Otto-von-Guericke-Universität Magdeburg

Institut für Verteilte Systeme
AG Eingebettete Systeme und Betriebssysteme

In-situ Rekonfiguration von Mikrocontrollern

Diplomarbeit

Jörg Diederich

Sommersemester 2007

Betreuer:

Prof. Jörg Kaiser
Dipl. Inform. Michael Schulze

Danksagung

Die vorliegende Diplomarbeit entstand an der Arbeitsgruppe Eingebettete Systeme und Betriebssysteme des Instituts für Verteilte Systeme. An dieser Stelle möchte ich mich bei allen Personen bedanken, die zur Entstehung dieser Arbeit beigetragen sowie mich mit Rat und Tat unterstützt haben.
Bedanken möchte ich mich bei Professor Kaiser für die Bereitstellung und Unterstützung des Themas. Besonderer Dank gilt meinem Betreuer, Michael Schulze, der mir mit seinem Einsatz und seiner fachlichen Kompetenz zur Seite stand. Bedanken möchte ich mich auch bei den weiteren Angehörigen der Arbeitsgruppe. Insbesondere vertreten durch Sebastian Zug und Thomas Kiebel, standen sie meinen Fragen und Problemen geduldig und hilfsbereit gegenüber.
Bedanken möchte ich mich bei Professor Nett für seine Tätigkeit als Zweitgutachter der Arbeit.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Diplomarbeit „In-situ Rekonfiguration von Mikrocontrollern“ selbständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe. Alle wörtlichen und sinngemäßen Zitate sind als solche gekennzeichnet

Magdeburg, 22. August 2007
Jörg Diederich

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Abseits von der öffentlichen Aufmerksamkeit, führen eingebettete Systeme ein Schattendasein. Wie ihre Bezeichnung schon vermuten lässt, sind sie in anderen Systemen eingebunden und somit Teil eines größeren Ganzen. Sie besitzen daher meist keine direkte Schnittstelle zum Menschen, das umgebende System übernimmt diese Aufgabe. Ein eingebettetes System wird so schwerlich direkt als in Bewegung oder in Aktion angesehen werden können. Das umgebende System abstrahiert die beinhalteten Systeme vollkommen.

Aufgrund des zielgerichteten Einsatzes von Ressourcen machen eingebettete Systeme dennoch auf sich aufmerksam. Sie sind Kombinationen aus Hard- und Software, entworfen zum Ausführen einer ganz speziellen Funktion [3]. Jeweils zugeschnitten auf diese Funktion, werden idealerweise nur die Ressourcen aufgewendet, die zur Ausführung der Funktion auch wirklich erforderlich sind. So kommen eingebettete Systeme überall dort zur Anwendung, wo der alternative Einsatz normaler Rechnersysteme einen unverhältnismäßig hohen Aufwand erfordert. Nahe liegend kann das den finanziellen Aufwand betreffen. Auch Aufwendungen für die Behandlung von Umwelteinflüssen, etwa von Wärme, oder für den notwendigen Bauraum können Entscheidungskriterien sein. Mit der laufenden Debatte rund um die globale Erwärmung immer mehr ins Gewicht fällt der Energieaufwand zur Erfüllung einer bestimmten Funktion.

Oftmals zentrales und auch alleiniges Element aktueller eingebetteter Systeme ist ein Mikrocontroller. Bestehend aus den drei Komponenten Prozessor, Speicher und Peripherie, ist auf einem Mikrocontroller die geforderte Funktionalität vieler Anwendungsfälle bereits integriert. Seit dem erstem Auftreten in den 70er Jahren des vergangenen Jahrhunderts hat die von Mikrocontrollern angebotene Funktionalität kontinuierlich zugenommen. Dies betrifft alle Komponenten. Die ersten Mikrocontroller bestanden aus Prozessoren für Taschenrechner, zusammen mit wenig Speicher für Programme und Daten und nur einer Handvoll Eingabe- und Ausgabeports [18]. Heute bereits etablierte Mikrocontroller, beispielsweise repräsentiert durch Exemplare der AVR-Basic-Familie der Firma Atmel, verfügen über ungleich größere Möglichkeiten. So sind überwiegend mehr als ein Dutzend Ein- und Ausgabepins vorhanden, zusammen mit mehreren Kilobyte wiederbeschreibbarem, nichtflüchtigem Speicher und mehreren hundert Byte flüchtigem Speicher. Gleiches gilt für serielle Ein- bzw. Ausgabemöglichkeiten wie beispielsweise UART und SPI [44]. Auch Anschlussmöglichkeiten an Bussysteme mit Hilfe bereits integrierter Bausteine, wie beispielsweise an CAN (Controller Area Network) [35], sind nicht mehr nur Exoten vorbehalten.

Mit weiter zunehmenden Möglichkeiten der Integration kann angenommen werden, dass sich die aufgezeigte Entwicklung in Zukunft fortsetzt. Vom Standpunkt der Effizienz muss dieser Vorgang durchaus kritisch betrachtet werden. Aufgrund des hohen

Kostendrucks, wiederum hervorgerufen durch massenhaften Einsatz, sollten Mikrocontroller in einem ersten Ansatz nur die Funktionalität enthalten, die für das Ausführen der beabsichtigten speziellen Funktion notwendig ist. Um diesem Kriterium zu genügen, gleichzeitig jedoch auch für komplexe Aufgaben flexibel geeignet zu sein, bietet sich die Anwendung im Rahmen verteilter Systeme an. Verschiedene eigenständige Mikrocontroller führen ihre jeweils ganz spezielle Funktion aus. Dabei können sie über Kommunikationsschnittstellen mit anderen beteiligten Systemen zusammenarbeiten. Unter Ausnutzung existierender Ressourcen wird es so möglich, die Daten dort zu erfassen, auszuwerten und entsprechend zu reagieren, wo immer es erforderlich ist. Seinen bisherigen Höhepunkt findet dieses Vorgehen im Bereich der so genannten Sensornetze. Hier erfüllen einzelne eingebettete Systeme nur jeweils einen Bruchteil einer Gesamtfunktion. Erst durch Kommunikation und Zusammenarbeit vieler Teilnehmer eines Sensornetzes wird es möglich, die beabsichtigte Gesamtfunktion zu realisieren.

1.1. Motivation

Sowohl zunehmende Funktionalität als auch Zusammenarbeit in einem verteiltem System schließen sich nicht aus. So können an exponierten Positionen platzierte Mikrocontroller eine Vielzahl von Teilfunktionen ausführen, gleichzeitig aber auch als Datenquellen für verschiedene andere Mikrocontroller im verteilten System dienen. Beide Punkte beeinflussen aber den Lebenszyklus der auf den Mikrocontrollern eingesetzten Software deutlich.

Mit wachsenden Möglichkeiten der Hardware steht für die Software ein immer größerer Spielraum zur Verfügung. Potentiell erhöht sich damit die Komplexität der entwickelten Programme. Nach dem Grundsatz, dass ein eingebettetes System für eine spezielle Funktion entworfen wird, sollte ein Programm jede Funktionalität benutzen müssen, um letztendlich die beabsichtigte Funktion zu realisieren. Vor diesem Hintergrund ist die Modularisierung der Software ein bisher unausweichlicher Schritt, um in der Entwicklungsphase die Produktivität zu erhalten. Unter Beachtung der Schnittstellen können dedizierte Module unabhängig voneinander und somit parallel entwickelt werden. Die Wiederverwendung von bewährten Modulen spart nicht nur Entwicklungszeit, sondern hilft auch, die Anzahl von Fehlern zu begrenzen.

Dennoch lassen sich Fehler nicht vermeiden, im Gegenteil, ihre Anzahl steigt frei nach Murphy's Law mit der Komplexität der Software [15]. Mit Erkennung von Fehlern kann eine Rekonfiguration der auf den eingebetteten Systemen installierten Software notwendig werden. In der Entwicklungsphase ist dies während der Fehlersuche, dem so genanntem Debugging, ein üblicher und oft wiederholter Vorgang. Derartige Rekonfigurationen, im Folgenden auch Aktualisierungen genannt, sind aber auch Teil der Wartungsphase der Software. So können über Rekonfigurationen Änderungen eingepflegt werden, um das verteilte System an neue Anforderungen anzupassen. Neue Funktionen kommen hinzu, nicht mehr benötigte Funktionen werden entfernt.

In verteilten Systemen, in denen verschiedene Mikrocontroller kooperieren, sind Rekonfigurationen ungleich schwieriger durchführbar als in individuell eingesetzten Einzel-

systemen. Für ein manuelles Vorgehen ist der Zugang zu jedem einzelnen Mikrocontroller erforderlich. Einzelne Exemplare können aber nur eingeschränkt oder überhaupt nicht zugänglich sein. Beispiele dafür sind Einsatzgebiete wie entfernte Inseln, Kernreaktoren oder das Weltall. Vorstellbar sind auch inakzeptable Störungen der Funktion durch den Zugang zum jeweiligen Teilsystem. Diese Problematik betrifft aber ebenfalls Einzelsysteme. Jedoch steigt in einem verteilten System der Aufwand für eine Rekonfiguration allein mit der Anzahl der Teilsysteme linear an. Nur unter außergewöhnlichen Umständen kann der Aufwand möglicherweise noch akzeptiert werden. Verglichen werden kann dies mit Rückrufaktionen von beispielsweise Automobilherstellern, die trotz hohen Aufwands am Ende sinnvoller sind als eine mögliche nachfolgende Behandlung entstandener Schäden. Aber auch in diesem Szenario nähme der Aufwand bereits drastische Züge an, wäre der Rückruf einer ganzen Fahrzeugflotte eines Automobilherstellers notwendig. Schließlich weist jedes Fahrzeugmodell seine eigene Charakteristik auf. Dies kann als Sinnbild für die inhärent vorhandene Heterogenität eines aus mehreren kooperierenden Systemen bestehenden Systems gelten. Beim Entwurf des verteilten Systems werden unter Umständen bereits mehrere verschiedene Mikrocontroller verwendet, um mögliche individuelle Vorteile bei der Erfüllung von Teilfunktionen zu nutzen. Zudem werden Mikrocontroller in vielen Anwendungsgebieten über lange Zeiträume eingesetzt. Gelegentliche Ausfälle, Ergänzungen oder Anpassungen an neue Anforderungen resultieren in einer Vielzahl verschiedener, aber immer noch miteinander kooperierender Einzelsysteme. Die nahe liegende Lösung für die Problematik der Rekonfiguration eines verteilten Systems lautet Ausnutzung existierender Ressourcen. Da die einzelnen Teilsysteme zur Erfüllung der Gesamtfunktion miteinander kommunizieren müssen, kann diese Kommunikation auch zur Verbreitung und Installation von Aktualisierungen genutzt werden. Die mögliche Vielfalt der beteiligten Systeme erschwert die Realisierung der Lösung deutlich. Verschiedenste Kommunikationsmittel und Kommunikationspartner müssen berücksichtigt werden. Eine daraus resultierende Behandlung einzelner Teilsysteme, beispielsweise durch jeweils spezifische Arten der Kommunikation, macht die Reduzierung des Aufwands teilweise wieder zunichte.

1.2. Zielstellung

Die Heterogenität von verteilt arbeitenden Mikrocontrollern und der Modulcharakter der jeweils eingesetzten Software sind grundlegende Annahmen für die vorliegende Arbeit. Diese Annahmen sollen im Rahmen einer Entwicklung für ein Verfahren zur Rekonfiguration der Software berücksichtigt werden.

Die Abstrahierung von der zugrunde liegenden Hardware soll auf die Kommunkationsschnittstelle beschränkt bleiben. Dies lässt den jeweiligen Mikrocontrollern ein Maximum an Freiheiten zur individuellen Realisierung ihrer Funktion. Als gemeinsame Kommunikationsplattform soll mit COSMIC (COoperating SMart devICes) [21] eine nach dem Publish/Subscribe-Prinzip arbeitende Middleware verwendet werden. Nach diesem Prinzip können sich Teilsysteme für den Erhalt von ausgewählten Informationen anmelden.

Bei Verwendung von unterschiedlichen Informationen wird so auch eine Rekonfiguration von einzelnen oder von Gruppen von Teilsystemen möglich.

Beim Durchführen von Rekonfigurationen soll die bereits vorhandene Struktur des einzuspielenden Programms ausgenutzt werden. Liegen Programmteile bereits in einem System vor, erübrigt sich deren Übertragung. Es ist daher eine Möglichkeit vorzusehen, Informationen über bereits existierende Programmteile zu erhalten. Fehlende Programmteile müssen über die Kommunikationsschnittstelle verbreitet werden. Nach Erhalt müssen diese mit den bereits vorliegenden Teilen in Verbindung gebracht werden. Zu jedem Entwicklungsprozess gehört auch die Erprobung. Zu diesem Zweck soll die Entwicklung in Form eines Prototyps implementiert werden. Als Plattform vorgesehen ist der Mikrocontroller AT90CAN128 der Firma Atmel.

Ziel dieser Arbeit ist, Rekonfigurationen in eingebetteten Systemen ein erhöhtes Maß an Flexibilität und Transparenz zu verleihen. Zugleich sollen existierende und etablierte Vorgehensweisen beibehalten werden können. Angestrebt wird, durch den Einsatz einer Middleware ein zentrales Interface für Implementationen auf verschiedenen Plattformen zu schaffen. Dies erleichtert potentiell Entwicklungen für bislang unberücksichtigte Plattformen. Ebenso bietet die Middleware auch eine Kommunikationsmöglichkeit für die Software, welche die eigentliche Funktion eines Mikrocontrollers realisieren soll. Zwischen einer Rekonfiguration von komplett neuer oder lediglich geänderter Software soll möglichst kein Unterschied erkennbar sein. Bestehende und bewährte Programmteile brauchen nicht neu eingespielt werden, neue Programmteile fügen sich in das existierende Gebilde ein. Auf diesem Wege kann zudem die Sicherheit vor Fehlern verbessert und die Übertragung von Programmcode auf das notwendige Maß begrenzt bleiben.
Kein Ziel dieser Arbeit ist die Berücksichtigung des Energieaufwands. Sämtliche beteiligte Systeme verfügen über ausreichende Energiereserven. Weiterhin kein Ziel ist die Verbreitung der Rekonfiguration. Nachrichten erreichen nur die direkt benachbarten Mikrocontroller und werden dort nicht weiter verbreitet. Maßnahmen zu Fragen der Sicherheit betreffen ebenfalls nicht diese Arbeit. Jeder Kommunikationspartner verhält sich regulär. Lediglich die Möglichkeit eines Recovery, um im Falle eines Fehlers mit einem definierten Zustand neu zu beginnen, ist zu betrachten.

1.3. Aufbau der Arbeit

Die nachfolgenden Seiten verteilen sich auf mehrere, jeweils aufeinander aufbauende Kapitel. Kapitel 2 dient der Beschreibung der gegebenen Anforderungen. Vorgestellt werden sowohl Mikrocontroller als auch die Middleware COSMIC. Die Grundlagen der Rekonfiguration erläutert Kapitel 3. Es erfolgt eine Beschreibung des Vorgehens für ein einzelnes und für ein verteiltes System, verbunden mit Erläuterungen zum Format der betrachteten Daten. Kapitel 4 beinhaltet die Vorstellung bereits bestehender Lösungen. Durch die abschließende Abgrenzung wird eine Einordnung der vorliegenden Arbeit gegeben. Die Beschreibung der erarbeiteten Konzepte ist Aufgabe von Kapitel 5. Jeweils betrachtet wird das Konzept der kompletten Rekonfiguration, der Beschränkung auf funktionale Komponenten sowie der Nutzung von Modulen. Den detaillierten Entwurf

ausgewählter Möglichkeiten erläutert Kapitel 6. Im Mittelpunkt stehen die Konzepte der vollständigen Rekonfiguration sowie der funktionalen Aufteilung der Daten. Die Evaluation der auf Basis des Entwurfs entstandenen Implementation beschreibt Kapitel 7. Untersucht und verglichen wird das zeitliche Verhalten einzelner Verfahren. Auf Basis von Messungen erfolgt zusätzlich eine Bewertung des eingesetzten Kommunikationsprotokolls. Die Arbeit schließt mit der in Kapitel 8 gegebenen Zusammenfassung der Ergebnisse sowie dem Ausblick auf weitere Entwicklungsmöglichkeiten.

2. Anforderungen

Verglichen mit anderen Wissenschaften, ist die Entwicklung im Bereich der Informatik noch jung. Dennoch sind die Ansprüche an zu entwickelnde Lösungen bereits heute auf dem Niveau älterer Wissenschaften. Das betrifft Aspekte wie

- Leistungsfähigkeit
- Komplexität
- Zuverlässigkeit
- Möglichkeiten der nachfolgenden Weiterentwicklung
- termingerechte Fertigstellung

Im Gebiet der Ingenieurwissenschaften sind derartige Ansprüche schon länger bekannt. Hier sind über Jahre bereits systematische Vorgehensweisen und Methoden erarbeitet worden, um sämtliche Aspekte zu berücksichtigen. Die Richtlinien VDI 2221 [46] und VDI 2222 [47] fassen die Vorschläge beispielsweise für die Entwicklung und Konstruktion von technischen Produkten zusammen. Eine Anwendung der Vorschläge auch auf Entwicklungen im Bereich der Informatik ist aufgrund der generischen Natur möglich. Dieses Vorgehen ist unter dem Begriff Software-Engineering bekannt [9].

Grundlage im methodischen Entwickeln ist das Erkennen der Anforderungen, die an die zu entwickelnde Lösung gestellt werden. So kann in nachfolgenden Vorgängen zielgerichtet und problemorientiert vorgegangen werden [30].

2.1. Mikrocontroller

Der Begriff Mikrocontroller ist unbestimmt und schwer zu fassen. Vielfach definieren die Hersteller von Computersystemen den Begriff selbstständig und deklarieren ihre Produkte entsprechend. Eine in der Literatur allgemein akzeptierte Definition ist die eines eigenständigen Computersystems, welches Mikroprozessor, Speicher und Peripherie in einer Baugruppe (*Package*) zusammenfasst [33, 4, 18].

Als Herzstück eingebetteter Systeme sind Mikrocontroller dazu bestimmt, eine ganz spezielle Funktion auszuführen. Als Folge der Vielfalt möglicher Funktionen existieren verschiedenste Ausführungen, angeboten von einer Reihe von Herstellern. Die auf dem Markt erhältlichen Mikrocontroller lassen sich wie folgt unterteilen [33]:

- eigenständige (*embedded*) 8-Bit-Mikrocontroller

- 16- und 32-Bit-Mikrocontroller
- digitale Signalprozessoren (*DSP*)

Die Einteilung ist lediglich grob, ihre Grenzen sind fließend. Es sind auch kleinere Mikrocontroller verfügbar. Sie sind hauptsächlich für hohe Stückzahlen interessant, bei denen die Kostenersparnis die verminderte Leistungsfähigkeit aufwiegt. Digitale Signalprozessoren sind, im Gegensatz zu anderen Mikrocontrollern, allein auf die Verarbeitung von analogen Signalen spezialisiert. Zunehmenden Marktanteil erarbeiten sich 16- und 32-Bit-Mikrocontroller. Zum Teil handelt es bei diesen um die Zusammenstellung eines bereits etablierten Mikroprozessors mit Speicher und Peripherie, integriert in einen Baustein. Im historisch begründeten Sinne stehen jedoch 8-Bit-Mikrocontroller als Sinnbild für den klassischen Mikrocontroller. Im Gegensatz zu manchem Vertreter mit höherer Wortbreite integrieren sie alle Bestandteile in einem einzigen Chip. Zusätzliche Bauelemente, wie beispielsweise angeschlossener Speicher, sind für einen Einsatz nicht notwendig. Die Architektur dieser Mikrocontroller ist häufig bereits mit Blick auf die Integration von Mikroprozessor, Speicher und Peripherie entwickelt worden. Beispiele sind Vertreter der 8051- oder der AVR-Architektur. Die weitere Betrachtung im Rahmen dieser Arbeit beschränkt sich mit 8-Bit-Mikrocontrollern auf die klassischen Mikrocontroller. Die Vormacht dieser Vertreter zeigt sich auch in der Betrachtung von Mikrocontrollern in der Literatur [3, 33].

Steuerndes Element eines Mikrocontrollers ist die eingesetzte Software, welche in einem Speicher vorgehalten und vom Mikroprozessor ausgeführt wird. Diese soll im Folgendem mit dem Begriff Anwendung bezeichnet werden. Mit Hilfe individuell angepasster Anwendungen können gleiche Mikrocontroller in verschiedenen Gebieten zum Einsatz kommen. Zusätzlich bieten Mikrocontroller einen hohen Grad der Integration. Mit Auswahl eines geeigneten Exemplars steht die benötigte Funktionalität in einem einzigen Baustein zur Verfügung. Durch die geringe Anzahl an Teilen und Signalwegen kommen Vorteile von reduzierten Kosten, erhöhter Zuverlässigkeit und verbesserter Leistung hinzu. Mit dem flexiblen Einsatz und dem Grad der Integration kann die derzeit beherrschende Stellung der Mikrocontroller in der Welt der eingebetteten Systeme [4] begründet werden.

Verschiedene Faktoren stellen hohe Anforderungen an Mikrocontroller:

- Finanzieller Aufwand
  Jede Funktionalität eines Mikrocontrollers schlägt sich in finanziellem Aufwand nieder, sei es in der Entwicklung, der Fertigung oder der Anschaffung. Ungenutzte Funktionalität zur Durchführung der Aufgabe bedeutet Verschwendung finanzieller Mittel. Ist diese bei geringen Stückzahlen noch vertretbar, führt der breite Einsatz von Mikrocontrollern schnell zu erheblichen Möglichkeiten der Einsparung.

- Verbrauch von Energie
  Im Einsatz in Systemen, welche rund um die Uhr operieren, erhält der Energieverbrauch der Mikrocontroller eine Bedeutung. Lange Zeiträume der Aktivität lassen bereits eine geringe Reduzierung der benötigten Energie in eine erkennbare

Ersparnis resultieren. Für Mikrocontroller in autonomen Systemen ist der effiziente Einsatz von Energie von besonderem Interesse. Ohne fremde Stromversorgung, führen derartige Systeme die zu ihrem Betrieb notwendige Energie vielfach in Batterien mit sich. Der Verbrauch dieses begrenzten Vorrats an Energie legt häufig die Einsatzdauer autonomer Systeme fest.

