


Abbildung 14: Ausgangsport Konfiguration

Ebenfalls kann der Threshold Level für einige Eingangsports konfiguriert werden. Die Standardkonfiguration sieht einen TTL Level vor. Welche speziellen Level im anderen Fall eingestellt werden können ist dem jeweiligen Datenblatt zu entnehmen.

Die meisten Ports des C167 haben zahlreiche alternierende Funktionen, wobei jeweils nur eine aktiviert sein darf. Diese Portbelegung ist grob aus der Tabelle 4 ersichtlich, detaillierte Informationen finden Sie in [Si96a] im Abschnitt Parallel Ports ab Seite 6-1.

P0H (IO)	-	D8-15 (IO)	A8-15 (O)	AD8-15 (IO)
P0L (IO)	D0-7 (IO)	D0-7 (IO)	AD0-7 (IO)	AD0-7 (IO)
P1H (IO)	A8-15 (O)	CC24IO-CC27IO (I)	-	-
P1L (IO)	A0-7	-	-	-
P2 (IO)	CC0IO-CC15IO (IO)	EX0IN-EX7IN (I)	T7IN (I)	-
P3 (IO)	CLKOUT (O), SCLK (IO), BHE# (O), RxD0 (I), TxD0 (O), MTSR (IO), MRST (IO), T2IN-T4IN (I), T3EUD (I), T3OUT (O), CAPIN (I), T6OUT (O), T0IN (I)	WRH# (O)		
P4.0-7 (IO)	A16-23 (O)	A16-19 (O), P4.4 (IO), CAN_RxD (I), CAN_TxD (O), P4.7 (IO)	-	-
P5 (I)	AN0-15 (I)	T2EUD (I), T4EUD (I), T5IN (I),T6IN (I), T5EUD (I), T6EUD (I)	-	-
P6.0-7 (IO)	CS0#-CS4# (O), HOLD# (I), HDLDA# (O), BREQ# (O)	-	-	-
P7.0-7 (IO)	POUT0-3 (O), CC28IO - CC31IO (IO)	-	-	-
P8.0-7 (IO)	CC16IO - CC23IO (IO)	-	-	-

Tabelle 4: Alternierende Portbelegungen des C167

2.5.1 Beispiel zu den Ports des C167

An Port P2.0 ist eine LED anzuschließen, siehe Abbildung 15. Diese LED ist per Software zu toggeln.[i]

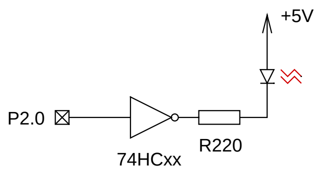


Abbildung 15: Beispiel - LED Beschaltung an Port P2.0

---

[i] C167 Port Pins liefern ca. 1mA, 74HCxx Bausteine jedoch ca. 35mA; In dieser Schaltung wird der Stromfluß durch den Widerstand auf ca. 15-20mA beschränkt.

```
#include <reg167.h>   /* in dieser Header Datei sind die Register des C167
                         definiert */

void main (void) {

  int i;
  sbit DP20 = DP2^0;  /* der Variablen DP20 wird bit 0 von DP2 zugewiesen */
  sbit P20 = P2^0;    /* der Variablen P20 wird bit 0 von P2 zugewiesen */

  DP20 = 1;           /* die Richtung von P2.0 wird auf Ausgang geschalten */
  P20  = 0;           /* an P2.0 wird eine 0 ausgegeben */

  while (1) {         /* Endlosschleife */
   for (i=0; i < 0x2625A0; i++) /* Software Warteschleife */
     ;
   P20 = 1;                     /* an P2.0 wird eine 1 ausgegeben;
                                   LED leuchtet */
   for (i=0; i < 0x2625A0; i++) /* Software Warteschleife */
     ;
   P20 = 0;                     /* an P2.0 wird eine 0 ausgegeben;
                                   LED dunkel */
  }
```

Wie aus diesem C-Sourcecode Listing ersichtlich ist, ist die CPU des C167 mit der Abarbeitung des Codes ständig beschäftigt. Viel eleganter wäre in diesem Zusammenhang eine Interrupt gesteuerte Lösung unter Verwendung eines Timers.

2.6 Der Resetvorgang des C167

Es kann auf drei verschiedene Arten ein Resetvorgang des C167 eingeleitet werden. Über einen *Hardware Reset* (dieser sollte mindestens 2 Taktzyklen lang andauern; allerdings ist eine längere Resetdauer empfehlenswert), einen *Software Reset* oder durch einen Reset über den *Watchdog Timer* kann der C167 in einen definierten Anfangszustand gebracht werden. Die Register nehmen hierbei den Wert an, der als Reset Wert bei der jeweiligen Beschreibung angegeben ist.

Während der Reset Phase wird der Pegel des Pins External Access (EA#) abgetastet. Ein aktiv lower Zustand bedeutet hierbei, daß ein externer Programmstart aus einem externen ROM vorgenommen wird, während bei EA#='1' ein interner Programmstart ab Adresse 0x0000 erfolgt. Zusätzlich wird in dieser Zeit der Zustand von Port 0 in diverse Register kopiert, siehe Abbildung 16.

Wird das Bit BSL = '0' abgetastet, so wird der Boostrap Loader Mode aktiviert. Hierbei erwartet der C167 Programm und Startcode von der asynchronen seriellen Schnittstelle. Im Bootstrap Mode erwartet der C167 über die asynchrone Schnittstelle zunächst ein Zero Byte (1 Start Bit, acht '0' Datenbits und danach ein Stopbit). Der C167 ermittelt aus der Dauer dieses Low Pulses die gesendete Baudrate und initialisiert damit die asynchrone Schnittstelle. Anschließend schickt er dem Host PC ein vordefiniertes Erkennungs-Byte[j]. Dies signalisiert dem Host PC, daß der C167

---

[j] Beim C167 hat das Erkennungsbyte den Wert 0xC5.

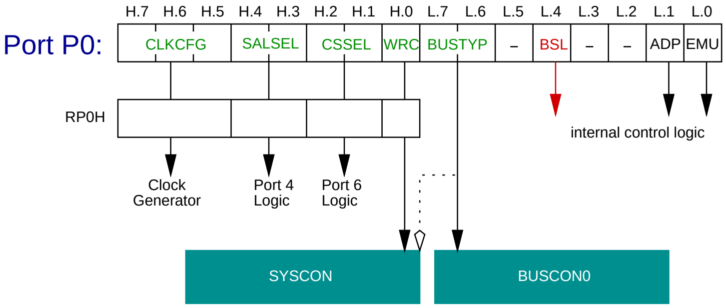


Abbildung 16: Port 0 während des Resetvorganges

die serielle Schnittstelle erfolgreich aktiviert hat. Danach schickt der Host dem C167 32 Code- und Datenbytes, die in der Folge das Programm an eine geeignete Stelle laden und den Resetvektor in der Interrupt Vektor Tabelle mit der Sprungadresse für den Programmstart einstellen. Wird hingegen BSL = ’1’ abgetastet, so erfolgt der Programmstart direkt von der Adresse 0x0000 – dem Resetvektor – weg. Über die Bitkombination BUSTYP wird festgelegt ob ein 8 oder 16-Bit sowie ein demultiplexed oder multiplexed Speicher am externen Datenbus angeschlossen ist. Das Bitfeld CSSEL legt schließlich die Anzahl der aktiven Chip Select Signale fest. Die Größe des Adreßraums und die Konfiguration für die Taktrate wird über die Bitfelder SALSEL und CLKCFG definiert, siehe auch [Si96a] Seite 17-7. Bei einem Software und einem Watchdog Timer Reset werden die niederwertigen sechs Bit an Port P0L ignoriert.

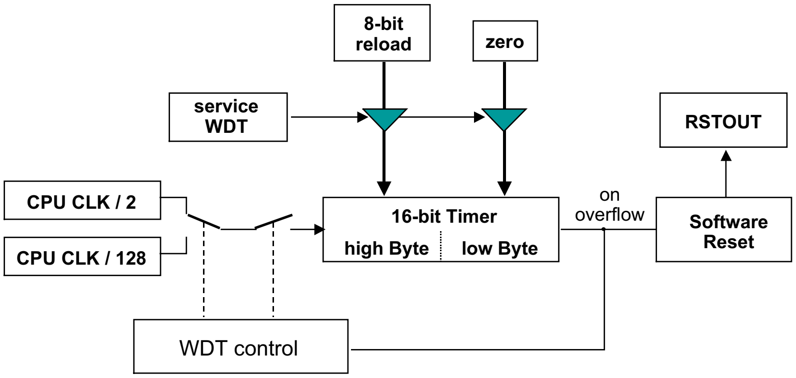


Abbildung 17: Der Watchdog Timer des C167

Der Watchdog Timer, siehe Abbildung 17, wird dazu verwendet um das System bei kurzen externen Störungen wieder in einen definierten Zustand zu bringen. Bei der Programmentwicklung ist allerdings darauf zu achten, daß der Watchdog Timer Reset nicht aktiviert wird solange der C167 im Bootstrap Loader Mode ist. Grundsätzlich muß der Watchdog Timer während der Initialisierungsphase aktiviert oder deaktiviert werden. Wird er aktiviert so generiert er bei einem Überlauf einen Reset des C167. Um dies zu verhindern muß dieser Timer periodisch während des Programmablaufes rückgesetzt werden. Der Sinn des Watchdog Timers liegt darin, daß die Wahrscheinlichkeit für eine Störung sehr groß ist, wodurch wiederum verhindert wird, daß der Watchdog Timer rückgesetzt wird. Dies bewirkt, daß der C167 in der Folge bei einem Überlauf des Watchdog

Timers resetiert und wieder in einen definierten Zustand versetzt wird. Beim Aufsetzen dieses Timers ist die Ausführungszeit des restlichen Programmes (mit Unterbrechungen durch ISRs) genau zu betrachten und der Watchdog mit einem etwas höheren Wert zu initialisieren. Generell können mit dem Watchdog Timer Zeitbereiche zwischen $25,6\mu s$ und $419ms$ abgedeckt werden. Nähere Informationen hierzu bietet der Abschnitt Watchdog Timer in [Si96a] Seite 12-1.

2.7 Der External Bus Controller

Der C167 verfügt über einen eigenen *External Bus Controller* (EBC) für die Steuerung von Zugriffen auf externe Bausteine. Dadurch wird die CPU stark entlastet und durch langsame externe Einheiten nicht unnötig angehalten und externe Chip-Select Logik kann meist gänzlich eingespart werden. Während des Resets wird unter anderem die Bus Konfiguration für Zugriffe auf den *default* Adreßbereich festgelegt. Wobei dieser default Adreßbereich all jene Adreßbereiche umschließt, die nicht von anderen Bereichen allokiert sind. Die Arbitrierung und Zuordnung von Adreßbereichen wird etwas später in diesem Abschnitt noch genauer erläutert. Die Konfiguration selbst wird in das Bitfeld BUSTYP von Register BUSCON0 kopiert und gibt an ob auf die externe Baugruppe im entsprechenden Adreßbereich mit 8 oder 16-Bit, mit einem multiplexed oder demultiplexed Buszyklus zugegriffen wird. Sämtliche BUSCON Register 0-4 haben hierbei das gleiche Layout. In Abbildung 18 ist hierbei exemplarisch das Registerlayout von BUSCON1 dargestellt. Alle BUSCON Register sind in [Si96a] auf Seite 8-18 dargestellt. Die Konfiguration einzelner Adreßbereiche wird über BUSCON1-4 während der Initialisierungsphase durchgeführt. Die Initialisierungsphase selbst wird mit dem Befehl EINIT (End of Initialization) abgeschlossen. Die einstellbaren Konfigurations-Möglichkeiten sind hierbei der Abbildung 18 zu entnehmen. Grundsätzlich ist bei den gewählten Einstellungen zwischen einem multiplexed und einem demultiplexed Bus Modus zu unterscheiden. Beim **multiplexed Bus Modus** werden über Port 0 die Adressen und Daten im Zeitmultiplex ausgegeben. Hierbei ist ein externes Zwischenspeichern der Adressen nötig. Der Buszugriff, wie er in Abbildung 19 dargestellt ist, beginnt mit der Aktivierung des *Address Latch Enable* (ALE) Signals. Danach werden die Adressen ausgegeben, die mit der fallenden Flanke des ALE Signals extern zu latchen sind. Anschließend liegen die Adressen noch einige Zeit an bevor Sie vom Bus weggeschaltet werden müssen. Der EBC aktiviert nun das entsprechende Steuersignal (RD# für einen Lesezugriff, WR# für einen Schreibzugriff, bzw. WRL# und WRH#). Danach werden die Daten entweder vom EBC oder dem externen Baustein angelegt. Nach einer Zeitspanne, die durch die Zugriffszeit auf die externe Einheit definiert ist, werden die Daten wieder ungültig.

Im **demultiplexed Bus Modus** werden die Adressen und Daten über getrennte Leitungen geführt. Der EBC legt die Adressen für den gesamten Buszyklus an Port 1 respektive Port 4 an. Weiters werden die entsprechenden Steuersignale (RD#, WR#, WRL# od. WRH#) aktiviert, die für eine programmierbare Zeitspanne anliegen. Im Anschluß werden die Daten entweder vom EBC oder der externen Einheit aktiviert. Diese sind aber erst nach der einstellbaren Zugriffszeit gültig.

- Bei Lesezyklen werden die einlangenden Daten gespeichert woraufhin das Steuersignal deaktiviert wird. In der Folge nimmt die externe Einheit die Daten vom Bus und schaltet diesen wieder auf Tri-State.

ddress: 0xFF14	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BUSCON1	CSWEN1	CSREN1	-	RDYEN1	-	BUSACT1	ALECTL1	-	BTYP		MTTC1	RWDC1	MCTC			
Reset: 0x0XX0	rw	rw	-	rw	-	rw	rw	-	rw	rw	rw	rw	rw	rw	rw	rw

Bit	Function
MCTC	**Memory Cycle Time Control (Number of memory cycle time wait states)** 0 0 0 0 : 15 waitstates (Number = 15 - <MCTC>) 1 1 1 1 : No waitstates
RWDCx	**Read/Write Delay Control for BUSCONx** ’0’: With read/write delay: activate command 1 TCL after falling edge of ALE ’1’: No read/write delay: activate command with falling edge of ALE
MTTCx	**Memory Tristate Time Control** ’0’: 1 waitstate ’1’: No waitstate
BTYP	**External Bus Configuration** 0 0 : 8-bit Demultiplexed Bus 0 1 : 8-bit Multiplexed Bus 1 0 : 16-bit Demultiplexed Bus 1 1 : 16-bit Multiplexed Bus Note: For BUSCON0 BTYP is defined via PORT0 during reset.
ALECTLx	**ALE Lengthening Control** ’0’: Normal ALE signal ’1’: Lengthened ALE signal
BUSACTx	**Bus Active Control** ’0’: External bus disabled ’1’: External bus enabled (within the respective address window, see ADDRSEL)
RDYENx	**READY Input Enable** ’0’: External bus cycle is controlled by bit field MCTC only ’1’: External bus cycle is controlled by the READY# input signal
CSRENx	**Read Chip Select Enable** ’0’: The CS# signal is independent of the read command (RD#) ’1’: The CS# signal is generated for the duration of the read command
CSWENx	**Write Chip Select Enable** ’0’: The CS# signal is independent of the write command (WR#,WRL#,WRH#) ’1’: The CS# signal is generated for the duration of the write command

Abbildung 18: Register BUSCON1

- Bei Schreibzugriffen wird zunächst das Steuersignal deaktiviert. Wenn programmiert werden die Daten und Adressen noch etwas länger aktiv gehalten.

Diese Abfolge ist in der Abbildung 20 detailliert dargestellt. Beim Übergang zwischen einem demultiplexed in einen multiplexed Buszyklus wird ein Idle Zyklus eingefügt.

Das Bustiming beim Zugriff auf externe Bausteine kann mittels folgender Parameter variiert werden (siehe auch Abbildung 18).

- ALE Control (ALECTL): (relevant für multiplexed Buszugriffe) Dieser Wert definiert die Länge des ALE Signals und die Hold Time der Adreßleitungen nach der fallenden Flanke von ALE.
- Memory Cycle Time (MCTC): Mit diesem Wert können 0-15 Wait-States eingestellt werden. Mit MCTC wird die Zugriffszeit auf externe Einheiten festgelegt.

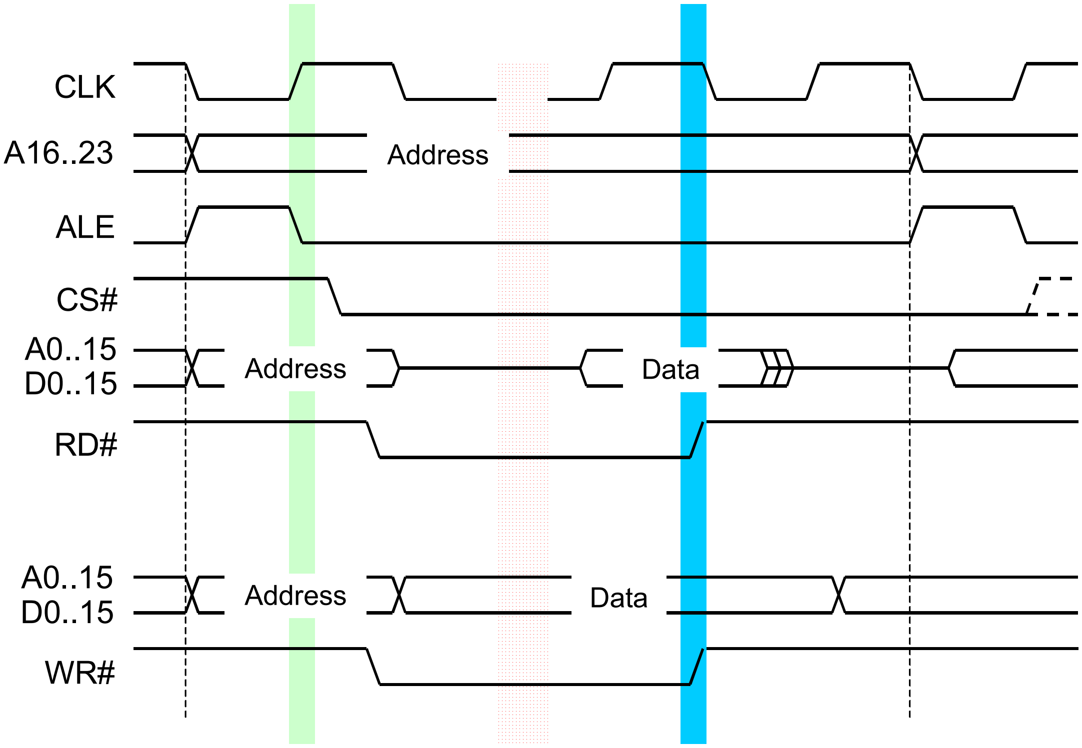


Abbildung 19: Multiplexed Bus Modus

- Memory Tristate Time (MTTCx): Dieser Wert definiert den Zeitpunkt für den Wechsel der Datenbustreiber vom aktiven in den inaktiven Zustand.
- Read/Write Delay (RWDCx): (relevant für multiplexed Buszugriffe) Gibt die Zeitspanne an nach der das Steuersignal (RD#, WR# bzw. WRL# und WRH#) in Bezug auf die fallende Flanke des ALE Signals aktiviert wird.
- READY Control (RDYENx): Werden mehr Wartezyklen benötigt als über das Feld MCTC programmiert werden können bzw. ist eine externe Terminierung des Buszykluses nötig, so ist dieses Bit zu setzen.

Wie bereits zuvor erwähnt können für verschiedene Adreßbereiche des C167 über den EBC Steuer-, Daten- und Adreßsignale generiert werden. Für jedes der vier Chip Select Signale (CSx#) ist jeweils ein BUSCONx/ADDRSELx Registerpaar zuständig. (z.B. BUSCON1 steuert den zeitlichen Ablauf der Steuersignale sowie die Datenbreite und Zugriffsart für CS1#; ADDRSEL1 gibt weiters an in welchem Adreßbereich diese Signale generiert werden, siehe Abbildung 21.) Hierfür können Adreßfenster im Bereich zwischen 4 KByte und 8 MByte selektiert werden, die sich auch gegenseitig überlappen können.Welches Fenster bei einer Adreßbereichsüberlappung dann jeweils selektiert wird, ist über die Adreßbereichs-Arbitrierung festgelegt (siehe Abbildung 22). Mit höchster Priorität werden Adreßbereiche, die über XBCONx und XADRSx selektiert werden arbitriert. Diese Bereiche sind für ein *X-Peripheral* (z.B: CAN, XRAM) – sofern bei einem C167er Derivat überhaupt vorhanden – zuständig und für dieses auch fest verdrahtet. In der Folge werden all

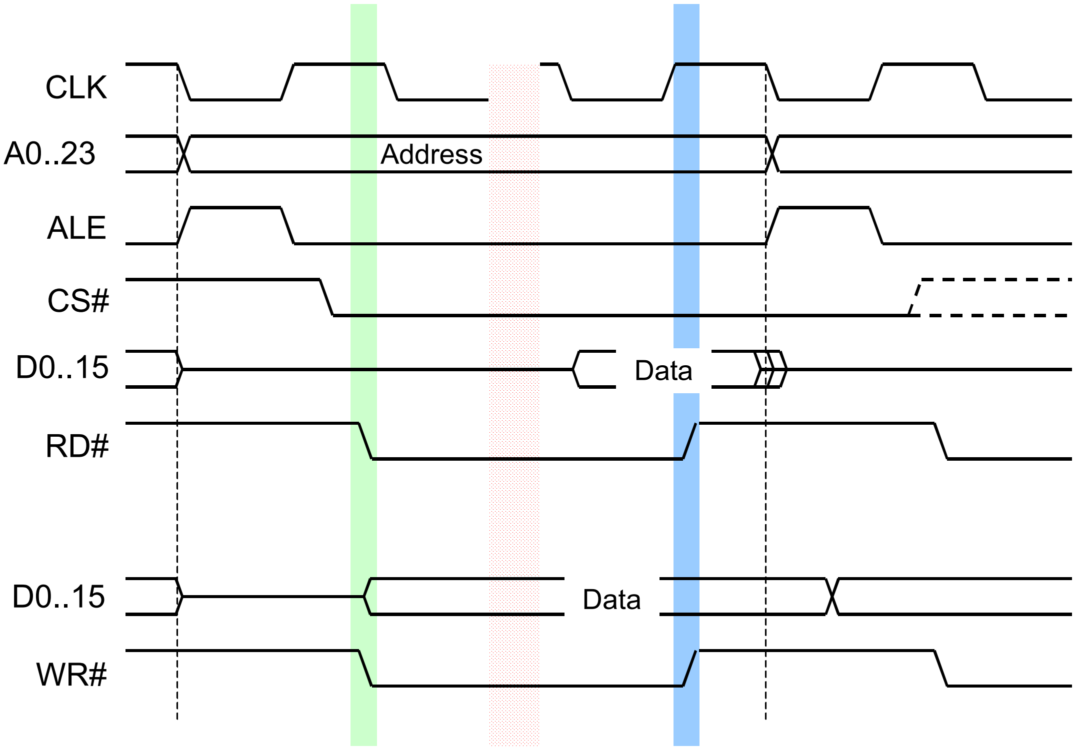


Abbildung 20: Demultiplexed Bus Modus

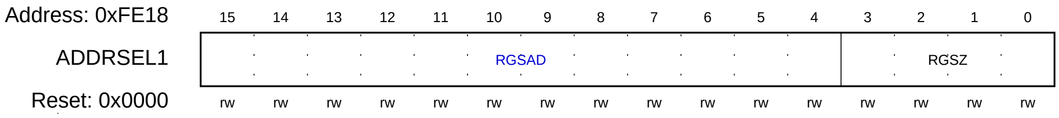
jene Bereiche arbitriert die über ADDRSEL2/4 bzw. BUSCON2/4 selektiert werden. Bereiche die über ADDRSEL2 und ADDRSEL4 ausgewählt werden dürfen sich hierbei nicht überlappen, dies gilt ebenso für die Bereiche von ADDRSEL1 und ADDRSEL3. Letztere werden nach den ADDRSEL2 und 4 Bereichen arbitriert. Der *default* Bereich umfaßt den gesamten Adreßraum des C167 und wird über BUSCON0 gesteuert. Dieser Bereich hat die niedrigste Priorität und wird nur dann selektiert wenn kein anderer Bereich aktiviert ist.

2.7.1 Timing Parameter des C167

In diesem Abschnitt sollen die im Datenblatt [Si95] angegebenen AC-Charakteristiken des C167 besprochen werden. Sehr detaillierte Angaben hierzu sind auch [MS95] zu entnehmen.

Die Grundlage jedes Mikroprozessors/controllers ist der Takt. Von diesem werden sämtliche internen Aktionen abgeleitet, so auch die Signale für den externen Bus. Jedem Signal, jeder Signalflanke liegt eine spezifische auslösende Taktflanke zugrunde. Man unterscheidet zwischen einem *nominellen Timing* bei dem keine Laufzeitverzögerungen berücksichtigt werden und dem *realen Timing*, bei dem alle Verzögerungen innerhalb des Bausteins berücksichtigt werden. Für diese Signal Delays gibt es zahlreiche Ursachen:

- Leitungslängen, Kapazitäten, Induktivitäten



Bit	Function
RGSZ	**Range Size Selection** Defines the size of the address area controlled by the respective BUSCONx/ ADDRSELx register pair. See table below.
RGSAD	**Range Start Address** Defines the upper bits of the start address (A23...A12) of the respective area. See table below. (X ... don't care)

Bit field RGSZ	**Resulting Window Size**	**Relevant Bits (R) of Start Address (A23..A12)**
0000	4 KByte	R R R R R R R R R R R R
0001	8 KByte	R R R R R R R R R R R X
0010	16 KByte	R R R R R R R R R R X X
0011	32 KByte	R R R R R R R R R X X X
0100	64 KByte	R R R R R R R R X X X X
0101	128 KByte	R R R R R R R X X X X X
0110	256 KByte	R R R R R R X X X X X X
0111	512 KByte	R R R R R X X X X X X X
1000	1 MByte	R R R R X X X X X X X X
1001	2 MByte	R R R X X X X X X X X X
1010	4 MByte	R R X X X X X X X X X X
1011	8 MByte	R X X X X X X X X X X X
11xx	Reserved	

Abbildung 21: Registerlayout von ADDRSEL1

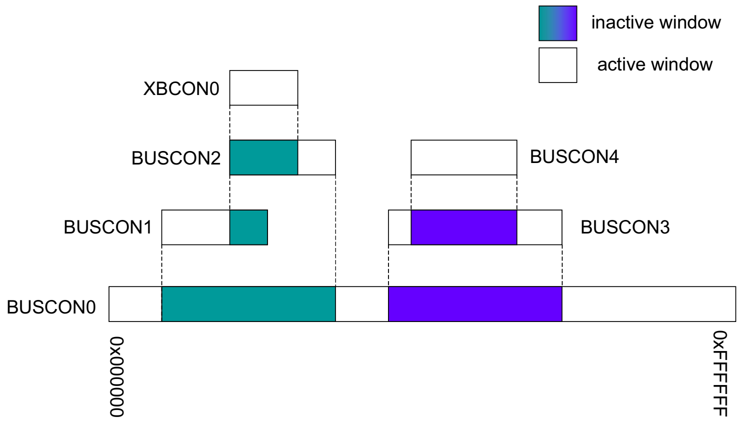


Abbildung 22: Adreßbereichsarbitrierung

- Umgebungstemperatur, Feuchtigkeit, Spannungsversorgung, etc.
- Variationen in der Fertigung
  - Lage des Bausteins am Wafer
  - Größe des Wafers (Verzerrungen durch Dotierungsverteilung, Maskenbelichtung)
  - Variationen im Fertigungsprozeß, etc.

Im Datenblatt werden für die jeweiligen Parameter Grenzwerte angegeben, die auf keinen Fall innerhalb der spezifizierten Betriebsbedingungen unter bzw. überschritten werden. Im Unterschied zum C166 verfügt der C167 über die Möglichkeit direkt Chip Select Signale für externe Baugruppen erzeugen zu können. Dadurch kann in den meisten Fällen zusätzliche externe Decodier Logik eingespart werden. Die intern erzeugten Chip Select Signale werden hierbei aus den Adressen abgeleitet. Erschwerend ist hierbei die Tatsache, daß diese Signale programmierbar sind. Dadurch wird die Erzeugung der Signale relativ komplex. Im folgenden sind einige dieser Timingparameter punktuell aufgezählt und mit den wichtigsten Erläuterungen versehen:

- ALE High Time t_5: Das Adreß Latch Enable (ALE) Signal wird unabhängig vom eingestellten Bus-Modus erzeugt, im Normalfall wird es jedoch nur im Multiplex Betrieb benötigt. Mit der fallenden Flanke sind extern die Adressen zu latchen. Die für ein externes Adreß-Latch geforderte Länge des Enable-Pulses muß kleiner als t_5 sein.
- Address Setup to ALE t_6: Diese Zeit gibt an, wann die Adressen spätestens vor der fallenden Flanke von ALE gültig sein müssen. D.h. die geforderte Setup-Time des externen Latches muß $\leq t_6$ sein.
- Address Hold after ALE t_7: Die Hold Zeit des Adreß-Latches muß $\leq t_7$ sein.
- ALE falling edge to RD#/WR# t_9 (t_8 with R/W delay): Dieser Parameter ist in manchen Anwendungen bedeutsam, wenn eine bestimmte Wartezeit vor der Aktivierung der Speichersteuersignale gefordert ist.
- Address float after RD#/WR# (Multiplex Betrieb) t_{10}, t_{11}: Diese Zeit gibt für den Fall eines Lesezugriffs an, wie lange die Adressen nach Aktivierung des RD# Signals noch aktiv sind. Problematisch ist hierbei, daß externe Bausteine wie z.B. Speicher zumeist den Datenbus mit Aktivierung des Speichersteuersignals treiben. Dies kann zu einem Buskonflikt führen! Mit R/W Delay ist diese potentielle Konfliktzeit wesentlich geringer als ohne R/W Delay; *d.h. im Multiplex Betrieb ist die Einstellung eines R/W Delay empfehlenswert* — es sei denn die externe Baugruppe aktiviert den Datenbus erst entsprechend nach dieser Zeitspanne.
- RD#, WR# low Time t_{12}, t_{13}: Dieser meist unkritische Parameter gibt die garantierte LOW Zeit dieser Signale an. Beim Lesen ist die fallende Flanke von RD# relevant, da die externen Bausteine auf diese Flanke folgend die Daten aktiv treiben. Beim Schreiben ist jedoch die steigende Flanke von WR# interessant, hier ist eine Setup-Zeit der Daten erforderlich.

- RD# to Valid Data In t_{14}, t_{15}: Als eine der wichtigsten Parameter gibt diese Zeit an, wann nach der fallenden Flanke von RD# die Lesedaten vom Speicher spätestens am Eingang des Controllers stabil anliegen müssen. Die Zeit Output Enable to Output Data Valid des externen Bausteins muß $\leq t_{14}$ (bzw. $\leq t_{15}$) sein. Hierfür müssen auch sämtliche externen Verzögerungen zwischen den beiden Bausteinen berücksichtigt werden.

- ALE LOW to Valid Data In t_{16}: Ist für das externe Latch eine Gültigkeit der Ausgänge ab der fallenden Flanke von ALE angegeben, so ist dieser Parameter im Multiplex Betrieb anstelle von t_{17} für die Berechnung der Speicherzugriffszeit heranzuziehen.

- Address to Valid Data In t_{17}: Die Zugriffszeit des externen Bausteins muß auf jeden Fall kleiner als t_{17} abzüglich sämtlicher Verzögerungen zwischen beiden Bausteinen sein.

- Data Hold after RD# rising edge t_{18}: Dieser unkritische Parameter gibt die Dauer an, für die der externe Baustein die Daten nach der steigenden Flanke von RD# noch aktiv halten muß.

- Data float after RD# t_{19}, t_{20}, t_{21}: Bei aufeinanderfolgenden Lesezyklen im Multiplex Betrieb muß der externe Baustein die Daten nach der Deaktivierung des RD# Signals vom Bus nehmen. Reicht diese Zeit nicht aus, so muß noch ein weiterer Tristate Waitstate eingefügt werden. Folgt jedoch ein Demultiplex Buszyklus, so müssen die Daten abgeschaltet sein bevor der Baustein des Folgezyklusses die Daten aktiv treibt. Dies ist bei einem Lesezyklus am kritischsten wo die Daten zumeist unmittelbar nach Aktivierung des RD# Signals angelegt werden. D.h. in diesem Zusammenhang müssen die erforderlichen Zeiten überprüft werden, und gegebenenfalls ist ein Tristate Waitstate einzufügen (MTTC). Besonders kritisch gestaltet sich der Übergang zwischen dem Demultiplex Betrieb in einen Multiplex Betrieb. Hier wird per Hardware automatisch ein leicht modifizierter Waitstate injiziert.

- Data Valid to WR# t_{22}: Da ein Speicher normalerweise die Daten mit der steigenden Flanke von WR# einliest, müssen die Daten entsprechend vorher gültig anliegen. Beim kalkulieren dieser Setup-Zeit sind eventuelle externe Verzögerungen zwischen den Bausteinen zu berücksichtigen.

- Data Hold after WR# t_{23}, t_{24}: Von den externen Bausteinen wird zumeist eine Hold Time nach der steigenden Flanke von WR# gefordert.

- ALE Rising Edge after RD#, WR# t_{25}, t_{26}: Diese Parameter geben den kürzesten Abstand des Read/Write Signals zum folgenden Bus-Zyklus an.

- Address Hold after RD#, WR# t_{27}, t_{28}: Die Adressen dürfen im Demultiplex Betrieb nicht vor den Steuersignalen abgeschalten werden. Oft wird von den externen Bausteinen sogar eine mindest geforderte Hold Time angegeben.

- ALE falling edge to CS# t_{38}: Dieser Parameter hat lediglich dann eine Relevanz, wenn der momentane Bus-Zyklus mit einem anderen Chip-Select als der vorherige arbeitet.

- CS# LOW to Valid Data In t_{39}: Wird ein Baustein über ein Chip-Select Signal des Controllers angeschlossen, so ist dieser Parameter für die Bestimmung der Zugriffszeit heranzuziehen.

- CS# Hold after RD#, WR# t_{40}, t_{41}: Die Haltezeit gilt für den Fall, daß im anschließenden Bus-Zyklus ein anderes Chip-Select erzeugt wird. Greift der nächste Bus-Zyklus auf denselben Bereich zu, so bleibt das Chip Select aktiv.

- RrCS#, WrCS# t_{42} bis t_{57} und t_{68}: Diese Parameter beziehen sich alle auf die optionalen Read/Write Chip Selects. Diese Signale werden prinzipiell wie die entsprechenden RD# bzw. WR# Signale generiert. Aufgrund der etwas aufwendigeren Logik sind diese jedoch etwas verzögert. Die prinzipiellen Anmerkungen bei den RD# und WR# Signalen gelten deshalb auch für RdCS# bzw. WrCS#.

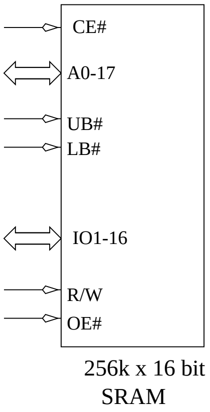
Für den Anschluß Wort-organisierter Bausteine besitzt der C167 zusätzliche Steuersignale, die einen Byte-weisen Zugriff ermöglichen. Hier werden anstelle der Adreßleitungen An-A0 die Leitungen An-A1 an den externen Baustein angeschlossen. Mit dem Signal A0 und der zusätzlichen Steuerleitung BHE# wird die Zugriffsart unterschieden, siehe Tabelle 5. Eine weitere Möglichkeit derartige Zugriffe zu machen besteht durch die Verwendung der Signale WRL# und WRH#. Dies erlaubt aber nur einen direkten Unterschied bei schreibenden Zugriffen. Bei lesenden Zugriffen kann der Controller ohnedies selbst die entsprechende Unterscheidung machen. In einigen seltenen Fällen wird bei manchen Peripheriebausteinen beim Lesen bestimmter Bytes ein Status Bit geändert. In diesem Fall muß auch beim Lesen die Byte Steuerung erfolgen; d.h. über A0 und BHE#.

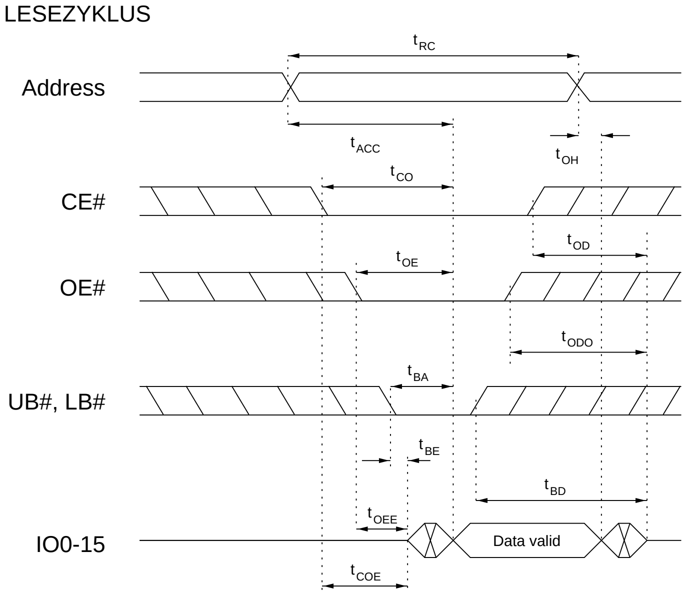
A0	BHE#	Bedeutung
0	0	Zugriff auf ein ganzes Wort
0	1	Zugriff auf das Low Byte
1	0	Zugriff auf das High Byte
1	1	illegale Kombination

Tabelle 5: Byte-weiser Zugriff auf Wort organisierte Bausteine

2.7.2 Beispiele zur Timinganalyse

Im folgenden Beispiel soll ein 256Kx16-Bit SRAM Baustein [To97] an den C167 angeschlossen werden. Die Timing-Daten für den C167, der mit **20 MHz** getaktet werden soll, sind aus dem Datenblatt des C167 [Si95] zu entnehmen. Die Daten des SRAM Speicherbausteins sind aus den nebenstehenden Tabellen und Grafiken ersichtlich.

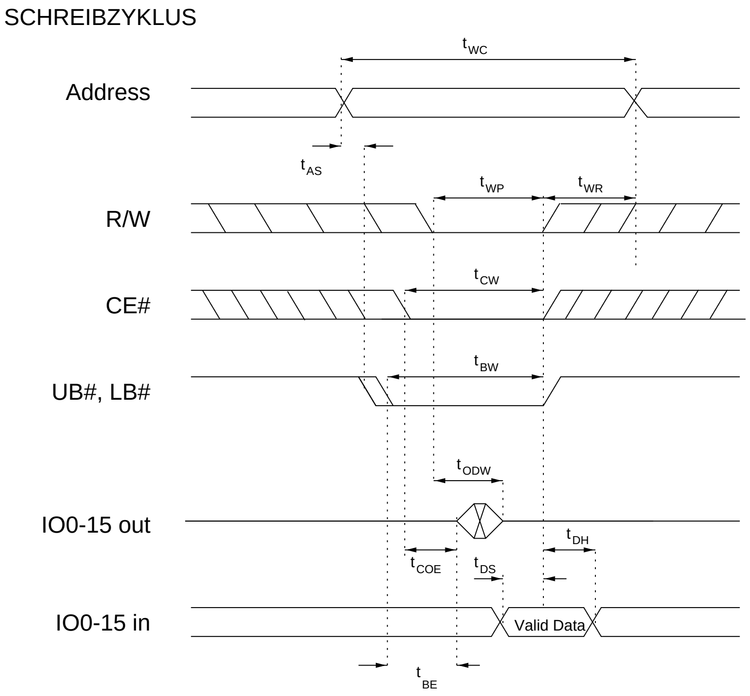




Parametertabelle zum 256K x 16-Bit SRAM:

Parameter	Symbol	min.	max.
Read Cycle Time	t_{RC}	100 ns	
Address Access Time	t_{ACC}		100 ns
Chip Enable Access Time	t_{CO}		100 ns
Output Enable Access Time	t_{OE}		50 ns
Output Data Hold Time	t_{OH}	10 ns	
CE# low to Output active	t_{COE}	10 ns	
OE# low to Output active	t_{OEE}	5 ns	
Output high-Z from CE# high	t_{OD}		35 ns
Data Byte Control Access Time	t_{BA}		50 ns
Data Byte Control to Output active	t_{BE}	5 ns	
Data Byte Control high to Output high-Z	t_{BD}		35 ns
Write Cycle Time	t_{WC}	100 ns	
Write Pulse Width	t_{WP}	60 ns	
CE# to End of Write	t_{CW}	80 ns	
Address Setup Time	t_{AS}	0	
Write Recovery Time	t_{WR}	10 ns	
Output high-Z from WE# low	t_{ODW}		35 ns
OE# high to Output high-Z	t_{ODO}		35 ns
Data Byte Control to End of Write	t_{BW}	60 ns	
Data Setup Time	t_{DS}	40 ns	
Data Hold Time	t_{DH}	0 ns	

Der Speicherbaustein ist mit getrennten Daten- und Adreßleitungen ausgeführt und auch entsprechend an den C167 angeschlossen. Für den C167 ist daher der *16-Bit Demultiplexed Bus Mode*



zu wählen. Da der Baustein über getrennte Byte Enable Leitungen (UB#, LB#) verfügt kann der Baustein folgendermaßen an den C167 angeschlossen werden:

- CE# wird an CS1# des C167 angeschlossen (frei gewählt).
- A0-17 werden an die Adreßleitungen A1-A18 des C167 angeschlossen.
- IO1-16 sind mit den Datenleitungen D0-D15 zu verbinden.
- OE# ist an die RD# Leitung, R/W an die WR# Leitung anzuschließen.
- Zur Unterscheidung der jeweils aktiven Bytes wird BHE# an UB#, und A0 an LB# angeschlossen.

Im folgenden werden nun die benötigte Anzahl an Waitstates ermittelt um diesen Speicher optimal ansprechen zu können:

- RD#, WR# LOW Time:
  - no R/W delay
    - C167: $t_{13} = 65ns + t_c$
    - Speicher: es gibt keine entsprechende Angabe im Lesezugriff
    - $t_{WP} = 60ns$

Der C167 muß das WR# Signal mindestens für die Dauer von t_{WP} aktiv treiben. Aus $t_{WP} \leq t_{13}$ folgt $-5ns \leq t_c$.

- RD#, WR# LOW Time:
  - with R/W delay
    - C167: $t_{12} = 40ns + t_c$
    - $t_{WP} = 60ns$

Die Länge des RD# bzw. WR# Pulses wird hier kürzer da die Buszykluslänge gleich bleibt das R/W Signal aber später aktiviert wird. Aus $t_{WP} \leq t_{12}$ folgt $20ns \leq t_c$.

- RD# to Valid Data In:
  - no R/W Delay
    - C167: $t_{15} = 55ns + t_c$
    - Speicher: $t_{OE} = 50ns$

Der Speicher muß spätestens t_{15} nach RD# die Daten gültig getrieben haben. Aus $t_{OE} \leq t_{15}$ folgt $-5ns \leq t_c$.

- RD# to Valid Data In:
  - with R/W Delay
    - C167: $t_{14} = 30ns + t_c$

Hier folgt aus $t_{OE} \leq t_{14}$ die Bedingung $t_c \geq 20ns$.

- Address to Valid Data In:
  - C167: $t_{17} = 70ns + 2t_A + t_c$
  - Speicher: $t_{acc} = 100ns$

Da für den Demultiplex das ALE Signal keine besondere Relevanz aufweist wird $t_A := 0ns$ gesetzt. Die Daten müssen vor t_{17}, nach Aktivierung der Adressen, beim Mikrocontroller eingelangt sein. Mit $t_{17} \geq t_{acc}$ ergibt sich $t_c \geq 30ns$.

- Data Hold after RD# rising edge:
  - C167: $t_{18} = 0ns$
  - Speicher: $t_{DH} = 0ns$

Der Speicher muß die Daten mindestens t_{18} nach RD# rising edge noch aktiv halten. Aufgrund der obigen Werte ist die Ungleichung $t_{DH} \geq t_{18}$ erfüllt.

- Data float after RD# rising edge:
  - no R/W delay
    - C167: $t_{21} = 15ns + t_F$
    - Speicher: $t_{ODO} = 35ns$

Der Speicher muß spätestens nach t_{21}, von RD# rising edge weg, die Daten vom Bus genommen haben. Somit ergibt sich aus $t_{21} \geq t_{ODO}$ die Bedingung $t_F \geq 20ns$.

- Data float after RD# rising edge:
  - with R/W delay
    - C167: $t_{20} = 35ns + t_F$

Hieraus folgt $t_F \geq 0ns$.

- Data Valid to WR#:
  - C167: $t_{22} = 25ns + t_c$
  - Speicher: $t_{DS} = 40ns$

Aufgrund dieser Data Setup Time müssen die Daten um t_{22} vor WR# aktiv und gültig sein. Dies ergibt $t_c \geq 15ns$.

- Data Hold after WR#:
  - C167: $t_{24} = 15ns + t_F$
  - Speicher: $t_{DH} = 0ns$

Der Mikrocontroller muß die Daten noch für t_{24}, nach WR#, aktiv halten. Die Bedingung $t_{24} \geq t_{DH}$ ist für alle Werte von t_F erfüllt.

- Address Hold after RD#, WR#:
  - C167: $t_{28} = 0ns + t_F$
  - Speicher: $t_{WR} = 10ns$

Nach der Deaktivierung der Steuersignale sind die Adressen noch für $t_{28} \geq t_{WR}$ aktiv zu halten. Dies ergibt $t_F \geq 10ns$.

- CS# to Valid Data In:

  - C167: $t_{39} = 55ns + t_c + 2t_A$
  - Speicher: $t_{CO} = 100ns$

Die Daten müssen vor $t_{39} \geq t_{CO}$, nach Aktivierung von CS#, beim Mikrocontroller eingetroffen sein. Daraus ergibt sich die Bedingung $t_c \geq 45ns$.

- CS# Hold after RD#, WR#

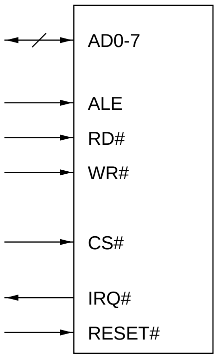
Für diese Zeitbeziehung gibt es beim Speicher keine direkte Entsprechung.

Aus all diesen Bedingungen ergeben sich nun folgende Schlußfolgerungen:

1. Konfiguration ohne R/W delay:
   - $t_F \geq 20ns := 2TCL(1 - MTTC)$. Aus dieser Gleichung ist ersichtlich, daß $MTTC = 0$ gewählt werden sollte. $(TCL = 25ns)$
   - $t_c \geq 45ns := 2TCL(15 - MCTC)$ bedingt einen Wert $MCTC = 14$, d.h. es ist ein Waitstate einzufügen.
   - Wie bereits erwähnt ist aufgrund des Demultiplex Betriebs normal ALE zu konfigurieren $ALECTL = 0$.
2. Konfiguration mit R/W delay:
   - $t_F \geq 10ns$ bewirkt ebenso $MTTC = 0$.
   - Die übrigen Werte sind analog zur Konfiguration ohne R/W delay vorzunehmen.

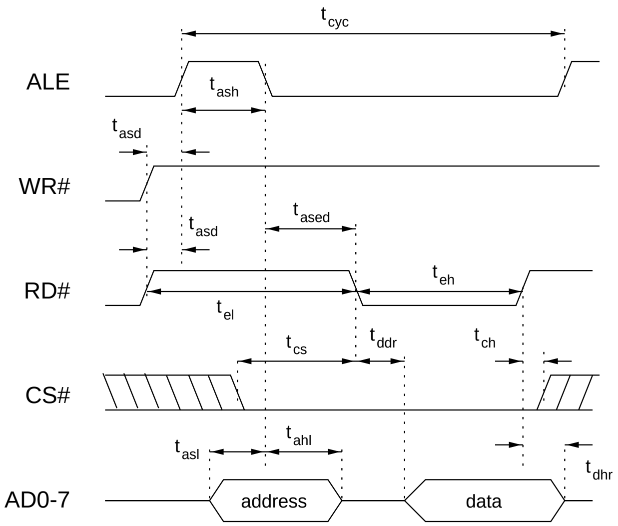
Aufgrund der Speichergröße von 256Kx16 ist ein 512KB großes Adreßfenster im zugehörigen ADDRSEL1 Register zu konfigurieren. Weiters muß die Basisadresse ein ganzzahlig vielfaches der Speicherbereichsgröße sein, d.h. in diesem Fall kann der 512KB große Speicher auf die Startadressen 0x0, 0x80000, 0x100000, 0x180000, usw. gemappt werden. Möchte man diesen Speicher nun ab der Adresse 0x80000 aufwärts in den Speicherbereich des C167 einblenden so ergibt sich für ADDRSEL1=0x0807, siehe [Si96a] Seite 8-20. Für das Register BUSCON1 ist weiters danach ein Wert von 0x048E entsprechend den obigen Daten einzustellen. Möchte man nun in diesem Speicher Programme oder Daten ablegen, so sind die Einstellungen im Linker diesbezüglich zu erweitern.