- Umgebungsbedingungen
  Eingebettete Systeme registrieren und verarbeiten Ereignisse aus ihrer Umgebung. Zu diesem Zwecke sind sie in der Umwelt platziert. Somit sind auch Mikrocontroller den Einflüssen der Umgebung ausgesetzt. Erschütterungen beanspruchen einen Mikrocontroller beispielsweise mechanisch. Schroffe Temperaturwechsel resultieren in thermische Belastungen. Umgebende Gase und Flüssigkeiten greifen einen Mikrocontroller chemisch an. Innerhalb der erkannten und berücksichtigten Bedingungen muss die Funktionsfähigkeit über die beabsichtigte Lebensdauer hinweg gewährleistet sein.

Als Resultat der Anforderungen sind die Möglichkeiten der Mikrocontroller sehr detailliert bemessen. Zum Einsatz kommen kostengünstige, bewährte und energiesparende Technologien genau in dem Maße, wie sie für die beabsichtigte Aufgabe vonnöten sind. Eine konkrete Angabe der zur Verfügung stehenden Möglichkeiten ist jedoch aufgrund der Menge verschiedener Mikrocontroller unmöglich. Die folgenden Angaben sind somit lediglich als Anhaltswerte zu betrachten, um eine ungefähre Vorstellung zu vermitteln:

- Taktfrequenz des Prozessors kleiner oder gleich 16 Megahertz
- Verfügbarer flüchtiger Speicher von weniger oder gleich vier Kilobyte
- Verfügbarer Festspeicher von weniger oder gleich 128 Kilobyte
- Bis zu drei Schnittstellen zur seriellen Kommunikation

Über freie Pins, zumeist aber über die existierenden seriellen Schnittstellen können weitere Peripheriebausteine angeschlossen werden. Geradezu klassische Vertreter sind Kommunikationsbausteine für beispielsweise drahtlose Verbindungen zu anderen Systemen. Zusammen mit dem Mikrocontroller bilden diese Bausteine die wesentlichen Elemente von Platinen bzw. Boards. Hinzu kommen Bauelemente der Stromversorgung sowie häufig weitere unterstützende Elemente wie Taktgeber, Taster oder Leuchtdioden.

Die dargelegten Möglichkeiten haben Auswirkungen auf die Entwicklung von Software. Im Vergleich zur Welt der Personalcomputer sind die Ressourcen, welche einer Software zur Verfügung stehen, sehr stark beschränkt. Der überwiegende Mehrzahl der im PC-Bereich existierenden Software bleibt aus diesem Grunde eine Anwendung auf Mikrocontrollern verwehrt. Dies betrifft insbesondere Software wie Tools, Bibliotheken und Betriebssysteme, welche gemeinhin als selbstverständlich angesehen werden. Verschieden ist auch die Ausführung von Software. Mikrocontroller besitzen keine Speicherhierarchie mit vielen Ebenen. Wie Abbildung 2.1 zeigt, schließt sich unmittelbar an die Register des Mikroprozessors der Hauptspeicher an. Während der Ausführung einer

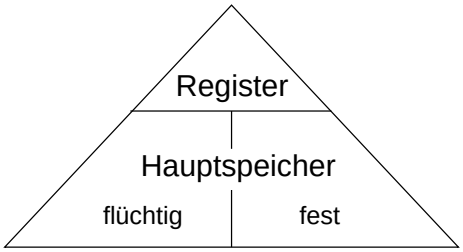


Abbildung 2.1: Speicherhierarchie eines Mikrocontrollers (in Anlehnung an [39])

Anwendung adressiert der Mikroprozessor Befehle und Operanden, die im Hauptspeicher vorgehalten werden. Neben dieser Rolle fungiert der Hauptspeicher auch als Permanentspeicher. Der nichtflüchtige Teil, im Folgendem auch als Festspeicher bezeichnet, bewahrt die Anwendung über einen Reset des Mikrocontrollers hinaus auf.

Mikroprozessoren von Mikrocontrollern, die nach der Harvard-Architektur arbeiten, unterteilen den Hauptspeicher in Programm- und Datenspeicher. Der Programmspeicher beinhaltet die Instruktionen einer Anwendung. Der Datenspeicher umfasst die variablen Operanden von Instruktionen. Aufgrund seiner vielfach besseren und einfacheren Zugriffscharakteristika ist die Verwendung des flüchtigen Hauptspeichers als Datenspeicher nahe liegend. Variablen werden häufig referenziert und modifiziert, ein Bewahren ihres Zustands über mehrere Ausführungen der Anwendung hinweg ist aber im Regelfall nicht notwendig. Hingegen sind Instruktionen unveränderliche Bestandteile einer Anwendung, welche aber dauerhaft vorgehalten werden müssen. Entsprechend bietet sich die Verwendung des nichtflüchtigen Teils des Hauptspeichers für den Programmspeicher an. Mikroprozessoren der von-Neumann-Architektur, welche ebenfalls in Mikrocontrollern zur Anwendung kommen, kennen die Trennung zwischen Programm- und Datenspeicher nicht. Operanden und Instruktionen können in beliebigen Speichern vorliegen. Aus den bereits beschriebenen Gründen bleibt der flüchtige Speicher dennoch meist den Variablen einer Anwendung vorbehalten. Da eine Unterscheidung der Speicher nicht erfolgt, ist ein gemeinsamer Adressraum möglich. Im Gegensatz dazu verwenden Mikroprozessoren der Harvard-Architektur für jeden Speicher einen eigenen Adressraum. Abbildung 2.2 stellt die unterschiedliche Adressierung der beiden Architekturen schematisch dar.

Die Mehrheit der derzeit in Mikrocontrollern zum Einsatz kommenden Mikroprozessoren realisiert die Harvard-Architektur. Beispiele sind Mikrocontroller der 8051-, AVR- oder PICMicro-Architekturen. Mikroprozessoren der von-Neumann-Architektur sind Bestandteile von Mikrocontrollern der 68HC-Reihen der Firma Freescale [33].

2.2. COSMIC

Jeder der verschiedenen auf dem Markt erhältlichen Mikrocontroller besitzt seine eigenen Details und Eigenheiten. Bei Berücksichtigung mehrerer verschiedener Mikrocontroller steht der Programmierer vor der Aufgabe, für gleiche Aufgaben neue Lösungen zu entwickeln.

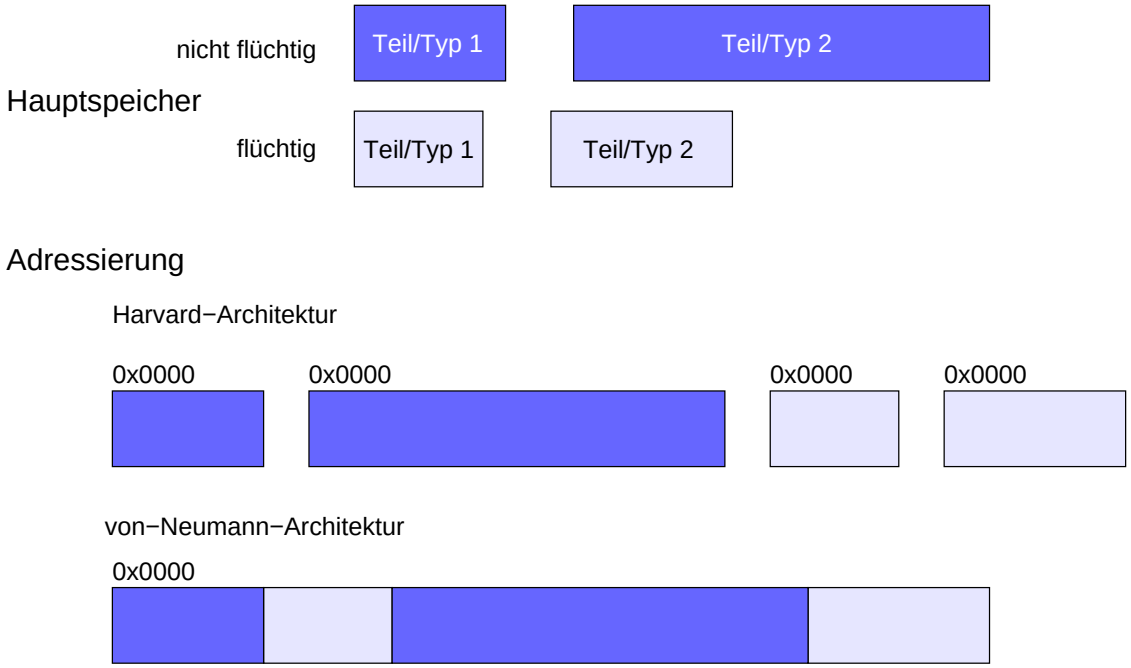


Abbildung 2.2: Adressierung nach Harvard- bzw. von-Neumann-Architektur

Für Aufgaben der Kommunikation existiert mit COSMIC ein Vorschlag zur Vereinfachung. COSMIC abstrahiert zu diesem Zweck vom verwendeten Kommunikationssystem. Anwendungen können für die angebotenen Schnittstellen entwickelt werden, ohne dass sich der Programmierer mit dem Ablauf der Kommunikation auf den niederen Übertragungsebenen beschäftigen muss.

Mit seiner definierten Schnittstelle und seinem Zugriffsprotokoll entspricht COSMIC einer Middleware [38], die zwischen Anwendungen und Kommunikationsschnittstellen angesiedelt ist. Eine aufsetzende Anwendung kann für verschiedenste Einsatzgebiete implementiert werden, ohne selbst ihre Kommunikationsschnittstelle ändern zu müssen. COSMIC ist für die Zusammenarbeit von Mikrocontrollern konzipiert worden. Mikrocontroller registrieren Ereignisse ihrer Umgebung, reagieren mit der Generierung von Daten und verbreiten diese in einem verteilten System. Bei Empfang können andere Teilsysteme die Daten eines Ereignisses verarbeiten und ebenfalls reagieren. Aus übergeordneter Sicht interagieren die einzelnen Teilsysteme miteinander. Das Kommunikationsmodell von COSMIC ist ereignisbasiert. Anwendungen verbreiten Ereignisse mit Hilfe von Daten über die Middleware. Dies kann eine Reaktion auf ein registriertes Ereignis aus der Umwelt oder auf eine interne Zustandsänderung der Anwendung sein. Im Gegenzug werden Anwendungen von der Middleware über die von anderen Anwendungen verbreiteten Ereignisse informiert. COSMIC betrachtet die beteiligten Systeme als autonome Einheiten. Daher bestehen drei Anforderungen an eine Interaktion:

- Ereignisse müssen spontan und jederzeit erzeugt und verbreitet werden können
- Die Kommunikation erfolgt zwischen mehreren Teilnehmern

- Es erfolgt keine Kontrollübergabe vom übertragenden Teilsystem an einen der Empfänger eines Ereignisses

Mit Blick auf die Forderungen erfolgt die Interaktion der Anwendungen nach dem *Publish/Subscribe*-Prinzip, im Folgendem kurz P/S genannt. Eine Anwendung, welche ein Ereignis verbreitet, tritt als *Publisher* auf. Anwendungen, welche am Empfang von Ereignissen interessiert sind, stellen entsprechend die *Subscriber* dar. Zur Unterscheidung der verschiedenen Arten von Ereignissen verwendet COSMIC einen Bezeichner. Gleich klassifizierte Ereignisse sind von den Anwendungen unter dem gleichen Bezeichner zu publizieren. Mit Hilfe des Bezeichners können sich Anwendungen für den Empfang entsprechender Ereignisse an COSMIC anmelden.

Mit Hilfe des P/S können die Anwendungen - und damit die Mikrocontroller - unabhängig voneinander und selbstständig arbeiten. Ein Publisher verbreitet Ereignisse völlig unabhängig von der Anzahl existierender Subscriber. So können eine oder mehrere interessierte Anwendungen im verteilten System vorhanden sein. Möglich ist auch, dass sich keine Anwendung für ein bestimmtes Ereignis interessiert. Aus Sicht des Publishers besteht für keinen der Fälle ein Unterschied. Auf der anderen Seite muss jeder Subscriber erst in dem Falle aktiv werden, in dem ein Publisher die Ereignisse verbreitet, für die sich registriert wurde. Somit findet eine Filterung der Ereignisse statt. Ereignisse, für die keine Anwendung registriert (*subscribed*) ist, brauchen bereits auf Ebene der Middleware nicht weiter verfolgt werden. Selbstverständlich können für eine Art von Ereignissen zu jedem Zeitpunkt mehrere Publisher bestehen.

COSMIC erweitert P/S um das Konzept der Ereigniskanäle (*Event Channel*). Für eine Anwendung ist ein Ereigniskanal die Schnittstelle zu den Kommunikationsmechanismen und zugleich deren Abstraktion. Eine als Publisher auftretende Anwendung verwendet einen Ereigniskanal, um Ereignisse unter einem gleichen Bezeichner zu verbreiten. Jede Anwendung, die Ereignisse empfangen möchte, verwendet für diesen Zweck ebenfalls einen oder mehrere Ereigniskanäle. Durch Registrieren eines Ereigniskanals für einen Bezeichner ist sichergestellt, dass nur entsprechend gekennzeichnete Ereignisse über diesen Ereigniskanal empfangen, aber auch nur gesendet werden. Durch die Verwendung von Ereigniskanälen können die Möglichkeiten der Verbreitung von Ereignissen beschrieben werden. Diesem Zweck dienen die Attribute, mit denen jeder Ereigniskanal versehen werden kann. Angaben wie beispielsweise zur Vorhersehbarkeit der Verbreitung, zu auftretenden Verzögerungen oder auch zur Periodizität von Ereignissen erlauben eine Abstimmung zwischen Publisher und Subscriber.

Für Subscriber besteht keine Möglichkeit, ein Ereignis einem Ursprung zuzuordnen. In seiner Funktion als Middleware verbirgt COSMIC sämtliche Details der Kommunikation vor den Anwendungen. Logisch gesehen, ist das gesamte verteilte System in diesem Sinne eine einzelne, Ereignisse verbreitende Ressource. Eine Zuordnung und damit Trennung der Ressourcen widerspricht der Transparenz dieser Kommunikation. Ein einzelner Ereigniskanal auf Seiten jedes Subscribers fasst den Empfang sämtlicher gleichartiger Ereignisse unter einem Dach zusammen. Die Zusammenfassung ist unabhängig von der Anzahl der Publisher oder deren Kommunikationsverbindung.

3. Grundlagen

Eine Anwendung, die auf einem Mikrocontroller zu Ausführung gebracht werden soll, muss auf selbigem vorliegen. Je nach Art des dafür verwendeten Speichers können vielfach auch nachträgliche Änderungen des Speicherinhalts durchgeführt werden. Mit dieser Möglichkeit ergibt sich die Idee zur Rekonfiguration der auf dem Mikrocontroller laufenden Anwendung. Der Bedarf nach Rekonfiguration hat mehrere Ursachen:

- Fehlerkorrektur
  Nicht akzeptable Fehler in der Anwendung erfordern eine Korrektur. In diesem auch *bug fixing* genannten Prozess werden Veränderungen in der Anwendung notwendig.

- Iterationen in der Entwicklung der Anwendung
  Jeder Softwareentwicklungsprozess ist durch eine Menge von Iteration gekennzeichnet, bestehend aus Entwurf, Implementierung und Test. Spätestens die Testphase benötigt eine Anpassung der auf einem Mikrocontroller laufenden Software.

- Hinzufügen und Entfernen von Funktionalität
  Die Anforderungen an die Anwendung sind fließend. Neue Forderungen an die Funktionalität kommen hinzu, andere fallen heraus. Die Realisierung der veränderten Anforderungen resultiert in der Anpassung der Anwendung.
  Auch aus Kapazitätsgründen des Speichers kann eine Anpassung der Anwendung notwendig werden. Die lange Einsatzdauer von Mikrocontrollern äußert sich potentiell im Bedarf nach verschiedener Funktionalität zu verschiedenen Zeitpunkten. Aus Platzgründen kann vielfach nicht die Funktionalität für alle Zeitpunkte im Speicher vorgehalten werden. Eine Anpassung der Anwendung über den gesamten Einsatzzeitraum wird notwendig.

- Parametrisierung
  Oftmals können die im Einsatz vorherrschenden Bedingungen nur im Einsatz selbst bestimmt werden werden. Eine Anwendung berücksichtigt Bedingungen unter anderem über Parameter, um sich so an das Einsatzgebiet anzupassen. Eine hinreichend genaue Vorhersage sämtlicher Parameter ist jedoch nur in Ausnahmefällen möglich. Im Regelfall erfordert die Anpassung an die nachträglich erkannten Bedingungen Änderungen der Parameter und somit Änderungen an der Anwendung.

Wie angedeutet, kommt dem Speicher, in welchem die Anwendung für den Mikroprozessor bereit steht, eine tragende Bedeutung zu. Für nachträgliche Änderungen muss dieser Speicher wieder und wieder beschreibbar sein. Für diese Art Speicher kommen in Mikrocontrollern derzeit zumeist die folgenden Bausteine zum Einsatz:

- statischer Speicher mit wahlfreiem Zugriff (*Static Random Access Memory*, kurz *SRAM*)
- dynamischer Speicher mit wahlfreiem Zugriff (*Dynamic Random Access Memory* kurz *DRAM*)
- elektrisch löschbarer, programmierbarer Nur-Lese-Speicher (*Electrically Erasable Programmable Read-Only Memory*, kurz *EEPROM*)
- Flash, eine Variante des EEPROM

Zusätzlich können über die Peripherie weitere Speicherbausteine angeschlossen sein. Um semantisch zwischen derartig angeschlossenen Speicher und auf dem Mikrocontroller integriertem Speicher zu unterscheiden, erhalten erstgenannte den sinnbildlich passenden Präfix „extern“, letztgenannte entsprechend den Präfix „intern“. Externe Speicher können aus einer der bereits aufgeführten Arten von Speicher bestehen, aber auch andere Arten verwenden. Ein Beispiel ist löschbarer, programmierbarer Nur-Lese-Speicher(*EPROM*), dessen auffälligste Eigenschaft das Löschen mit Hilfe von ultraviolettem Licht ist.

Für die Speicherung des Programmcodes einer Anwendung in Betracht kommen im allgemeinen Fall die nichtflüchtigen Speicher wie EEPROM und Flash. Somit steht die Anwendung auch im Falle eines Ausfalls der Stromversorgung weiterhin zur Verfügung. Flash-Speicher wurde als Ersatz für EPROM-Speicher aufgestellt und bietet ähnlich hohe Integrationsdichten wie DRAM [16]. Verglichen mit EEPROM-Speicher, ist Flash-Speicher billig herzustellen. Flash-Speicher ist mittlerweile nicht nur in Mikrocontrollern weitaus populärer als EEPROM [3], auch in Geräten der Kommunikations- und Unterhaltungselektronik ist diese Art Speicher überaus verbreitet. Das hat dazu geführt, dass ein Einspielen von neuem Programmcode bereits umgangssprachlich als so genanntes *Flashen* bezeichnet wird.

3.1. Rekonfiguration

Die Rekonfiguration einer Anwendung erfordert zunächst eine erneute Iteration im Entwicklungsprozess dieser Software. Eine Besonderheit der Mikrocontroller ist die Verwendung eines Fremdsystems für diesen Prozess. Aufgrund der beschränkten Ressourcen von Mikrocontrollern werden üblicherweise sämtliche notwendigen Schritte auf leistungsfähigeren Systemen ausgeführt. Mit ihrer ausreichenden Leistung und ihrer weiten Verbreitung haben sich für diese Zwecke handelsübliche Personalcomputer etabliert. Die Softwareentwicklung für Mikrocontroller unterscheidet sich bis zum Abschluss der Implementation nicht von der für andere Softwaresysteme. Allgemein gültige Verfahren und Prozesse können hier zur Anwendung kommen. Ist die Entwicklung abgeschlossen, erfolgt der Übergang auf das eigentliche Zielsystem, den Mikrocontroller.

Der Übergang auf das Zielsystem schließt die Übertragung des Programmcodes sowie dessen Speicherung und Ausführung auf dem Mikrocontroller mit ein. Speziell das Speichern des Programmcodes wird in diesem Zusammenhang als Programmieren bezeichnet. Mit der weitreichenden Unterstützung des *In-System-Programming (ISP)* sind

sämtliche Schritte mittlerweile deutlich vereinfacht. Musste vormals der Mikrocontroller in spezielle Apparaturen eingesetzt werden, kann der Mikrocontroller heute beim Programmieren im verbauten System verbleiben. Die Kommunikation zwischen dem Sender des Programmcodes, im Folgendem kurz mit Host bezeichnet, und einem Mikrocontroller erfolgt gewöhnlich über serielle oder parallele Schnittstellen [19]. Ist diese Schnittstelle nicht auf einem der beteiligten Systeme verfügbar, können so genannte Programmieradapter zum Einsatz kommen. Auf Seite des Hosts übernimmt ein geeignetes Programm die Aufgaben der Kommunikation. Die verschiedenen Vertreter derartiger Programme werden, entsprechend ihrem Zweck, unter dem Begriff Programmiersoftware zusammengefasst. Das entsprechende Gegenstück auf Seiten des Mikrocontrollers ist ein fest im Mikrocontroller verankertes Programm.

Weitergehende Möglichkeiten erlauben die Verwendung eigener Software auf Seiten des Mikrocontrollers zur Programmierung. Das Gegenstück zur Programmiersoftware kann so für den Anwendungsfall individuell erstellt werden. Die Beschränkungen durch fest verdrahtete Algorithmen, beispielsweise durch zu verwendende Kommunikationsschnittstellen, entfallen. Da eine Anwendung das Programmieren durchführt, wird dies gelegentlich auch als *In-Application-Programming (IAP)* bezeichnet. Allein für den Zweck des ISP entwickelte Anwendungen haben vielfach eine Sonderstellung inne. Sie können aufgrund einer oder mehrerer Eigenschaften, beispielsweise ihrer Position im Speicher, Programmcode anderer Anwendungen schreiben und löschen. Damit sind sie auch für den Start der Anwendungen prädestiniert. Aus diesem Grund werden derartige Programme auch als Bootloader bezeichnet.

Das ISP lässt sich in drei Phasen unterteilen:

- Einlesen des Programmcodes
- Übertragung
- Programmieren des Speichers

Der Programmcode einer Anwendung als Ergebnis des Softwareentwicklungsprozesses liegt für die Übertragung auf einem Host vor. Hier erfolgt dementsprechend das Einlesen durch die Programmiersoftware. Im Anschluss überträgt diese den Programmcode mit Hilfe eines Protokolls an den Kommunikationspartner auf dem Mikrocontroller. Dieser nimmt wiederum die Programmierung des Speichers mit den empfangenen Daten vor. Je nach Art der im Entwicklungsprozess eingesetzten Tools liegt der Programmcode zu Beginn des ISP in unterschiedlichen Formaten vor. Um die Programmiersoftware von der Vielfalt der in der Entwicklung eingesetzten Tools zu trennen, werden gesonderte Formate als Eingabeformate akzeptiert. Daher müssen die Ausgabedaten des Entwicklungsprozesses für das ISP konvertiert werden. Die Konvertierung erfolgt mit Hilfe spezieller Programme oder, falls vorhanden, über spezielle Einstellungen der bei der Entwicklung verwendeten Tools [16]. Je nach Hersteller des eingesetzten Mikroprozessors unterscheiden sich die üblicherweise zum Einsatz kommenden Formate. Für Prozessoren der Firma Motorola wird der Programmcode häufig in eine S19-, S-Record- oder auch SREC-Format genannte Form konvertiert. Auch verwandte Prozessoren anderer Hersteller und Prozessorfamilien setzen auf dieses Format. Ebenfalls häufig zum

Einsatz kommt das Intel-Hex-Format. Beide Formate können durch ihre Verbreitung als die Standardformate der Eingabe für das ISP angesehen werden. Ihnen gemein ist die Darstellung der Daten in Form von ASCII-kodierten Zeichenketten. Hinzu kommen Angaben zur Zieladresse und zur Anzahl. Zur Erkennung von Fehlern versehen beide Formate die Daten mit Prüfsummen.

3.2. Network Programming

Für die Rekonfiguration von Mikrocontrollern in verteilten Systemen ist das ISP über die klassischen Schnittstellen ungeeignet. Grund ist die 1:1-Beziehung zwischen Host und einem einzelnen System zu jedem Zeitpunkt. Daraus ergeben sich mehrere Probleme:

- Vorausgesetzt, ein Host ist der Kommunikationspartner für mehrere Mikrocontroller, nimmt die Rekonfiguration des verteilten Systems viel Zeit in Anspruch. Jeder Mikrocontroller muss sequentiell behandelt werden. Zudem beansprucht dieses Vorgehen das Kommunikationsmedium durch das Datenaufkommen, da die gleichen Daten mehrfach übertragen werden müssen.
- Ist die Verbindung eine strikte 1:1-Verbindung über beispielsweise ein dediziertes Kabel, kann die Verbindung eines Hosts zu jedem einzelnen Mikrocontroller nicht dauerhaft sein. Zur Rekonfiguration eines verteilten Systems muss daher die Verbindung jeweils neu hergestellt werden. Das damit verbundene manuelle Vorgehen bringt mehrere Schwierigkeiten mit sich:
  - Ein Zugang zu jedem einzelnen Mikrocontroller ist notwendig. Für außergewöhnlich exponiert platzierte Mikrocontroller kann dieser Zugang nur durch hohen Aufwand hergestellt werden.
  - Der Zeitaufwand steigt linear mit der Anzahl der zu rekonfigurierenden Mikrocontroller.
  - Verbindungen und Schnittstellen aller Mikrocontroller werden mechanisch beansprucht, insbesondere bei häufigen Rekonfigurationen.
  - Die Belastung für den Anwender steigt, da wiederholt die gleichen und somit monotonen Tätigkeiten auszuführen sind.

Im Bereich der Sensornetze, als extreme Beispiele für verteilte Systeme aus Mikrocontrollern, sind diese Probleme der Rekonfiguration sehr ausgeprägt. Unter dem Sammelbegriff des *Network Programming* sind hier bereits alternative Verfahren zur Rekonfiguration eines verteilten Systems entwickelt worden.
Der Vorgang des Network Programming lässt sich in Anlehnung an [19] grundsätzlich in drei Phasen unterteilen:

1. Kodieren der zu verbreitenden Daten
2. Verbreitung

3. Installation

Obwohl nicht explizit aufgeführt, steht als Absicht hinter dem Vorgang stets die nachfolgende Ausführung eines Programms. Ein Programm im Sinne der Informatik ist ein durch eine Computersprache ausgedrückter Algorithmus, welcher im Gegensatz zum allgemeinen Algorithmusbegriff unter Umständen jedoch nicht terminiert [23]. Programme werden von Menschen geschrieben, jedoch von einem Mikroprozessor ausgeführt. Beide Seiten verwenden unterschiedliche Computersprachen. Ein Mikroprozessor kann für Menschen lesbare Programme im Allgemeinen nicht ausführen. Im Gegenzug kann die vermutlich überwiegende Mehrheit der Menschen den so genannten Maschinencode eines Mikroprozessors nicht verstehen. Für den zur Programmausführung notwendigen, semantisch korrekten Informationsfluss zwischen diesen Sprachen existieren zwei Ausführungsmodelle (*Execution Models*):

- Interpretation
  Die Übersetzung eines Programms in eine für den beabsichtigten Mikroprozessor verständliche Sprache wird auf den Zeitpunkt der Ausführung verschoben. Wesentlich ist, dass jede erkannte Anweisung des Programms als eigenständige Einheit übersetzt und ausgeführt wird. Dazu analysiert ein so genannter Interpreter das Programm. Erkannte Anweisungen werden vom Interpreter in semantische Äquivalente der Maschinensprache eines Mikroprozessors übersetzt und umgehend zur Ausführung gebracht. Aus Sicht des ausgeführten Programms stellt der Interpreter die ausführende Einheit dar, nicht der Mikroprozessor selbst. In diesem Sinne repräsentiert der Interpreter einen eigenen Mikroprozessor. Abgesehen von speziellen Hardwarelösungen ist der Interpreter jedoch kein Mikroprozessor. Daher wird für ihn häufig der Begriff der so genannten virtuellen Maschine (*Virtual Machine*, kurz VM) verwendet.

- Übersetzung
  Im Gegensatz zur Interpretation liegen alle Bestandteile eines Programms vor der Ausführung bereits in maschinenlesbarer Form vor. Auf einen zusätzlichen Interpreter zur Ausführungszeit kann verzichtet werden, der einzige aktive Interpreter des Programms ist zu jedem Zeitpunkt ein Mikroprozessor. Der Quellcode, welcher ein Programm in einer für Menschen lesbare Form darstellt, wird vollständig in die Maschinensprache des beabsichtigten Mikroprozessors übersetzt. Oftmals liegt der Quellcode in mehreren, sich aufeinander beziehenden Bestandteilen vor. Das Auflösen dieser Beziehungen ist in diesem Fall ein notwendiger Schritt, um ein ausführbares Programm zu erhalten.

Die Wahl einer dieser Möglichkeiten hat direkten Einfluss auf das Format der im Network Programming zu betrachtenden Daten. Mit der Entscheidung für das verwendete Format entscheidet sich jedes Verfahren zur Rekonfiguration für eines der beiden Modelle.

3.3. Rekonfiguration unter Verwendung von übersetztem Code

Für einen Mikrocontroller ist das Vorliegen einer Anwendung in Maschinencode der unmittelbare Weg zu ihrer Ausführung. Liegt ein Programm in Maschinencode vor, können die enthaltenen Anweisungen direkt vom Mikroprozessor interpretiert und ausgeführt werden.

Die Arbeitsschritte zum Erzeugen eines direkt ausführbaren Programms lassen sich unter dem Begriff Kompilieren zusammenfassen. Abbildung 3.1 zeigt die typischen Aufgaben beim Kompilieren und ihre Beziehung zueinander. Ausgangspunkt des Kompilie-

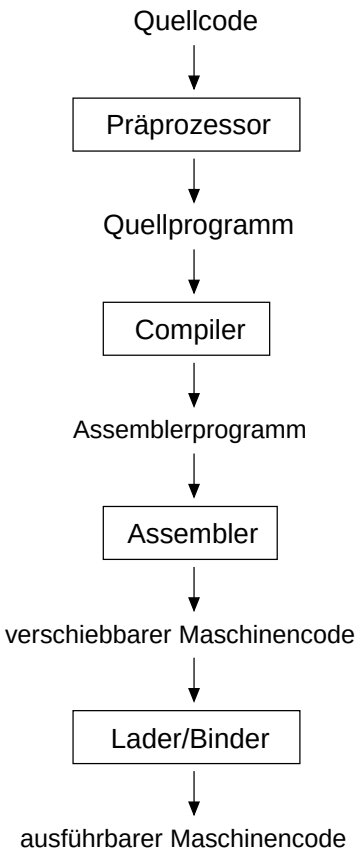


Abbildung 3.1: Typische Aufgaben im Prozess des Kompilierens (nach [1])

rens ist der Quellcode. Nach Bearbeitung durch den Präprozessor erfolgt die Übersetzung der einzelnen Bestandteile in die Assemblersprache. Für diese Tätigkeit zur Anwendung kommt ein Übersetzer (*Compiler*) genanntes Programm. Assembler stellt eine Zwischensprache auf dem Weg zur Maschinensprache dar. Durch die sprachliche Trennung wird es möglich, einen Compiler für verschiedene Ausprägungen von Assembler zu verwenden. Die Übersetzung des erzeugten Assemblerprogramms in Maschinencode erfolgt durch einen weiteren Übersetzer, den so genannten Assembler.

Es entspricht der natürlichen Intention, ein zu entwickelndes Programm in mehrere Bestandteile zu zerlegen. Auf diese Weise bleibt der Überblick erhalten. Zugleich können einzelne Bestandteile erneut verwendet werden. Vorteilhaft ist die Anwendung des beschriebenen Übersetzungsvorgangs für einzelne Teile des Programms. Im Fall einer Änderung braucht nur der betroffene Bestandteil erneut übersetzt werden. Zwei Tätigkeiten sind bei dieser Vorgehensweise im Anschluss an die Übersetzung durch den Assembler notwendig:

- Auflösen von Referenzen
  Bestandteile eines Programms referenzieren sich untereinander. Klassische Beispiele sind die gegenseitige Verwendung von Variablen oder der Aufruf von Routinen untereinander. Die Zieladressen der Referenzen sind durch die individuelle Übersetzung unbekannt. Der Compiler drückt Referenzen mit Hilfe von Symbolen aus. Die Zuordnung dieser abstrakten Namen zu Adressen erfolgt im Prozess des Bindens (*Linkens*).

- Relokation
  Die individuelle Übersetzung einzelner Bestandteile generiert Maschinencode, der im Allgemeinen jeweils an Adresse Null beginnt. Somit überlappt sich der Maschinencode mehrerer Bestandteile. Zur Zusammenfassung muss die Überlappung beseitigt werden. Dabei wird der jeweilige Maschinencode verschoben. Neben dem Zuweisen einer neuen Startadresse müssen auch jeweils die Adressen der Anweisungen und Variablen korrigiert werden, um der neuen Position des Maschinencodes im Speicher gerecht zu werden. Dieser Relokation genannte Vorgang kann ebenfalls vom Binder (*Linker*) durchgeführt werden. Auch der Lader (*Loader*), welcher das Programm zur Ausführung in den Arbeitsspeicher lädt, kann die Relokation während seiner Tätigkeit vornehmen.

Eine Ausnahme stellt positionsunabhängiger Code (*Position Independent Code*) dar. Positionsunabhängiger Code beinhaltet keine absoluten Adressen auf sich selbst, sondern nur relative Angaben. Er braucht daher nicht relokiert werden. Zur Erstellung ist jedoch die Unterstützung des Compilers notwendig. Weiterhin muss der Mikroprozessor derartigen Code mit entsprechenden Maschinenbefehlen unterstützen. Beides sind spezielle und nicht allgemein zu erfüllende Anforderungen.

Der Begriff des Kompilierens ist missverständlich. Die Übersetzung des Quellcodes in Assemblercode durch den Compiler wird ebenfalls mit Kompilieren bezeichnet. Oftmals sind Compiler, Assembler und Linker jedoch eng miteinander verknüpft, so dass häufig der Compiler als Sinnbild für alle beteiligten Tools angesehen wird.

Das Kompilieren ist Teil der Softwareentwicklung für Mikrocontroller. Von den daran beteiligten verschiedenen Rechnersystemen übernimmt, wie in Abschnitt 3 erläutert, üblicherweise der Host die Durchführung sämtlicher Aufgaben. Somit liegt als Resultat des Kompilierens der ausführbare Maschinencode auf dem Host vor. Zur Ausführung muss dieser anschließend, unter Verwendung geeigneter Mechanismen, seinen Weg in den Programmspeicher der Mikrocontroller finden. Vergleichsweise neu ist der Ansatz, Aufgaben im Prozess des Kompilierens an die Mikrocontroller zu delegieren. Ziel dieses

Ansatzes ist es, die sich aus der Aufgabenverteilung ergebenden Möglichkeiten vorteilhaft auszunutzen.

3.4. Rekonfiguration unter Verwendung von Interpretern

Interpreter sind ein Weg, um die im Quellcode eines Programms ausgedrückte Intention der Entwickler direkt zu berücksichtigen. Obwohl Übersetzungen in einen Zwischencode, den so genannten Bytecode, durchaus üblich sind, erfolgt die endgültige Übersetzung in Anweisungen des Mikroprozessors erst mit Hilfe des auf diesem Mikroprozessor laufenden Interpreters. Der Interpreter bzw. die virtuelle Maschine führt den Quellcode oder zumindest einen verwandten Code aus. Die Programmausführung ist aus diesem Grunde nahe an sprachlichen Elementen und Konstrukten, welche die Entwickler verwendeten. Somit ist auch die mit dem Quellcode verbundene Absicht deutlich einfacher zu erkennen.

Die Verwendung von virtuellen Maschinen bietet mehrere Vorteile:

- Eine im Quellcode oder im Bytecode vorliegende Anwendung kann auf unterschiedlichen Mikroprozessoren zur Ausführung gebracht werden. Jeweils spezifische Interpreter entwickeln aus den Anweisungen der Anwendung für den jeweiligen Mikroprozessor geeignete Anweisungen.

- Durch die freie Wahl der Anweisungen auf Seite der Eingabe können Interpreter an Anwendungen angepasst werden. Es können Anweisungen spezifiziert werden, mit deren Hilfe sich Anwendungen sehr kompakt ausdrücken lassen. Gerade für die gelegentliche Erstellung von Anwendungen durch Anfänger oder reine Anwender lässt sich mit Hilfe von Interpretern ein einfaches Programmiermodell realisieren. Häufig eingesetzte Abläufe eines spezifischen Einsatzbereichs, beispielsweise der Empfang von Nachrichten, können zusammengefasst und in einfacher und intuitiver Form bereit gestellt werden.

- Eine zu interpretierenden Anwendung ist typischerweise durch kompakteren und kleineren Code ausgedrückt als ihr Pendant im Maschinencode. Da somit weniger Daten zu übertragen sind, eignet sich eine zu interpretierende Anwendung gut für die Rekonfiguration eines entfernten Systems. Da zudem mit dem Quell- oder dem Bytecode auf einer hohen sprachlichen Ebene gearbeitet wird, lassen sich Änderungen zu vorhergehenden Anwendungen unmittelbar dort lokalisieren, wo sie vom Programmierer getätigt wurden. Dies kann die Menge der zu übertragenden Daten weiter reduzieren.

- Mit der Abstraktion von der zugrunde liegenden Hardware wird die Verbreitung und Ausführung einer Anwendung auch in heterogenen Umgebungen möglich.

Neben den genannten Vorteilen bringt die Verwendung von virtuellen Maschinen aber deutliche Probleme mit sich.

Zunächst nahe liegend ist eine niedrigere Ausführungsgeschwindigkeit einer Anwendung. Das Abholen, Dekodieren und Ausführen einer Anweisung durch einen Interpreter nimmt Zeit in Anspruch. Verglichen mit der Implementierung im Maschinencode, fällt dieser Overhead für komplizierte Anweisungen niedriger aus als für einfache Anweisungen. Insbesondere im Gebiet der Sensornetze fällt der damit verbundene erhöhte Energieaufwand ins Gewicht. Hier kann sich im Falle häufiger Rekonfigurationen dennoch der Aufwand rechnen. Die häufig hohen Kosten der Kommunikation, welche mit jeder Rekonfiguration aufzubringen sind, werden durch den kompakten Code reduziert. Bei hinreichend kurzer Dauer der Ausführung zwischen Rekonfigurationen wird mehr an Energie eingespart, als durch die zusätzlichen Kosten der Ausführung verbraucht wird.

Ein weiteres, gleichwohl bedeutendes Problem sind die Ressourcen, welche eine virtuelle Maschine benötigt. Auf Mikrocontrollern mit ihren beschränkten Ressourcen müssen für die Vergabe von bedeutenden Ressourcen wie Arbeits- oder nichtflüchtigem Programmspeicher triftige Gründe bestehen. Dem steht eine akute Beschränkung der von virtuellen Maschinen angebotenen Funktionalität gegenüber, welche die Erstellung von Anwendungen behindert. Zusätzliche Funktionalität in Form von Befehlen und Funktionen erfordert jedoch die Aufwendung von inhärent knappen Ressourcen.

4. Verwandte Arbeiten

Insbesondere die Forschung im Gebiet der Sensornetze treibt die Rekonfiguration für Mikrocontroller voran. In einer Reihe von Arbeiten wurden hier verschiedene Verfahren mit jeweils eigenen Schwerpunkten entwickelt. Der Charakteristik der Sensornetze entsprechend, ist das Minimieren der für die Rekonfiguration notwendigen Energie oberstes Ziel.

Anhand der in Abschnitt 3.2 vorgestellten Ausführungsmodelle lassen sich die Verfahren unterscheiden. Bisher besondere Verbreitung haben Verfahren erlangt, welche den übersetztem Code als Eingabe verwenden. Mit Blick auf den jeweils verwendeten Ansatz lassen sich diese Verfahren weiter unterteilen.

4.1. Verfahren mit Verwendung von ausführbarem Maschinencode

Verfahren, welche als Eingabe für eine Rekonfiguration den ausführbaren Maschinencode verwenden, werden derzeit am häufigsten eingesetzt. Die Verfahren haben den Vorteil, dass der Mikrocontroller als Empfänger von jeglichen Arbeitsschritten zur Erzeugung des Codes befreit ist. Von Bedeutung ist dies aufgrund der beschränkten Ressourcen. Zudem ist eine Rekonfiguration ein Prozess, die den Mikrocontroller bzw. das eingebettete System in der Durchführung seiner beabsichtigten Funktion behindert. Benötigte Ressourcen stehen anderen Prozessen zunächst nicht zur Verfügung. Das betrifft nicht nur die Ausführungszeit auf dem Mikroprozessor, dazu zählen auch die Allokation von Programmspeicher und RAM. Nicht bestritten werden kann auch der Verbrauch von Energie, wie im Bereich der Sensorknoten stets erwähnt wird.

Am Ausgangspunkt der Entwicklung stand die Rekonfiguration per klassischem ISP. Vertreter für derartige Lösungen waren und sind die Möglichkeiten beispielsweise der AVR-Mikrocontroller per SPI oder JTAG, oder auch der Monitor-Modus bei HC08-Mikrocontrollern der Firma Motorola. Wie in Abschnitt 3.2 darlegt, sind diese Verfahren für verteilte Systeme nicht praktikabel. Entwicklungen zum Network Programming haben das Vorgehen, vermutlich aufgrund seiner Einfachheit, jedoch aufgegriffen.

Vornehmlich anzutreffen ist die Ersetzung des kompletten Speicherabbilds. Die im Zuge einer Rekonfiguration durchzuführenden Aufgabe der Programmierung beschränkt sich auf das Einspielen eines komplett erhaltenen neuen Programms, unter Ersetzung des alten Speicherinhalts. Aufgrund der vergleichsweisen Einfachheit dieser Aufgabe konzentrieren sich die existierenden Entwicklungen in diesem Bereich auf Details zum vorher zu tätigenden Schritt der Übertragung. Im Prinzip sind die vorgeschlagenen Möglichkeiten daher auch zur Verteilung und Aktualisierung von beliebigen Informationen

in einem verteilten System geeignet. Die Intention hinter diesen Entwicklungen, das Realisieren einer Rekonfiguration, gerät aus diesem Grund leicht aus dem Blickfeld. Weitergehende Ansätze versuchen, Differenzen zwischen den alten und neuen Daten im Zuge einer Rekonfiguration zu erkennen und auszunutzen. Grundlage für den bei diesen Differenzansätzen (*Diff-based Approaches*) getriebenen Aufwand ist die Vermutung, dass sich zu aktualisierende Daten häufig nur in wenigen Details unterscheiden. Ein Austausch des kompletten Datensatzes erscheint in solchen Fällen ineffizient. Vorteilhaft wäre statt dessen lediglich die Aktualisierung der Daten, die sich im Vergleich zur vorherigen Version geändert haben.

4.1.1. XNP

XNP [8][20] ist eine im Rahmen des TinyOS-Projekts entstandene Entwicklung. TinyOS ist ein an der University of California in Berkeley entwickeltes Betriebssystem speziell für eingebettete, vernetzte Systeme [42]. TinyOS hat sich in mehreren Jahren eine Gemeinde von Anwendern erarbeitet. Durch das frühe Einbinden in TinyOS kann XNP auch als erste allgemein anwendbare Entwicklung zum automatischen Austausch von Programmcode bezeichnet werden.

Der Algorithmus beim Aktualisieren mit XNP ist sehr einfach und lässt sich in drei Phasen unterteilen:

- In der *Program Download Phase* überträgt ein Sender ein komplettes Programm an einen oder mehrere Empfänger. Der auch als Basisstation bezeichnete Sender unterteilt dazu das Speicherabbild des Programms in einzelne Teile (*Capsules*) und überträgt diese nacheinander. Jeder Empfänger speichert die *Capsules* in einem externen Speicher aufeinander folgend ab. Während [20] explizit den externen EEPROM für diesen Zweck benennt, kommt in [8] der externe Flash zur Anwendung.

- Hat die Basisstation alle Daten übertragen, erfolgt in der *Query Phase* eine Kontrollübergabe. Um das Fehlen von Programmteilen aufgrund von Übertragungsfehlern zu korrigieren, überprüft jeder Empfänger die empfangenen Daten und fordert fehlende Bestandteile neu an. Die Basisstation sendet angeforderte Teile als Broadcast, um es so auch anderen Empfängern zu ermöglichen, existierende Lücken in den übertragenen Daten ohne einen eigenen Request zu schließen.