Im zweiten Beispiel soll eine Real Time Clock [Da98] an den C167 angeschlossen werden. Hierbei sind diesesmal die Timingwerte für eine Taktfrequenz von 10MHz zu ermitteln. Die Timingparameter zum C167 sind aus dem Datenblatt [Si95], jene zur Real Time Clock wiederum aus den folgenden Abbildungen und Tabellen zu entnehmen.



Real Time Clock

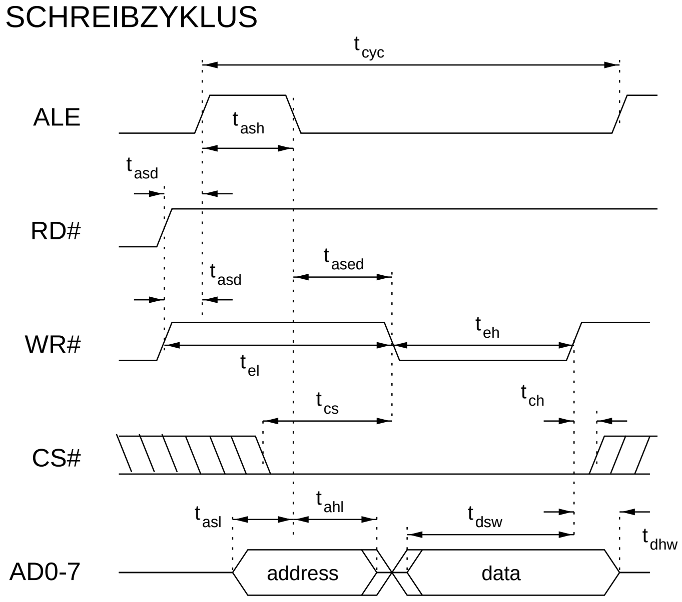
LESEZYKLUS



Parameter	Symbol	min.
Cycle Time	t_{cyc}	385 ns
Pulse Width ALE high	t_{ash}	60 ns
Delay Time RD# high to ALE high	t_{asd}	20 ns
Delay Time ALE low to RD# low	t_{ased}	40 ns
Pulse Width RD# high	t_{el}	150 ns
Pulse Width RD# low	t_{eh}	125 ns
CS# low to RD# low	t_{cs}	20 ns
Output Data delay from RD# low	t_{ddr}	120 ns
Chip Select Hold Time	t_{ch}	0 ns
Address Valid to ALE low	t_{asl}	30 ns
Address Hold Time	t_{ahl}	10 ns
Read Data Hold Time	t_{dhr}	10 ns
Data Setup Time	t_{dsw}	100 ns
Write Data Hold Time	t_{dhw}	0 ns

Die Real Time Clock ist mit gemultiplexten Daten- und Adreßleitungen ausgeführt und auch entsprechend an den C167 angeschlossen. Für den C167 ist daher der *8-Bit Multiplexed Bus Mode* zu wählen.

- CS# wird an CS2# des C167 angeschlossen (frei gewählt).
- AD0-7 werden an die Leitungen AD0-AD7 des C167 angeschlossen.
- RD#, WR# und ALE sind an die gleichnamigen Ports des C167 anzuschließen.
- Da Interrupt Leitungen in den meisten Fällen als Open Drain Ausgänge ausgeführt sind, ist die IRQ# Leitung über einen Pull-Up Widerstand an einen Fast External Interrupt des C167 anzuschließen.



- Das Reset# Signal ist entweder vom Systemreset oder über einem I/O Pin des C167 zu kontrollieren.

Im folgenden sind die Timingparameter für das Interfacing dieser beiden Bausteine aufeinander abzustimmen, wobei sich jeweils die halbe Taktperiode TCL aus

$$TCL = \frac{1}{2 * 10MHz} = 50ns$$

ergibt.

- ALE High Time: $t_5 = TCL - 10ns + t_A \geq t_{ash} = 60ns$
- Address Setup to ALE: $t_6 = TCL - 15ns + t_A \geq t_{asl} = 30ns$
- Address Hold after ALE: $t_7 = TCL - 10ns + t_A \geq t_{ahl} = 10ns$
- ALE falling edge to RD#, WR#: (with R/W delay) $t_8 = TCL - 10ns + t_A \geq t_{ased} = 40ns$
- RD#, WR# Low Time: (with R/W delay) $t_{12} = 2TCL - 10 + t_c \geq t_{eh} = 125ns$
- RD# to Valid Data In: (with R/W delay) $t_{14} = 2TCL - 20 + t_c \geq t_{ddr} = 120ns$
- Data Hold after RD# rising edge: $t_{18} = 0ns \leq t_{dhr} = 10ns$
- Data Valid to WR#: $t_{22} = 2TCL - 25 + t_c \geq t_{dsw} = 100ns$
- Data Hold after WR#: $t_{23} = 2TCL - 15ns + t_F \geq t_{dhw} = 0ns$
- ALE rising edge after RD#, WR#: $t_{25} = 2TCL - 15 + t_F \geq t_{asd} = 20ns$

- CS# Low to Valid Data In: $t_{39} = 3TCL - 20ns + t_c + 2t_A \geq t_{cs} + t_{ddr} = 40ns$
- CS# Hold after RD#, WR#: $t_{40} = 3TCL - 15ns + t_F \geq t_{ch} = 0ns$

Mit einem R/W Delay ergibt sich nun folgende Konfiguration:

- $t_A \geq 20ns := TCL * ALECTL$, daraus ergibt sich ALECTL zu 1.
- $t_c \geq 40ns := 2TCL * (15 - MCTC)$ ergibt für MCTC den Wert 14, also 1 Waitstate.
- Der Wert für die Tristate Time ergibt sich zu $t_F = 0ns$ und somit $MTTC = 1$.

Sollen die Register der Real Time Clock z.B. ab der Adresse 0x90000 angesprochen werden, so ist das zugehörige ADDRSEL2 Register mit dem Wert 0x0900 zu laden. Ein Adreßfenster von 4 KByte reicht für diesen Baustein aus, da ohnedies nur 2^8 Adressen generiert werden können. Für das BUSCON2 Register ergibt sich aus den ermittelten Timingparametern ein Wert von 0x067E, siehe [Si96a]. Ein Register auf der Adresse 0x90000 könnte man in “C” z.B. über folgende Makro Sequenz ansprechen:

```
#define RTC_REGISTER_ADDR 0x90000
#define MVAR(object, addr) ((object volatile far *) (addr))
#define RTC_REGISTER MVAR(unsigned char, RTC_REGISTER_ADDR)
```

2.8 Timer

Die C167er Familie ist mit neun Timern ausgestattet wovon fünf als sogenannte General Purpose Timer verwendet werden können. Diese fünf Timer, die in zwei funktionelle Gruppen unterteilt sind (GPT1 und GPT2), werden in diesem Abschnitt etwas detaillierter besprochen. Die Timer Gruppe GPT1 verfügt über drei Timer, die miteinander verschaltet werden können und die eine Auflösung von 400ns bei einer CPU Taktfrequenz von 20MHz aufweisen. Eine zeitliche Auflösung von 200ns läßt sich mit den beiden Timern des GPT2 Blockes erzielen. Für alle Timer gilt, daß sie in verschiedensten Betriebsarten programmiert werden können, in denen sie sowohl für sich alleine als auch in Kombination miteinander funktionell operabel sind.

2.8.1 General Purpose Timer Unit 1

Die Gruppe GPT1 besteht aus den Timern T2, T3 und T4. In Abbildung 23 ist ein Funktionsschaltbild von Timer T3 wiedergegeben.

Im Timer Mode (T3M) wird T3 über den CPU Clock getaktet. Mit Hilfe eines Prescaler Faktors (T3I) wird der CPU Clock in Zweierpotenz Schritten zwischen 8 und 1024 geteilt bevor er zum eigentlichen Timer geführt wird. Neben der normalen Timer Betriebsart gibt es den Gated Timer Modus wobei programmiert werden kann ob das Gate Signal high oder low aktiv berücksichtigt wird. Der Timer läuft in dieser Betriebsart nur im aktiven Bereich des Gatesignals, das über Port

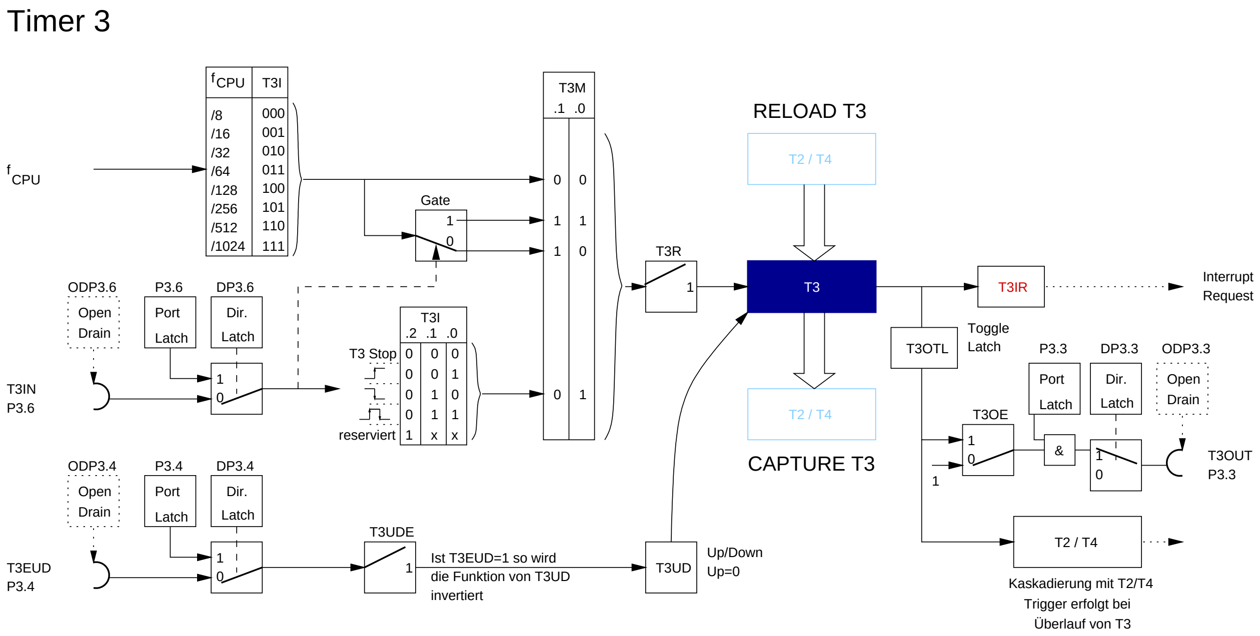


Abbildung 23: Funktionsschaltbild von Timer T3

P3.6 extern angelegt wird. T3 läßt sich auch als Counter betreiben. In diesem Fall erfolgt ein Inkrementieren bzw. Dekrementieren des Counters über eine programmierbare Signalflanke. Die Zählrichtung kann einerseits per Software (T3UDE=‘0‘) über das Konfigurationsbit T3UD (down = ‘1‘, up = ‘0‘) oder im anderen Fall (T3UDE=‘1) über den Port Pin P3.4/T3EUD festgelegt werden (T3EUD=‘1‘ ... down, T3EUD=‘0‘ ... up). In letzterem Fall kann die Zählrichtung zusätzlich über T3UD invertiert werden, siehe hierzu [Si96a] Seite 9-4. Bei einem Überlauf des Timers (0xFFFF →0x0000 bzw. 0x0000 → 0xFFFF) wird jeweils ein Interrupt Request generiert indem das Bit T3IR gesetzt wird. Zusätzlich wird hierbei das Output Toggle Latch einmal getoggelt. Dieses kann einerseits über den Port Pin P3.3[k] nach außen geführt werden und andererseits zur Kaskadierung mit den Timern T2 oder T4 herangezogen werden. Die Timer T2 und T4 (siehe Abbildung 24) können als Reload Register für T3 fungieren. Der Reload wird hierbei entweder durch das Toggeln von T3OTL oder durch einen Flankenwechsel an T2IN / T4IN getriggert. Im Reload Modus hält der entsprechende Reload Timer T2 / T4 unabhängig vom entsprechenden Run Bit (T2R / T4R) des Timers.

Die Capture Betriebsart ist ein weiterer Modus indem der Wert des Timers T3 entweder in das Timer Register von T2 oder T4 kopiert wird. Dieser Modus wird entweder durch einen Flankenwechsel an T2IN oder T4IN ausgelöst. In diesem Modus hält der entsprechende Timer (T2 oder T4) unabhängig vom entsprechenden Run Bit. Sowohl beim Reload als auch beim Capture kann ein Interrupt Request generiert werden indem das entsprechende Bit T2IR bzw. T4IR gesetzt wird. Die beiden Timer T2 und T4 sind ähnlich zu T3 aufgebaut. Beide können jeweils mit T3 über das Output Toggle Latch zu einem 32 oder 33-Bit breiten Timer zusammengefügt werden. Im Unterschied zu T3 besitzt weder T2 noch T4 ein derartiges Toggle Latch und auch keinen zugeordneten Ausgangsport.

---

[k] Achtung! Hierfür muß der Port P3.3 zuvor mit einer ‘1‘ initialisiert werden.

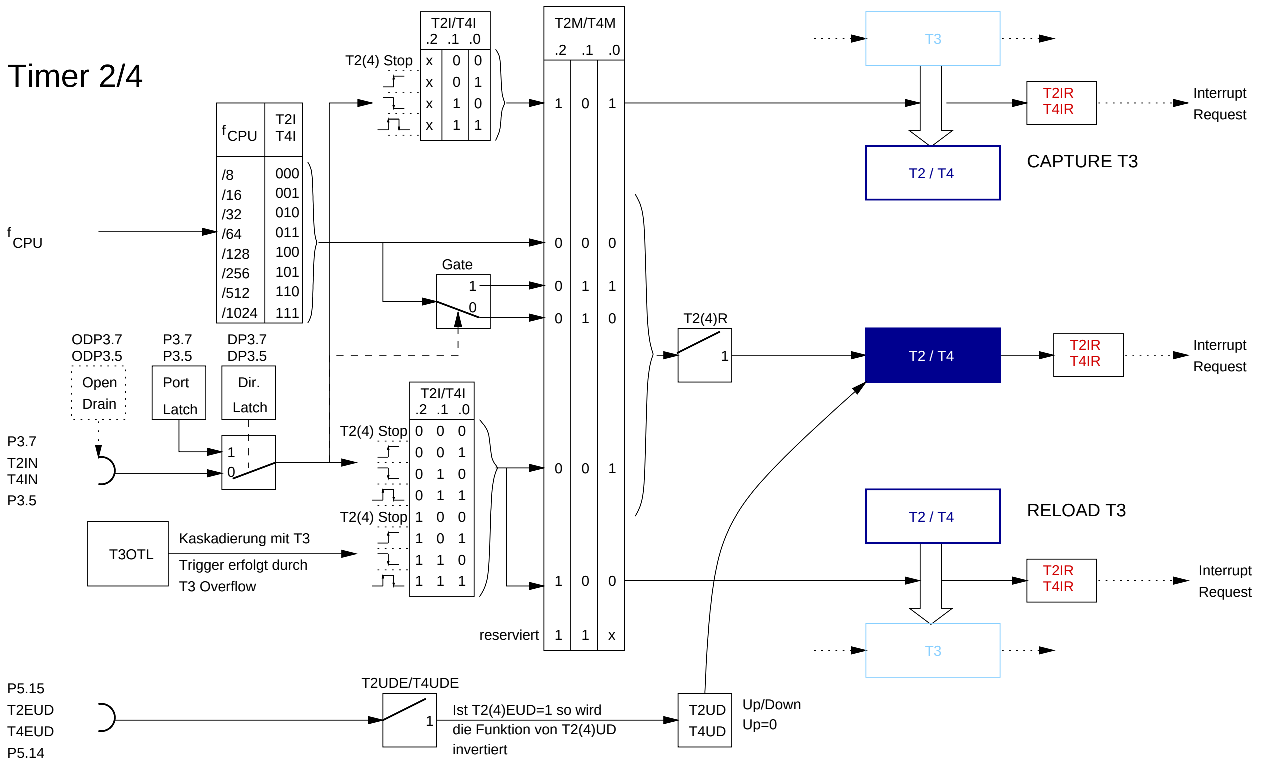


Abbildung 24: Funktionsschaltbild der Timer T2 und T4

2.8.2 Beispiel - Lauflicht mit dem Timer Block GPT1

Aufgabenstellung: Mit einem an Port P2.0-P2.8 angeschlossenen LED Balken ist ein Lauflicht zu generieren. Der Anschluß des LED Balkens erfolgt analog zu Abbildung 15.

Der folgende Programmcode illustriert eine Lösung, die aus einem Hauptprogramm und einer Interrupt Service Routine für Timer T3 besteht. Im Hauptprogramm werden zunächst die Ausgangsports initialisiert, sprich Port P2.0-P2.8 wird auf Ausgang geschaltet und die extern angeschlossenen LEDs werden in einen sinnvollen Anfangszustand gesetzt. Danach wird T3 als aufwärtszählender Timer initialisiert und das zugehörige T3 Register wird mit einem Anfangswert programmiert, sodaß die Periode (Anfangswert bis Maximalwert) den geforderten 250ms entspricht. Nach einem Überlauf würde T3 bei 0x0000 weiterzählen. Um jedoch für die folgenden Intervalle die gleiche Periodendauer zu erhalten wird Timer T2 als Reload Register für Timer T3 konfiguriert und mit demselben Anfangswert programmiert. D.h. nach einem Überlauf von T3 inkrementiert dieser nicht beginnend bei 0x0000 sondern ab dem Reload Wert von T2. Schließlich wird für T3 ein sinnvoller Interrupt Level programmiert und lokal freigegeben. Im Anschluß erfolgt eine globale Interruptfreigabe und Timer T3 wird gestartet. Ab diesem Zeitpunkt inkrementiert T3 in Abhängigkeit vom internen Takt beginnend vom Initialwert, der in das Register T3 programmiert wurde. Am Ende des Hauptprogrammes ist eine Endlosschleife erforderlich, in der die CPU abgeschaltet, d.h. in den Idle Modus versetzt wird.

Für Timer T3 wurde weiters eine Interrupt Service Routine aufgesetzt, die bei jedem Überlauf aktiviert wird. Die Zuordnung dieser Routine zu T3 erfolgt über die Schlüsselworte “interrupt 0x23”, wobei der Wert 0x23 entsprechend der Tabelle 3 statisch festgelegt ist. In dieser ISR wird das Bitmuster einer Laufvariable auf Port P2 ausgegeben, und auch entsprechend für das nächste Intervall neu berechnet.

```
/* Beispielprogramm Lauflicht */
#include <reg167.h>
#include <intrins.h>

int i;  /* globale Laufvariable */

/* Bei einem Ueberlauf des Timer 3 von 0xFFFF->Reload Wert
   wird diese ISR getriggert. */
void timer3_interrupt_service_routine (void) interrupt 0x23 {

     P2=1<<i;   /* Eine 1 wird um den Wert der Variablen i
                   nach links geshiftet. */
     if (i<7)   /* Diese Bedingung hält die Variable i */
        i++;    /* im Wertebereich zwischen 0 und 7 */
     else
        i=0;
}

void main (void) {
  i=0;          /* Initialisierung der Laufvariablen */
  DP2=0x00ff;   /* Port P2.0-P2.8 -> Ausgang */
  P2=0x0000;    /* Dunkelschaltung der LEDs */
  T3CON=0x0004; /* Pre-Scaler=128 -> Periode=420ms */
  T2CON=0x0027; /* Reload Mode bei jedem Flankenwechsel von T3OTL */
  T3=0x679F;    /* =0xFFFF-0x9860; 0x9860 == 250ms */
  T2=0x679F;    /* Reload Wert fuer T3 */
  T3IC=0x0047;  /* IE=1; ILVL=1; GLVL=3 */
  IEN=1;        /* globale Interruptfreigabe im PSW */
  T3R=1;        /* startet den Timer 3 */
  while(1){     /* Endlosschleife */
    _idle_();
  }
}
```

2.8.3 General Purpose Timer Unit 2

Diese Gruppe besteht aus den Timern T5 und T6 die ebenfalls miteinander kaskadiert werden können. Der Wert des Timers T5, siehe Abbildung 25, kann über die an einem externen Pin angelegte Flanke rückgesetzt bzw. in das CAPREL Register umkopiert werden. Dieser Timer kann im Gegensatz zu anderen Timern bei Überlauf nicht mit einem einstellbaren Wert automatisch neu geladen werden.

Sowohl bei einem Überlauf als auch bei einem Capture von T5 in das CAPREL Register kann ein entsprechender Interrupt Request generiert werden.

Der Timer T6, siehe Abbildung 26, kann bei einem Überlauf über das CAPREL Register neu geladen werden. Timer T6 verfügt zusätzlich über ein Output Toggle Latch wie Timer T3. Über

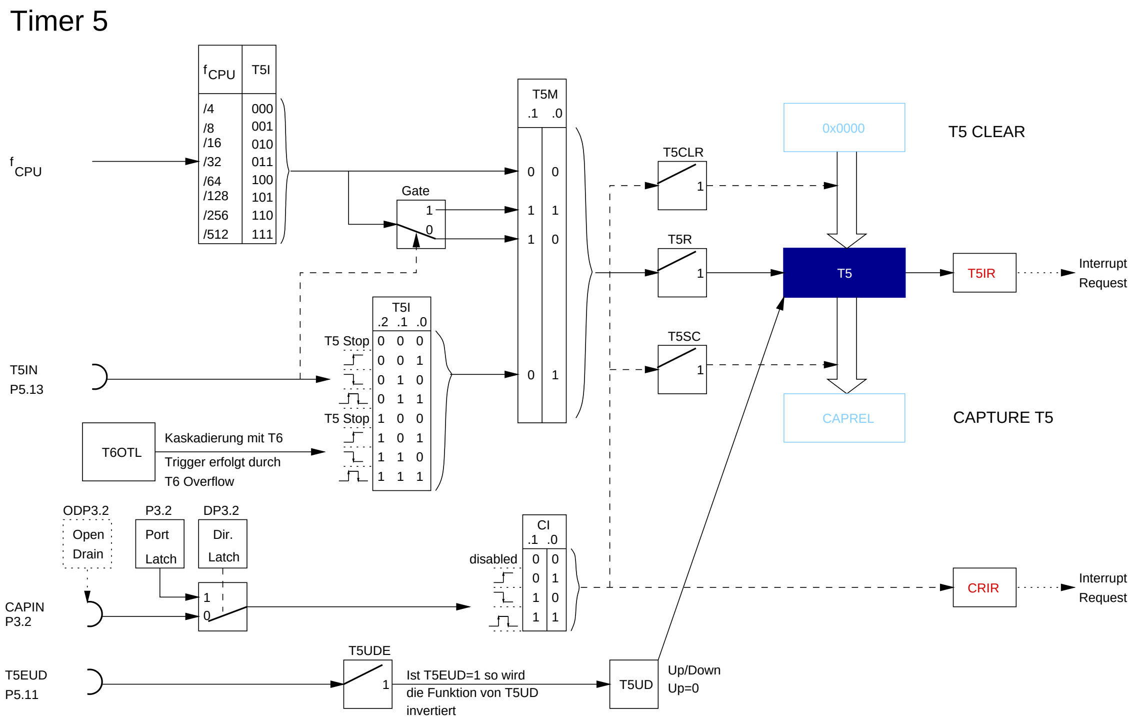


Abbildung 25: Blockschaltbild zu Timer T5

dieses Latch kann Timer T5 mit dem Timer T6 kaskadiert werden. Der direkte Ausgang von Timer T6 kann als ein Eingang für die CAPCOM Timer T0, T1, T7 und T8 Verwendung finden.