- Empfängt die Basisstation keine Anfragen über einen vorgegebenen Zeitraum, kann die *Reprogram Phase* eingeleitet werden. Nach Aufforderung transferieren Empfänger die nun komplette Aktualisierung in den internen Flash und starten sich selbst anschließend neu. Da der interne Flash dem Mikrocontroller als Programmspeicher dient, wird auf diesem Wege das neu eingespielte Programm zur Ausführung gebracht.

Für XNP sind ein Programm auf der Basisstation sowie auf jedem Empfänger entsprechende Gegenstücke notwendig. Den auf jedem Mikrocontroller laufenden Kommuni-

kationspartner unterteilt die vorgeschlagene Implementation in jeweils zwei Komponenten. Eine ist die auf dem eingebetteten System laufende TinyOS-Anwendung selbst. Mit einem eingebundenen XNP-Modul kann die Anwendung die Aufgaben der Kommunikation übernehmen. Den abschließenden Transfer der empfangenen Daten führt die zweite Komponente aus, der Bootloader. Im Gegensatz zur Anwendung besitzt dieser durch seine Position im Programmspeicher die Möglichkeit, den Programmspeicher zu beschreiben.

Der Austausch von Programmcode via XNP bleibt auf die unmittelbaren Nachbarn der Basisstation beschränkt, der *1-Hop-Umgebung*. Eine weitere Verbreitung des Programmcodes durch vormalige Empfänger ist im Algorithmus nicht vorgesehen. Das Aufspüren von Lücken in den empfangenen Daten in der *Query Phase* wird durch die Verwendung des S19-Formats möglich. Jede *Capsule* entspricht einer Zeile einer entsprechenden Datei. Da die Zeilen sequentiell aufeinander folgen und auch entsprechend gekennzeichnet sind, lassen sich fehlende Zeilen bzw. Daten leicht feststellen.

4.1.2. Deluge

Wie XNP aus dem vorigem Abschnitt 4.1.1, ist auch Deluge [17] eine Entwicklung aus dem Umfeld von TinyOS. Mit Version 1.1.8 von TinyOS wurde Deluge in das Betriebssystem aufgenommen. Seit Version 1.1.14 wird es im Hauptzweig (*Main Tree*) geführt und steht somit auch für Anwender zur Verfügung.

Deluge, das ins Deutsche übersetzt in etwa „Überschwemmung“ oder „Regenguss“ bedeutet, ist ein Verfahren zur Verteilung und Aktualisierung größerer Datenmengen. Die Motivation zur Entwicklung von Deluge war die Beobachtung zum Bedarf nach Rekonfiguration von Programmen in Sensornetzen. Bestehende Verteilungsverfahren boten bislang nur die Möglichkeit, kleine Datenmengen in der Größenordnung von wenigen Bytes zu bearbeiten. Rekonfigurationen, welche im Allgemeinen mehrere Kilobyte umfassen, konnten somit nicht durchgeführt werden. Erklärtes Ziel ist es, zuverlässig große Datenmengen von einem oder mehreren Sensorknoten an alle Teilnehmer eines drahtlosen Sensornetzes zu verbreiten. Die Verbreitung von Daten erfolgt in drei Schritten:

1. Ankündigen (*Advertise*)
2. Anforderung (*Request*)
3. Verbreitung (*Data*)

Mit Hilfe von Ankündigungen (*Advertisements*) informieren sich miteinander kommunizierende Systeme über ihre jeweils bereits vorliegenden Daten. Um den Kommunikationsaufwand gering zu halten, beinhaltet ein Advertisement lediglich ein Objektprofil (*Object Profile*). Dieses stellt den aktuellen Stand der Daten dar. Stellt ein Empfänger eines Advertisements fest, dass sein Objektprofil auf eine veraltete Version hinweist, wird er eine entsprechende Anforderung (*Request*) nach neuen Daten auslösen. Auf diese Weise verbreiten sich die Daten durch ein ganzes Netz von miteinander verbundenen Systemen. Da ein Sender alle seine Nachbarn sinnbildlich mit neuen Informationen infiziert, wird auch von einem epidemischem Vorgehen gesprochen.

Für die Übertragung teilt Deluge die zu übertragenden Daten logisch in eine zweistufige Hierarchie auf. Auf der unteren Ebene werden die Daten in Pakete (*Packets*) aufgeteilt. Diese werden auch als Übertragungseinheit verwendet. Auf der oberen Ebene fassen Seiten (*Pages*) jeweils eine Menge von N aufeinander folgenden Paketen zusammen. Abbildung 4.1 verdeutlicht die logische Hierarchie der zu übertragenden Daten.

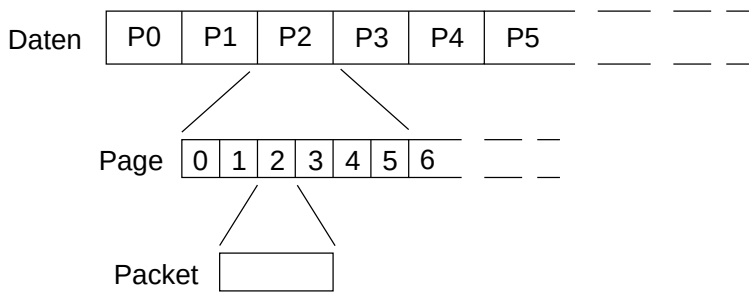


Abbildung 4.1: Hierarchie der Daten in Deluge

Die Page stellt die obere Übertragungseinheit dar. Deluge überträgt Daten, indem es zunächst eine Page überträgt, dann die nächste usw. Dies hat folgende Vorteile:

- Ein Request nach lediglich einer Page kann leicht einen Bitvektor der Länge N enthalten, welcher die bereits erhaltenen Pakete kennzeichnet. Dies verringert das notwendige Datenvolumen. Wird immer nur eine Page von einem Sender übertragen, kann der Verwaltungsaufwand bei der Übertragung der Pakete gering bleiben. Es gilt immer nur maximal N Pakete zu betrachten, welche die Bestandteile einer Page sind. Auf Empfängerseite ist so nur ein Puffer in der Größe einer Page erforderlich.

- Betrifft eine Aktualisierung lediglich einzelne Pages, kann auf die Übertragung unveränderter Pages verzichtet werden. Dazu sind Informationen erforderlich, in welcher vorhergehenden Version jede einzelne Page zuletzt geändert wurde. Zu diesem Zweck beinhaltet das Objektprofil neben der Versionsnummer eine weitere Information, den *Age Vector*. Mit dem Abstand zur Versionsnummer ist hier für jede Page die Version ihrer letzten Änderung aufgeführt. Der Empfänger eines Advertisements kann mit den Abständen in seinem Objektprofil durch einfachen Vergleich leicht die anzufordernden Pages ermitteln.

- Eine räumliche Auftrennung und damit schnellere Verbreitung der Daten wird möglich. Für dieses sog. *Spatial Multiplexing* nimmt ein Empfänger eine komplett empfangene Seite umgehend in sein Objektprofil auf und sendet ein Advertisement. Dies geschieht unabhängig davon, ob bereits sämtliche Pages einer Rekonfiguration empfangen wurden. Somit können mögliche weitere Empfänger bereits Daten erhalten, während ein Sender zugleich noch mit der Komplettierung der Rekonfiguration beschäftigt ist.

Wie bereits in Abschnitt 4.1 angedeutet, liegt auch bei Deluge der Schwerpunkt auf der Verbreitung der Daten. Die Installation einer Aktualisierung erfolgt unter Zuhilfenahme

eines *TOSBoot* genannten Bootloaders. Deluge speichert empfangene Daten im externen Flash eines Mikrocontrollers. Wurde eine Rekonfiguration komplett empfangen, wird dies im internen Flash gekennzeichnet. Einzige Aufgabe von TOSBoot ist es nun, nach einem Reset diese Kennung auszuwerten und in einer möglichen Reaktion den Transfer der Daten in den internen Flash vorzunehmen. Liegt keine entsprechende Kennzeichnung vor, bringt TOSBoot schlicht die Daten im internen Flash zur Ausführung.

4.1.3. Verfahren von Reijers und Langendoen

Das Verfahren von Reijers und Langendoen [34] basiert auf zwei Annahmen. Vornehmlich wird herausgestellt, das Kommunikation unter energetischen Aspekten teuer ist. Dies gilt insbesondere in Umgebungen wie drahtlosen Sensornetzen, in denen Energie zumeist die wertvollste Ressource ist. Gleichzeitig werden die häufig nur geringen Änderungen bei der Aktualisierung von Programmcode im Vergleich zur vorherigen Version hervorgehoben.

Der Energieaufwand für die Kommunikation ist ungleich höher als für das Verarbeiten von Daten. Daher wird eine Änderung der Tätigkeiten während einer Rekonfiguration vorgeschlagen. Grundlage ist die potentielle Ähnlichkeit zwischen originalem und modifiziertem Maschinencode. Auf einer Basisstation als Auslöser einer Rekonfiguration werden lediglich die Änderungen zwischen aktuellem und neuem Maschinencode ermittelt. Diese Änderungen werden verbreitet. Mit Hilfe des vorliegenden Maschinencodes und der empfangenen Änderungen können die zu rekonfigurierenden Systeme den neuen Maschinencode selbstständig konstruieren. Die Schritte zur Bestimmung und Einarbeitung der Veränderungen im Maschinencode bilden den Schwerpunkt des Verfahrens. Der Kommunikation zwischen den beteiligten Systemen wird keine Aufmerksamkeit geschenkt.

Für die Ermittlung der Änderungen kommt ein Algorithmus ähnlich dem des diff-Algorithmus unter Unix zum Einsatz. Unix *diff* versucht, das kürzeste sog. *Edit Script* zwischen zwei Strings zu berechnen. Die verwendeten Operationen unterscheiden sich jedoch grundlegend:

- Ein *Copy* ist die Anweisung, eine Folge von Bytes aus dem Original schlicht zu kopieren. Angenommen wird, dass Fragmente der neuen Daten auch bereits im Original an möglicherweise nur anderen Positionen enthalten sind. Beispiele können der jeweilige Prolog und Epilog von Funktionsaufrufen sein, oder auch Verschiebungen, hervorgerufen durch eingefügten Code.

- Die Operation *Insert* fordert zum Einfügen von Daten auf. Diese Operation ist daher mit der entsprechenden Folge von Daten versehen.

- Eine *Repair*-Operation ist eine Konzession an die Granularität von Modifikationen. Angenommen wird, das einige Fragmente in den originalen und den neuen Daten lediglich sehr ähnlich sind, aber nicht vollkommen identisch. Für derartige Fälle wäre auch für kleinste Differenzen eine Operationsfolge, bestehend aus *Copy-Insert-Copy*, notwendig. Mit einem *Repair* kann dies vermieden werden.

Aufbauend auf einer *Copy*- Operation, werden einzelne Wörter nachträglich korrigiert.

Eine Rekonfiguration lässt sich durch eine Sequenz von Operationen darstellen. Diese sind im Edit Script zusammengefasst, welches von der Basisstation verbreitet wird. Das Ausführen der Rekonfiguration beschränkt sich für die Empfänger auf das Abarbeiten der Folge von Operationen, basierend auf dem vorliegenden Maschinencode.

Das Vergleichskriterium gegenüber der Übertragung der kompletten Daten einer Aktualisierung ist die Größe des Edit Scripts. Sind die Unterschiede lediglich marginal, so steht der Kompromiss zwischen Aufwand zur Kommunikation und Bearbeitung in Frage. In der Evaluation mit den existierenden Operationen wurde für die Entwickler des Verfahrens deutlich, dass bereits eine kleine Änderung im Programmcode in ein vergleichsweise großes Edit Script resultiert. Als Grund benannt werden die Auswirkungen von Verschiebungen. Referenzen auf verschobene Bestandteile des Maschinencodes können sich potentiell überall im Maschinencode befinden, müssen aber ebenfalls durch das Verfahren berücksichtigt werden.

Zwei Ansätze werden zur Minimierung des Problems vorgeschlagen. Vorstellbar ist, durch zusätzlichen Puffer im Maschinencode das Verschieben von Fragmenten zu vermeiden. Funktionen können wachsen oder schrumpfen, ohne andere Bestandteile des Codes zu beeinflussen. Eine andere Möglichkeit ist die Verwendung einer *Patch List*. Dafür wird angenommen, das Änderungen im Maschinencode zumeist lokal beschränkt sind. Häufig verschiebt sich eine Menge von Anweisungen um einen gleichen Betrag. Ist dieser Offset bekannt, können mehrere Referenzen vorteilhaft im Zuge einer *Copy*-Operation korrigiert werden. Dazu wird bei dieser Operation jedes zu kopierende Byte überprüft. Ist das Byte ein Operand eines Maschinenbefehls zur Verwendung einer Referenz, beispielsweise in einem *CALL*, und ist der Offset für den Operand bekannt, wird dieser auf den Operanden aufgeschlagen.

Problematisch sind Fälle, in denen der Operand keine Referenz darstellt, sondern beispielsweise einen konstanten Wert. Potentiell würde dies in falsch korrigierten Operanden resultieren. Jedoch wird die Patch List vor der Anwendung des *diff*-Algorithmus auf der Basisstation erstellt und bereits auf den neu einzuspielenden Programmcode angewendet. Fehlerhaft erzeugte Referenzen werden so als Unterschiede erkannt und nachträglich im Rahmen beispielsweise einer *Repair*-Operation korrigiert.

4.1.4. Verfahren von Jeong und Culler

Einen zum Verfahren aus dem vorigen Abschnitt 4.1.3 recht ähnlichen Ansatz verfolgen Jeong und Culler. Auch hier wird versucht, die Effizienz einer Rekonfiguration durch Kenntnis von Veränderungen zu erhöhen.

Zum Erkennen von Veränderungen zwischen vorliegendem und neuem Code kommt eine Variante des Rsync-Algorithmus zum Einsatz [45]. Dieser wurde dafür entwickelt, binäre Daten über Kommunikationsverbindungen mit geringer Bandbreite zu aktualisieren. Damit erscheint der Rsync-Algorithmus für Aktualisierungen in verteilt arbeitenden eingebetteten Systemen zunächst prädestiniert. Als problematisch angesehen wird von

den Entwicklern jedoch die Ermittlung der Unterschiede zwischen vorliegendem und neuem Programmcode. Diese umfasst im ursprünglichen Rsync-Algorithmus die Berechnung von Checksummen und Hashwerten auf den empfangenden Systemen. Dies würde für Mikrocontroller größeren Aufwand bedeuten. Statt dessen wird angenommen, dass der Host den Speicherinhalt der Empfänger kennt. Da weiterhin angenommen werden kann, dass der Host über weitaus mehr Ressourcen verfügt, wird ihm ein Großteil der Tätigkeiten zur Verbreitung des Programmcodes übertragen. Der Ablauf zur Erstellung der Differenzen gliedert sich wie folgt:

1. Der Host unterteilt den bekannten, aktuellen Maschinencode in Blöcke einer festen Größe B. Für jeden dieser Blöcke berechnet er eine Checksumme und einen Hashwert.

2. Der neue Maschinencode wird Byte für Byte mit einem Rahmen der Größe B durchlaufen. Für jeden Rahmen wird die Checksumme gebildet. Existiert im aktuellem Maschinencode ein Block mit der gleichen Checksumme, wird der Hashwert des Rahmens ermittelt und mit dem des gefundenen Blocks verglichen. Stimmen auch die Hashwerte überein, kann der Block aus dem aktuellem Maschinencode übernommen werden. In diesem Fall kann ebenfalls um B Bytes vorwärts gesprungen werden.

3. Findet sich zu einem Byte und dem dazugehörigem Rahmen keine entsprechende Checksumme, oder stimmen die Hashwerte nicht überein, so wird um ein Byte weiter vorwärts gelaufen. Zusammenhängende Bereiche nicht übereinstimmender Bytes werden gekennzeichnet.

Das Vorgehen zur Aktualisierung ist weitgehend identisch zu dem Vorgehen aus Abschnitt 4.1.3. Es existieren jedoch nur zwei Operationen, die mit *Copy* und *Download* aber lediglich anders benannt sind. Im ersten Entwurf des Verfahrens wurde von einem Empfänger jede empfangene Operation umgehend ausgeführt. Aufgrund von daraus resultierenden Problemen in der Bearbeitungszeit und bei Übertragungsfehlern puffert der aktuelle Entwurf die Operationen in einer *Script Queue*. Die abschließende Ausführung der Operationen aus der Script Queue entspricht dem Abarbeiten eines Edit Scripts im Verfahren aus Abschnitt 4.1.3.

In der Beschreibung des Verfahrens herausgestellt wird die Entwicklung einer kompletten und hardwareunabhängigen Möglichkeit zur Aktualisierung. Neben dem Verfahren zur Erstellung der zu übertragenden Daten kommt für die Übertragung selbst das bereits in Abschnitt 4.1.1 vorgestellte XNP zum Einsatz. Durch die allgemeine Anwendbarkeit das Verfahrens entfallen aber hardwarespezifische Optimierungen. Beispielsweise wird auf das Patchen von Adressen zur Behandlung von Referenzen auf verschobenen Code verzichtet. Wie die Evaluation zeigt, fällt die Einsparung im übertragenem Datenvolumen, auch im Vergleich zum Verfahren aus dem vorigem Abschnitt, selbst bei kleinen Änderungen im Quellcode gering aus.

4.1.5. Verfahren von Koshy und Pandey

Das Verfahren von Koshy und Pandey [24] greift ebenfalls die Idee der Differenzansätze auf. Jedoch wird der Maschinencode einer Anwendung gezielt manipuliert, um Unterschiede zwischen zwei Versionen zu minimieren.

Ausgangspunkt der Überlegungen ist die bereits beschriebene Ursache für große Differenzen: Programmteile können an beliebiger Stelle referenziert sein, so dass kleine, auch lokal beschränkte Änderungen globale Auswirkungen haben können. Klassisches Beispiel ist der Aufruf einer Funktion von verschiedenen Punkten im Quellcode. Verändert die Funktion aus unterschiedlichen Gründen ihre Position bzw. Adresse im Maschinencode, müssen sämtliche Referenzen aktualisiert werden. Der Schwerpunkt des Verfahrens liegt daher auf dem Vermeiden von Verschiebungen. Das Erzeugen der Differenzen, ihre Beschreibung in einem Skript sowie dessen Übertragung und Abarbeitung werden als bereits bekannt angesehen und daher in den Hintergrund der Betrachtungen gestellt.

Zum Vermeiden der als *Code Shifting* bezeichneten Verschiebungen wird das Einfügen von zusätzlichem Puffer für jede Funktion angewendet. Diese Möglichkeit wurde im bereits beschriebenen Verfahren von Reijers und Langendoen in Abschnitt 4.1.3 angedeutet. Mit Hilfe des *Slop Space* genannten Puffers kann eine Funktion im Verlauf von Aktualisierungen an Umfang zu- oder abnehmen, ohne dass sich ihre Adresse ändert. Sämtliche Referenzierungen bleiben daher gleich, die Auswirkungen von getätigten Änderungen bleiben lokal begrenzt. Erst für den Fall, dass eine Funktion mehr Platz benötigt, als ihr in ihrem Slop Space noch zur Verfügung steht, sind Umgruppierungen im Maschinencode notwendig.

Um überhaupt Funktionen betrachten zu können, greift das Verfahren in den Bindeprozess ein. Im letzten Schritt der Übersetzung werden abstrakte Namen auf konkrete Werte abgebildet [27]. Ein Linker weist den Symbolen Adressen zu. Sämtliche Adressen aller Symbole machen zusammen das Speicherlayout eines Programms aus. Im Bindeprozess ist eine Funktion, wie alle anderen Bestandteile eines Programms auch, durch ein Symbol repräsentiert. Mit Verwendung eines modifizierten Linkers kann für jedes Symbol der Slop Space geschaffen werden, indem die Vergabe der Adressen kontrolliert wird. Zur inkrementellen Behandlung ist dies noch nicht ausreichend. Ohne Berücksichtigung eines vorherigen Speicherlayouts können Symbole vollkommen frei auf neue Adressen abgebildet werden. Für diesen Zweck beinhaltet der modifizierte Linker den *Memory Manager*. Dieser kennt das vorherige Speicherlayout. Er weist Symbolen initial ihren Speicherplatz zu. Bei erkannten Veränderungen im Zuge einer Rekonfiguration wird vom Memory Manager der für ein Symbol zur Verfügung stehenden Slop Space überprüft. Ist er nicht ausreichend, können verschiedene Strategien zum Einsatz kommen. Unter Umständen ist es günstiger, nachfolgenden Code zu verschieben, um den Slop Space zu vergrößern. Dies wäre vorteilhaft, wenn im gesamten Programm weniger auf den zu verschiebenden Code referenziert wird als auf das geänderte Symbol selbst. Im allgemeinen Fall wird aber der Code verschoben und bestehende Referenzen werden aktualisiert. Die entstehende Lücke im Speicherlayout kann für neue Zuweisungen verwendet oder dem Slop Space des vorherigen Codes zugeordnet werden.

Da die Änderungen bereits beim Linken berücksichtigt werden, kann der ausführende Linker als *Incremental Linker* bezeichnet werden. Dieser führt seine Aufgabe auf dem Host durch. Die Entwickler des Verfahrens argumentieren, dass ein Linken auf Mikrocontrollern zu aufwendig und nur schwerlich zu realisieren ist. Als Beispiele angeführt werden die Datenstrukturen des Linkers, deren Übertragung und Speicherung potentiell viele Ressourcen in Anspruch nimmt. Der Vorgang des Linkens auf dem Host für ein entferntes System lässt sich entsprechend als *Remote Incremental Linking* bezeichnen. Hat der Linker seine Arbeit beendet, werden die Unterschiede zum vorherigem Maschinencode bestimmt. Die Differenzen werden, ähnlich zu den Verfahren aus den Abschnitten 4.1.3 und 4.1.4, an die Empfänger verbreitet und dort eingespielt.

Das Linken und Erstellen des Speicherlayouts auf einem Fremdsystem hat den Nachteil, dass sämtliche Empfänger das gleiche, vorherige Speicherlayout besitzen müssen. Besitzen ein oder mehrere Empfänger Maschinencode mit einem anderen Speicherlayout, beispielsweise durch spezifische Programmmodule, führt das Einpflegen der Differenzen zu falschen Referenzen und damit zu fehlerhaft installiertem Code. Der Linker, im Detail der Memory Manager, muss daher den Status aller potentiellen Empfänger einer Rekonfiguration berücksichtigen. Sind Unterschiede bekannt und liegen damit unterschiedliche Speicherlayouts vor, muss jedes Layout einzeln berücksichtigt werden. Dies betrifft ebenfalls die anschließende Verbreitung der Differenzen an die spezifischen Empfänger.