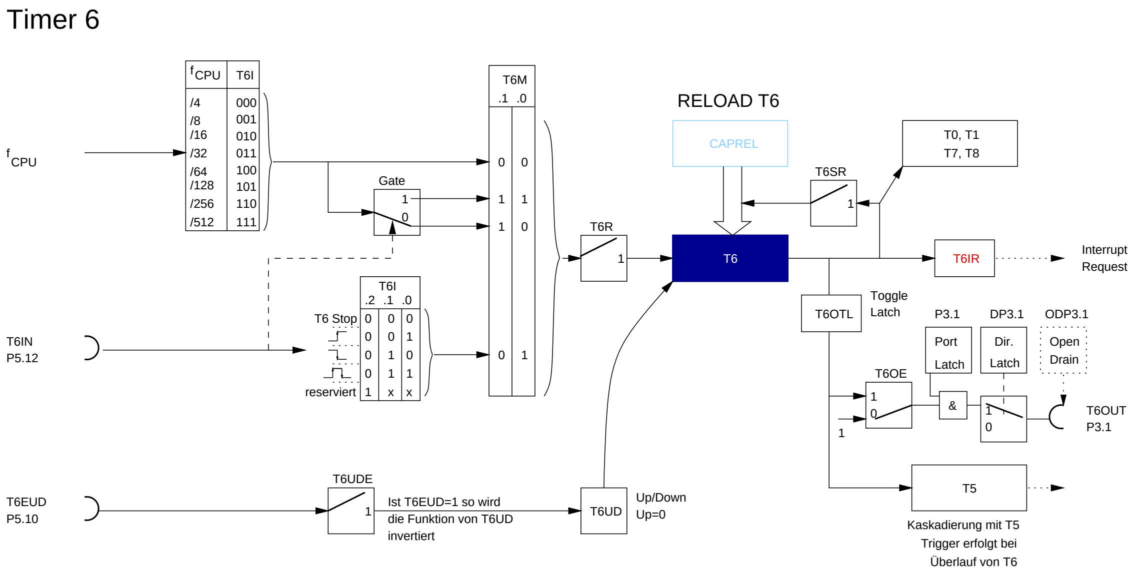


Abbildung 26: Blockschaltbild zu Timer T6

2.8.4 Beispiele zu GPT2

Aufgabenstellung: Am CAPIN Eingang liegt ein digitales Signal an. Es soll ein Programm realisiert werden, welches mit der feinsten Auflösung von Timer T5 die Zeitdauer zwischen zwei aufeinanderfolgenden Flanken ermittelt. Beträgt die Meßdauer mehr als 5 Sekunden, so soll die Messung abgebrochen werden.
Die folgende Realisierung verwendet den CAPIN Eingang in Kombination mit dem Timer T5. Bei jeder steigenden oder fallenden Flanke an diesem Eingang wird gleichzeitig der aktuelle Wert des Timers T5 in das CAPREL Register gesichert, Timer T5 wird auf den Initialwert 0x0000 rückgesetzt und der Interrupt Request CRIR wird ausgelöst. In der entsprechenden ISR wird beim ersten Aufruf Timer T5 gestartet und die Variable zum Zählen von T5 Perioden (zur Überwachung der Meßdauer von 5 Sekunden) wird initialisiert. Bei jedem Überlauf von Timer T5 wird diese Variable inkrementiert, solange bis die 5 Sekunden Meßdauer verstrichen sind, danach wird die Messung abgebrochen und Timer T5 angehalten. Tritt ein zweiter Flankenwechsel an CAPIN auf bevor die Meßdauer abgelaufen ist, so werden der Stand des Timers T5 aus dem CAPREL Register, und die Anzahl der verstrichenen T5 Perioden gesichert. Danach wird ebenso Timer T5 angehalten und die Messung wird beendet. Die Zeitdauer zwischen den beiden Flanken ergibt sich somit aus $(periodenanzahl * 65535 + impulsanzahl) * 200ns$.

```
/* Ausmessen der Zeitdauer zwischen zwei aufeinanderfolgenden Flanken. */
/* Die Messung wird spätestens nach 5 Sekunden beendet. */
#include <reg167.h>
#include <intrins.h>
int i;                    /* Zählt die Periodenanzahl von Timer T5. */
unsigned char flankennr;  /* unterscheidet zw. 1ter und 2ter Flanke */
int impulsanzahl;         /* ermittelt die Anzahl der Pulse in der
                             aktuellen Timer Periode */
int periodenanzahl;       /* zählt die Anzahl an abgelaufenen Perioden von T5 */
sbit DP32 = DP3^2;        /* Richtung für CAPIN */
sbit DP20 = DP2^0;        /* Richtung für GPIO Pin zur LED Steuerung */
sbit P20 = P2^0;          /* GPIO Pin zu LED Steuerung */

void t5isr (void) interrupt 0x25 {
        if (i<382)      /* Sind 5 Sekunden Messzeit abgelaufen? */
           i++;         /* Nein! -> Inkrementiere i */
        else {
           P20=0;       /* Ja!  -> Schaltet die LED dunkel da die
                           Messung abgebrochen wurde. */
           T5R=0;      /* T5 Stop */
        }
}

void crisr (void) interrupt 0x27 {
        if (flankennr==0) {     /* erste Flanke? */
                T5R=1;          /* Startet Timer T5 */
                P20=1;          /* Ausgangs LED hell -> Meßbeginn */
                flankennr=1;    /* JA -> es wird nur noch eine weitere Flanke
                                   erwartet */
                i=0;            /* Der Zähler für die T5 Perioden wird
```

```
                                rückgesetzt */
        }
        else {                         /* zweite Flanke! */
                impulsanzahl = CAPREL; /* Sichern des aktuellen T5 Timer Standes */
                periodenanzahl = i;    /* Sichern der verstrichenen Periodenanzahl */
                flankennr=0;           /* -> bereit für eine weitere Messung */
                i=382;                 /* i auf Wert>5sec. setzen */
                P20=0;                 /* LED dunkel */
                T5R=0;                 /* T5 Stop */
        }
}
void main (void) {
        flankennr=0;    /* 0...es ist noch keine Flanke aufgetreten */
        i=382;          /* 382 * 13.107ms (T5 Periodendauer) > 5s */
        DP20=1;         /* Setzt die Richtung von DP2.0 auf Ausgang. */
        P20=0;          /* Die an P2.0 angeschlossene LED leuchtet nicht da
                           die Messung noch nicht aktiv ist. */
        DP32=0;         /* Setzt die Richtung von DP3.2 auf Eingang. */
        T5CON=0xF000;   /* Timer T5 Konfiguration:
                           T5SC=1 : Capture into Register CAPREL enabled
                           T5CLR=1 : Timer T5 is cleared on a capture
                           Timer Mode, Count Up, Prescaler=4 d.h.
                           feinste Auflösung (200ns) */
        T5IC=0x0045;    /* T5IE=1; ILVL=1; GLVL=1; */
        CRIC=0x0044;    /* CRIE=1; ILVL=1; GLVL=0; */
        IEN=1;          /* globale Interruptfreigabe */
        while(1) {      /* Endlosschleife */
          _idle_();     /* Abschalten der CPU -> Strom sparen */
        }
}
```

Aufgabenstellung: In dem zweiten Beispiel soll die Frequenz des am CAPIN Eingang anliegenden Signals vervielfacht in seiner Frequenz über den Ausgang von T6 (T6OUT) ausgegeben werden. Timer T5 wird bei dieser Realisierung derart konfiguriert, daß eine steigende Flanke am CAPIN Eingang die Sicherung von T5 in das CAPREL Register und gleichzeitig eine Initialisierung von T5 mit 0x0000 bewirkt. Nach einer kurzen Initialisierungsphase enthält dann das CAPREL Register den Wert der Periodendauer des Eingangssignals. Der hier angeführte Lösungsvorschlag setzt jedoch voraus, daß diese Periodendauer kürzer ist als jene von T5 und entsprechend länger als die Auflösung von T5. Für diese Fälle müßte das Programm entsprechend erweitert werden. Der Wert im CAPREL Register dient in der Folge dem Timer T6 als Reload Wert. T6 zählt mit einer höheren Frequenz jeweils von diesem Wert abwärts auf 0x0000. Bei jedem Underflow wird der Ausgang von T6 getoggelt. D.h. der Wert der Frequenzvervielfachung wird in dieser Realisierung über das Geschwindigkeitsverhältnis von Timer T5 zu T6 eingestellt. Ändert sich die Periode des Eingangssignals, so ändert sich auch der zugehörige Wert des CAPREL Registers und entsprechend über T6 die Periode des Ausgangssignals.

```
/* Frequenz Vervielfachung */
#include <reg167.h>
#include <intrins.h>
```

```
sbit DP32=DP3^2;        /* Richtung des Ports P3.2 */
sbit DP31=DP3^1;        /* Richtung des Ports P3.1 */
sbit P31=P3^1;          /* Port3.1 */

void main (void) {
        DP32=0;         /* CAPIN ist der Signaleingang */
        DP31=1;         /* Port 3.1 dient als Ausgang für T6 */
        P31=1;          /* Der Port muß einmal beschrieben werden damit
                           der alternate Ausgang T6OUT auch nach
                           außen gelangt. */
        T5CON=0xD004;   /* T5SC=1: Capture -> Register CAPREL
                           T5CLR=1: T5 -> 0x0000 bei Capture
                           CI=01: Capture erfolgt bei der steigenden
                                  Flanke von CAPIN;
                           count up; Timer Mode; Pre-Scaler=64 */
        T6CON=0x8682;   /* T6SR=1: Reload über CAPREL
                           T6OE=T6OTL=1; Ausgang und Toggle Latch aktiv
                           count down; Timer Mode; Pre-Scaler=16 */
        T5R=1;          /* Starte Timer T5 */
        T6R=1;          /* Starte Timer T6 */
        while(1){       /* Endlosschleife */
          _idle_();     /* Stromspar Modus */
        }

}
```

2.9 Capture Compare Unit (CAPCOM)

Die Capture Compare Unit des C167 dient zum Erfassen und Vergleichen von externen, digitalen Signalzuständen. Ebenso können mit diesem Modul recht elegant Pulsmuster erzeugt werden. Dieses Modul ist in zwei funktionelle Gruppen unterteilt wobei der CAPCOM Unit 1 die Timer T0 und T1 und der CAPCOM Unit 2 die Timer T7 und T8 zugeordnet sind. In Abbildung 27 ist ein funktionelles Blockschaltbild des Timer T0 dargestellt. Timer T0 kann als Timer oder Counter verwendet werden und verfügt über ein eigenes Reload Register. Der Timer T1, siehe Abbildung 28

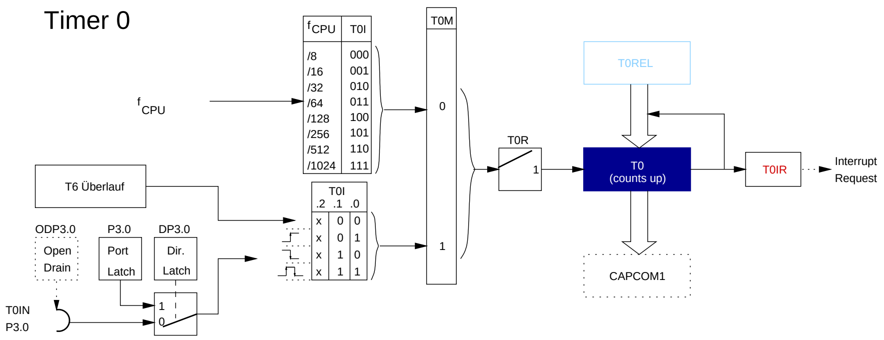


Abbildung 27: Timer T0

,kann nur als Timer verwendet werden, verfügt jedoch ebenso wie Timer T0 über ein eigenes Reload Register. Der Timer T7 ist analog zu Timer T0 und T8 analog zu T1 ausgeführt. Weiters können alle Timer der Capture Compare Unit mit Timer T6 kaskadiert werden.

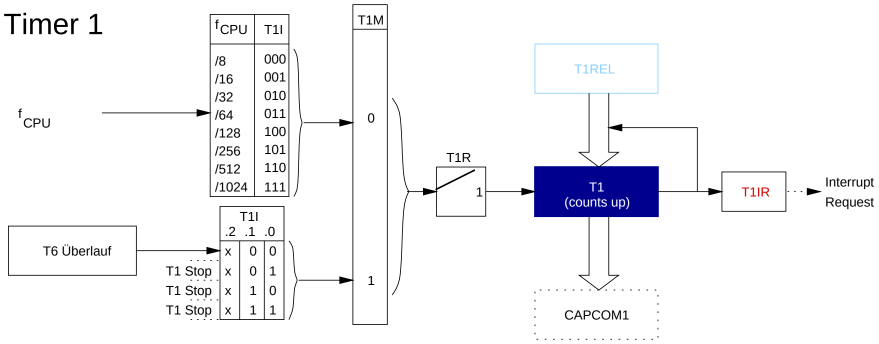


Abbildung 28: Timer T1

Beide Units verfügen über jeweils 16 Capture/Compare Register, wobei jedes dieser Register über einen separaten I/O Pin verfügt. Jedes kann individuell für den Capture oder einen der Compare Modi programmiert werden. Darüberhinaus kann jedes Register einen Timer der jeweiligen Unit verwenden um die jeweilige Funktion zu erfüllen. Um diese unabhängige Gestaltung der einzelnen Register auch voll nutzen zu können ist jedem Register explizit ein Interrupt Request zugeteilt.

2.9.1 Capture Mode

Im Capture Mode wird bei Auftreten eines Flankenwechsels — wobei die Flankenart programmiert werden kann — der Inhalt des zugeordneten Timers in das entsprechende Capture Register kopiert. Hierbei wird zusätzlich ein Interrupt Request generiert.

2.9.2 Compare Mode 0

Im Compare Mode 0 wird jedesmal ein Interrupt Request generiert, wenn der Wert des entsprechend zugeordneten Timers mit dem Wert im Compare Register übereinstimmt. Innerhalb einer Periode kann der Wert des Compare Registers öfters umgesetzt werden, sodaß mehrere Request innerhalb einer Timer Periode ausgelöst werden können. In der Abbildung 29 wird z.B. der Compare Wert cv1 in der ISR jeweils auf den Wert cv2 und anschließend wieder auf den Wert cv1 umgesetzt.

2.9.3 Compare Mode 1

Der Compare Mode 1 entspricht funktionell dem Mode 0, nur wird zusätzlich der zugeordnete Ausgangspin bei jedem Event getoggelt. Dieser Modus wird in der Abbildung 29 veranschaulicht.

2.9.4 Compare Mode 2

Im Compare Mode 2 wird nur ein Interrupt innerhalb einer Timer Periode generiert. Dies ist jeweils bei der ersten Gleichheit zwischen dem Compare Register und dem Timer Wert der Fall.

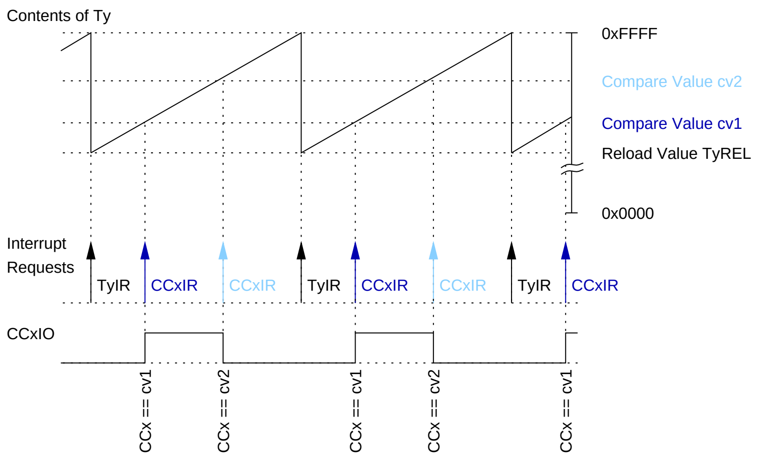
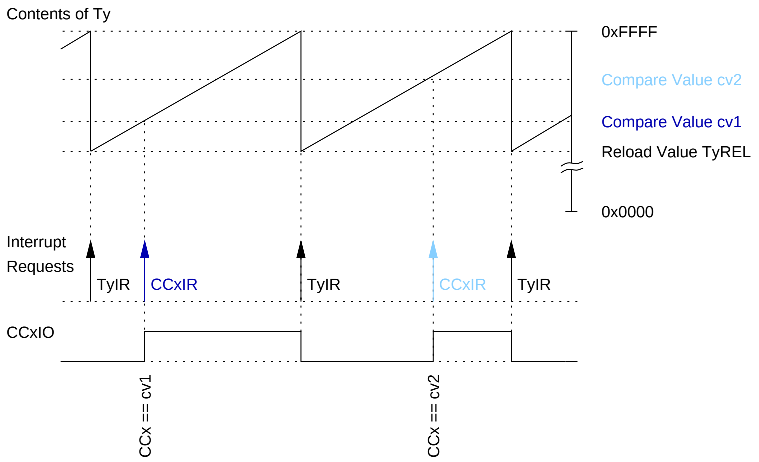


Abbildung 29: Compare Mode 1



Anmerkung: Nach dem ersten Vergleich wird der Compare Value des CCx Registers von cv1 auf cv2 umgesetzt. Dies wirkt sich erst in der zweiten Timer Periode aus.

Abbildung 30: Compare Mode 3

2.9.5 Compare Mode 3

Wie im Mode 2 wird im Compare Mode 3 verfahren, nur wird hier zusätzlich bei der ersten Gleichheit des Registerwertes mit dem Timerinhalt der zugehörige Ausgangsport auf ‘1‘ gesetzt. Bei einem Überlauf des Timers wird dieser wieder auf ‘0‘ rückgesetzt. Dies wird auch in der Abbildung 30 illustriert.

2.9.6 Double Register Compare Mode

Double Register Compare ist ein weiterer Compare Mode in dem ein zugeordnetes Registerpaar jeweils einen Ausgangspin steuert. Welche Register in diesem Fall zusammenwirken ist aus Tabelle 6 ersichtlich. Wird z.B. ein Register der Bank 1, programmiert auf den Mode 1, mit dem

CAPCOM Unit 1			CAPCOM Unit 2		
Register Pair		Associated	Register Pair		Associated
Bank 1	Bank 2	Output Pin	Bank 1	Bank 2	Output Pin
CC0	CC8	CC0IO	CC16	CC24	CC16IO
CC1	CC9	CC1IO	CC17	CC25	CC17IO
CC2	CC10	CC2IO	CC18	CC26	CC18IO
CC3	CC11	CC3IO	CC19	CC27	CC19IO
CC4	CC12	CC4IO	CC20	CC28	CC20IO
CC5	CC13	CC5IO	CC21	CC29	CC21IO
CC6	CC14	CC6IO	CC22	CC30	CC22IO
CC7	CC15	CC7IO	CC23	CC31	CC23IO

Tabelle 6: Registerzuordnung für den Double Register Compare Mode

Wert cv1 (CC2 = 0x2000) geladen. Das entsprechende Register der Bank 2, programmiert auf den Mode 0, wird mit dem Wert cv2 (CC10 = 0x5000) beschrieben, dann wird der Ausgangspin bei jeder Übereinstimmung des Timerwertes mit einem der beiden Registerwerte getoggelt. Hiermit können elegant Pulse mit definierter Länge erzeugt werden, wie dies auch in der Abbildung 31 schematisiert ist.

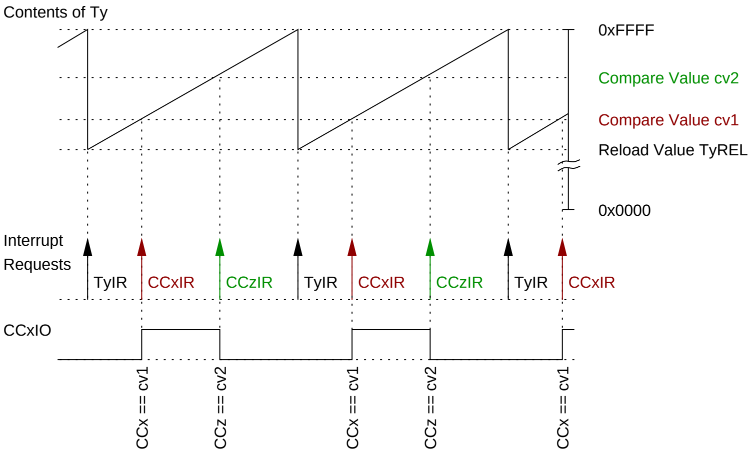


Abbildung 31: Double Register Compare Mode

2.9.7 Beispiele zur Capture/Compare Einheit

Aufgabenstellung: Über einen Capture/Compare Eingang sollen die Flankenwechsel eines digitalen Signals in ihrer zeitlichen Abfolge erfaßt werden.

Die Realisierungsidee zu diesem Beispiel soll an Hand von Abbildung 32 illustriert werden. Das Capture Register CC0 wird dem Timer T0 zugeordnet und nimmt bei jedem Flankenwechsel am zugehörigen Pin den Wert von T0 auf. Bei jedem Capture Event (steigende oder fallende Flanke) wird im Anschluß über einen PEC Transfer der gesicherte Wert von T0 im CC0 Register in ein

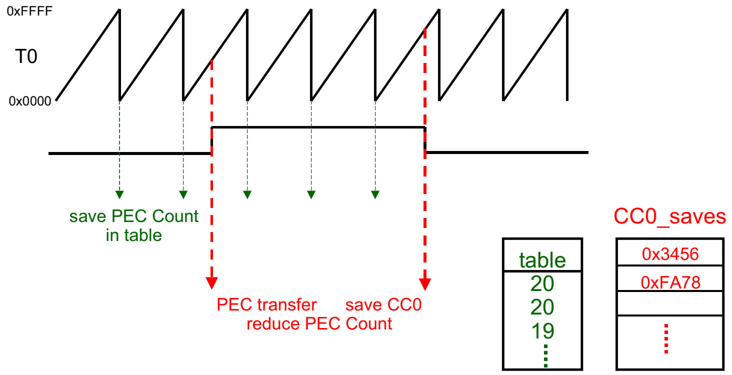


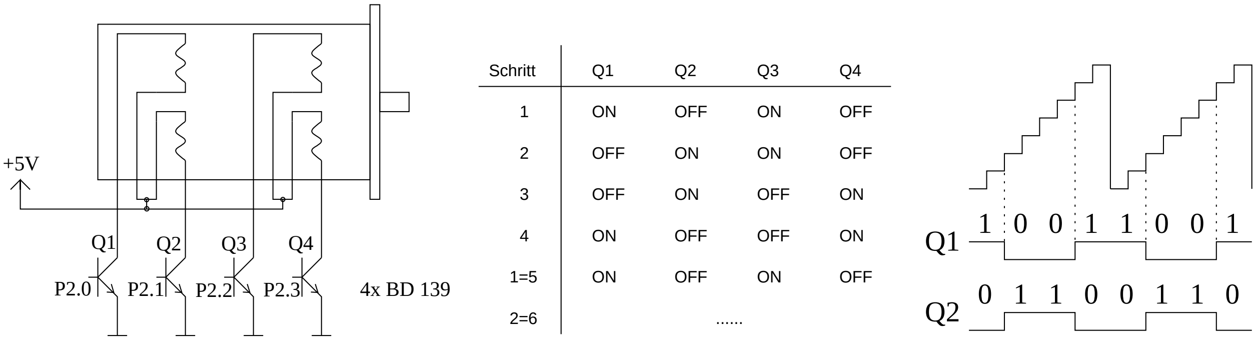
Abbildung 32: Beispiel zeitliche Flankenerfassung

Variablenfeld umkopiert. Der PEC, der dem Capture Interrupt zugeordnet ist, dekrementiert nach jedem Transfer den Zähler für die Anzahl an erfaßten Flanken und inkrementiert gleichzeitig den *Destination Pointer*, d.h. der folgende PEC Transfer wird in das Speicherfeld mit dem nächst höheren Index transferiert. Damit die zeitliche Erfassung sehr feinkörnig erfolgen kann, wird die Zeitbasis für T0 auf den niedrigsten Wert gesetzt. Treten sämtliche zu erfassenden Flankenwechsel jedoch nicht innerhalb einer Timer Periode von T0 auf, so muß die Anzahl der T0 Überläufe und der Wert des PEC Counters in der ISR gesichert werden. Daraus ergeben sich zwei Tabellen, wie Sie in Abbildung 32 skizziert sind. An Hand der Kombination dieser beiden Tabellen lassen sich recht einfach die Zeitpunkte der einzelnen Flankenwechsel rekonstruieren. Veranschaulicht an dieser Skizze ergibt dies für den ersten Flankenwechsel $(65535*2+0x3456)*400ns$. Wurde die definierte Anzahl an PEC Transfers durchgeführt, so wird, wenn der PEC Counter den Wert 1 erreicht hat, die entsprechende ISR unmittelbar nach dem letzten PEC Transfer aktiviert. Nach der festgelegten Anzahl an Flankenwechsel bzw. wenn die Meßzeit verstrichen ist terminiert die Messung indem Timer T0 angehalten wird. Die Funktionalität und das Aufsetzen eines PEC Transfers wurde bereits detailliert in Abschnitt 2.4 beschrieben.

```
#pragma pecdef (0)      /* PEC pointer reservieren */
#include <reg167.h>     /* Register Definitionen des C167 */
#include <intrins.h>    /* bausteinspezifische Funktionen */

/* definiert die maximale Anzahl an zu erfassenden
   Flankenwechseln */
#define MAX_FLANKEN_ANZAHL 10
/* definiert die maximale Anzahl an Timer T0 Perioden
   die für die Messung verwendet werden soll */
#define MAX_MESSZEIT 15

int i;
sbit DP20=DP2ˆ0;                /* Richtung von P2.0 */
sbit P20=P2ˆ0;                  /* P2.0; hier liegt das Digitalsignal an */
unsigned char startpegel;       /* Speichert den Startpegel des Signals */
/* Dieses Feld speichert die Timer T0 Zeitstempel der auftretenden Flanken */
int flanke[MAX_FLANKEN_ANZAHL];
/* Dieses Feld legt die Anzahl der Timer T0 Überläufe fest. */
```

```
unsigned char t0_overrun[MAX_MESSZEIT];

/* T0 Interrupt Service Routine wird bei jedem Überlauf von T0 aktiviert. */
void t0isr (void) interrupt 0x20 {
        if (i>MAX_MESSZEIT)             /* Messzeit abgelaufen? */
           T0R=0;                       /* Ja. Stoppe Timer T0 */
        else {                          /* Nein. Sichere den Wert des PEC Counters */
           t0_overrun[i]=PECC0 & 0x00FF;
           i++;                         /* Zähler für die Messzeit inkrementieren */
        }
}

/* Capture/Copmpare 0 Interrupt Service Routine; wird aktiv wenn der vorherige
   PEC Counter = 1 war. */
void cc0isr (void) interrupt 0x10 {
        T0R=0;                          /* Stoppe Timer T0 */
}

/* Dieses Unterprogramm initialisiert den PEC Transfer und startet Timer T0 */
void messung (void) {
        /* Inkrementiere Destination Pointer DSTPx um 2 ->
           Word Transfer; Führe "MAX_FLANKEN_ANZAHL" Transfers durch.
        */
        PECC0=0x0200 | (MAX_FLANKEN_ANZAHL & 0x00ff);
        SRCP0=_sof_(&CC0);      /* Source Adresse ist das CC0 Register */
        DSTP0=_sof_(flanke);    /* Destination Adresse ist das Feld "flanke" */
        startpegel=P20; /* Anfangspegel des Signals sichern */
        T0R=1;          /* Starte Timer T0 */
}

void main (void) { /* Das Hauptprogramm */
        for (i=0;i<MAX_FLANKEN_ANZAHL;i++) /* Initialisierung */
                flanke[i]=0;
        for (i=0;i<MAX_MESSZEIT;i++)
                t0_overrun[i]=0;
        i=0;    /* Setzt den Counter für die Messzeit initial auf 0 */
        DP20=0; /* Richtung von P2.0 -> Eingang */
        T01CON=0x0000;  /* Timer Mode für T1 und T0
                           Pre-Scaler für beide 8
                           d.h. Auflösung 400ns */
        CCM0=0x0003;    /* CC0 -> Capture bei jeder Flanke an CCxIO
                           zugeordnet zu T0 */
        CC0IC=0x0078;   /* CC0IE=1; ILVL=14; GLVL=0; */
        T0IC=0x007A;    /* T0IE=1; ILVL=14; GLVL=2; */
        messung();      /* Initialisiere die Messung */
        IEN=1;          /* globale Interruptfreigabe */
        while(1){       /* Endlosschleife */
           _idle_();    /* Stromsparmodus */
        }
}
```

Aufgabenstellung: Im zweiten Beispiel soll ein Schrittmotor im Vollschrittmodus angesteuert werden. Der Anschluß und die Ansteuerung des Motors hierfür ist der Abbildung 33 zu entnehmen.



Schritt	Q1	Q2	Q3	Q4
1	ON	OFF	ON	OFF
2	OFF	ON	ON	OFF
3	OFF	ON	OFF	ON
4	ON	OFF	OFF	ON
1=5	ON	OFF	ON	OFF
2=6		......		

Abbildung 33: Schrittmotor im Vollschrittmodus

In der Tabelle ist die Ansteuerung der einzelnen Wicklungen für aufeinanderfolgende Schritte in Linksrichtung wiedergegeben. D.h. für Linksrichtung müssen die Transistoren mit einem Pulsmuster entsprechend der Schritte 1-4 angesteuert werden um eine volle Umdrehung der Achse des Schrittmotors in dieser Richtung zu bewirken. Für weitere Umdrehungen wiederholt sich dieses Ansteuerschema in der gleichen Reihenfolge. Für Rechtsdrehung ist die Reihenfolge einfach umzudrehen, d.h. die Schritte müssen in der Reihenfolge 4, 3, 2, 1, 4, 3, 2, . . . durchlaufen werden. Die Ansteuerung in dem gezeigten Beispiel wird mit Hilfe der CAPCOM Einheit vorgenommen. Für die Wicklungen Q1 und Q2 ist dies exemplarisch in Abbildung 33 schematisiert dargestellt. Die CAPCOM Unit wird hierfür im Double Register Compare Modus betrieben. Q1 wird über den Pin CC0IO gesteuert, der wiederum den Registern CC0 und CC8 zugeordnet ist. Durch entsprechendes setzen der Compare Schwellwerte, wird am Ausgang ein Pulsmuster wie in Abbildung 33 rechts gezeigt realisiert. Dieses Bitmuster entspricht den Ansteuerungen der einzelnen Wicklungen des Schrittmotors für die Schritte 1-4. Indem die Periode und die Schwellwerte entsprechend verlängert oder verkürzt programmiert werden, läßt sich der Motor schneller/langsamer betreiben. Im angeführten Programm wird zunächst die entsprechende CAPCOM Unit initialisiert. Mit den Unterprogrammen *smot_links*() bzw. *smot_rechts*() werden die Schwellwerte entsprechend obigem Schema gesetzt und der zuständige Timer wird gestartet. In der ISR von Timer T0 wird die Anzahl der Umdrehungen überwacht und nach Erreichen der programmierten Schrittanzahl wird T0 gestoppt.

```
/* Schrittmotoransteuerung im Vollschrittmodus */
#include <reg167.h>     /* Register Definitionen des C167 */
#include <intrins.h>    /* bausteinspezifische Funktionen */

int inc,i,reload_value;
void smotor_links (void) {
        P2=0x0006;  /* Initialisiert die Ausgangspins CCxIO */
        /* Der folgende Block initialisiert CCx für Double
           Register Compare Modus für Linkslauf - bei Gleichheit des
           Registerwertes mit dem Timer T0 wird der zugehörige
           Ausgangspin getoggelt. */
        /* CC0IO -> CC0 und CC8 */
        CC0=reload_value+(0xffff-reload_value)*(1.0/4.0);
        CC8=reload_value+(0xffff-reload_value)*(3.0/4.0);
```

```
        /* CC1IO -> CC1 und CC9 */
        CC1=reload_value+(0xffff-reload_value)*(3.0/4.0);
        CC9=reload_value+(0xffff-reload_value)*(1.0/4.0);
        /* CC2IO -> CC2 und CC10 */
        CC2=reload_value;
        CC10=reload_value+(0xffff-reload_value)*(2.0/4.0);
        /* CC3IO -> CC03 und CC11 */
        CC3=reload_value+(0xffff-reload_value)*(2.0/4.0);
        CC11=reload_value;
        i=0;            /* Der Schrittzähler wird rückgesetzt */
        T0R=1;          /* starte T0 */
}
void smotor_rechts (void) {
        P2=0x000A;  /* Initialisiert die Ausgangspins CCxIO */
        /* Der folgende Block initialisiert CCx für Double
           Register Compare Modus für Rechtslauf - bei Gleichheit des
           Registerwertes mit dem Timer T0 wird der zugehörige
           Ausgangspin getoggelt. */
        /* CC0IO -> CC0 und CC8 */
        CC0=reload_value+(0xffff-reload_value)*(3.0/4.0);
        CC8=reload_value+(0xffff-reload_value)*(1.0/4.0);
        /* CC1IO -> CC1 und CC9 */
        CC1=reload_value+(0xffff-reload_value)*(1.0/4.0);
        CC9=reload_value+(0xffff-reload_value)*(3.0/4.0);
        /* CC2IO -> CC2 und CC10 */
        CC2=reload_value+(0xffff-reload_value)*(2.0/4.0);
        CC10=reload_value;
        /* CC3IO -> CC03 und CC11 */
        CC3=reload_value;
        CC11=reload_value+(0xffff-reload_value)*(2.0/4.0);
        i=0;            /* Der Schrittzähler wird rückgesetzt */
        T0R=1;          /* starte T0 */
}

/* Die Timer T0 Interrupt Service Routine wird bei jedem
   Überlauf von t0 von 0xFFFF -> reload_value aktiviert. */
void timer0_isr (void) interrupt 0x20 {
        if (i<inc-1)    /* Solange nicht alle Schritte ausgeführt
                           sind wird nur i --der Schrittzähler--
                           inkrementiert */
           i++;
        else {
           T0R=0;       /* inc Schritte gemacht! -> T0 Stop */
           i+=2;
        }
}
void main (void) {
        DP2|=0x000f;            /* Port P2.0-3: Ausgang */
        T01CON=0x0006;          /* beeinflußt die Geschwindigkeit */
        T0IC=0x45;              /* IE=1; ILVL=1; GLVL=1 */
        /* Double Register Compare Mode */
        CCM0=0x5555;            /* CC0-3: Compare Mode 1 */
        CCM2=0x4444;            /* CC8-11: Compare Mode 0 */
```

```
        i=0;                    /* Zähler für die Anzahl der Schritte */
        inc=2;                  /* Anzahl der Umdrehungen */
        reload_value=0x3;       /* Reload Wert für Timer Null - beeinflußt
                                   die Geschwindigkeit */
        T0=T0REL=reload_value;  /* initialisert T0 und das Reload Register */
        IEN=1;                  /* globale Interruptfreigabe */
        smotor_links();         /* Linkslauf; startet T0 und setzt CCx */
        while (i<inc)           /* warte bis alle Schritte gemacht sind */
           ;
        smotor_rechts();        /* Rechtslauf; startet T0 und setzt CCx */
        while (i<inc)           /* warte bis alle Schritte gemacht sind */
           ;
        while(1){               /* Endlosschleife */
          _idle_();             /* Stromsparmodus */
        }
}
```

2.10 Pulse Width Modulation (PWM) Unit

Der C167 verfügt über vier voneinander unabhängige PWM Kanäle mit einer eigenen Zeitbasis. Die maximale Auflösung beträgt hierbei 50 ns bei einer CPU Taktfrequenz von 20 MHz. Diese Kanäle können in vier verschiedenen Betriebsarten betrieben werden, die über die Register PWMCON0 und 1 konfigurierbar sind. Diese sollen im folgenden im Zusammenhang mit dem Blockschaltbild in Abbildung 34 etwas genauer beschrieben werden.

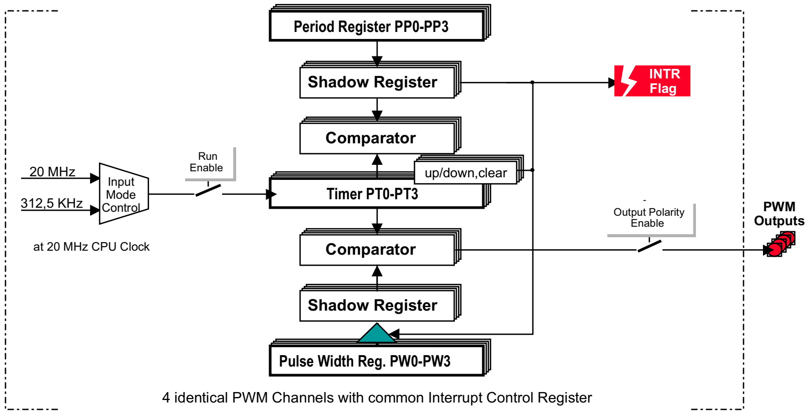


Abbildung 34: PWM Blockschaltbild

2.10.1 Standard PWM - Edge Aligned (Mode 0)

Der entsprechende Timer PTx wird ständig inkrementiert. Wird der Wert des entsprechenden *Period Shadow Registers* (zugehörig zu PPx) erreicht, so wird der Timer im nächsten Taktschritt auf 0x0000 zurückgesetzt. Der zugehörige Ausgang des PWM Kanals wird auf ‘1‘ gesetzt, wenn der

Wert des Timers größer/gleich dem Wert des *Pulse Width Shadow Registers* (zugehörig zu PWx) ist. Beim Rücksetzen des Timers PTx auf 0x0000 wird gleichzeitig auch der Ausgangsport auf ‘0’ retourgesetzt. In Abbildung 35 ist ein Beispiel für eine derartige PWM wiedergegeben.

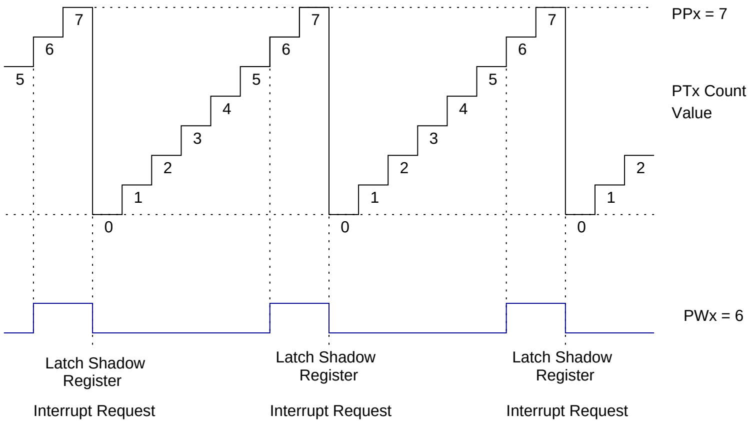


Abbildung 35: Standard PWM

2.10.2 Symmetrische PWM (Mode 1)

In dieser Betriebsart zählt der Timer aufwärts bis er den Wert im zugehörigen Period Shadow Register erreicht. Ist dies der Fall so wird die Zählrichtung invertiert bis der Zähler den Wert 0x0000 erreicht. Bei diesem Wert wird die Zählrichtung abermals invertiert. Ein symmetrisches Ausgangssignal wird erreicht indem der Ausgang auf ‘1‘ gesetzt wird, solange der Wert des Counters größer/gleich dem Wert des Pulse Width Shadow Registers ist, siehe hierzu Abbildung 36.

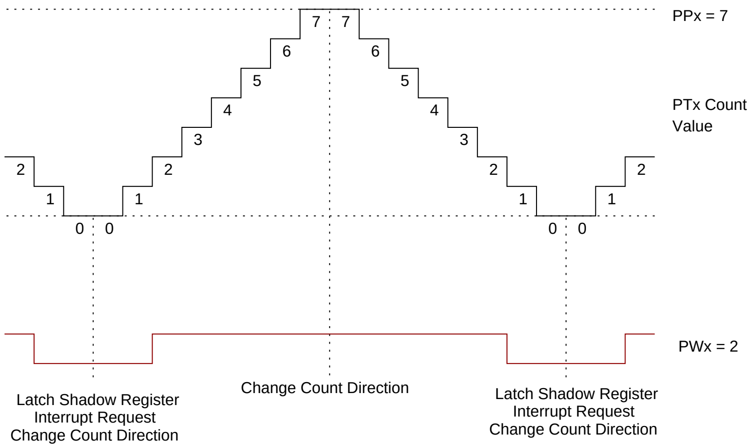


Abbildung 36: Symmetrische PWM

2.10.3 Burst Mode

Die Signale von den PWM Kanälen 0 und 1 werden auf den Ausgangsport von Kanal 0 über eine logische UND Verknüpfung hinausgeführt. Der Ausgang von Kanal 0 ist durch diese Modifikation selbst nicht betroffen und kann in seiner ursprünglichen Funktion weiter verwendet werden. In diesem Modus können beide Kanäle sowohl entweder im Modus 0 oder aber auch im Modus 1 betrieben werden. Eine Signalform die dieser Betriebsart entspricht ist in Abbildung 37 dargestellt.

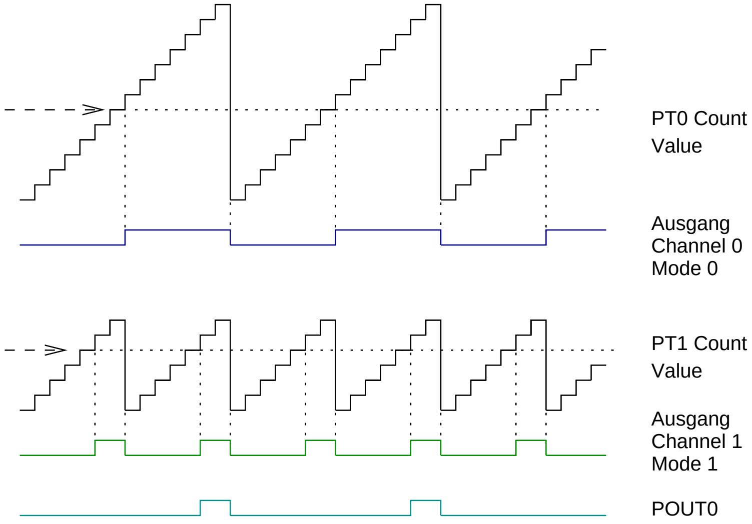


Abbildung 37: Burst Mode

2.10.4 Single Shot Mode

Hier wird der jeweilige Timer PTx per Software gestartet und zählt solange aufwärts bis er den Wert im Period Shadow Register erreicht. Im anschließenden Taktzyklus wird der Timer gestoppt und auf 0x0000 rückgesetzt. Der entsprechende Ausgang wird auf ‘1‘ gesetzt wenn der Inhalt des Timers größer/gleich dem Wert des Pulse Width Shadow Registers ist. Wird der Timer gelöscht, so wird auch der Ausgangspin auf ‘0‘ rückgesetzt. Bemerkenswert ist, daß in dieser Betriebsart durch Umprogrammierung des Timer Wertes sowohl die Delay Zeit vom Startzeitpunkt des Timers bis zum Setzen des Ausgangspins als auch die Pulslänge direkt modifiziert werden kann, siehe Abbildung 38.

2.10.5 Beispiel zum PWM Modul

Aufgabenstellung: Im folgenden Beispiel soll die Funktion des PWM Moduls anhand einer *edge-aligned* PWM illustriert werden.
Im Hauptprogramm wird die Periode der PWM, die Betriebsart und das entsprechende Interrupt Control Register aufgesetzt. Gleichzeitig mit der Interrupt Freigabe wird Timer PT0 mit dem Takt CPU/64 gestartet. In der zugehörigen ISR muß zunächst der entsprechende Interrupt Request gelöscht werden. Letzteres ist innerhalb des C167 nur für das PWM und das CAN Modul nötig.

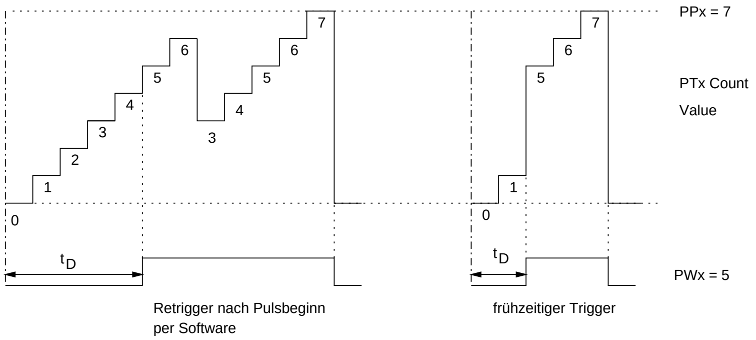


Abbildung 38: Single Shot Mode

Danach wird zyklisch der Duty Cycle grob variiert wodurch die Pulslänge des Ausgangssignals zyklisch gestreckt bzw. verkürzt wird.

```
#include <reg167.h>     /* Registerdefinitionen des C167 */
#include <intrins.h>    /* Spezialfunktionen des C167 */
#define PWM_PERIODE 5999 /* Periodendauer */

int duty_cycle=6000;     /* initial 0% Duty Wert */
/* In der PWM Interrupt Service Routine wird auf ein
   jeweils anderes Tastverhältnis weitergeschaltet.
   Diese ISR wird bei einem Wechsel von PPx -> 0
   aktiviert. */
void pwm_isr (void) interrupt 0x3F {
   PWMCON0&=0xefff;             /* Löschen des Interrupt Requests
                                   bei PWM extra nötig */
   switch(duty_cycle) {
        case 6000: {            /* schalte auf 25% Duty Cyle */
                duty_cycle=4500;
                PW0=duty_cycle;
                break;
        }
        case 4500: {            /* schalte auf 50% Duty Cyle */
                duty_cycle=3000;
                PW0=duty_cycle;
                break;
        }
        case 3000: {            /* schalte auf 75% Duty Cyle */
                duty_cycle=1500;
                PW0=duty_cycle;
                break;
        }
        case 1500: {            /* schalte auf 100% Duty Cyle */
                duty_cycle=0000;
                PW0=duty_cycle;
                break;
        }
        default:   {            /* schalte auf 0% Duty Cyle */
                duty_cycle=6000;
```

```
                PW0=duty_cycle;
                break;
        }
    }
}

void main (void) {
        DP7|=0x0001;             /* Richtung PWM Kanal 0 -> Ausgang */
        PT0=0;                   /* Startwert für die Timer PT0 */
        PP0=PWM_PERIODE;         /* Periodendauer der PWM */
        PW0=duty_cycle;          /* 0% Duty Cycle -> Initialisierung*/
        PWMCON1=0x0001;          /* PWM Betriebsart - edge aligned */
        PWMIC=0x004A;            /* PWMIE=1; ILVL=2; GLVL=2; */
        IEN=1;                   /* globale Interruptfreigabe */
        PWMCON0=0x0111;          /* starte die Timer PT0-3; CPU Clock/64;
                                    Interrupts enabled */
        while(1){                /* Endlosschleife */
          _idle_();              /* Stromsparmodus */
        }
}
```

2.11 Analog Digital Converter (ADC)

Die C167er Familie verfügt über einen integrierten Analog Digital Converter , siehe Abbildung 39, mit folgenden Eigenschaften:

- 10-Bit Auflösung
- Wandlungsverfahren: Sukzessive Approximation
- Wandlungszeit: minimal $9,6 \mu s$
- 16 analoge Eingangskanäle

2.11.1 Fixed Channel Conversion Modes

Es wird jeweils ein fix zugeordneter Eingangskanal (ADCH) gewandelt. Nach jeder Wandlung wird ein Interrupt Request generiert und das Wandlungsregister mit der zugehörigen Kanalnummer im Register ADDAT gespeichert.

Im *Single Conversion Mode* wird der ADC nach einer Wandlung angehalten und die Bits ADC busy (ADBSY) und ADC Start (ADST) werden zurückgesetzt. Im *Continuous Conversion Mode* startet der ADC automatisch nach jeder Wandlung mit einem weiteren Zyklus. Wird das ADST Bit per Software rückgesetzt obwohl eine Wandlung im Laufen ist, so wird diese zu Ende geführt und der ADC erst anschließend angehalten. In Abbildung 40 ist das Konfigurationsregister des ADC mit sämtlichen Einstellmöglichkeiten wiedergegeben.