4.2. Verfahren mit Verwendung von verschiebbarem Maschinencode

An der Entwicklung der Verfahren aus Abschnitt 4.1 zeigt sich eine zunehmende Berücksichtigung der Effizienz des Vorgehens. Die naive Übertragung der kompletten Anwendung ist einfach, verbraucht gleichwohl aber Ressourcen wie Zeit, Übertragungskapazität und Energie, insbesondere auch für bereits bekannte und installierte Daten. Die vorgeblich intelligenteren, auf Differenzen arbeitenden Ansätze aus den Abschnitten 4.1.3 und 4.1.4 können aber bislang nicht überzeugen. Die Größe der erkannten Änderungen korreliert im allgemeinen Fall nicht mit den Änderungen im Quellcode der zu aktualisierenden Anwendung.

Verschiedene Entwickler beschreiten mit unterschiedlichen Verfahren einen anderen Weg. Allen Verfahren gemein ist die Annahme, dass für intelligentere Rekonfigurationen mehr Informationen über die Struktur von Anwendungen notwendig sind. Der ausführbare Maschinencode als Informationsquelle wird für diese Zwecke als unzulänglich betrachtet. Der Blick auf den Kompilierprozess in Abbildung 3.1 verdeutlicht dies. Informationen über die Gliederung des Quellcodes in verschiedene Bestandteile, wie sie vom Entwickler eingebracht wurden, werden im Verlauf des Kompilierprozesses vernichtet. Für das Erkennen der ursprünglichen Struktur einer Anwendung kommt eine Analyse des ausführbaren Maschinencodes daher zu spät.

Um dennoch an benötigte Informationen zu gelangen, wird aktiv in den Übersetzungsprozess eingegriffen. Angesichts weiter steigender Ressourcen von Mikrocontrollern kann begründet die Absicht verfolgt werden, die Mikrocontroller aktiv am Übersetzungsprozess zu beteiligen. Diese Systeme sind für die Speicherung und Ausführung des erzeugten Maschinencode verantwortlich. Daher sind sie unmittelbare Ansprechpartner für maschinenspezifische Informationen im Übersetzungsprozess.

4.2.1. Impala

Motivation für die Entwicklung von Impala [29] ist die Ansicht der Entwickler, dass monolithische Programme bzw. Anwendungen in Sensornetzen benachteiligt sind. Bei einer angestrebten Berücksichtigung aller möglicher Anwendungsfälle wird der Code vielfach groß und schwerfällig. Gleichzeitig steigt die Komplexität, dies erschwert die Entwicklung und begünstigt das Entstehen von Fehlern. Auch bei Rekonfigurationen werden große monolithische Anwendungen als benachteiligt angesehen, da häufig große Fragmente an Code zu transportieren sind. Gleichzeitig wird von den Entwicklern der Standpunkt vertreten, dass aus der Heterogenität von Anwendungen in kooperierenden Systemen Vorteile resultieren. Oftmals herrschen für jedes System spezifische Bedingungen, deren Eigenschaften mit individuell angepassten Anwendungen vergleichsweise besser ausgenutzt werden können als mit Standardlösungen.

Mit einer Middleware als zusätzlicher Schicht zwischen Hardware und Anwendungen wird versucht, beide Aspekte zu berücksichtigen. Die Middleware dient den Anwendungen als Betriebssystem und Ereignisfilter. Zwei Hauptbestandteile der Middleware sind mit der Verwaltung der Anwendungen betreut. Der *Application Adapter* führt die Anpassung des Systems an sich ändernde Bedingungen durch. Liegen mehrere Anwendungen fertig installiert vor, entscheidet der Application Adaptor, welche Anwendung am besten geeignet ist. Anschließend wird die ausgewählte Anwendung zur Ausführung gebracht. Zu jedem Zeitpunkt wird so genau eine Anwendung ausgeführt. Zum Erhalt der Kriterien für die Entscheidung werden die Anwendungen befragt. Die zweite Hauptkomponente der Middleware ist der *Application Updater*. Dieser behandelt sowohl den Empfang als auch die Installation von neuen Versionen von Anwendungen. Demzufolge ist er mit dem Ausführen von Rekonfigurationen beauftragt. Um die Probleme von monolithischen Anwendungen zu vermeiden, werden Anwendungen von Impala als eine Kollektion von Modulen betrachtet. Das Verbinden bzw. Linken einzelner Module zu einer Anwendung wird vom Application Updater eines jeden Empfängers durchgeführt.

Durch die Betrachtung einer Anwendung als Kollektion von Modulen entspricht das Durchführen der Rekonfiguration einem Verbreiten und Installieren einzelner Module. Für eine Rekonfiguration enthält jedes Modul eine Versionsnummer, ebenso wie jede Anwendung. Mit Hilfe der Versionierung von Modulen brauchen nur ausgewählte Bestandteile einer Anwendung übertragen werden. Ein Empfänger kann bereits Module einer neuen Version vorrätig haben. Auf eine Übertragung dieser Module kann in diesem Fall verzichtet werden. Der gesamte Verbreitungsalgorithmus lässt sich in drei Phasen unterteilen:

1. Jedes System verbreitet in Advertisements die Versionsnummern der Module und der Anwendung für jede installierte Anwendung.

2. Mit Erhalt der Advertisements kann ein Empfänger bestimmen, welcher Sender die höchste Versionsnummer für eine Anwendung verbreitet. Wird mit Blick auf die eigene Versionsnummer deutlich, dass der eigene Code zu aktualisieren ist, werden fehlende Module von einem Sender angefordert.

3. Empfangene Anforderungen werden von jedem System mit den Daten der entsprechenden Module beantwortet.

Dies ist das gebräuchliche Vorgehen der auch in den vorigen Abschnitten beschriebenen Verfahren.

Eine Besonderheit ist das Erfassen und Pflegen von unvollständigen Aktualisierungen durch den Application Updater. Die Begründung für dieses Vorgehen liegt in der Annahme, dass die Übertragung einer Anwendung vergleichsweise lange dauert. Somit sind Unterbrechungen zu erwarten, bis hin zu Rekonfigurationen, die aufgrund von fehlendem Modulen möglicherweise niemals durchgeführt werden können. Mit der Pflege von mehreren Versionen für jede Anwendung wird die Wahrscheinlichkeit erhöht, jemals alle Module einer neuen Version einer Anwendung vollständig vorliegen zu haben. Tritt dieser Fall für eine Version schließlich ein, kann die Rekonfiguration erfolgreich zu Ende gebracht werden. Die empfangenen Module werden gebunden und integriert. Unvollständige, niedriger versionierte Anwendungen besitzen nun keine Bedeutung mehr und können im Anschluss entfernt werden. Zur Integration von Anwendungen in die Middleware wird eine Tabelle von Funktionszeigern verwendet. Die *Application Activation Table* beinhaltet für jede installierte Anwendung einen Funktionszeiger. Im Laufe einer Aktualisierung spielt der Application Updater die neue Adresse einer Anwendung in diese Tabelle ein. Der Funktionszeiger wird von der Middleware zum Ausführen der Anwendung verwendet.

Für die Implementation von Impala gingen die Entwickler von vergleichsweise hohen Hardwarevoraussetzungen aus. Verwendet wurden iPAQ Handhelds mit über 200 Megahertz Taktfrequenz sowie mehreren Megabyte Arbeits- und Programmspeicher. Derartige Voraussetzungen lassen einen Einsatz auf den wesentlich beschränkteren Mikrocontrollern fraglich erscheinen. Weitere Nachteile sind im Verfahren selbst begründet. Module, welche gebunden werden, dürfen keine Referenzen untereinander aufweisen. Zudem ist jedes Modul in seiner Größe beschränkt. Ungleich schwerer wiegt aber, dass unklar bleibt, gegen welchen Programmcode die einzelnen Module gelinkt werden. Dem eigentlichen Prozess den Linkens wird vom Verfahren keine Bedeutung zugemessen. Anhand des erläuterten Programmiervorgangs kann lediglich vermutet werden, dass jedes Modul auf die Middleware gelinkt wird.

4.2.2. Contiki

Wie am Vorgehen aus dem vorigen Abschnitt 4.2.1 deutlich wird, können Aktualisierungen leicht auf Systemen durchgeführt werden, welche das Laden von Modulen er-

lauben. Die Aktualisierung beschränkt sich auf neue oder veränderte Module, die es zu übertragen und einzubinden gilt. Das speziell für eingebettete Systeme entwickelte Betriebssystem Contiki [7] bietet derartige Möglichkeiten.

Die Möglichkeiten der ursprünglichen Version von Contiki sehen das Einbinden von Modulen vor, die als *Pre-linked* bezeichnet werden. Derartige Module wurden bereits zum Zeitpunkt des Übersetzens, der *Compile Time*, gelinkt. Somit sind alle Adressen auf Funktionen und Variablen des referenzierten Systems aufgelöst, der Maschinencode des Moduls enthält die entsprechenden Adressen. Die notwendige Relokation erfolgt durch Contiki beim Einbinden eines Moduls. Dazu muss das Modul in einem Datenformat vorliegen, welches sowohl den Maschinencode als auch Informationen zur Relokation beinhaltet. Der Loader von Contiki wertet diese Informationen aus, passt den Maschinencode entsprechend an und lädt das Modul in den für Contiki reservierten Speicher. Der nachfolgende Aufruf der Initialisierungsfunktion des Moduls durch den Loader registriert das Modul abschließend in Contiki selbst.

Eine neue Möglichkeit von Contiki ist das dynamisches Binden (*Dynamic Linking*) von Modulen. Im Gegensatz zu *Pre-linked*-Modulen entfällt das Auflösen der Referenzen während der Übersetzung. Statt dessen wird der Vorgang beim Einbinden des Moduls in Contiki durchgeführt. Zu diesem Zweck kennt jede Instanz von Contiki das eigene Speicherlayout und ist in der Lage, Referenzen selbstständig auflösen. Ein solches Vorgehen hat den Vorteil, dass das Problem der verschiedenen Speicherlayouts gelöst wird. Zur Compile Time muss das Speicherlayout der Empfänger einer Rekonfiguration nicht bekannt sein. Es ist ausreichend, wenn die individuelle Auflösung der Referenzen sichergestellt ist.

Zwei Voraussetzungen müssen gegeben sein, damit das dynamisches Binden eines Moduls erfolgreich durchgeführt werden kann.

- Die Referenzen des Moduls müssen durch den Linker als solche erkannt werden. Entsprechend müssen bei der Verbreitung neben dem eigentlichen Maschinencode weitere Informationen übertragen werden.

- Die Ziele der Referenzen eines Moduls müssen dem Linker in Form von Adressen bekannt sein. Eine Referenz ist durch ein Symbol repräsentiert. In Symboltabellen sind die Abbildungen von Symbolen auf Adressen in einfacher Weise festgehalten.

Die Art und Weise der Erfüllung der Voraussetzungen charakterisiert das dynamische Linken in Contiki. Ein dynamisch zu linkendes Modul ist in Contiki stets eine Objektdatei (*Object File*). Im Übersetzungsprozess, wie er in Abbildung 3.1 dargestellt ist, erzeugt der Compiler zusammen mit dem Assembler Objektdateien. Jede Objektdatei beinhaltet die für eine Quelldatei erzeugten Befehle und die durch die Befehle bearbeiteten Daten [27]. Die Zusammensetzung und der Inhalt einer Objektdatei hängt vom verwendeten Objektformat ab. Contiki setzt das *Executable and Linkable Format* [43] ein, im Folgendem mit ELF bezeichnet. Dieses Format ist weit verbreitet, insbesondere auf UNIX-Systemen stellt es seit einigen Jahren das Standardformat für Objektdateien dar. Daher existieren bereits eine Reihe von Programmen für den Umgang mit diesem Format. Objektdateien im ELF enthalten eine Symboltabelle für die in der Datei

verwendeten Symbole. Ebenfalls enthalten sind Tabellen mit Relokationsinformationen (*Relocation Tables*). Diese beschreiben sämtliche aufzulösenden Referenzen.

Beinahe alle Voraussetzungen zum Linken eines Moduls sind durch die Zusatzinformationen aus dem ELF erfüllt. Lediglich die in einem Modul referenzierten Symbole, welche an anderer Stelle bereitgestellt werden, können noch nicht aufgelöst werden. Contiki beschränkt diese, aus Sicht eines Moduls externen Symbole (*External Symbols*) auf ausschließlich vom Betriebssystem bereitgestellten Symbole. Zu diesem Zweck stellt Contiki eine Symboltabelle des Betriebssystems bereit. Jedes dynamisch einzubindende Modul kann in der Relokationstabelle ausschließlich auf Symbole aus dieser Tabelle sowie auf eigene Symbole verweisen. Hält sich ein Modul nicht an diese Vereinbarung, entstehen potentiell unaufgelöste Referenzen(*Undefined References*), und das Linken schlägt fehl. Das Einbringen neuer Symbole kann nur durch eine Rekonfiguration von Contiki selbst erfolgen. Von einzelnen Modulen bereitgestellte Symbole (*Exported Symbols*) können nicht für das Linken weiterer Module verwendet werden, da sie nicht von Contiki in die Symboltabelle eingearbeitet werden. Eine direkte Referenzierung zwischen dynamisch geladenen Modulen ist somit unmöglich. Konnte das Linken eines Moduls erfolgreich durchgeführt werden, ist es zur Ausführung bereit. Der Linker übernimmt auch den Schritt des Ladens. Die Bestandteile des Programmcodes werden dabei an die ihnen angedachten Adressen kopiert und gegebenenfalls initialisiert. Anschließend kann das neue Programm von Contiki gestartet werden.

Das Betriebssystem Contiki zielt auf die Anwendung in kleinen Architekturen von Mikrocontrollern ab, wie beispielsweise der AVR-8Bit-Architektur der Firma Atmel. Das ELF wurde hingegen für die Verarbeitung auf 32Bit-Rechnern entworfen. Für Mikrocontroller einer Architektur mit geringeren Wortbreiten bedeutet die Verwendung von ELF Overhead. In ihren Prozessoren finden derartige Wortbreiten bei der Adressierung keine Verwendung, da der Adressbus entsprechend beschränkt ist. Daher bleiben die oberen Bytes der Strukturen im ELF typischerweise ungenutzt. Aus diesem Grunde schlagen die Entwickler mit *Compact ELF* (CELF) ein auf die Bedürfnisse der kleinen Architekturen angepasstes ELF vor. Objektdateien im CELF tragen die gleiche Information wie im ELF, jedoch durch kleinere Strukturen von acht oder 16 Bit Breite ausgedrückt .

4.2.3. FlexCup

Die Entwickler von FlexCup [14] stellen ein Verfahren für die Rekonfiguration vor, welches das schon aus dem Abschnitten 4.1.1 und 4.1.2 bekannte Betriebssystem TinyOS zur Grundlage hat. FlexCup ermöglicht das Einspielen von neuen Komponenten in ein laufendes TinyOS-System und so die gezielte Aktualisierung einer Anwendung. Der Verbreitungsmechanismus der Komponenten steht völlig im Hintergrund und wird nur am Rande erwähnt.

Obwohl TinyOS den Begriff der Komponente durchgängig verwendet, ist es in seiner ursprünglichen Form ein monolithisches System. Die Programmiersprache von TinyOS ist *nesC*, eine gerade für komponentenbasierte Architekturen entwickelte Sprache. So ist TinyOS selbst auch aus strukturellen Komponenten aufgebaut. Eine Trennung zwischen Anwendung und Betriebssystem existiert jedoch nicht. Im Gegensatz zu beispielsweise

Contiki aus Abschnitt 4.2.2 existiert keine Schicht, auf der Anwendungen aufsetzen. Statt dessen wird eine Anwendung zusammen mit den von ihr benötigten Komponenten von TinyOS kompiliert. Als Resultat liegt eine einzelnes Programm vor, welche sowohl die Bestandteile des Betriebssystems als auch die der Anwendung enthält. Eine Rekonfiguration einer auf TinyOS basierenden Anwendung geht damit bisher mit einer Rekonfiguration des Betriebssystems einher.

Um dennoch einzelne Komponenten in ein laufendes TinyOS einpflegen zu können, bedient sich FlexCup einer noch neuen Möglichkeit von nesC. Mittels Binärkomponenten (*Binary Components*) können seit Version 1.2 Komponenten unabhängig von einer Anwendung kompiliert und verwendet werden. Für eine Anwendung, die eine Binärkomponente verwenden will, generiert der nesC-Compiler entsprechend Referenzen auf die noch unbestimmten Funktionen. Zudem werden externe Symbole für Funktionen der Anwendung angelegt, welche von der Binärkomponente referenziert werden. Beides geschieht umgekehrt beim Kompilieren der Binärkomponente. Schnittstelle zwischen beiden Partnern ist die so genannte Komponentendefinition, welche die gegenseitig angebotenen und verwendeten Konstrukte beschreibt. Sie muss beiden Partnern zur Verfügung stehen, damit die gegenseitigen Referenzen korrekt aufgelöst werden können. Durch Binärkomponenten wird es in TinyOS möglich, Komponenten zur Verfügung zu stellen, deren Quellcode nicht bekannt ist. Vor der Ausführung müssen aber beide Partner zu einer Einheit gebunden werden. Eine Änderung der Komponentendefinition von einer Seite erfordert so auch eine Anpassung und erneute Übersetzung des entsprechenden Gegenstücks.

Zum Erhalt einer ausführbaren Anwendung ist der Prozess des Linkens der Binärkomponenten durchzuführen. Dieser kann auf dem Host durchgeführt werden, so dass als Resultat ein komplett neues Speicherabbild vorliegt. Zur Durchführung einer selektiven Rekonfiguration verlagert FlexCup den Prozess aber auf den Mikrocontroller. Wie auch schon bei Contiki aus dem vorigen Abschnitt 4.2.2, sind erneut Objektdateien Ausdrucksform der nun mit dem Präfix „binär“ versehenen Komponenten. Vom nesC-Compiler erzeugte Objektdateien bzw. Binärkomponenten liegen im ELF vor. Mit Hinweis auf die Effizienz verwendet FlexCup beim Durchführen der Rekonfiguration ein eigenes Format. Dieses besteht aus dem entsprechendem Maschinencode sowie aus Metadaten. Letztere beinhalten die Symbol- und Relokationsinformationen einer Objektdatei, umgewandelt aus dem ELF in eigene Formen der Repräsentation.

Das Übersetzen in verschiebbaren Maschinencode sowie das Umwandeln der Ergebnisse in das eigene Format werden auf dem Host durchgeführt. Zur Verbreitung der Daten verwendet FlexCup einen eigenen Verbreitungsalgorithmus aus einer früheren Entwicklung. Aus Sicht des Erstellens einer lauffähigen Anwendung ist die Verbreitung ein unabhängiges Problem und wird daher vollkommen transparent behandelt. Sind die Daten einer Komponente, sowohl Maschinencode als auch Metadaten, vollständig übertragen, stoppt der Empfänger eine laufende Anwendung und startet den Prozess zum Einbinden der Komponente in das laufende System. In diesem Prozess sind mehrere, sequentiell aufeinander folgende Schritte notwendig:

1. Der Maschinencode der entsprechenden vorherigen, auf dem System bereits installierten Komponente muss ersetzt werden. Der Fall einer komplett neuen Komponente kann nicht auftreten. Dies würde eine komplett neue Schnittstelle bedeuten, worauf die installierte Anwendung nicht vorbereitet ist.
   FlexCup speichert den Maschinencode jeder Komponente in einem Bereich des externen Flash. Da kein Puffer zwischen den gespeicherten Komponenten vorgesehen ist, kann das Ersetzen einer Komponente bei Größenveränderungen zu Verschiebungen des nachfolgenden Codes führen.

2. Die Symbole aller Komponenten werden in einer einzigen Tabelle verwaltet. Die Symboltabelle wird ebenfalls im externem Flash festgehalten, um so auch nach einem Neustart des Systems wieder zur Verfügung zu stehen. Durch eine geänderte Komponente können neue Symbole hinzugekommen und Adressen bestehender Symbole verändert worden sein. Diese Änderungen müssen in die Symboltabelle eingearbeitet werden. Da die Symboltabelle aus Gründen der Zugriffsgeschwindigkeit sortiert vorgehalten wird, gestaltet sich das Einfügen eines neuen Symbols schwierig. Nachfolgend einsortierte Symbole müssen dann im Flash verschoben werden, was aufwendige Schreiboperationen erfordert. Ein Entfernen eines Symbols ist in FlexCup bisher nicht realisiert.

3. Jede neue Komponente bringt in den übertragenen Metadaten auch seine Relokationstabelle mit. Diese muss persistent gespeichert werden, um im Falle eines Neustarts einen konsistenten Status für weitere Aktualisierungen vorzufinden. Hierfür findet wiederum der externe Flash Verwendung.

4. Der wesentliche Schritt ist das Aktualisieren der Referenzen. Nahe liegend ist die Anpassung der einzuspielenden Komponente auf die Adressen der existierenden Anwendung. Gleiches gilt im Umkehrschluss für Referenzen aus der existierenden Anwendung auf die rekonfigurierte Komponente. Hinzu kommen Referenzen, die aufgrund des Verschiebens des Codes in Schritt 1 angepasst werden müssen. Beide zuletzt genannten Schritte sind nur möglich, da sämtliche Relokationstabellen für alle installierten Komponenten vorgehalten werden.

5. Der nun vorbereitete Maschinencode der Anwendung muss abschließend vom externen in den internen Flash des Mikrocontrollers übertragen werden.

Abgesehen vom letzten Schritt, führt der *FlexCup-Linker* sämtliche Aufgaben durch. Zentraler Arbeitsbereich ist der externe Flash des Mikrocontrollers, der neben den Relokationstabellen und der Symboltabelle auch den Maschinencode sämtlicher installierter Komponenten enthält.
Das Übertragen der Anwendung in den internen Flash wird vom *FlexCup-Bootloader* realisiert. Dieser führt, in klassischer Manier der Bootloader, nach Beendigung seiner Aufgabe einen Sprung an den Reset-Vektor des Mikrocontroller aus und startet auf diesem Wege die soeben rekonfigurierte Anwendung.