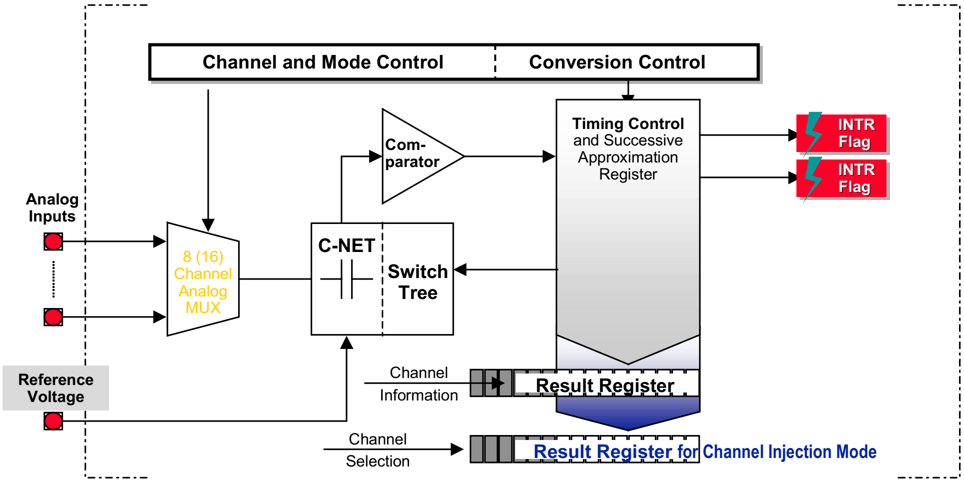


Abbildung 39: ADC Blockschaltbild

2.11.2 Auto Scan Conversion Modes

In dieser Betriebsart wird jedesmal eine ganze Sequenz von analogen Kanälen der Reihe nach analog-digital gewandelt. Im Bitfeld ADCH wird jener Kanal spezifiziert mit der die Wandlungs-Sequenz beginnt; sie endet mit Kanal Null. Nach jeder einzelnen Wandlung wird ein Interrupt Request generiert, der angibt, daß im ADDAT Register ein aktuelles Wandlungsergebnis steht. Auch in dieser Betriebsart wird zwischen einem Single und einem Continuous Conversion Mode unterschieden. Das Verhalten entspricht jenem bei der Fixed Channel Conversion mit dem Unterschied, daß hier jeweils eine Sequenz von Eingangskanälen gehandhabt wird.

2.11.3 Wait for Read Control

In den zuvor beschriebenen Modi wird ein Wandlungsergebnis stets überschrieben und geht verloren wenn die CPU das ADDAT Register nicht rechtzeitig ausliest. Um dies zu verhindern kann der Wait for Read Control Modus programmiert werden. Hier wird der Wert einer neuen Wandlung — sofern die vorhergehende noch nicht aus ADDAT ausgelesen wurde — in einen temporären Buffer kopiert und der ADC wird für weitere Wandlungen angehalten. Wird zu einem späteren Zeitpunkt das ADDAT Register mit dem älteren Wandlungswert ausgelesen, so wird im Anschluß automatisch der Wert des temporären Buffers in das ADDAT Register umkopiert. Es wird hierbei außerdem ein Interrupt Request generiert um zu signalisieren, daß ein neues Wandlungsergebnis im ADDAT Register steht. Mit dem Umkopiervorgang wird gleichzeitig der ADC wieder gestartet.

2.11.4 Channel Injection Mode

Während einer kontinuierlichen Wandlung kann die Wandlung eines spezifischen Kanals eingeschoben werden ohne die Betriebsart zu wechseln. Hierfür muß das Wait for ADDAT Read Mode

Address: 0xFFA0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADCON	ADCTC		ADSTC		AD CRQ	AD CIN	AD WR	AD BSY	ADST	-	ADM		ADCH			
Reset: 0x0000	rw	rw	rw	rw	rw	rw	rw	r	rw	-	rw	rw	rw	rw	rw	rw

Bit	Function
ADCH	**ADC Analog Channel Input Selection**
ADM	**ADC Mode Selection** 00: Fixed Channel Single Conversion 01: Fixed Channel Continuous Conversion 10: Auto Scan Single Conversion 11: Auto Scan Continuous Conversion
ADST	**ADC Start Bit**
ADBSY	**ADC Busy Flag** ADBSY=1: A conversion is active
ADWR	**ADC Wait for Read Control**
ADCIN	**ADC Channel Injection Enable**
ADCRQ	**ADC Channel Injection Request Flag**
ADSTC	**ADC Sample Time Control** 00: (Sample Clock) tsc = tcc 01: (Sample Clock) tsc = tcc*2 10: (Sample Clock) tsc = tcc*4 11: (Sample Clock) tsc = tcc*8
ADCTC	**ADC Conversion Time Control fCPU = 1/(2*TCL)** 00: tcc = TCL*24 10: tcc = TCL*96 11: tcc = TCL*48

Abbildung 40: ADCON Registerlayout

Bit gesetzt sein. Das Ergebnis der eingeschobenen Wandlung wird in das Register ADDAT2 gesichert. Der Zeitpunkt zu dem die eingeschobene Wandlung durchgeführt wird ist durch Setzen des ADCRQ Bits oder durch einen Event des Capture/Compare Registers CC31 festgelegt. Letzteres Ereignis setzt ebenfalls das ADCRQ Bit.

2.11.5 Beispiele zum ADC

Aufgabenstellung: Die an den analogen Ports (P5.0-P5.7) anliegenden Signale sollen zyklisch gewandelt werden.
In diesem Beispiel wird der ADC auf Auto Scan Continuous Mode programmiert.Nach jeder Wandlung wird die zugehörige ISR aktiviert in der das Wandlungsergebnis aus dem Ergebnisregister in ein Speicherfeld umkopiert wird. Das Ergebnisregister ADDAT ist jeweils mit dem hexadezimalen Wert 0x03FF zu maskieren, da in dem obersten Nibble die gewandelte Kanalnummer codiert ist. Wurden eine programmierte Anzahl an Wandlungen durchgeführt so wird letztendlich der ADC gestoppt und sämtliche Wandlungsergebnisse sind im entsprechenden Speicherfeld abgelegt.

```
#include <reg167.h>      /* Register Definitionen des C167 */
#include <intrins.h>     /* bausteinspezifische Funktionen */

#define ANZ 16           /* Anzahl der Wandlungen */
```

```
unsigned int adc_ergebnisse[ANZ]; /* Buffer für die Wandlungsergebnisse */
unsigned int i;                   /* Zählvariable für die Wandlungen */


/* In der ADC Interrupt Service Routine wird das Wandlungsergebnis
   ausgelesen und in das Feld adc_ergebnisse kopiert. */
void adcisr (void) interrupt 0x28 {
        if(i<ANZ) {                     /* Alle Wandlungen durchgeführt? */
                /* Nein! -> Sichere das Wandlungsergebnis; Maskiere die
                   Kanalnummer aus. */
           adc_ergebnisse[i]=ADDAT & 0x03FF;
           i++; /* Inkrementiere den Wandlungszähler. */
        }
        else {
           adc_ergebnisse[i]=ADDAT & 0x03FF;
           ADST=0;      /* JA! Stoppe den ADC */     }
}

void main (void) {
        i=0;    /* Initialisierung des Wandlungszählers. */
        ADCON=0x0237;   /* Auto Scan Continuous Conversion der Kanäle 7-0
                           Wait for Read Control aktiv; tcc=24TCL
                           Sample clock: tsc=tcc */
        ADCIC=0x4C;     /* ADCIE=1; ILVL=3; GLVL=0 */
        IEN=1;          /* globale Interruptfreigabe */
        ADST=1;         /* starte den ADC */
        while(1){       /* Endlosschleife */
          _idle_();     /* Stromsparmodus */
        }
}
```

Aufgabenstellung: Im zweiten Beispiel soll ein Analogsignal periodisch abgetastet und analog-digital gewandelt werden. Nach jeder Wandlung soll das Ergebnis mit einem PEC Transfer weggesichert werden.
In der illustrierten Lösung wird mit Timer T3 periodisch jede Sekunde ein Interrupt Event ausgelöst. Timer T2 fungiert hier als Reload Register von T3 um die Periodendauer von einer Sekunde für sämtliche Wandlungen zu gewährleisten. Der Interrupt Event von T3 wird benützt um eine festgelegte Anzahl von AD-Wandlungen mittels PEC Transfer vom ADDAT Ergebnisregister des ADC in ein Speicherfeld zu kopieren. Beim letzten PEC Transfer wird das Interrupt Request Flag automatisch nicht retourgesetzt und eine normale ISR, zugeordnet zu T3, wird ausgeführt. In dieser werden sowohl T3 als auch der ADC, der zuvor auf kontinuierliche Wandlung programmiert wurde, angehalten.

```
#include <reg167.h>     /* Register Definitionen des C167 */
#include <intrins.h>    /* bausteinspezifische Funktionen */

#define ANZ 3
unsigned int adc_ergebnis[ANZ]; /* Ergebnisfeld */
/* Bei jedem Überlauf von 0x0000 -> T2 (Reload Wert) wird ein
```

```
   PEC Transfer durchgeführt und der PEC Counter dekrementiert.
   Ist letzterer == 1  so wird ein letzter PEC Transfer durchgeführt
   und anschliesend sofort die Interrupt Service Routine von T3 aktiviert.
   Zusätzlich wird bei einem Reload von T2 nach T3 über T2 ein
   Interrupt Request generiert. Da dieser aber nicht enabled wird
   ist er hier nicht weiter relevant. */
void t3isr (void) interrupt 0x23 {
        ADST=0;                         /* Stoppt den ADC */
        T3R=0;                          /* Stoppt T3 */
}

void main (void) {
        T3CON=0x0086;   /* Pre-Scaler: 512; Timer Mode; Count Down */
        T3=0x9860;      /* := 1sec */
        T2CON=0x0027;   /* Reload Mode; -> aktiv bei jedem Flankenwechsel von
                           T3OTL */
        T2=0x9860;      /* Reload Wert := 1sec */
        ADCON=0x0210;   /* Fixed Channel Continuous Conversion von Kanal 0;
                           Wait for Read Control aktiv; Conversion clock: tcc=24TCL
                           Sample clock: tsc=tcc */
        T3IC=0x78;      /* T3IE=1; ILVL=14; GLVL=0 => selektiert glz. PEC Kanal 0 */
        PECC0=0x0200 | (ANZ & 0x00FF);  /* Increment DSTP0; Word Transfer; ANZ PEC Transfers */
        SRCP0=_sof_(&ADDAT);            /* Source Adresse für den PEC Transfer */
        DSTP0=_sof_(adc_ergebnis);      /* Destination Adresse für den PEC Transfer
                                           -> DSTP0 wird nach jedem Transfer
                                           inkrementiert */
        IEN=1;          /* globale Interruptfreigabe */
        T3R=1;          /* starte Timer 3 */
        ADST=1;         /* starte den ADC */
        while(1){      /* Endlosschleife */
           _idle_();  /* Stromsparmodus */
        }
}
```

2.12 Serielle Schnittstellen

Der C167 ist mit einer rein synchronen sowie einer asynchron/synchronen Schnittstelle ausgestattet. Der hierfür benötigte Baudratengenerator ist integriert.

2.12.1 Asynchrones / Synchrones Interface ASC0

Bei einer Vollduplex[l] Datenübertragung kann mit einer Transferrate bis zu 625 KBaud[m] und bei einer Halbduplex Übertragung bis zu 2,5 MBaud übertragen werden (bei 20 MHz CPU Takt). Dies wird unter anderem durch zusätzliche Buffer Register im Receive als auch im Transmit Zweig

---

[l] Bei einer Vollduplex Übertragung werden Daten gleichzeitig in beide Richtungen übertragen. Bei einer Halbduplex Verbindung ist dies nur unidirektional möglich.

[m] Mit dem Begriff Baud wird die Anzahl der übertragenen “Symbole pro Sekunde” angegeben. Dieses Maß stimmt somit nicht automatisch mit der Einheit “Bit/s” überein.

erreicht. Im Konfigurationsregister, siehe [Si96a] Seite 10-2, können mehrere Einstellungen zur Übertragung vorgenommen werden:

- 1 oder 2 Stop-Bits,
- 7, 8 oder 9 Daten-Bits,
- odd oder even Parity sowie
- diverse Mechanismen zur Erkennung von Fehlern

Das Blockschaltbild des asynchronen/synchronen seriellen Interfaces ist in Abbildung 41 dargestellt.

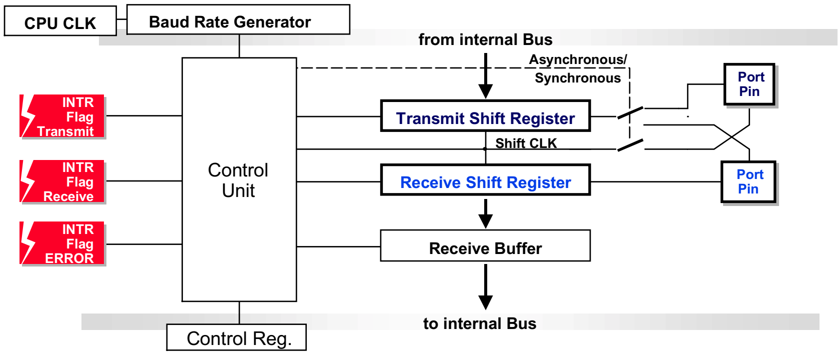


Abbildung 41: Blockschaltbild der ASC0

Für die asynchrone Datenübertragung ist der Baudraten Generator entsprechend

$$SOBG = \frac{f_{CPU}}{16(2+<SOBRS>)B_{async}} - 1$$

zu programmieren. Für den synchronen Betrieb ergibt sich dieser Reload Wert aus

$$S0BG = \frac{f_{CPU}}{4(2+<SOBRS>)B_{sync}} - 1.$$

Der Wert für <S0BRS> kann hierbei entweder mit ‘1‘ oder mit ‘0‘ frei gewählt werden. Soll z.B. mit 9600 Baud asynchron bei einer CPU Taktrate von 20 MHz übertragen werden so ergibt sich mit S0BRS=0 für S0BG=64d=0x0040. Diese Schnittstelle verfügt über drei verschiedene Interrupt Request Leitungen, siehe Abbildung 42.

Beim Versenden einzelner Zeichen sollte der S0TIR Request verwendet werden. Erfolgt jedoch eine back-to-back Übertragung mehrerer Zeichen hintereinander so sollte anstelle des Transmit Requests der Buffer Request S0TIR verwendet werden.

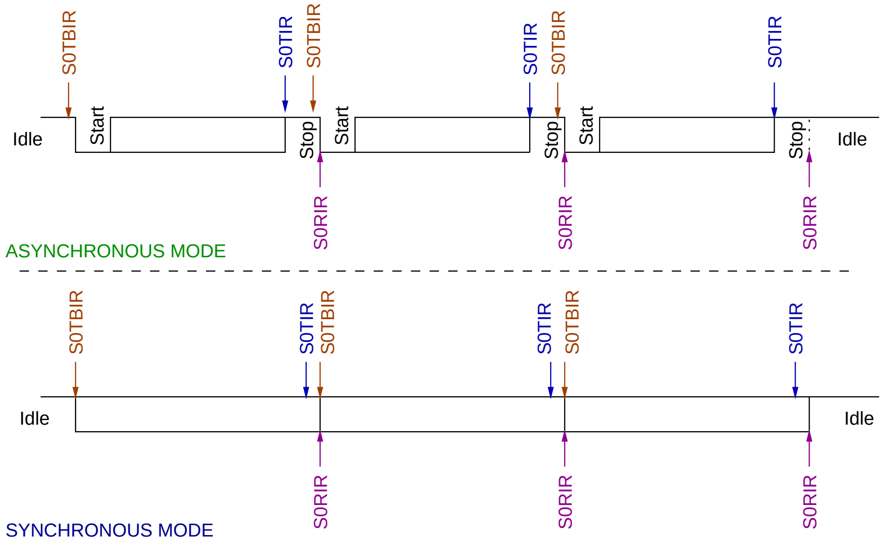


Abbildung 42: Interrupt Requests von ASC0

2.12.2 High Speed Synchronous Serial Interface SSC

Mit diesem Interface kann ebenso sowohl im Vollduplex als auch im Halbduplexbetrieb gearbeitet werden. Bei einer CPU Taktrate von 20 MHz können hiermit bis zu 5 MBaud übertragen werden. Bei diesem Interface sind die Datenbreite, die Shift Richtung, die Clock Polarität sowie die Phase programmierbar, siehe [Si96a] Seite 11-3. Weiters kann dieses Interface mit SPI (Serial Peripheral Interface) tauglichen Geräten kommunizieren. Die Baudrate wird hierbei über $SSCBG = \frac{f_{CPU}}{2B_{ssc}} - 1$ ermittelt.

2.12.3 Beispiel zur seriellen Schnittstelle

Aufgabenstellung: Über die asynchrone serielle Schnittstelle ist die Zeichenfolge “Hello World!” auszugeben.
Im ersten Schritt wird die asynchrone serielle Schnittstelle mit derselben Baudrate konfiguriert, mit der auch das zweite Kommunikationsendgeräte arbeitet. Nach der Interrupt Freigabe wird eine Variable, die zum Zählen der Zeichen des zu übertragenden Strings dient auf Null initialisiert. Damit das erste Zeichen übertragen werden kann, wird dieses in den Transmit Buffer von ASC0 kopiert. Die serielle Schnittstelle übernimmt automatisch das Zeichen aus diesem Buffer in das Ausgangsschieberegister, von dem aus das Zeichen in der Folge bitweise versendet wird. Sobald das letzte Bit des Zeichens übertragen wurde, wird die Transmit ISR angestoßen. In dieser wird der Zeichenzähler erhöht und das entsprechende Zeichen in den Transmit Buffer umkopiert. Die ASC0 Schnittstelle verfährt mit diesem Zeichen auf dieselbe Art und Weise wie mit dem ersten Zeichen. Zeigt der Zeichenzähler auf das End-of-String Zeichen so wird der Transmit Interrupt diabled und die Übertragung somit angehalten.

```
/* ASC0 */
#include <reg167.h>
```

```
int i,s;                         /* Laufvariable */
char hello[14]="Hello World!"; /* Ausgabestring */
void init_asc0 (void) {         /* Initialisierung von ASC0 */
        DP3|=0x0400;            /* Richtung -> Transmit */
        DP3&=0xF7FF;            /* Richtung -> Receive */
        P3|=0x0400;             /* Transmit setzen */
        S0TIC=0x45;             /* ILVL=1, GLVL=0, S0TIE=1 */
        S0BG=0x0040;            /* Baudrate = 9600 Baud */
        S0CON=0x8011;           /* 8 Bit Daten, 1 Stop Bit,
                                   Starte Baudrate Generator */
}
/* Interrupt Service Routine zur Befüllung des Transmit Buffers
   von ASC0 -> wird aktiviert wenn ein Zeichen aus dem Buffer
   S0TBUF in das Transmit Shift Register kopiert wird */
void asc_transmit_isr (void) interrupt 0x47 {
        s=0;
}
/* Die Funktion putchar() kopiert ein Zeichen in den Transmit
   Buffer S0TBUF */
void put_char (char c)  {
   S0TBUF=c;
   while(s!=0)     /* hier erfolgt das eigentliche "rauspollen" */
      ;
}
void main (void) {
        s=1;
    init_asc0();    /* Initialisierung von ASC0 */
    IEN=1;          /* globale Interruptfreigabe */
    while (hello[i]!=’\0’) {  /* Solange ein String nicht
                                 vollständig ausgegeben wurde ...*/
       put_char(hello[i]);    /* Ein Zeichen ausgeben */
       i++;     /* Zeichenzähler inkrementieren */
       s=1;     /* -> Zeichen "rauspollen" - s wird in der ISR rückgesetzt */
    }
    while(1){ ; }      /* Endlosschleife */
}
```

Anmerkung: Dieses Programm weist zahlreiche Schwachstellen auf. So wird hier z.B. nicht geprüft ob die serielle Schnittstelle zum Versenden einer Nachricht bereit ist. Es sind weiters keinerlei Möglichkeiten vorgesehen eine langsam konfigurierte Schnittstelle von einem zu rasch arbeitenden Mikroprozessor zu entkoppeln. D.h. wenn der Prozessor Daten zu rasch an die serielle Schnittstelle weitergibt bevor diese verschickt werden, so können Daten verloren gehen. Eine elegantere Lösung würde hier z.B. mit Hilfe eines Ringbuffers eine Geschwindigkeitsentkopplung zwischen Schnittstelle und CPU vornehmen und mittels einer Statusabfrage die Bereitschaft der beiden Kommunikationsendgeräte überprüfen.

2.13 Das Controller Area Network (CAN)

Das von der Firma Bosch definierte Controller Area Network (CAN) Protokoll findet neben seinem Haupteinsatzgebiet in mobilen Systemen der Kraftfahrzeugsindustrie auch in Bereichen der

Gebäudeautomatisierung und der Produktionsautomatisierung seine Einsatzgebiete. In diesem Abschnitt soll in erster Linie das in der C167er Familie implementierte CAN Modul erläutert werden. Für detailliertere Informationen zum CAN Protokoll wird auf weiterführende Literatur verwiesen, siehe [Bos91], [Ets94] oder [Si96b].

Die Struktur eines CAN Knotens ist in einer groben Anlehnung an das OSI-7 Schichten Modell i.a. wie in Abbildung 43 dargestellt festgelegt.

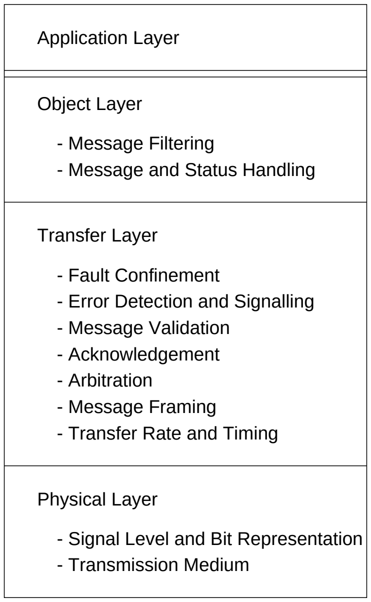


Abbildung 43: Schichten Modell eines CAN Knotens

- Die physikalische Schicht definiert *wie* Signale übertragen werden.
- Der Transfer Layer bildet das Kernstück des CAN Protokolls. In diesem Teil wird beschrieben wie Nachrichten vom darüberliegenden Object Layer empfangen bzw. wie sie an diesen weitergegeben werden. Diese Schicht ist zuständig für das Bit-Timing, die Synchronisation, das Paketieren von Nachrichten, die Arbitration, das Acknowledgement von Nachrichten und die Detektion von Fehlern, deren Signalisierung sowie Beschränkung.
- Der Object Layer schließlich dient zur Filterung von Nachrichten und zur Statusüberwachung.
- Der Application Layer ist für die jeweilige Anwendung reserviert.

CAN ist ein asynchrones serielles Bussystem mit einer logischen Busleitung. Es besitzt eine offene, lineare Busstruktur mit gleichberechtigten Stationen. Dem System muß nicht bekannt sein, wieviele Teilnehmer am Bus angeschlossen sind. Dies erlaubt das Abklemmen von Stationen (z.B. nach Ausfall), ohne die Kommunikation der übrigen Stationen zu beeinträchtigen; auch können bei Bedarf neue Teilnehmer hinzugefügt werden.

Die Buslogik entspricht dem „Wired-AND”-Mechanismus, wonach rezessive Bits (logisch ‘1‘) von dominanten Bits (logisch ‘0‘) überschrieben werden. Solange kein Busteilnehmer dominante Bits sendet, ist der Bus im rezessiven Zustand; ein dominantes Bit genügt, um den Bus in den dominanten Zustand zu versetzen. Für CAN kommt daher jedes Übertragungsmedium in Frage auf dem dominante und rezessive Zustände übertragen werden können. Kostengünstig ist eine elektrisch differentielle Zweidrahtleitung, die je nach Umgebungsbedingungen unverdrillte oder verdrillte Adern (*Twisted Pair*) besitzt und entweder galvanisch ge- oder entkoppelt ist, siehe Abbildung 44.