Mit FlexCup ist es möglich, eine auf TinyOS basierende Anwendung selektiv zu rekonfigurieren. Im Vergleich mit anderen, bereits beschriebenen Verfahren können sich

einzelne Module erstmals untereinander referenzieren. Der Prozess zum Erstellen einer ausführbaren Anwendung entspricht, obwohl zwei verschiedene Systeme beteiligt sind, dem gebräuchlichen Vorgehen im Rahmen der Softwareentwicklung. Von Nachteil ist jedoch der für Mikrocontroller hohe Bedarf an Speicher, da eine komplette Kopie der laufenden Anwendung samt Metainformationen vorgehalten wird. Trotz des Speicheraufwands ist die Rekonfiguration von statischer Natur. FlexCup erlaubt nur den Austausch von Modulen. Somit ist die Granularität der Modularisierung mit der ersten Installation der Anwendung festgelegt. Aktualisierte Module können zusätzliche Funktionen einbringen. Diese können jedoch nur von bereits bekannten Modulen in Anspruch genommen werden, wobei die entsprechende Referenzierung im Rahmen nachfolgender Rekonfigurationen eingearbeitet werden kann.

4.3. Verfahren mit Verwendung von interpretiertem Code

Für Mikrocontroller existieren im Vergleich zur Ausführung von übersetztem Code nur wenige Verfahren, welche das Ausführungsmodell der Interpretation verwenden.

4.3.1. Matè

Matè [28] ist ein auf dem Betriebssystem TinyOS laufender Interpreter von Bytecode. Matè bietet dem Programmierer einen kompakten Befehlssatz. Jeder Befehl ist ein Byte lang, inklusive eines Operanden. Außerdem wird es dem Anwender ermöglicht, bis zu acht eigene Anweisungen zu definieren. Dies muss jedoch schon bei der Übersetzung der virtuellen Maschine erfolgen.

Matè behandelt jeweils 24 Anweisungen als eine *Code Capsule*. Besitzt eine Anwendung größeren Umfang, verteilt sie sich über mehrere dieser Bestandteile. Matè führt eine Anwendung aus, indem es die Befehle der Code Capsules sequentiell dekodiert und zur Ausführung bringt. Die Code Capsule stellt auch die Einheit dar, in der Matè eine Anwendung an andere Mikrocontroller verbreitet. Diesem Zweck dienen zwei Befehle des Befehlssatzes. Wird auf einen solchen Befehl gestoßen, erfolgt die Übertragung einer kompletten Code Capsule an benachbarte Mikrocontroller. Führt eine dortige Instanz der virtuellen Maschine die Anweisungen aus, verbreitet sich der Bytecode entsprechend weiter.

4.3.2. VM*

Die Entwickler von VM* [25] verfolgen den Ansatz einer komponentenbasierten virtuellen Maschine. Anstatt einen kompletten, aber auch großen Befehlssatz zur Verfügung zu stellen, bietet eine Instanz von VM* nur die Funktionalität, welche die darauf aufsetzende Anwendung auch wirklich benötigt. Das dafür notwendige Zusammensetzen (*Synthesizing*) der virtuellen Maschine erfolgt durch die Analyse des Bytecodes der auszuführenden Anwendung. VM* basiert auf dem Befehlssatz der Java Virtual Machine

(JVM). Im Prozess des Synthesizing wird der Bytecode der Anwendung auf benötigte Anweisungen und Funktionen untersucht. Die Komponenten von VM*, welche diese Anforderungen bereitstellen, werden anschließend zur virtuellen Maschine zusammengestellt.

Durch die enge Bindung zwischen virtueller Maschine und Anwendung ist eine Trennung der Daten bei der Rekonfiguration der Anwendung nicht immer möglich. Wird neue Funktionalität gefordert, oder alte Funktionalität nicht mehr benötigt, ändert sich auch die Instanz von VM* entsprechend. Das bereits in Abschnitt 4.1.5 vorgestellte Verfahren des Remote Incremental Linking wird zur Durchführung der Rekonfiguration vorgeschlagen.

4.4. Zusammenfassung und Bewertung

Die erläuterten Verfahren stammen ausschließlich aus dem Forschungsgebiet der drahtlosen Sensornetze. Hier begann die Entwicklung der automatisierten Rekonfiguration. Nachfolgende Entwicklungen blieben bisher diesem Gebiet vorbehalten. Tabelle 4.1 gibt eine Zusammenfassung. Die zum Einsatz kommende drahtlose Kommunikation ist oftmals nicht explizit auf einen Standard festgelegt. Verfahren, welche mit Blick auf eine spezifische Sensorplattform entwickelt wurden, beziehen sich auf die dort gegebenen Möglichkeiten. Andere Verfahren berücksichtigen allgemeingültige Merkmale der drahtlosen Kommunikation, ohne sich auf die Hardware im Detail zu beschränken. Fast ausnahmlos gleich sind hingegen erhöhte Anforderungen an die Hardware durch den Gebrauch von externem Festspeicher. Verfahren, welche sich auf den einzig in einem Mikrocontroller integrierten Festspeicher verlassen, sind deutlich in der Minderheit. Wie vom ISP bekannt, führt ein Host die Rekonfiguration benachbarter Mikrocontroller durch. Eine Auswahl der Empfänger wird nur bei wenigen Verfahren vorgenommen. Gebräuchlich für eine Auswahl ist die Verwendung von Versionsnummern. Empfänger können auf diesem Wege ihre Teilnahme an einer Rekonfiguration bestimmen und gegebenenfalls deren weiteren Ablauf veranlassen. Die Alternative in Form einer vom Inhalt der Rekonfiguration unabhängigen Auswahl findet nur geringe Anwendung. In den Verfahren kommen verschiedene Formate zur Beschreibung des Inhalts der Rekonfiguration zum Einsatz. Bis auf wenige Ausnahmen, stehen diese in direktem Zusammenhang mit dem jeweiligen Typ.

Die Verfahren lassen sich nicht nur speziell in drahtlosen Sensornetzen, sondern allgemein für die Rekonfiguration von Mikrocontrollern verwenden. Die Ersetzung des kompletten Programmspeichers mehrerer Mikrocontroller entspricht der Weiterführung des Prinzips des ISP. Differenzansätze stellen die ersten Möglichkeiten der selektiven Rekonfiguration dar. Mit erkannten Unterschieden zwischen Anwendungen werden ausgesuchte Bereiche des Programmspeichers manipuliert. Verfahren mit Verwendung von Modulen erhöhen den Abstraktionsgrad der Rekonfiguration. Ohne die Ergebnisse des Übersetzungsvorgangs zu kennen, kann der Programmierer mit der Auswahl von Modulen Einfluss auf den Inhalt des Programmspeichers nehmen. Virtuelle Maschinen erhöhen den Abstraktionsgrad weiter. Auf die Rekonfiguration nimmt allein der Quellcode Ein-

Verfahren	Typ	Format	verwendete Hardware	Kommunikation Kardinalität	Kommunikation Schnittstelle	Merkmale
XNP	komplette Ersetzung	SREC-Format	EEPROM, externer Flash	Broadcast	drahtlos	1-Hop-Umgebung
Deluge	komplette Ersetzung	nicht festgelegt	externer Flash	Broadcast	drahtlos (433 MHz, Zigbee)	Verbreitung beliebiger Daten
Reijers und Langendoen	Differenzansatz	Edit Script	externer EEPROM	bislang 1:1	unbekannt	Patch-List
Jeong und Culler	Differenzansatz	Edit Script im SREC-Format	externer Flash	Broadcast	drahtlos	XNP, Rsync
Koshy und Pandey	Differenzansatz	Edit Script	interner Flash	nicht festgelegt	nicht festgelegt	Remote Incremental Linking
Impala	Einspielen neuer Module	unbekannt	iPAQ Handheld	Broadcast	drahtlos	Middleware
Contiki	Einspielen neuer Module	ELF oder CELF	nicht festgelegt	Broadcast	drahtlos (868 MHz, Zigbee)	Linken gegen OS
FlexCup	Austausch von Modulen	eigenes Format	externer Flash	Multicast	drahtlos (CC1000)	TinyOS-Binärkomponenten
Matè	Ersetzung von Quellcode	Capsules	unbekannt	Multicast	drahtlos (916 MHz)	Virtuelle Maschine
VM*	Differenzansatz auf Quell- und Binärcode	Edit Script	interner Flash	nicht festgelegt	drahtlos (CC1000)	konfigurierte Virtuelle Maschine

Tabelle 4.1: Vergleich der verwandten Verfahren

fluss, eine Berücksichtigung der zur Ausführung verwendeten Ressourcen und Prozesse ist nicht notwendig.

Die erläuterten Verfahren sind aber auch an sehr spezielle Anforderungen angepasst:

- drahtlose Kommunikation mit unzuverlässigen Verbindungen, beschränkten Reichweiten und hohem Energieaufwand
- inhärent angespannte Energiesituation
- vergleichsweise große und frei verfügbare Festspeicher zur Aufzeichnung von Sensordaten

Dabei setzen die einzelnen Verfahren unterschiedliche Prioritäten. Verfahren wie XNP, Deluge oder Impala berücksichtigen insbesondere mögliche Übertragungsschwierigkeiten und Probleme der Reichweite. Grundlage für die Differenzansätze ist der hohe Energieaufwand der Kommunikation, welcher den Aufwand für zusätzliche Berechnungen des Mikroprozessors rechtfertigt. In verminderter Form gilt dies auch für modulbasierte Verfahren. Hier stößt zudem die Verwaltung der notwendigen Metainformationen an die Grenzen des internen Festspeichers. Daher wird fast ausnahmslos von den Möglichkeiten des externen Festspeichers Gebrauch gemacht. Virtuelle Maschinen nehmen eine Sonderstellung ein. Die Möglichkeit der Rekonfiguration ist ein Nebenprodukt ihrer Entwicklung. Eine zielgerichtete Anpassung an die Anforderungen erfolgt nicht. Durch Ausnutzen der Interpretation von Quellcode bleibt der Energieaufwand der Rekonfiguration dennoch berücksichtigt.

Die enge Bindung an die Anforderungen, insbesondere die der Kommunikation, steht einer allgemeinen Anwendung der Verfahren entgegen. Mikrocontroller können potentiell auf eine Vielzahl von Kommunikationsmöglichkeiten zurückgreifen. Diese können gleiche, lediglich ähnliche oder zuweilen auch gänzlich andere Anforderungen stellen. Beispiele sind kabelgebundene Lösungen, welche eine unmittelbare Kommunikation mit allen Teilnehmern bei hoher Zuverlässigkeit bieten. Im Falle einer nahezu unbegrenzten Energieversorgung kann die Kommunikation zudem unabhängig vom Energieaufwand betrachtet werden. Hingegen sind die erweiterten Speicher von Sensoren im Einsatzgebiet vieler Mikrocontroller nicht vorgesehen. Eine Anwendung der erläuterten Verfahren unter veränderten Anforderungen würde die spezifischen Bedingungen nicht berücksichtigen, die neuen Möglichkeiten nicht nutzen und wäre somit nicht effizient.

Ein allgemeingültiger Ansatz zur automatisierten Rekonfiguration verlangt die strukturelle Trennung von Aufgaben. Bleibt die Kommunikation einer eigenen Schicht vorbehalten, können die dort herrschenden Bedingungen individuell und unabhängig von anderen Aufgaben berücksichtigt werden. Im Gegenzug bleiben die Details der Kommunikation vor den Algorithmen der Rekonfiguration verborgen. Eine Wechsel der Kommunikationsmittel kann so vollkommen transparent erfolgen. Die Rekonfiguration beschränkt sich allein auf die dazu notwendigen Vorgehensweisen. Differenz- und Modulansätze, vormals allein durch den Energieaufwand der Kommunikation motiviert, verlieren durch die Trennung von Kommunikation und Rekonfiguration nicht ihre Berechtigung. Ihre Existenzberechtigung wird statt dessen um neue Punkte erweitert, wie beispielsweise

die Dauer der Kommunikation oder den Grad der Selektivität der Rekonfiguration. Die Unabhängigkeit von Kommunikation und Rekonfiguration erleichtert die Entwicklung von Verfahren auch für andere Anforderungen. Verglichen mit anderen eingebetteten Systemen, sind die Speichermöglichkeiten von drahtlosen Sensoren sehr flexibel und großzügig bemessen. Angepasst an weitaus beschränktere Möglichkeiten, können Entwicklungen die Rekonfiguration einer ganze Reihe gleichartiger eingebetteter Systeme ermöglichen, ohne die verwendete Kommunikation berücksichtigen zu müssen.

Mit dem Einsatz der Middleware COSMIC wird die erläuterte Trennung von Kommunikation und Rekonfiguration beabsichtigt. Die Basis der Interaktion zwischen kommunizierenden Systemen sind Ereignisse. Diese stellen eine gekennzeichnete Information dar. Von den Details der Kommunikation, wie dem Übertragungsformat, der verwendeten Schnittstelle oder dem verwendetem Medium, wird auf diesem Wege abstrahiert. Das zum Einsatz kommende Publish/Subscriber-Prinzip bildet die Grundlage für die Rekonfiguration verschiedenster Mikrocontroller. Entsprechende Vertreter abonnieren dazu den Empfang von Ereignissen im Rahmen ihrer Klassifizierung. Im Gegenzug verbreiten Publisher die zur Rekonfiguration notwendigen Informationen mit Hilfe von Ereignissen. Durch die mit den Ereignissen verbundene Kennzeichnung der Informationen wird die Rekonfiguration adressiert. Die Teilnehmer der Rekonfiguration sind somit der Publisher und entsprechende Subscriber. Alle verfolgen das Ziel der Rekonfiguration mit dem gleichen Verfahren. Durch das Abonnement können die beteiligten eingebetteten Systeme selbstständig entscheiden, an welchem Verfahren sie teilnehmen.

5. Konzeption

Die Rekonfiguration der auf einem Mikrocontroller laufenden Anwendung verlangt, wie bereits in Abschnitt 3 erläutert, die Änderung eines nichtflüchtigen Speichers. Auf diesem Wege bleiben die Änderungen auch nach einem Neustart des Mikrocontrollers erhalten. Bereits die Anforderung zum Einsatz der Middleware COSMIC aus Abschnitt 2.2 legt die Durchführung der Rekonfiguration in die Hände von Software. Entsprechend der Terminologie aus Abschnitt 3.1 kann so von einem Verfahren des IAP gesprochen werden. Für COSMIC existieren sowohl Konzept als auch verschiedene bereits vorbereitete Implementationen. Diese lassen sich von zu übersetzenden Anwendungen auf direktem und einfachem Wege verwenden. Der Einsatz im Rahmen eines Interpreters ist als weitaus komplizierter und aufwendiger zu bewerten. Zudem ist die Übersetzung von Anwendungen bei Mikrocontrollern überaus gebräuchlich. Der Einsatz eines Interpreters würde die Änderung eingespielter und bewährter Verfahren erfordern und ist damit nicht zu rechtfertigen.

Die Verwendung von COSMIC unterteilt jede Anwendung in zwei Bestandteile. Abbildung 5.1 skizziert die Aufteilung. Entsprechend zunehmendem Abstraktionsgrad von

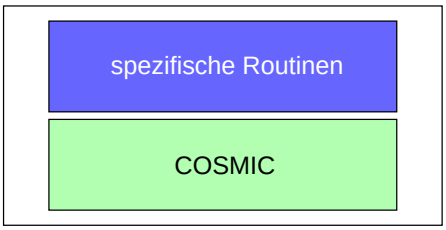


Abbildung 5.1: Bestandteile einer Anwendung

der Hardware sind die Routinen der Middleware im unteren Block zusammengefasst. Auf diese setzen weitere Routinen auf, welche zusammen die Funktionen der Anwendung definieren. Mit entsprechender Funktionalität versehen, kann die Anwendung die Rekonfiguration selbst durchführen.

Bedeutend für die Struktur einer Anwendung ist das Fehlen einer nebenläufigen Ausführung. Der Mikroprozessor bearbeitet ausschließlich die Anweisungen einer Anwendung. Dabei wird er lediglich durch Interrupts unterbrochen. Aus diesem Grunde müssen die Routinen der Middleware, der Rekonfiguration und der Realisierung der eigentlichen Aufgabe in einer Anwendung zusammengefasst sein. Zwei oder mehrere nebenläufig ausgeführte Anwendungen, von denen eine die Aufgabe der Rekonfiguration übernimmt, sind unter diesen Voraussetzungen unmöglich. Ihre Berechtigung haben Anwendungen, deren alleinige Aufgabe die Rekonfiguration ist. Sie können als definierter Ausgangszustand betrachtet werden, auf dem nachfolgende Rekonfigurationen aufbauen. In diesem

Fall kann auf Routinen für weitere Aufgaben verzichtet werden. Verzichtet die Anwendung dagegen auf die Komponenten zur Rekonfiguration, ist der Weg für weitere Rekonfigurationen versperrt.

Wie in Abschnitt 2.1 beschrieben, liegen sämtliche Instruktionen der Anwendung zunächst im Festspeicher vor. Sie bilden die Grundlage für die spätere Programmausführung, unabhängig vom dort verwendeten Speicher. Da sich die Instruktionen frei untereinander referenzieren können, müssen ihre Adressen nicht zusammenhängend sein. Abbildung 5.2 stellt eine mögliche Aufteilung der Bestandteile an verschiedenen Adressen dar. Die Routinen von COSMIC bilden einen zusammenhängendem Block, hier

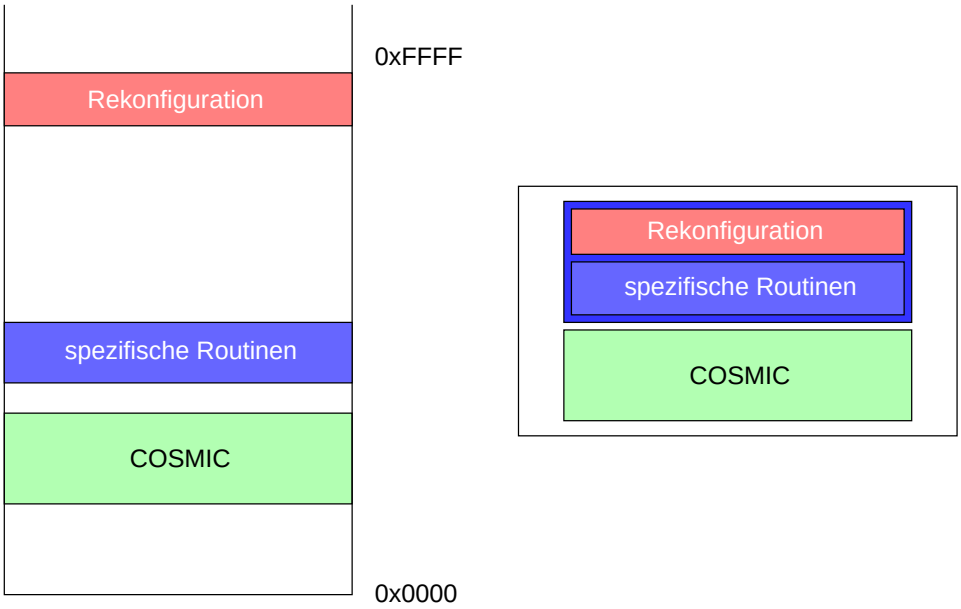


Abbildung 5.2: Verteilung der Bestandteile einer Anwendung im Adressraum

im unterem Adressbereich positioniert. Einen zweiten Block bilden die Routinen zur Rekonfiguration. Den dritten Block stellen die Routinen der eigentlichen Anwendung. Letztgenannte sind für jede Anwendung spezifisch. Sie dienen der Erfüllung der Aufgabe, für welche die Anwendung entworfen wurde. Daher unterscheiden sie sich je nach Einsatzgebiet, Problem und Verfahren zur Lösung.

Mit dem Festspeicher betrifft die Rekonfiguration den Speicher, aus dem die Rekonfiguration selbst ausgeführt wird. Somit besteht die Gefahr, dass der Programmcode der Anwendung, die aktuell die Rekonfiguration durchführt, in einen inkonsistenten Zustand gerät. Die Folgen für den Vorgang der Rekonfiguration, für das Verhalten und die Umwelt des verteilten Systems sind unvorhersehbar. Sie können von nicht mehr ansprechbaren Mikrocontrollern über beschränkt einsatzfähige verteilte Systeme bis hin zu lebensbedrohlichen Situationen für Menschen reichen. Aus diesem Grunde müssen Vorgehensweisen entwickelt werden, welche die Modifikation von aktuell laufendem Programmcode ausschliessen. Mikrocontroller mit Mikroprozessoren nach der von-Neumann-Architektur sind hier prinzipiell bevorteilt. Durch ihre Fähigkeit zur Befehlsausführung aus einem beliebigen Speicher besteht die Möglichkeit, die Ausführung

der Anwendung in einen anderen Speicher zu verlagern. Ein Beispiel für einen anderen Speicher wäre der flüchtige Speicher. In diesem Szenario dient der Festspeicher nur als Permanentspeicher, aus dem eine Anwendung vor ihrer Ausführung geladen wird. Somit kann der Festspeicher ohne Rücksicht auf die Programmausführung verändert werden. Der Größenvergleich der Speicher in Abschnitt 2.1 machen deutlich, dass dieses Vorgehen lediglich in Einzelfällen durchführbar ist. Der Festspeicher eines Mikrocontrollers stellt die mit Abstand umfangsreichste Möglichkeit zum Speichern und Ausführen von Instruktionen dar. Anwendungen, welche problemlos Platz im Festspeicher finden, können leicht das verfügbare Platzangebot anderer Speicher überschreiten. Eine künstliche Größenbeschränkung von Anwendungen würde bedeutende Ressourcen ungenutzt lassen. Zudem setzt die Mehrzahl der bekannten Mikrocontroller auf die Harvard-Architektur. Wie in Abschnitt 2.1 erläutert, ist für das Ausführen von Instruktionen einzig der Festspeicher vorgesehen. Nicht zuletzt auch aus Gründen der Einfachheit wird daher im Folgendem davon ausgegangen, dass eine Rekonfiguration den Speicher betrifft, aus dem sie auch vorgenommen wird.

Die Aufgaben der nachfolgend beschriebenen Vorgehensweisen orientieren sich an der Entwicklung der in Abschnitt 4 erläuterten Verfahren.