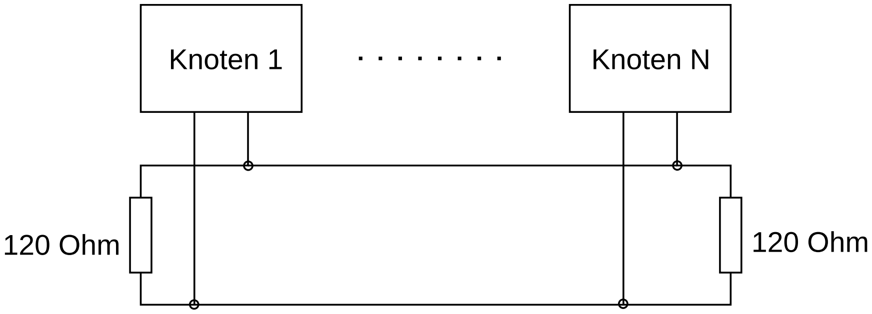


Abbildung 44: CAN Feldbussystem

Die binären Daten werden mit Hilfe des Non-Return-to-Zero-Codes dargestellt (LOW-Pegel: dominanter Zustand, HIGH-Pegel: rezessiver Zustand). Zur Vermeidung von zu starker Anhebung bzw. Absenkung des Gleichspannungspegel auf der Busleitung wird *Bit-Stuffing* durchgeführt. Dies bedeutet, daß bei einer Übertragung einer Nachricht maximal fünf hintereinanderfolgende Bits dieselbe Polarität haben dürfen. Tritt eine derartige oder längere Sequenz auf, so muß vom Sender ein zusätzliches Bit mit umgekehrter Polarität in den Bitstrom eingefügt werden. Vom Empfänger wird dieses Bit dann wieder entfernt (*Destuffing*).

Das CAN-Protokoll ist nicht teilnehmer-, sondern nachrichtenorientiert. Deshalb werden nicht die Bus-Stationen adressiert, sondern der Inhalt einer Nachricht. Dies geschieht mit Hilfe eines Identifiers, der durch seinen Wert gleichzeitig auch die Priorität der Nachricht festlegt. Hierbei gilt: Je niedriger der Identifier, desto höher ist die Priorität.

Die Buszuteilung erfolgt durch das Verfahren CSMA/CD mit NDBA (*Carrier Sense Multiple Access / Collision Detection* mit *Non-Destructive Bitwise Arbitration*). Möchte ein Busteilnehmer A eine Nachricht an einen oder mehrere andere Knoten senden, prüft er zunächst ob der Bus frei ist, d.h. ob er sich im rezessiven Zustand befindet (*Carrier Sense*). Ist dies der Fall und hat gleichzeitig auch keine andere Station einen Sendewunsch, wird A zum Busmaster und sendet seine Nachricht. Alle anderen Teilnehmer gehen automatisch auf Empfang da Sie selber nicht senden und am Bus einen dominanten Zustand erkennen. Nach dem korrekten Empfang prüfen alle Stationen anhand des Identifiers der Nachricht, ob die empfangenen Daten für Sie von Bedeutung sind oder nicht. Im ersten Fall werden die Daten weiterverarbeitet, sonst verworfen. Wollen nun zwei oder mehrere Busteilnehmer zum selben Zeitpunkt den Bus belegen (*Multiple Access*), wird eine drohende Kollision der Botschaften mit Hilfe des Wired-AND Prinzips („dominant überschreibt rezessiv”) und des Identifiers durch bitweise Arbitrierung vermieden (*Collision Detection / NDBA*). Jeder Knoten gibt Bit für Bit den Identifier seiner Nachricht auf den Bus und beobachtet den Buspegel. Eine

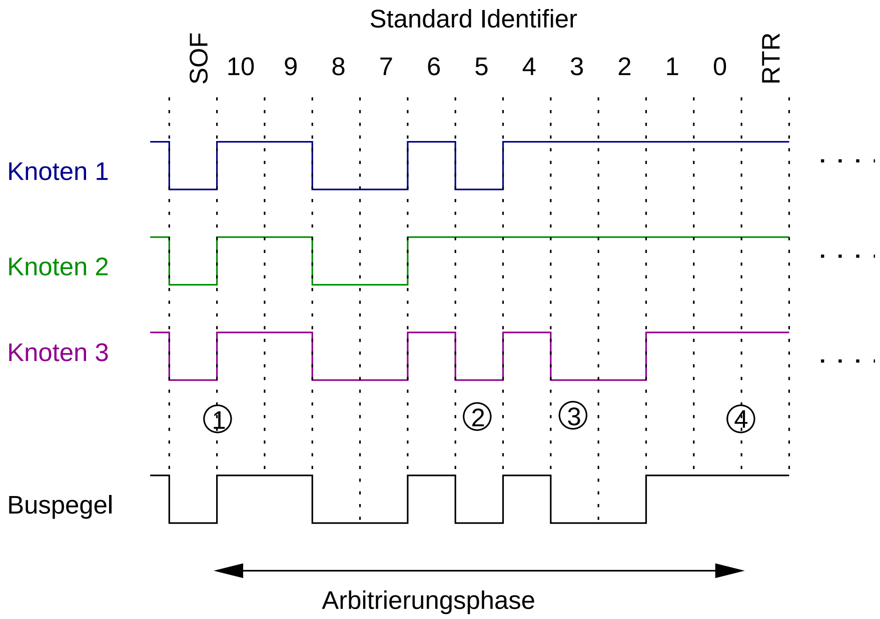


Abbildung 45: Beispiel eines Arbitrierungsvorganges im CAN Protokoll. Die Teilnehmer 1, 2 und 3 beginnen gleichzeitig einen Arbitrierungsversuch (1). Teilnehmer 2 verliert zum Zeitpunkt (2), Teilnehmer 1 zum Zeitpunkt (3) das Buszugriffsrecht. Beide Teilnehmer gehen damit in den Empfangszustand; am Ende der Arbitrierungsphase besitzt nur noch Teilnehmer 3 das Buszugriffsrecht und schaltet seine Nachricht auf den Bus.

Station, die ein rezessives Bit sendet, jedoch ein dominantes Bit zurückliest, verliert in diesem Moment den Wettstreit um den Bus und geht auf Empfang (siehe Abbildung 45), da der Identifier eines konkurrierenden Teilnehmers niedriger und damit die Priorität von dessen Nachricht höher ist.

Der Busknoten mit der wichtigsten Nachricht setzt sich also ohne Zeit- und Bitverlust durch (*NDBA*), alle übrigen Sendestationen wiederholen den Sendeversuch automatisch, sobald der Bus wieder frei ist. Es ist nicht erlaubt, daß verschiedene Teilnehmer Nachrichten mit demselben Identifier senden, da dies wirklich zu Buskollisionen führen kann und Fehler verursacht.

Für Synchronisationszwecke wird die Dauer eines Bits, die *Bit-Zeit*, in vier Abschnitte unterteilt die ihrerseits jeweils aus einer fixen Anzahl aus sogenannten *Zeitquanten* bestehen, siehe Abbildung 46.

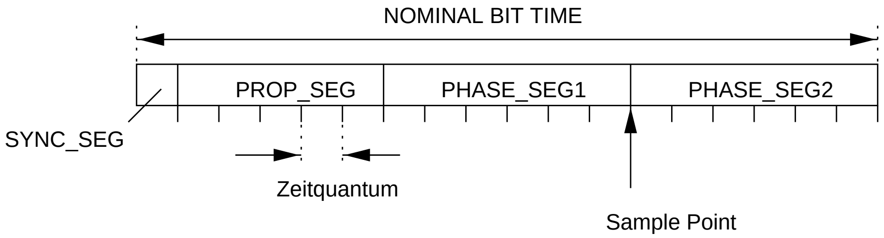


Abbildung 46: Bit-Zeit

Der *SYNC_SEG* Teil dient zur Synchronisation verschiedener Knoten am Bus. Es wird erwartet, daß eine auftretende Flanke innerhalb dieses Segments zu liegen kommt. Dieser Bereich ist genau einen Zeitquant lang. Der *PROP_SEG* Teil ist programmierbar in einem Bereich zwischen einem und acht Zeitquanten. Dieser Bereich wird verwendet um physikalische Verzögerungen, verursacht durch Laufzeiten innerhalb des Netzwerkes, zu kompensieren. Die Zeitdauer von PROP_SEG entspricht zweimal der maximalen Laufzeit am Bus und zusätzlich der Verzögerung durch den Eingangskomparator bzw. dem Ausgangstreiber. Die Phasen-Buffer Segmente *PHASE_SEG1* und *PHASE_SEG2* kompensieren Phasenfehler der Signalflanken. Diese Bereiche können durch die Resynchronisation entsprechend verlängert bzw. verkürzt werden. Am Ende des PHASE_SEG1 Segments wird das Bit gesampelt. Im C167 werden PROP_SEG und PHASE_SEG1 unter der Bezeichnung *Tseg1* zusammengefaßt, während PHASE_SEG2 als *Tseg2* bezeichnet wird (siehe [Si96a] Seite 23-10). Die Dauer einer einzelnen Bit-Zeit kann im Bereich von 8 bis 25 Zeitquanten programmiert werden. Ein Zeitquantum wiederum ergibt sich aus $t_q = (BRP + 1) * 2 * t_{CLK}$ (BRP entspricht der eingestellten Übertragungsrate).

Der Transfer von Nachrichten auf dem Medium CAN wird mittels vier verschiedener Nachrichtenformate durchgeführt.

- Ein *Datenrahmen* (siehe Abbildung 47) transportiert Daten vom Sender zum Empfänger.
- Über einen *Remote Frame* fordert ein Knoten die Übertragung einer Nachricht von einem anderen Knoten mit demselben Identifier, die auch der Remote Frame hat, an.
- Ein *Error Frame* wird von jedem Knoten generiert sobald dieser einen Fehler am Bus detektiert.
- Mit einem *Overload Frame* wird ein zusätzliches Delay zwischen Daten oder Remote Frames eingefügt.

Zusätzlich gibt es noch den sogenannten *Interframe Space* der vor/nach Daten oder Remote Frames eingefügt wird.

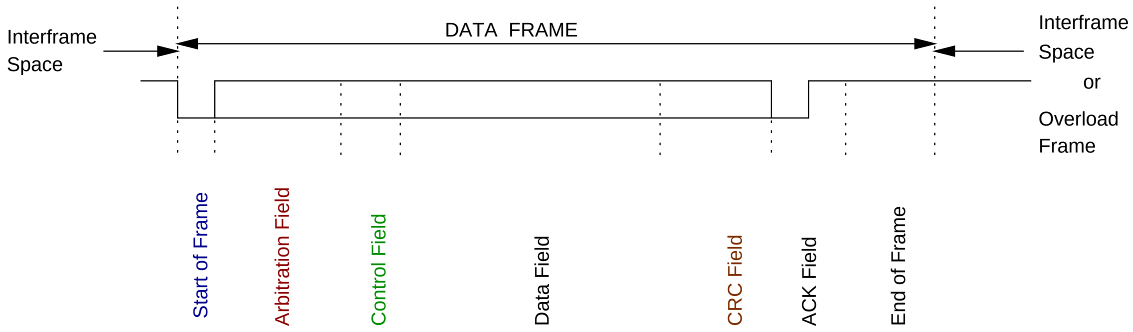


Abbildung 47: Aufbau eines Datenrahmens

Weiters sind im CAN Standard 2.0A nur Rahmen im *Standard Format* definiert. In einer späteren Revision wurde der Standard 2.0B herausgegeben, der neben dem Standard Format ein sogenanntes *Extended Format* beherrscht. Im Standard Format besteht das Arbitration Feld aus einem 11-Bit

Identifier und einem Remote Transmission Request Bit (RTR), welches in Datenframes dominant und in Remote Frames rezessiv gesendet werden muß. Im Extended Format besteht das Arbitration Feld aus einem 11-Bit Identifier, einem rezessiven Substitute Remote Request Bit, einem Identifier Extension Bit, weiteren 18-Bit Identifier und ebenfalls einem RTR Bit. Nähere Information zu den einzelnen Rahmen und deren Felder ist aus der begleitenden Literatur [Bos91] zu entnehmen. In der CAN Spezifikation werden allgemein folgende Eigenschaften für den CAN Bus angegeben:

- Systemart: Multimaster
- Topologie: Bus
- Leitungslänge: 1 km bei einer Übertragungsrate von 50 kBit/s; 40 m bei einer Übertragungsrate von 1 MBit/s
- Übertragungsmedium: (verdrillte) Zweidrahtleitung, Lichtwellenleiter
- Buszugriffsverfahren: CSMA/CD mit NDBA
- Bitcodierung: NRZ
- Datentelegramm - **Standard Format**: 1 Start, 11 Identifier, 1 RTR, 6 Control, 0-64 Daten, 15 CRC und 1 CRC Delimiter, 2 Acknowledge und 7 End of Frame Bits - **Extended Format**: 1 Start, 11 Identifier, 1 SRR, 1 IDE, 18 Identifier, 1 RTR, 6 Control, 0-64 Daten, 15 CRC und 1 CRC Delimiter, 2 Acknowledge und 7 End of Frame Bits
- Datensicherung: 15-Bit CRC, Hamming Distanz 6
- Nutzdaten: 0-64 Bit
- Overheaddaten: 44-64 Bit; zwischen den Paketen sind zumindest 3-Bit Interframe Space
- Knotenanzahl[n]: typisch 30

Das im C167 implementierte CAN Modul besteht aus 15 *Message Objects*, wobei jedes mit einem eigenen Satz an Registern ausgestattet ist. Über jedes dieser Subsysteme können Nachrichten verschickt oder empfangen werden. Neben diesen einzelnen Message Objects gibt es einen Satz an allgemeinen Registern über die eine globale Konfiguration des CAN Moduls vorgenommen wird. Letzteres besteht aus den Mask Registern, einem Bit-Timing Register, dem Interrupt Register und einem allgemeinen Control/Status Register (vgl. Abbildung 48).

Jedes einzelne Message Object wiederum enthält, ihm zugeordnet eigene Register zur Steuerung der Nachrichten. Über diese werden festgelegt ob die zugeordneten Nachrichten verschickt oder empfangen werden, wieviel Datenbytes benötigt werden und ob ein Standard oder ein Extended

---

[n] Die Teilnehmeranzahl ist in einem CAN-Netzwerk nur vom Physical Layer abhängig. Es existieren keine Stationsadressen, sondern Nachrichten werden weiterverarbeitet, wenn sie das individuelle Akzeptanzfilter eines Teilnehmers passiert haben. Mit den 11 Identifier Bits des Standard Frames, die auch zur Arbitrierung herangezogen werden, sind 2048 verschiedene Nachrichtenarten möglich. Das Extended CAN Protokoll bietet 29 Identifier Bits und somit sind theoretisch 2^{29}verschiedene Nachrichtenarten möglich.

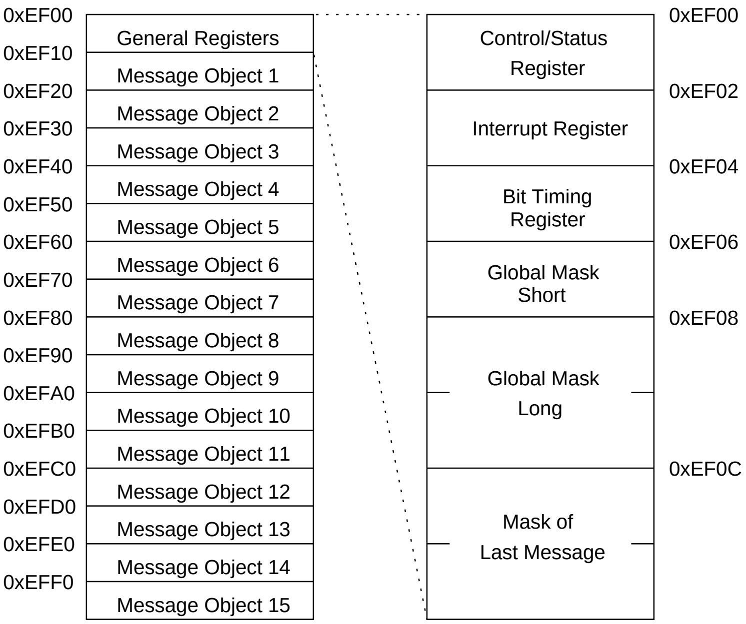


Abbildung 48: Subsysteme des CAN Moduls

Identifier verwendet wird. Für alle Message Objects wird die Bitrate in den globalen Registern des CAN Moduls festgelegt. Diese muß mit der Bitrate aller anderen Knoten am Bus übereinstimmen. Weiters gibt es zwei globale Masken Register, die es erlauben einlangende Nachrichten zu filtern. Angenommen IDMO ist der Identifier eines Message Objects, das Nachrichten empfangen soll. Weiters sind IDIM der Identifier einer einlangenden Nachricht und MASK der Wert des globalen Masken Registers. Eine Nachricht wird in diesem Fall nur dann empfangen wenn gilt:

(IDIM & MASK) == (IDMO & MASK)

Über das globale Maskenregister wird somit eine Vorselektion durchgeführt, welche Bits im Message Object nun tatsächlich verglichen werden. Wird z.B. eine 11-Bit (Standard Identifier) Maske mit 111 1110 0000b aufgesetzt so bedeutet dies, daß der Knoten die letzten fünf Bits aller einlangenden Nachrichten Identifier ignoriert. Auf diese Weise nimmt ein Message Object mit einem Identifier 100 1110 0000b sämtliche Nachrichten im Bereich zwischen 100 1110 0000b und 100 1111 1111b an. Auf diese Weise ist es möglich in den letzten fünf Bits Applikations spezifische Informationen zu verpacken. Sämtliche Message Objekte teilen sich eine gemeinsame Interrupt Request Leitung. Damit die CPU die exakte Interruptquelle bestimmen kann, muß sie das Interrupt Register auslesen. Die Verwendung des CAN Moduls läßt sich aus der Sicht des Programmierers in drei Abschnitte unterteilen: Initialisierung des CAN Knotens, Aufsetzen der Message Objekte und der Ablauf beim Senden und Empfangen von CAN Nachrichten.

2.13.1 Initialisierung des CAN Moduls

Das Bit Timing Register ist während des normalen Betriebs vor falschen Schreibzugriffen geschützt. Dies wird durch ein Löschen des Configuration Change Enable (CCE) Bits im Control/Status Register erreicht. In der Initialisierungsphase ist dieses Register jedoch mit der richtigen Baudrate zu beschreiben. Hierfür muß also zunächst dieses Bit und das INIT Bit Im Control/Status Register gesetzt werden.

```
CTL_ST defa 0EF00h ; define Control/Status Register
mov R0, #0041h ; set Lock Bits CCE and INIT to start
mov CTL_ST, R0 ; the initalization
```

Im nächsten Schritt ist der Wert für das Bit Timing Register zu ermitteln. Möchte man z.B: einen Knoten mit einer CPU Taktrate von 20 MHz an ein CAN Netzwerk mit einer Übertragungsrate von 500 kBit/s anschließen so kann wie folgt vorgegangen werden. Wählt man z.B. 20 Zeitquanten für eine Bit-Zeit (Synchronisation + Tseg1 + Tseg2 = 1 + 16 + 3)[o] so ergibt sich ein einzelner Zeitquant bei einer Rate von 500 kBit/s zu 100 ns. Daraus ergibt sich für den Baud Rate Prescaler $BRP = \frac{(Zeitquantum)*f_{CPU}}{2} - 1$ ein Wert von Null.

```
BIT_TIM defa 0EF04h ; define Bit Timing Register
mov R0, #2700h ; TSEG1 = 15, TSEG2 = 2, SJW = 0, BRP = 0
mov BIT_TIM, R0
```

Schließlich sind noch die globalen Masken Register zu setzen. Möchte man z.B. Extended Identifier verwenden und nur Nachrichten durchlassen deren Identifier exakt mit dem der Message Objekte übereinstimmt, so können die Maskenregister wie folgt initialisiert werden:

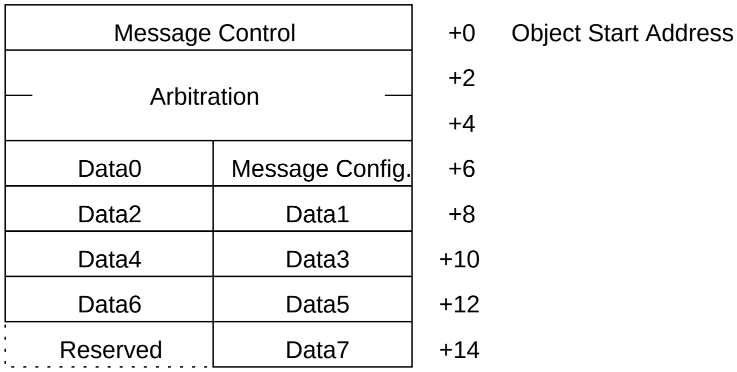
```
UP_GLM_L defa 0EF08h ; define Upper Global Mask (extended ID)
LO_GLM_L defa 0EF0Ah ; define Lower Global Mask (extended ID)
mov UP_GLM_L, ONES
mov LO_GLM_L, ONES
```

Um die Initialisierung abzuschließen sind die Lock Bits (CCE und INIT) im Control/Status Register zu löschen.

```
mov CTL_ST, ZEROS ; Clear bits CCE and INIT
```

2.13.2 Initialisierung der Message Objects

Jedes dieser Objekte verfügt über ein eigenes Message Control Register. Die einzelnen Felder dieses Registers werden hierbei jeweils über zwei Bits angesprochen, dadurch wird das Setzen bzw. Löschen von Bits erheblich erleichtert da die CAN Register nicht bitweise adressierbar sind. Mit der Bit Kombination 10b wird ein Element gesetzt, mit 01b rückgesetzt und die Kombination 11b läßt das entsprechende Feld unverändert. Bei jedem anderen Register, das nicht bitweise adressierbar ist, kann eine Modifikation nur über einen READ-MODIFY-WRITE Zyklus erfolgen.



Message Control		+0	Object Start Address
Arbitration		+2	
		+4	
Data0	Message Config.	+6	
Data2	Data1	+8	
Data4	Data3	+10	
Data6	Data5	+12	
Reserved	Data7	+14	

Abbildung 49: Layout eines Message Objects

Sämtliche Message Object Register werden über einen Offset zur jeweiligen Startadresse des entsprechenden Objektes angesprochen, siehe Abbildung 49.

Soll z.B. eine einzige Nachricht von einem Knoten A an einen weiteren Knoten B gesendet werden so kann folgendermaßen vorgegangen werden: Verwendet wird nur Message Object 1, übertragen wird nur ein Byte und für die Nachricht wird ein Extended Identifier mit dem Wert Null vergeben.

```
; -- CAN Node A Message Object 1 (Transmitter) --
MCSG_CTL_1 defa 0EF10h
mov R0, #5595h ; set field MSG_VAL
mov MSG_CTL_1, R0 ; message object 1 is valid
; set arbitration registers of message object 1
mov (MSG_CTL _1 + 2), ZEROS
mov (MSG_CTL_1 + 4), ZEROS
; set message data block size, direction and identifier type
mov R0, #1Ch
mov (MSG_CTL_1 + 6), R0 ; DLC = 1 (data length in bytes),
                            DIR = 1 (transmit),
                            XTD = 1 (extended)
```

Am Knoten B, dem Empfänger, wird dieselbe Initialisierung vorgenommen – einzig die Übertragungsrichtung wird auf Empfang eingestellt.

```
mov R0, #14h
mov (MSG_CTL_1 + 6), R0 ; DLC = 1, DIR = 0 (receive), XTD = 1
```

Die Initialisierung wird dadurch abgeschlossen indem sämtliche unbenützten Message Objects als ungültig gekennzeichnet werden. Dies wird durch Rücksetzen aller Bitfelder in den entsprechenden Message Control Registern erreicht.

---

[o] Tseg1 = TSEG1 + 1; Tseg2 = TSEG2 + 1

2.13.3 Senden und Empfangen einer CAN Nachricht

Eine Assembler Routine die ein Byte von A nach B schickt könnte z.B. folgendermaßen aussehen:

```
; data byte to be transmitted is stored in the CPU Register RL0
mov R1, #0EAFFh
mov MSG_CTL_1, R1 ; set TXRQ and CPUUPD
movb (MSG_CTL_1 + 7), RL0 ; place data in the message object
mov R1,#0F7FFh
mov MSG_CTL_1, R1 ; clear CPUUPD to invoke the transmission
```

Auf der Empfängerseite kann entweder auf das Bitfeld NEWDAT gepollt werden (dadurch ist allerdings die CPU ständig blockiert) oder eine Interrupt Service Routine aufgesetzt werden. Die empfangenen Daten sind in der Folge aus dem Message Object auszulesen und das NEWDAT Feld ist zurückzusetzen.