5.1. Rekonfiguration der gesamten Anwendung

Ein trivialer Ansatz der Rekonfiguration ist die Ersetzung der gesamten Anwendung. Die durchzuführenden Schritte sind einfach, da keine Rücksicht auf bereits existierende Bestandteile genommen werden muss. Nachteilig ist die Übertragung von vergleichsweise großen Datenmengen. Sämtliche Bestandteile einer Anwendung müssen komplett übertragen werden. Der zuverlässigen Übertragung der Daten kommt so eine hohe Bedeutung zu, Verluste oder Fehler führen unmittelbar zu einer falsch installierten Anwendung. Die Qualität der Kommunikation ist jedoch Aufgabe von COSMIC und kann aus Sicht der Rekonfiguration vollkommen transparent betrachtet werden.

Mit dem Ersetzen der kompletten Anwendung ergibt sich die Möglichkeit zur Verwendung gänzlich neuer Mechanismen. Wie Abbildung 5.3 verdeutlicht, können die Algorithmen zur Rekonfiguration oder zur Kommunikation verändert werden. Anwendungen verzichten dazu schlicht auf die entsprechenden Routinen oder bringen eigene Varianten mit. Wie bereits in Abschnitt 5 dargelegt, muss die neue Anwendung sowohl COSMIC als auch die Routinen zur Rekonfiguration beinhalten, um weitere Rekonfigurationen im Sinne dieser Arbeit zu erlauben.

Gemäß Abschnitt 5 muss verhindert werden, dass eine Anwendung ihren eigenen Bereich des Festspeichers modifiziert. Ein erster und einfacher Ansatz ist es, die Verantwortung dafür dem Programmierer zu übertragen. Eine von ihm vorgenommene Übersetzung muss garantieren, dass nur freie Bereiche des Festspeichers eingenommen werden. Eine Rekonfiguration schreibt somit nur fremde Adressen. Problematisch sind vom Mikrocontroller vorgegebene spezielle Abschnitte des Festspeichers. Adressen wie beispielsweise für Interruptvektoren können nicht frei gewählt werden, sondern sind unveränderlich. Potentiell verwenden beide Anwendungen, sowohl die zu ersetzende als

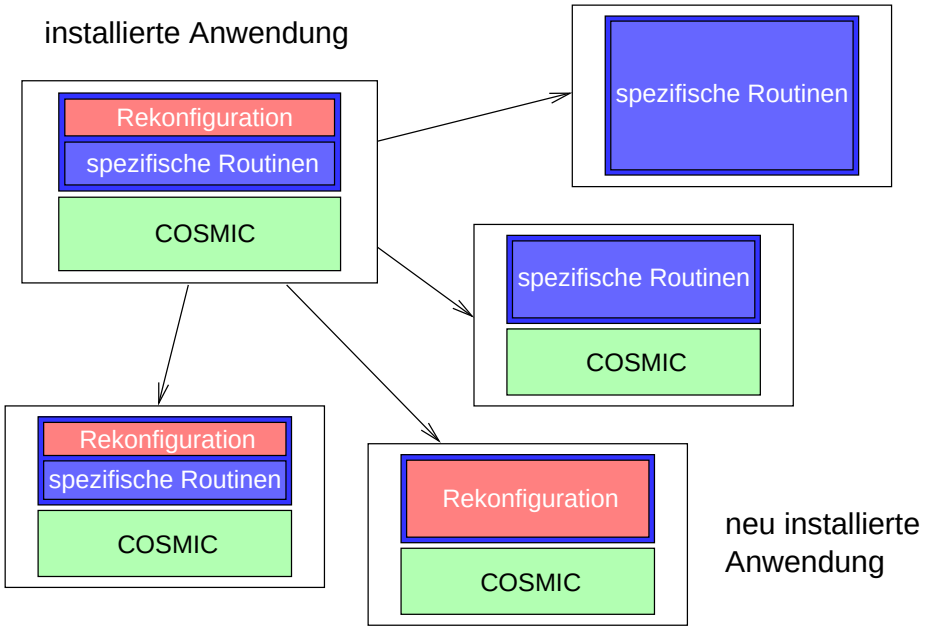


Abbildung 5.3: Mögliche Rekonfigurationen einer Anwendung

auch die neue, derartige Speicherbereiche. Ein häufig anzutreffendes Beispiel ist der Resetvektor. In ihm ist die Anweisung festgehalten, welche unmittelbar im Anschluss an einen Reset ausgeführt wird. Üblicherweise ist dies ein Sprung an eine Adresse, an der die auszuführende Anwendung beginnt. Der Resetvektor beinhaltet vor der Rekonfiguration die Adresse der erste Instruktion der ursprünglichen Anwendung. Nach erfolgreicher Rekonfiguration muss in ihm die erste Adresse der neuen Anwendung vermerkt sein. Gemeinsame Speicherbereiche lassen sich somit nicht vollständig vom Programmierer vermeiden. Verschiedene Vorgehensweisen lösen das Problem.

5.1.1. Explizites Berücksichtigen gemeinsamer Bereiche

Der überwiegende Teil des Festspeichers ist an keine spezielle Funktionalität gebunden. Mit einer Anpassung der Anwendung durch den Programmierer ist somit sichergestellt, dass die zu speichernden Daten in der Mehrzahl Ziele freier Speicherbereiche sind. Die wenigen Ausnahmen müssen durch die schreibende Anwendung erkannt und berücksichtigt werden. Mit Empfang von Programmcode über einen Ereigniskanal überprüft die Anwendung die angesprochenen Adressen. Sind gemeinsam genutzte Speicherbereiche betroffen, muss der entsprechende Programmcode zunächst zwischengespeichert werden. Andernfalls erfolgt umgehend das Speichern der Daten. Nach Empfang der gesamten Anwendung können die im Zwischenspeicher aufbewahrten Anweisungen an ihre angedachte Position geschrieben werden.

Die Vorteile des Vorgehens sind seine Geschwindigkeit und der geringe Bedarf an zusätzlichem Speicher. Wann immer möglich, werden die empfangenen Daten umgehend an ihrer Adresse gespeichert. Zwischenspeicher wird nur für Ausnahmefälle notwendig.

Nachteilig ist der Aufwand zum Erkennen von gemeinsamen Bereichen. Empfangener Programmcode muss generell auf seinen Speicherplatz überprüft werden. Da sich Mikrocontroller verschiedener Baureihen in den Adressen von speziellen Speicherbereichen unterscheiden können, ist stets eine individuelle Lösung erforderlich. Weiterhin setzt das Vorgehen die geeignete Anpassung der neuen Anwendung durch den Programmierer voraus. Dieser muss für jede Rekonfiguration die Einteilung des Festspeichers berücksichtigen und seine Arbeitsschritte entsprechend variieren. Liegt die zu aktualisierende Anwendung in verschiedenen Layouts auf den Mikrocontrollern vor, wird das Vorgehen deutlich erschwert. Um einen gemeinsamen freien Speicherbereich zu bestimmen, muss zusätzlicher Aufwand vom Programmierer betrieben werden.

5.1.2. Umleiten gemeinsamer Bereiche

Speicherbereiche wie die für die Interruptvektoren binden spezifische Funktionalität an vorgegebene Adressen. Dennoch sind sie häufig mit der Absicht entwickelt, die Verwendung von anwendungsspezifischen Programmcode zu ermöglichen. Dazu referenzieren sie ihrerseits andere Adressen. Ist diese Voraussetzung für alle derartigen Bereiche gegeben, kann eine weitere Stufe der Indirektion die Einflussnahme der Rekonfiguration auf die laufende Anwendung verhindern.

Dazu verweisen gemeinsam genutzte Speicherbereiche von alter und neuer Anwendung auf konstante Adressen. Die vom Mikrocontroller vorgegebenen Bereiche brauchen demzufolge nicht im Verlauf einer Rekonfiguration modifiziert werden. Der kritische Bereich hat sich nun auf die referenzierten Adressen verlagert. An diesen wird wiederum auf andere Adressen verwiesen, welche durch eine Anwendung zur Laufzeit gesetzt und verändert werden können. Liegen diese Adressen nicht im Festspeicher, bleiben sie von Änderungen durch die Rekonfiguration verschont. Im Zuge der Ausführung einer Anwendung werden die Referenzen auf die Adressen individuell gesetzt. Der Rekonfiguration bleibt der Einfluss auf diesen Teil der Laufzeitumgebung der aktuell ausgeführten Anwendung verwehrt.

Am Beispiel der Interruptvektoren soll das Vorgehen verdeutlicht werden. Interruptvektoren ermöglichen eine individuelle und effiziente Reaktion auf Ereignisse. Abbildung 5.4 stellt die Strukturen und Beziehungen der Indirektion schematisch dar. Die Reaktion auf Interrupts, in der Abbildung beispielsweise ausgelöst von Timern, sind Sprünge an feste Adressen. Inhalt der angesprungenen Adressen sind erneut Sprungbefehle. Im Gegensatz zur ersten Indirektion sind deren Operanden aber nicht direkt vorgegeben. Statt dessen beinhalten fest vorgegebene Adressen die Ziele der Sprünge. Indem eine Anwendung den Wert der vorgegebenen Speicheradressen modifiziert, kann sie die Umleitung steuern. Wesentlich für die Rekonfiguration ist die Initialisierung der dynamischen Referenzen und deren Gebrauch.

1. Mit dem Resetvektor erfolgt der Sprung in die Initialisierung der dynamischen Referenzen. Sowohl der Festspeicher als auch der flüchtige Speicher bieten die Möglichkeit, zur Laufzeit beschrieben zu werden. Aus Sicht der Rekonfiguration darf der Festspeicher nicht verwendet werden, da sich in diesem Fall das Pro-

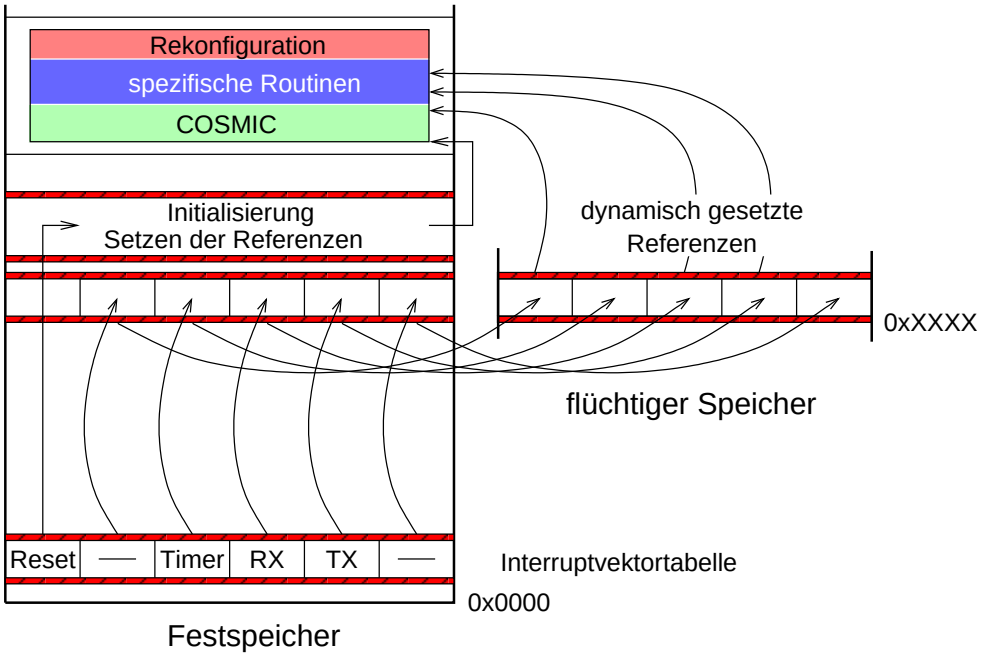


Abbildung 5.4: Strukturen und ihre Beziehung für die Umleitung von Interruptvektoren

blem der gemeinsamen Bereiche lediglich verlagert. Dem kommt zugute, dass für die Speicherung von dynamischen Daten, wie in Abschnitt 2.1 aufgeführt, der flüchtige Speicher benutzt wird. Die Referenzen werden im Festspeicher dauerhaft vorgehalten und bei Programmausführung an die vordefinierten Adressen im flüchtigen Speicher kopiert. Dies ist ein Teil der Initialisierung einer Anwendung.

2. Mit vorliegenden Referenzen im flüchtigem Speicher können die Interrupts umgeleitet werden. In Reaktion auf ein Ereignis wird nach einer Sprungadresse an einer definierten Position im flüchtigem Speicher gesucht. Im Erfolgsfall kann an die gefundene Adresse gesprungen werden.

Das Beispiel des Resetvektors zeigt, dass die Umleitung eine Vorbereitung verlangt. Ein Reset stellt eine besondere Form der Unterbrechung der Arbeit eines Mikroprozessors dar, da er umfangreiche Auswirkungen hat. Eine davon ist das Löschen des flüchtigen Speichers. Daher kann der Resetvektor nicht unmittelbar umgeleitet werden. Damit eine Referenz verfügbar ist, muss vorher die Initialisierung durchgeführt werden. Erst im Anschluss kann an eine durch die Anwendung vorgegebene Adresse verzweigt werden. Die Initialisierung einer Anwendung hat die Eigenschaft, dass sie nur einmal im Verlauf der Ausführung durchgeführt wird. Eine Rekonfiguration kann daher die Initialisierung ohne Auswirkungen auf die laufende Anwendung modifizieren. Sichergestellt werden muss lediglich, dass für die Umleitung der Interrupts auf die gleichen Adressen im flüchtigen Speicher verwiesen wird. Andernfalls wird auf Adressen zugegriffen, die als Daten der laufenden Anwendung noch nicht gültig sind. Eine Rekonfiguration der Initialisierungssequenz kann dazu benutzt werden, um im Anschluss an die Initialisierung an den

weiteren, neuen Programmcode der Anwendung zu verweisen. Das explizite Weiterleiten des Resetvektors erübrigt sich in diesem Fall. Alternativ kann auch die Umleitung für den Sprung an den nachfolgenden Programmcode verwendet werden. Dazu muss die Tabelle der Referenzen lediglich einen Eintrag aufweisen, der im Rahmen der Initialisierung mit der Adresse des weiteren Programmcodes besetzt wird. Kann vorausgesetzt werden, dass die Initialisierungssequenz unverändert bleibt, braucht eine Rekonfiguration des gesamten vorbereitenden Codes nicht durchgeführt werden. Das Verwenden der Umleitung erfordert jedoch einen Eingriff in die Initialisierung einer Anwendung. Es kann nicht davon ausgegangen werden, dass die zumeist von Bibliotheken bereitgestellte Sequenz das Abfragen einer Tabelle vorsieht, um den weiteren Programmcode einer Anwendung zu erreichen. Üblicherweise wird nach Abschluss aller vorbereitenden Maßnahmen direkt gesprungen. Somit stellt das Vorgehen eine Änderung etablierter Abläufe dar. Die Rekonfiguration der Initialisierung, möglicherweise nur zur Änderung des abschließend verwendeten Sprungziels, ist aus diesem Grunde zu bevorzugen.

Mit der Umleitung erfolgt eine Delegation der Aufgaben. Im Gegensatz zur direkten Verwendung anwendungsspezifischen Programmcodes kann die Anwendung zur Laufzeit entscheiden, welcher Programmcode zur Ausführung kommt. Für die Rekonfiguration von Vorteil ist, dass Speicherbereiche, welche durch die alte und neue Anwendung genutzt werden, nicht verändert werden müssen. Einmal beschrieben, bleiben ihre Inhalte unangetastet. Änderungen beziehen sich auf Bereiche, die keinen Einfluss auf die laufende Anwendung haben. Den Vorteilen stehen eine Reihe von Nachteilen entgegen. Zunächst muss garantiert sein, dass die Indirektion für alle betroffenen Speicherbereiche möglich ist. Bereiche, in denen allein der Wert vom Mikroprozessor verarbeitet wird, verbieten die Verwendung des Vorgehens. Wie Abbildung 5.4 zeigt, müssen mit Initialisierung, Sprungtabelle und dem Code zur Umleitung insgesamt drei Abschnitte des Speichers an feste Adressen gebunden bleiben. Zusätzlich müssen die Anwendungen, wie im Vorgehen aus Abschnitt 5.1.1, derart verschoben sein, dass sich ihre Adressbereiche nicht überschneiden. Insbesondere letztgenannte Forderung zwingt den Programmierer, sich mit Details der Rekonfiguration zu befassen. Die Umleitung erfordert einen Teil des flüchtigen Speichers. Der Umfang entspricht dem der Bereiche des Festspeichers, welche vom Mikrocontroller an feste Adressen gebunden sind. Erforderlich ist ebenfalls Festspeicher für den Programmcode zur Verfolgung der Referenzen. Dessen Bearbeitung nimmt Zeit in Anspruch, daher geht die Umleitung mit Performanceeinbußen einher.

5.1.3. Zwischenspeichern und Verschieben der neuen Anwendung

Auch wenn lediglich kleine Bereiche des Festspeichers der Erfüllung spezieller Zwecke dienen, wird damit dem Programmierer die Möglichkeit zur vollständigen Platzierung seiner Anwendung entzogen. In konsequenter Weiterführung dieses Gedankens wird ein Berücksichtigen des bestehenden Speicherlayouts durch den Programmierer überflüssig. Besteht ein ausreichend dimensionierter Zwischenspeicher, kann statt dessen eine komplette Anwendung ohne Anpassungen entwickelt werden. Im Zuge einer Rekonfiguration

wird die neue Anwendung in den Zwischenspeicher geschrieben. Nach Erhalt aller Daten wird die Anwendung aus dem Zwischenspeicher in den Festspeicher kopiert und steht somit zur Ausführung bereit.

Das unveränderte Fortführen des Entwicklungsprozesses einer Anwendung ist der Vorteil des Vorgehens. Der Programmierer kann wie gewohnt für einen Mikrocontroller entwickeln. Nachteilig sind zwei Voraussetzungen, die erfüllt sein müssen. Zum einen dürfen die Routinen zur Ausführung des abschließenden Kopiervorgangs nicht im Adressraum der neuen Anwendung liegen. Zum anderen muss der Zwischenspeicher entsprechend Platz aufweisen. Im Vergleich zum Festspeicher sind andere Speicher, wie in Abschnitt 2.1 beschrieben, bedeutend kleiner. Da der Zwischenspeicher den Umfang der neuen Anwendung begrenzt, stellt eine Aufteilung des Festspeichers in zwei Hälften den besten Kompromiss dar. Abbildung 5.5 stellt den Inhalt des Festspeichers im Verlauf des Vorgehens dar. Zunächst liegt nur eine Anwendung vor, welche die Rekonfiguration

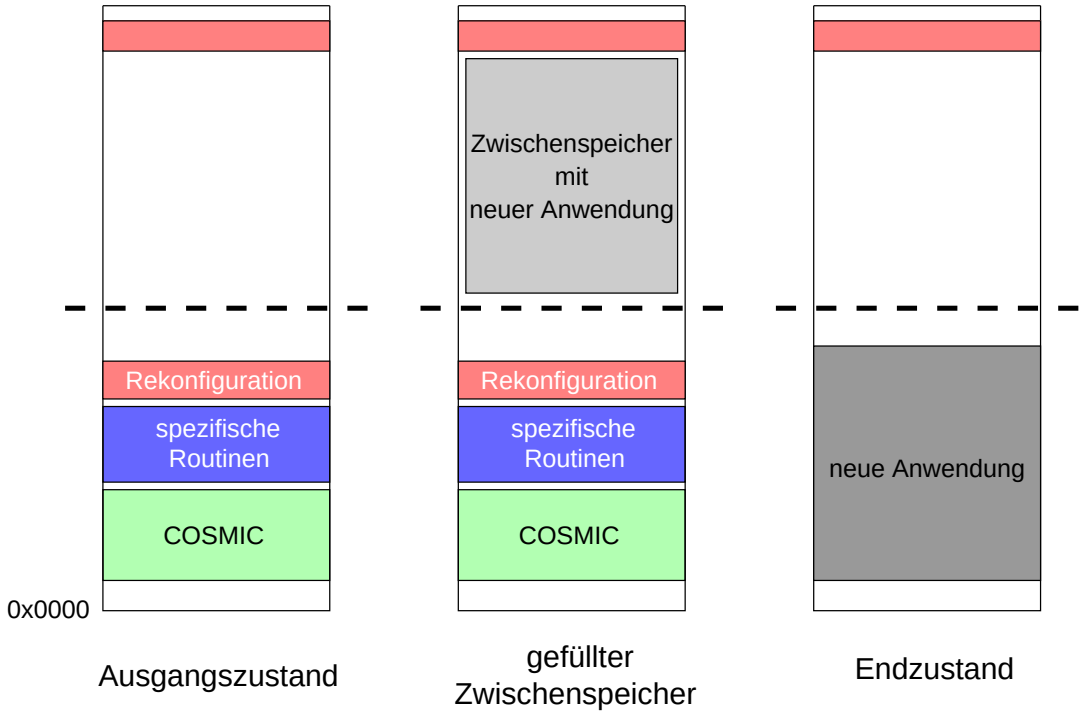


Abbildung 5.5: Zustände beim Zwischenspeichern und Verschieben einer Anwendung

durchführt und zu diesem Zweck über COSMIC kommuniziert. Die Anwendung füllt den Zwischenspeicher in der anderen Hälfte des Festspeichers. Abschließend übergibt sie die Ausführung an weitere Routinen, welche die Kopie in die andere Hälfte durchführen und dabei die alte Anwendung überschreiben.

Die Verwendung des Zwischenspeichers bietet die Möglichkeit zum Recovery. Die Voraussetzungen zum Wiederherstellen eines funktionierenden Zustands sind gegeben, wenn ein Austausch der Speicherinhalte von neuer und alter Anwendung vorgenommen wird. Im Verlauf des Kopierens des Zwischenspeichers wird zu diesem Zweck der Inhalt der betroffenen Zieladressen an einer dritten Stelle gesichert. Wurde der Kopiervorgang ab-

geschlossen, kann die Sicherung an die Stelle des Zwischenspeichers kopiert werden. Eine Unterteilung des Festspeichers in einen weiteren Zwischenspeicher zur Aufnahme der Sicherung schränkt die Möglichkeiten des Vorgehens weiter ein. Ein sequentielles Kopieren von Teilen des Zwischenspeichers ist besser geeignet. Mit entsprechender Einteilung kann ein weiterer Speicher, beispielsweise der flüchtige Speicher, für das Aufbewahren des überschriebenen Programmcodes verwendet werden. Das stückweise Vorgehen birgt die Gefahr, im Falle eines Fehlers sowohl im Zwischenspeicher als auch im Speicher der Anwendung eine Mischung aus altem und neuem Programmcode zu erhalten. War der Austausch dagegen erfolgreich, liegt eine funktionierende Version der Anwendung im Zwischenspeicher bereit. Auf diese kann durch die neue Anwendung selbstständig zurück gegriffen werden, wenn von ihr ein Fehler im eigenen Programmcode erkannt wurde. Dazu ist ein erneuter Austausch der Speicherinhalte und ein anschließender Neustart erforderlich. Mit Wiederherstellung des vormals funktionierenden Zustands ist der Weg zur Rekonfiguration mit einer wiederum verbesserten Anwendung frei.