2.13.4 Beispiel zum CAN Interface

Ein Sensor Knoten schickt über den CAN Bus eine zwei Byte lange Nachricht an einen Host Knoten. Das erste File enthält einige allgemeine Header und Makrodefinitionen sowie eine Intialisierungsroutine des C167 CAN Moduls.

```
/* can.h - Header File */
#include <reg167.h>        /* C167 Registerdefinitionen */
#include "canregs.h"       /* C167 CAN Registerdefinitionen */

/* Makro zum Befüllen der Arbitration Register mit dem Identifier */
#define ARBITR(id) ((unsigned long)(id) >> 21 & 0x000000ff | \
(unsigned long)(id) >>  5 & 0x0000ff00 | \
(unsigned long)(id) << 11 & 0x00ff0000 | \
(unsigned long)(id) << 27)
     /* Message Configuration Register Makro */
#define MSG_CFG(dlc,dir,xtd) ((dlc) << 4 | (dir) << 3 | (xtd) << 2)
     /* 11-bit ID wahlweise festgelegt auf: 101 0101 0110 */
#define CAN_MSG (0x556ul << 18)
     /* Die Länge des verwendeten Message Objekts */
#define LEN_CAN_MSG 2
     /* Resetiert INTPND im zugehörigen Message Control Register */
#define CAN_OBJECT_INT_RESET 0xfffd
     /* Welches Message Object wird verwendet ? (1-15) */
#define MSG 2

/* can.c - File */
#include "can.h"        /* Definition von Makros und Nachrichten IDs */
/* Die Funktion setup_can() initialisiert das CAN Modul
   -> wird der Funktion eine 1 übergeben so soll das Objekt
   die Nachricht verschicken
```

```
   -> wird eine 0 übergeben so soll das Objekt eine Nachricht empfangen */
void setup_can (int direction) {
   int object_number;
   /* Setze Bit:
         INIT in Control/Status Register
                -> Startet die Initialisierung des CAN Controllers
         CCE erlaubt der CPU Zugriff auf das Bit-Timing Register
   */
   CAN_CTL_STAT = CAN_INIT_ | CAN_CCE_;
   /* Bit Timing:
      0x2440 => 1 MBit/s
      0x49c4 => 125kBit/s
      ....
   */
   CAN_BIT_TIMING = 0x49C4;
   /* Markiere alle CAN Objekte ungültig */
   for (object_number = 1; object_number <= 15; object_number++) {
      CAN_MSGOBJ[object_number].msg_ctl = MSGVAL_CLR & CPUUPD_SET;
   }
   /* Die folgenden mask values geben an, daß alle Bits der
      message ID beim Vergleichen mit einer empfangenen ID
      berücksichtigt werden
   */
   CAN_MASK_SHORT = 0xffff;
   CAN_UMASK_LONG = 0xffff;
   CAN_LMASK_LONG = 0xf8ff;
   /* notwendig für das BASIC CAN feature */
   CAN_UMASK_LAST = 0x0000; /* Setze Message15 Mask */
   CAN_LMASK_LAST = 0x0000; /* Setze Message15 Mask */
   /* Wenn das Programm Message Objekt 15 nicht verwendet ist es nicht nötig
      mask of last message Register zu initialisieren
      (CAN_UMASK_LAST und CAN_LMASK_LAST).
   */
   /* setzt die Arbitration Register */
   CAN_MSGOBJ[MSG].arbitr = ARBITR(CAN_MSG);
   /* Initialisiert das Message Configuration Register
      -> Länge der Message
      -> Richtung 1: Transmit, 0: Receive
      -> 11 Bit Identifier
   */
   CAN_MSGOBJ[MSG].msg_cfg = MSG_CFG(LEN_CAN_MSG, direction, 0);
   /* Markiert das verwendete Message Objekt als gültig */
   CAN_MSGOBJ[MSG].msg_ctl =
      /* clear bits     set bits */
         INTPND_CLR &
         RXIE_CLR &
         TXIE_CLR &
                        MSGVAL_SET &
         NEWDAT_CLR &
         CPUUPD_CLR &
         TXRQ_CLR &
         RMTPND_CLR;
   /* CAN_IE_ wird gesetzt -> damit werden CAN Sende und Empfangs Interrupts
```

```
      akzeptiert.
   */
   CAN_CTL_STAT = CAN_IE_;
}
```

Das folgende File setzt einen Knoten als Empfänger von Message Objekt 2 mit einer zwei Byte langen Nachricht auf.

```
#include <stdio.h>       /* Standard I/O Routinen z.B. printf() */
#include <intrins.h>     /* C167 Spezial Funktionen z.B. _idle_() */
#include <reg167.h>      /* Registerdefinitionen des C167 */
#include "can.h"         /* enthält Makros sowie die ID der Nachricht */

sbit DP45 = DP4^5;     /* Portdefinition -> CAN_RxD */
sbit DP46 = DP4^6;     /* Portdefinition -> CAN_TxD */

int rcv_buffer[2];

/* Die Funktion zur Initialisierung des CAN Moduls befindet sich in can.c */
extern void setup_can (int direction);

/* CAN Interrupt Service Routine */
void canisr (void) interrupt 0x40 {

  unsigned char status;
  unsigned char intid;

  while(intid=CAN_INTID){
    status = CAN_CTL_STAT;          /* sichere Status Info. -> status */
    CAN_CTL_STAT &= 0x00ff;         /* lösche CAN Status Register */

    switch(intid){     /* Welches Message Object hat den Interrupt aktiviert ? */
      /* Message 2 Interrupt ? */
    case 0x04: {
      /* -> Setze das Objekt retour */
      CAN_MSGOBJ[MSG].msg_ctl = CAN_OBJECT_INT_RESET;
      /* sichert die beiden Bytes die der Sender geschickt hat */
      rcv_buffer[0] = CAN_MSGOBJ[MSG].msg[0];
      rcv_buffer[1] = CAN_MSGOBJ[MSG].msg[1];
      /* Gibt das Message Objekt wieder frei */
      CAN_MSGOBJ[MSG].msg_ctl |= NEWDAT_CLR;
      break;
    }
    /* Es werden keine anderen auslösenden Quellen in diesem Bsp. erwartet! */
    default: break;
    }
  }
}

/* H A U P T P R O G R A M M des E M P F Ä N G E R S */
int main (void)
{
```

```
   DP45 = 0;          /* Port 4.5 = input */
   DP46 = 1;          /* Port 4.6 = output */
   SYSCON |=0x0004;   /* enables X-Peripherials (z.B. CAN) */
   setup_can(0);      /* set up CAN interface -> 0: receive */
   /* Wird ein CAN Frame empfangen soll der Interrupt aktiviert werden */
   CAN_MSGOBJ[MSG].msg_ctl = RXIE_SET;
   XP0IC = 0x005a;    /* enables CAN interrupt */
   PSW |= 0x0800;    /* all interrupts enabled */

   /* Run */
   while (1) {       /* infinite loop */
      _idle_();
   }
}
```

Schließlich illustriert das folgende File den Sender einer zwei Byte langen Nachricht über CAN Message Objekt 2.

```
#include <stdio.h>         /* Standard I/O Routinen z.B. printf() */
#include <intrins.h>       /* C167 Spezial Funktionen z.B. _idle_() */
#include <reg167.h>        /* Registerdefinitionen des C167 */
#include "can.h"           /* enthält Makros sowie die ID der Nachricht */

sbit DP45 = DP4^5;         /* Portdefinition -> CAN_RxD */
sbit DP46 = DP4^6;         /* Portdefinition -> CAN_TxD */

/* Die Funktion zur Initialisierung des CAN Moduls befindet sich in can.c */
extern void setup_can (int direction);

/* CAN Interrupt Service Routine */
void canisr (void) interrupt 0x40 {
  unsigned char status;
  unsigned char intid;

  while(intid=CAN_INTID){
    status = CAN_CTL_STAT;          /* sichere Status Info. -> status */
    CAN_CTL_STAT &= 0x00ff;         /* lösche CAN Status Register */
    switch(intid)   /* Welches Message Object hat den Interrupt aktiviert ? */
      {
        /* Message 2 Interrupt ? -> Setze das Objekt retour */
      case 0x04: { CAN_MSGOBJ[MSG].msg_ctl = CAN_OBJECT_INT_RESET;
      break;
      }
      default: break; /* Keine andere Quellen in diesem Bsp. */
    }
  }
}

/* H A U P T P R O G R A M M des S E N D E R S */
void main(void)
{
   DP45 = 0;          /* Port 4.5 = input */
```

```
   DP46 = 1;          /* Port 4.6 = output */
   SYSCON |=0x0004;   /* enables X-Peripherials (e.g. CAN) */
   setup_can(1);      /* set up CAN interface -> 1: transmit */
   XP0IC = 0x005a;
   PSW |= 0x0800;    /* all interrupts enabled */
   /* Message Control Register: NEDDAT_SET -> Es werden neue Daten in ein
                                              Message Object geschrieben
                                CPUUPD_SET -> Das Message Object soll noch nicht
                                              verschickt werden -> es werden erst
                                              Daten eingetragen */
   CAN_MSGOBJ[MSG].msg_ctl = NEWDAT_SET & CPUUPD_SET;
   CAN_MSGOBJ[MSG].msg[0] = 0xa;           /* es wird ein 0xa eingetragen */
   CAN_MSGOBJ[MSG].msg[1] = 0xb;           /* es wird ein 0xb eingetragen */
   /* Message Control Register: TXRQ_SET -> Verschicke Nachrichten
                                CPUUPD_CLR -> Nachrichten stehen bereits in den
                                              Message Objects
                                TXIE_SET -> generiere einen Interrupt nach dem
                                            Versenden einer Nachricht */
   CAN_MSGOBJ[MSG].msg_ctl = TXRQ_SET & CPUUPD_CLR & TXIE_SET;
   while (1) {       /* Endlosschleife */
     _idle_();       /* Stromsparmodus */
   }
}
```

3 Real Time Operating Systems (RTOS)

Einen wesentlichen Kernpunkt der Mikrocomputer Architektur bildet u.a. der Bereich der *Embedded Systems*. Darunter versteht man i.a. applikationsspezifische Mikrocomputerschaltungen mit dedizierter Software integriert in eine zu steuernde Umgebung. Dazu zählt jedes Gerät oder jede Anlage die sich “intelligent” in ihre Umgebung integriert, angefangen vom Videorecorder, dem Faxgerät bis hin zu aufwendigen, komplexen Systemen im Bereich der Industriesteuerung. Die meisten Applikationen im Bereich der Embedded Systems erfordern ein hohes Maß an paralleler Verarbeitung und kurze Reaktionszeiten auf externe Ereignisse. Um die Entwicklungszeiten kurz zu halten werden in diesem Bereich Multitasking Betriebsysteme bzw. Real Time Kernels eingesetzt. Diese Real Time Operating Systems müssen vom Entwickler i.a. lediglich auf die entsprechende Zielhardware angepaßt werden. Ein RTOS erlaubt ein flexibles Aufteilen von Systemresourcen (CPU, Speicher, etc.) auf einzelne *Tasks* (Prozesse). Die Aufgaben eines Betriebsystems umfassen i.a.:

- Taskverwaltung (Scheduler, Dispatcher, Taskerzeugung, -terminierung, etc.)
- Betriebsmittelverwaltung (Programm- und Arbeitspeicherverwaltung, I/O-Handling, Filesystem, etc.)
- Zuverlässigkeitsmaßnahmen (Übertragungs-, Speichersicherung, Fehlertoleranzmechanismen, etc.)
- Schutzmaßnahmen (Zugriffsberechtigung, Objektschutz, etc.)

In den folgenden Abschnitten sollen einige Konzepte kurz angeführt werden, da diese eng mit der Mikrocomputer Architektur verwandt sind. Für eine ausführlichere Beschreibung der diversen Mechanismen sei hier auf weiterführende Literatur verwiesen [RL94], [SBS92].

3.1 Das Taskkonzept

Tasks sind die funktionalen und strukturierenden Einheiten von Betriebsystemen. Ein Task ist ein ausführbares Programm zusammen mit den dazugehörigen aktuellen Werten des *Programm Counters*, der Register, Variablen und Betriebsmittel (Speicher, I/O, Signale, Mailboxen, etc.). Für die Implementierung einer Task unterhält das Betriebsystem eine eigene Tabelle, den *Task Control Block* (TCB). Der TCB enthält die Prozeßinformation sowie den CPU und Systemkontext für den jeweiligen Task. Die Prozeßinformation wiederum definiert den Prozeßzustand und die Prozeßpriorität. Die einzelnen Taskzustände sind:

- existent
- non-existent
- ready: Eine Task ist bereit, wenn sie alle Betriebsmittel außer die CPU zugeteilt bekommen hat.
- running: Eine Task läuft, wenn sie die CPU gerade benützt.
- blocked: Eine Task ist blockiert oder suspended, wenn sie ihre Operation nicht ausführen kann, weil sie auf die Zuteilung von Betriebsmittel wartet.
- terminated: Eine Task ist beendet, wenn sie alle Anweisungen abgearbeitet und die ihr zugeteilten Betriebsmittel freigegeben hat.

Anhand der Prozeßpriorität entscheidet der *Scheduler* welchem Prozeß die CPU zugeordnet wird. Die meisten RTOS unterstützen hierbei im Regelfall ein *preemptives Scheduling*[p]. D.h. wenn ein Task mit höherer Priorität ansteht, so wird dem laufenden Task die CPU entzogen obwohl dieser eventuell noch nicht vollständig abgearbeitet wurde. Der Dispatcher ist für das ordnungsgemäße Umschalten des CPU und Systemkontextes von einem Task auf den nächsten zuständig.

3.2 Tasksynchronisation und Kommunikation

Tasks müssen während ihrer Arbeit auf Betriebsmittel (Speicher, Peripherie, etc.) zugreifen, die aber in der Regel nicht exklusiv benutzt werden können. Man spricht in diesem Fall von wechselseitigen Ausschluß (*mutual exclusion*) zwischen nebeneinander laufenden Prozessen. In diesem

---

[p] Auch das RTX166-Full RTOS der Fa. Keil unterstützt diese Scheduling Methode, nicht jedoch die abgespeckte Version RTX166-Tiny, die nach dem *Round-Robin* Verfahren arbeitet. Beim Round-Robin Verfahren ist jeder Task für eine vordefinierte Timeout Periode aktiv bevor er wieder suspendiert und die CPU einem anderen Task überlassen wird.

Fall muß das Betriebsystem *Semaphore* zur Synchronisation der einzelnen Tasks zur Verfügung stellen. Weiters arbeiten desöfteren Tasks an einer gemeinsamen Aufgabe, wobei mitunter die Reihenfolge der Abarbeitung der einzelnen Tasks eine wichtige Rolle spielt. Daten werden z.B. erst von einem Prozeß erzeugt, bevor diese von einem anderen weiterverarbeitet werden können. Hierfür hat sich der Begriff der *cooperative Tasks* eingebürgert.
Die *Inter-Process-Communication* (IPC) dient zum Informationsaustausch und zur Synchronisation von kooperierenden Tasks. Dieser Kommunikationsmechansimus kann entweder asynchron über *Mailboxes* oder synchron über *Pipes* erfolgen, siehe Abbildung 50. Beim asynchronen System senden die Tasks ihre Nachrichten asynchron an die Mailbox. Ebenso asynchron nehmen die Empfängerprozesse die Nachrichten von dort entgegen, wobei sie hier zumeist die erste Nachricht in der Warteschlange erhalten. Falls ein Prozeß seine Nachrichten nicht in der gesendeten Reihenfolge benötigt müssen zusätzliche Mailboxes eingerichtet werden. Eine gravierende Schwäche dieses sehr weit verbreiteten Systems ist, daß die kommunizierenden Prozesse ihre eigenen Regeln und Methoden für die Kommunikation definieren müssen. Die Koordinierung des Datenaustauschs wird somit Aufgabe der Applikation und ist somit über viele Prozesse verstreut. Beim synchronen Message Passing Verfahren werden Nachrichten direkt von einem Prozeß zum anderen weitergesendet und haben zu jeder Zeit eine Identität, einen Sender, einen Empfänger und einen Besitzer. In diesem Fall wird nur die Nachricht versendet und die Daten wechseln in der Regel zumeist einfach nur den Besitzer. Nur in dem Fall wenn der Empfängerprozeß in einem anderen Adreßraum arbeitet müssen die Daten umkopiert werden.

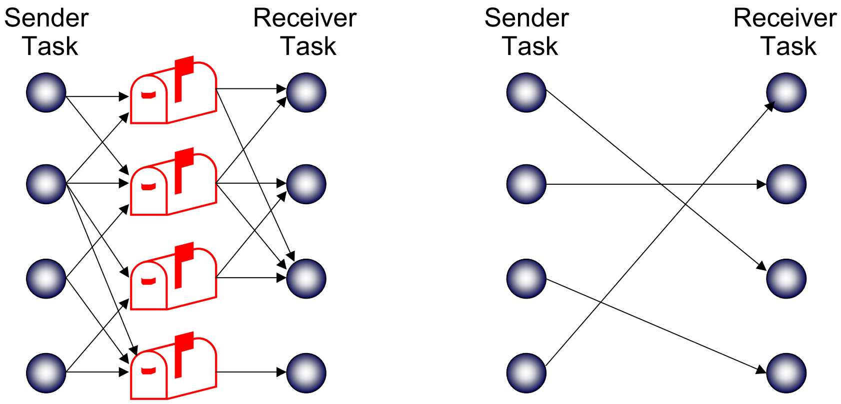


Abbildung 50: Kommunikation über Mailboxes (links) und mittels Message Passing (rechts)

3.3 Beispiel zur Illustration von RTOS Eigenschaften

Das folgende Beispiel wurde aus der Beispielsammlung zur Entwicklungsumgebung der Fa. Keil entnommen. Dieses Beispiel implementiert vier Software Zählschleifen, die mit Hilfe von vier verschiedenen Tasks realisiert werden. Im ersten Task *job0* werden drei Tasks erzeugt. Danach wird in einer Endlosschleife der Zähler *counter0* erhöht. Über die *os_wait* Funktion wird 5 Ticks lang in dieser Schleife verweilt, bis der Task suspendiert wird. Zuvor wurde bereits Task *job1* erzeugt, der *counter1* erhöht, solange bis dieser Task nach 10 Ticks mit Hilfe der *os_wait* Funktion ebenfalls suspendiert wird. Nach dem Task *job1* wurde der Task *job2* erzeugt, der *counter2* inkrementieren läßt, solange bis der Task bei einem Zählerüberlauf ein Signal an Task *job3* sendet, bei dem er ebenso suspendiert wird. Der Task *job3* wird jeweils nur angestoßen, wenn er von *job2* ein Signal erhält und bewirkt ein erhöhen der Zählvariable *counter3*.

```
#include <rtx166t.h>                  /* RTX-166 tiny functions & defines     */

int counter0;                         /* counter for task 0                   */
int counter1;                         /* counter for task 1                   */
int counter2;                         /* counter for task 2                   */
int counter3;                         /* counter for task 2                   */

/* Task 0 'job0':  RTX-166 tiny starts execution with task 0 */
void job0 (void) _task_ 0  {
  os_create_task (1);                 /* start task 1                         */
  os_create_task (2);                 /* start task 2                         */
  os_create_task (3);                 /* start task 3                         */

  while (1)  {                        /* endless loop                         */
    counter0++;                       /* increment counter 0                  */
    os_wait (K_TMO, 5, 0);            /* wait for timeout: 5 ticks            */
  }
}

/*   Task 1 'job1':  RTX-166 tiny starts this task with os_create_task (1)    */
void job1 (void) _task_ 1  {
  while (1)  {                        /* endless loop                         */
    counter1++;                       /* increment counter 1                  */
    os_wait (K_TMO, 10, 0);           /* wait for timeout: 10 ticks           */
  }
}

/*    Task 2 'job2':  RTX-166 tiny starts this task with os_create_task (2)   */
void job2 (void) _task_ 2  {
  while (1)  {                        /* endless loop                         */
    counter2++;                       /* increment counter 2                  */
    if (counter2 == 0)  {             /* signal overflow of counter 2         */
      os_send_signal (3);             /* to task 3                            */
    }
  }
}

/*    Task 3 'job3':  RTX-166 tiny starts this task with os_create_task (3)   */
void job3 (void) _task_ 3  {
  while (1)  {                        /* endless loop                         */
    os_wait (K_SIG, 0, 0);            /* wait for signal                      */
    counter3++;                       /* process overflow from counter 2      */
  }
}
```

4 Bemerkungen

In dem vorliegenden Skriptum sind bewußt stark vereinfachte Beispiele enthalten. Das Hauptaugenmerk wurde zumeist auf die grundlegende Darstellung der jeweiligen Funktionen gelegt. Es ist jeder Übungsteilnehmer herzlichst aufgefordert Verbesserungen diesbezüglich anzubringen. Auf der WWW Seite zur Übung http://mc.ict.tuwien.ac.at ist ein kleines Archiv mit weiteren Beispiele einsichtig, die so manches ausführlicher und technisch “sauberer” implementieren. Weiters befinden sich dort zahlreiche Links und Hinweise zu weiteren Quellen, die dem interessierten Leser sicherlich dienlich sein können. Durch die verfügbare Demosoftware und ein kurzes Tutorial wird dies ergänzt.

Konstruktive Kritik, weiterführende Beispiele und Hinweise richten Sie bitte am geeignetesten per e-mail an:

jandl@ict.tuwien.ac.at

Danksagung

Ich möchte an dieser Stelle ganz besonders Hrn. Ing. Brezovits von der Fa. Infineon für seine tatkräftige Unterstützung, sei es durch Informationen oder auch durch zur Verfügung Stellung von Übungs-Hard- und Software sowie zahlreichen Datenblättern und Manuals, recht herzlich danken.

Martin HORAUER

Literatur

[Si96a] Infineon AG, “C167 Derivates User´s Manual”, Infineon AG München, Version 2.0 Edition 03.96.

[Ets94] Etschberger K., “CAN Controller Area Network”, Carl Hanser Verlag 1994.

[Si96b] Infineon AG, “CAN-Anschluß für C166 / C500”, Application Note 03/96.

[Si95] Infineon AG, “C167 Data Sheet”, 1995.

[Yer98] Yeralan P.E., “Introductory Experiment in Controller Area Networks: Voice over CAN”, CONTACT Magazin Ausgabe 1, http://www.spacetools.com, 1998.

[FL94] Flik Th., Liebig H., “Mikroprozessortechnik”, Springer Verlag 1994.

[Doe95] Dörrhofer S., “Messen, Steuern und Regeln mit dem Mikrocontroller 80C166”, Franzis Verlag 1995.

[Bos91] Robert Bosch GmbH, “CAN Specification Version 2.0”, 1991.

[Joh93] Johannis R., “Handbuch des 80C166”, Infineon 1993.

[MS95] Mattheis K-H., Storandt S., “MC-Tools 17 - Arbeiten mit C166-Controllern”, Ottmar Feger Verlag 1995.

[D78] DIN Norm 44 300, “Normen über Informationsverarbeitung”, Beuth Verlag, 1978.

[To97] Toschiba, “Datenblatt: TC554161F”, http://www.toshiba.com, 1997.

[Da98] Dallas Semiconductor, “Datenblatt zum Baustein: DS12887, Real Time Clock”, http://www.dalsemi.com, 1998.

[SBS92] Schildt G.H., Blieberger, J. Schmid U., Stöckler S., “Informatik”, Springer Verlag Wien 1992.

[RL94] Rembold U., Levi P., “Realzeitsysteme zur Prozeßautomatisierung”, Hanser Verlag, 1994.

[HP94] Hennesy J.L., Patterson D.A., “Computer Architecture, A Quanitative Approach”, Morgan Kaufmann Publishers Inc., 1994.

[Tan90] Tannenbaum A.S., “Structured Computer Organization”, Prentice Hall, 1990.

[Sch98] Schwiegelshohn U., “Technische Informatik I+II”, Vorlesungsskriptum der Technischen Universität Dortmund, http://www-ds.e-technik.uni-dortmund.de/WEB-E/Lehre, 1998.

[Sch99] Schwiegelshohn U., “Parallele Rechnersysteme I+II”, Vorlesungsskriptum der Technischen Universität Dortmund, http://www-ds.e-technik.uni-dortmund.de/WEB-E/Lehre, 1999.

[Tab95] Tabak D., “Advanced Micorprocessors”, McGraw-Hill, 1995.

Index