Aus der Verwendung des Zwischenspeichers ergeben sich auch mehrere Nachteile. Offensichtlich ist, dass für die Anwendung maximal die Hälfte des Festspeichers zur Verfügung steht. Bedingt durch das abschließende Kopieren, nimmt das Rekonfigurieren zusätzlich Zeit in Anspruch. Weiterhin ist der Festspeicher häufig nur für eine begrenzte Anzahl an Schreibvorgängen konzipiert. Aus diesem Grund reduziert das temporäre Speichern die Einsatzdauer eines Mikrocontrollers. Mit der Verfügbarkeit von Flash als Festspeicher kann die Problematik weitestgehend entspannt betrachtet werden. Zwar variiert die garantierte Anzahl möglicher Schreibvorgänge je nach Hersteller. Allgemeine Aussagen lassen sich so weder treffen noch aus der Literatur ableiten. Jedoch wird bereits die als untere Grenze hinreichend akzeptierte Zahl von 10000 Schreibzyklen üblicherweise nicht in der Lebensdauer eines Mikrocontrollers erreicht.

5.1.4. Bewertung

Im Vergleich bietet das Zwischenspeichern und Verschieben aus Abschnitt 5.1.3 den einfacheren und flexibleren Lösungsweg. Die zu schreibenden Daten müssen nicht analysiert und implementierungsabhängig berücksichtigt werden. Der Programmierer braucht sich nicht mit den Details der Rekonfiguration befassen. Diesen Weg beschreiten auch die in den Abschnitten 4.1.1 und 4.1.2 erläuterten Verfahren. Die Halbierung des maximal erlaubten Umfangs einer Anwendung ist hingegen ein großer Nachteil. Insbesondere für Mikrocontroller mit kleinem Festspeicher sind daher die Vor- und Nachteile der beschriebenen Vorgehensweisen sorgfältig gegeneinander abzuwägen. Alternativ müssen andere Ansätze zur Rekonfiguration gewählt werden. Im Falle von ausreichenden Ressourcen bietet die Rekonfiguration der kompletten Anwendung jedoch einen einfachen Ansatz. Das Ignorieren des Speicherinhalts und das Speichern der gesamten Anwendung ähnelt dem Charakter des klassischen ISPs. Vorauszusetzen sind, neben einem ausreichend dimensionierten Festspeicher, weitreichende Kommunikationsmöglichkeiten. Stets muss der komplette Programmcode der Anwendung übertragen werden.

5.2. Rekonfiguration von funktionalen Bestandteilen

Die Vielfalt der Mikrocontroller und ihrer Einsatzgebiete gebietet, dass die Voraussetzungen zur Rekonfiguration aus dem vorigen Abschnitt 5.1 nicht immer gegeben sind. Die Gründe dafür können beispielsweise geringer Festspeicher oder beschränkte Kommunikationsmöglichkeiten sein. Die Rekonfiguration einer kompletten Anwendung ist unter derartigen Voraussetzungen nicht oder nur unter Aufbringung wertvoller Ressourcen möglich.

Die Aufteilung der Anwendung in einzelne Bestandteile bietet die Möglichkeit, die notwendigen Voraussetzungen für die Rekonfiguration deutlich zu senken. Zusammen mit der Lokalität von Veränderungen, beschränkt sich die Aktualisierung auf potentiell einzelne Bestandteile. Ein verringerter Bedarf nach Kommunikation und Speicher sind die direkte Folge. Bereits in Abschnitt 5 sind die funktionalen Bestandteile einer Anwendung erläutert. Abbildung 5.2 verdeutlicht die Aufteilung in die Bestandteile zur Erfüllung der eigentlichen Aufgabe und zur Rekonfiguration. Beide setzen auf die Middleware COSMIC auf, welche den dritten Bestandteil einer Anwendung bildet. Für die Aktualisierung im Sinne dieser Arbeit notwendig sind die Bestandteile der Middleware und der Rekonfiguration. Einmal durch die Anwendung initialisiert, können beide im Zusammenspiel eine Rekonfiguration selbstständig durchführen. Das bedeutet jedoch auch, dass ihr Programmspeicher für die Dauer der Rekonfiguration vor Veränderungen geschützt werden muss. Dagegen werden die spezifischen Routinen der Anwendung nicht unmittelbar benötigt. Ihr Überschreiben beeinflusst den Verlauf der Rekonfiguration nicht. Jedoch muss für diese Zeit auf die Bearbeitung der Aufgabe einer Anwendung verzichtet werden.

Da sich die Ausführung der drei Bestandteile in Zuge einer Rekonfiguration wie beschrieben unterscheidet, müssen unterschiedliche Verfahren zu ihrer Rekonfiguration zum Einsatz kommen.

- Sowohl die Bestandteile COSMIC als auch Rekonfiguration müssen unverändert bleiben. Daher lassen sie sich ausschließlich im Rahmen eines der Verfahren aus Abschnitt 5.1 aktualisieren.

- Die Routinen zur Erfüllung der Aufgabe der Anwendung sind unabhängig von den Prozessen der Rekonfiguration. Ihre Modifikation kann daher ohne Beeinträchtigung der Rekonfiguration durchgeführt werden. Aufgrund des reduzierten Umfangs verringert sich der Aufwand für Speicher und Kommunikation.

In Fortführung der Symbolik aus Abbildung 5.3 verdeutlicht Abbildung 5.6 die Unterscheidung. Der Wegfall der Bestandteile COSMIC oder Rekonfiguration, wie in den unten dargestellten Varianten, ist das Resultat einer kompletten Ersetzung nach Abschnitt 5.1. Alternativ ist auch vorstellbar, dass einer der Bestandteile lediglich nicht in die Anwendung eingebunden ist. In diesem Fall sind die jeweiligen Instruktionen noch im Festspeicher existent, kommen jedoch nicht zur Ausführung. Die Darstellungen geben entsprechend einzig die zur Ausführung kommende Funktionalität wieder. Bleiben

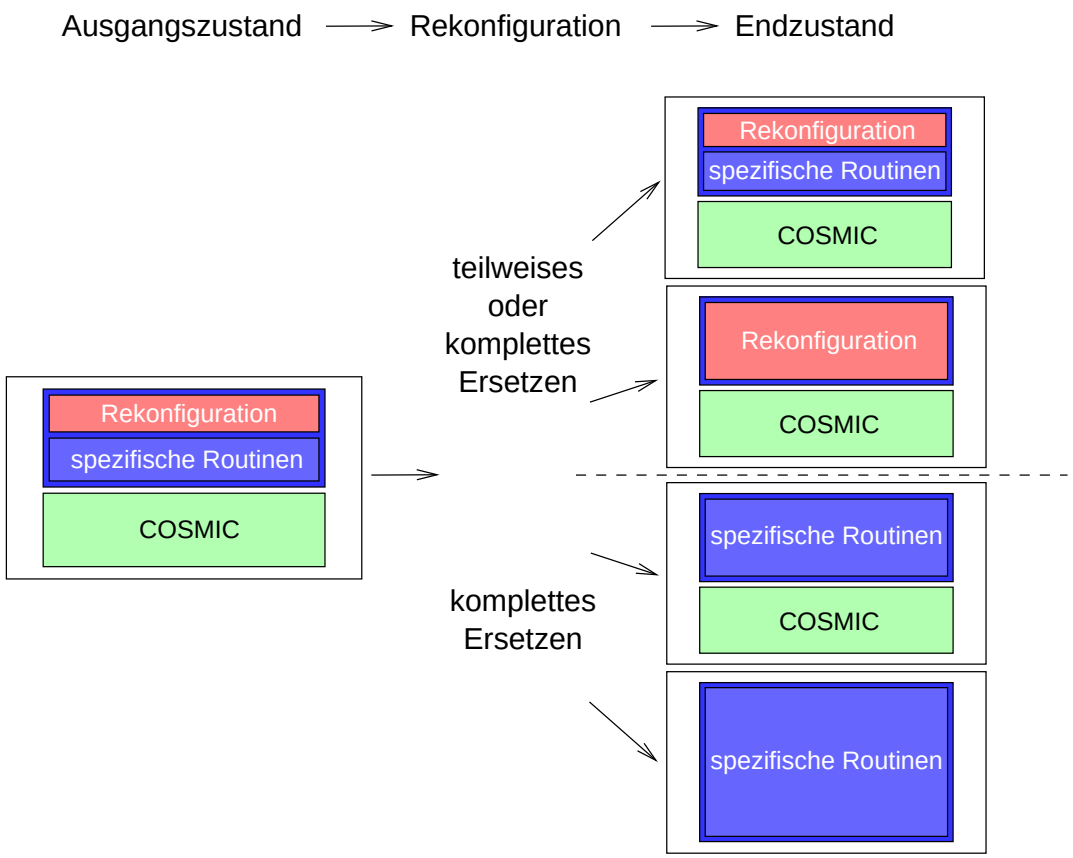


Abbildung 5.6: Einordnung von möglichen Rekonfigurationen einer Anwendung anhand der verwendeten Vorgehensweise

die Bestandteile wie in den oben abgebildeten Varianten dagegen unverändert, ist lediglich die Rekonfiguration der anwendungsspezifischen Routinen durchgeführt worden. Grundlage für zukünftige Modifikationen der Anwendung sind die für die Rekonfiguration notwendigen Bestandteile. Ihre Ablösung ist daher zumeist nicht das Ziel einer Rekonfiguration, eine Modifikation stellt aufgrund der möglichen Auswirkungen auf alle Anwendungen eine Ausnahme dar. Die Beschränkung auf den anwendungsspezifischen Teil kann somit als der Regelfall angesehen werden. Neben seinen Vorteilen besitzt das Vorgehen aber auch mehrere, nachfolgend beschriebene Besonderheiten.

5.2.1. Verhalten der Anwendung

Durch das Überschreiben der spezifischen Routinen einer Anwendung kann der Aufwand für eine Rekonfiguration reduziert werden. Wie bereits erwähnt, wird dies durch die Eigenständigkeit der Bestandteile zur Kommunikation und zur Rekonfiguration möglich. Die für die Rekonfiguration verantwortliche Komponente der Anwendung wird von COSMIC über relevante Ereignisse in Kenntnis gesetzt und beschreibt in Reaktion den Festspeicher. Ein Eingriff oder die Verwendung anderer Bestandteile der Anwendung ist nicht notwendig. Wird ohne Rücksicht auf bestehende Inhalte in den Festspeicher

geschrieben, kann ein temporäres Puffern des neuen Programmcodes unterbleiben. Damit bleibt auch der Festspeicher von zusätzlichen Schreibzyklen verschont. Durch dieses Vorgehen gerät jedoch die Anwendung potentiell in einen inkonsistenten Zustand. Programmcode der installierten und der neuen Anwendung kann zum Zeitpunkt der Ausführung gleichzeitig vorliegen. Wie in Abschnitt 5 beschrieben, sind die Auswirkungen vielfältig und nicht zu akzeptieren. Die Anwendung muss daher die Ausführung der betroffenen Routinen einstellen. Für den Zeitraum der Rekonfiguration kann eine Anwendung demnach nicht der Erfüllung der ihr zugedachten Aufgabe nachkommen.

Das Unterbinden der Ausführung der anwendungsspezifischen Routinen für die Dauer der Rekonfiguration darf keine Auswirkungen auf die Ausführung der zur Rekonfiguration notwendigen Routinen haben. Probleme ergeben sich aus Funktionalität, welche vom Mikrocontroller vorgegeben sowie von beiden Bestandteilen der Anwendung genutzt wird. Interrupts sind eine derartige Funktionalität. Verschiedene Möglichkeiten sind bei der Benutzung von Interrupts denkbar:

- Ein Abschalten aller Interrupts verhindert die Ausführung von Programmcode in Reaktion auf externe Ereignisse. Betroffen ist die gesamte Anwendung. Somit wird ebenfalls Einfluss auf die der Rekonfiguration dienenden Bestandteile genommen, welche auf die Funktionalität unter Umständen angewiesen sind. Nur wenn dies nicht der Fall ist, stellt das globale Abschalten der Interrupts eine Lösung dar. Die der Rekonfiguration dienenden Bestandteile dürfen damit ausschliesslich auf externe Ereignisse pollen, das heißt wiederholend das Vorliegen von Ereignissen überprüfen.

- Bei Beginn der Rekonfiguration führen die aufgerufenen Routinen den Stopp der anwendungsspezifischen Interrupts selbstständig durch. Über die Elemente des Mikrocontrollers, für die die Generierung von Interrupts freigegeben wurde, ist von außen nichts bekannt. Denkbar ist das Abschalten der Generierung von Interrupts von allen nicht benötigten Elementen. Dazu erforderlich sind Informationen über sämtliche Elemente. Eine weitere Möglichkeit ist das Modifizieren der Sprünge in anwendungsspezifischen Programmcode in Reaktion auf Interrupts. Die entsprechenden Mechanismen sind bereits in Abschnitt 5.1.2 erläutert. Wird durch die Anwendung bereits eine Tabelle zur dynamischen Umleitung von Interrupts verwendet, gestaltet sich das Modifizieren einfach. Die entsprechenden Einträge werden gelöscht, so dass betroffene Interrupts nicht weiter verfolgt werden. Besteht keine derartige Tabelle, muss die Interruptvektortabelle des Mikrocontrollers geändert werden. Ist diese Inhalt des Festspeichers, erfordert ihre Änderung zusätzlichen Aufwand. Der Vorteil des Vorgehens ist die Unabhängigkeit von der individuellen Gestalt der Anwendung. Eine Anpassung bei wechselnden Charakteristika kann somit unterbleiben. Eine generelle Lösung stellen die geschilderten Möglichkeiten aber nicht dar. Vorstellbar ist, dass anwendungsspezifische Routinen auf die gleichen Interrupts wie die Routinen der Rekonfiguration reagieren. Die Generierung solcher Interrupts darf für die Durchführung der Rekonfiguration nicht abgeschaltet werden. Somit muss die Reaktion durch anwendungsspezifischen Programmcode unterbunden, durch den Programmcode der Rekonfigurati-

on aber sichergestellt sein. Änderungen der zum Einsatz kommenden Umleitung sind erforderlich. Der Aufwand hängt von den Details der Mechanismen ab.

- Die anwendungsspezifischen Routinen übernehmen selbst die Verantwortung, um einem Aufruf in Reaktion auf Interrupts zu entgehen. Dies kann das Modifizieren von Registern bedeuten, welche die Generierung von Interrupts durch einzelne Elemente des Mikrocontrollers beeinflussen. Ist die Voraussetzung gegeben, ist ebenfalls ein Austragen aus einer Interrupttabelle vorstellbar. Da das Wissen über die verwendeten Ressourcen implizit vorhanden ist, gestaltet sich deren Freigabe einfach.

Ist die Unterbrechung der Arbeit eines Mikrocontrollers für das Einsatzgebiet nicht zu vertreten, muss ein zusätzlicher Schritt in den Prozess der Rekonfiguration eingefügt werden. Das Prinzip unterscheidet sich nicht von dem der kompletten Ersetzung aus Abschnitt 5.1. Auch dort darf die ausführende Instanz nicht während der Rekonfiguration modifiziert werden. Das als vorteilhaft bewertete Vorgehen mit einem Zwischenpuffer lässt sich analog bei der Aktualisierung nur eines Bestandteils der Anwendung einsetzen. Mit dem temporären Puffer ergibt sich zudem die Möglichkeit zum Recovery. Die spezifischen Routinen der laufenden und der neuen Anwendung tauschen zu diesem Zweck lediglich die Speicherplätze. Damit steht im Falle eines Fehlers noch immer ein funktionierender Zustand zur Verfügung. Tritt dagegen beim direkten Überschreiben des Festspeichers ein Fehler auf, ist der Mikrocontroller von weiteren Rekonfigurationen ausgeschlossen. Der vorherige Programmcode geht ohne zusätzliche Maßnahmen verloren.

5.2.2. Behandlung von Größenänderungen

Die Rekonfiguration der anwendungsspezifischen Routinen hat Auswirkungen auf den Umfang des benötigten Festspeichers. So kann mehr, weniger oder gleich viel Speicherplatz erforderlich werden. Ändert sich am Umfang nichts, überschreibt der neue Programmcode exakt die vorhergehenden Anweisungen. Abbildung 5.7 zeigt, dass bei Reduzierung des erforderlichen Speicherplatzes freier Speicher entsteht. Obwohl in diesem Fall offensichtlich Ressourcen ungenutzt bleiben, steht der Ausführung der Anwendung nichts entgegen. Auch dargestellt ist der Fall des Zuwachses an Programmcode. Die Instruktionen der modifizierten Anwendung liegen an Adressen, welche bereits für die Routinen der Rekonfiguration verwendet werden. Damit überschreibt eine Rekonfiguration den Programmcode, der gerade zur Ausführung kommt. Die weitere Ausführung der Rekonfiguration wird beeinflusst. Das Auftreten dieses Zustandes muss, wie in Abschnitt 5 beschrieben, vermieden werden.

Abbildung 5.7 lässt vermuten, dass die Auswirkungen von Größenveränderungen mit einem direkten Beschreiben des Festspeichers im Zusammenhang stehen. Überschreiben die Routinen zur Rekonfiguration ihren eigenen Programmspeicher, ist der weitere Verlauf nicht vorherzusehen. Aber auch das nachträgliche Kopieren eines vorher temporär gespeicherten Programmcodes ist von den Beschränkungen betroffen. Das Ziel des Inhalts des Zwischenspeichers bleibt, wie beim direkten Überschreiben, der Speicherplatz

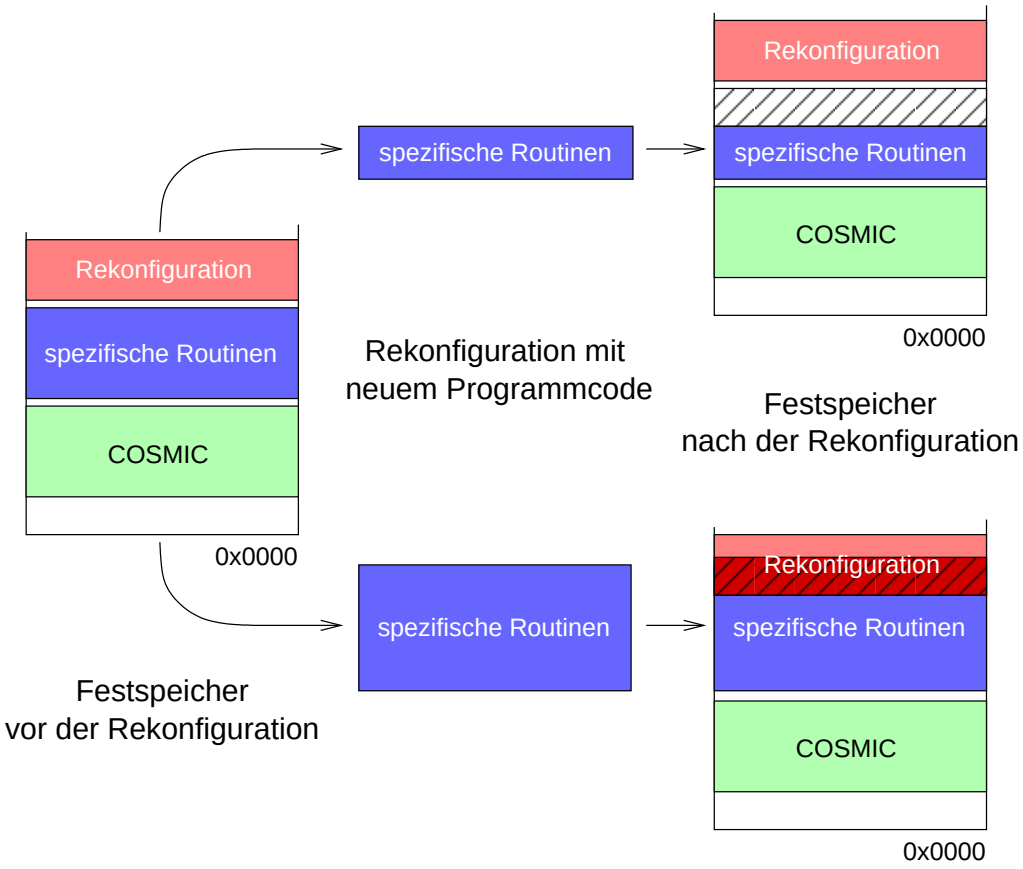


Abbildung 5.7: Auswirkungen von Größenveränderungen beim Ersetzen der anwendungsspezifischen Routinen

der vorhergehenden Routinen. Die Einschränkungen durch nachfolgenden Programmcode sind somit ebenfalls gegeben. Lediglich die Quelle der zu schreibenden Daten ist mit dem Zwischenspeicher eine andere.

Um ein Überschreiben von Programmcode zu vermeiden, sind verschiedene Vorgehensweisen bekannt:

a) Offensichtlich ist das Verschieben des nachfolgenden Programmcodes an höhere Adressen.

b) Mit freiem Speicher zwischen den einzelnen Bestandteilen kann der verwendete Programmcode an Umfang zu- oder abnehmen, ohne bei der Rekonfiguration andere Bestandteile der Anwendung in Mitleidenschaft zu ziehen.

c) Es muss sichergestellt sein, dass der zu ändernde Bestandteil die höchsten Adressen innerhalb der Anwendung einnimmt.

Eine korrekte Ausführung von verschobenen Instruktionen, wie sie für Punkt a) erforderlich ist, gewährleistet nur positionsunabhängiger Code. Nach Abschnitt 3.3 stellt derartiger Programmcode einen Ausnahmefall dar. Für Programmcode, der sich auf

vorgegebene Adressen bezieht, ist eine erneute Relokation notwendig. Ohne die Anpassung der absoluten und relativen Referenzen entsteht fehlerhafter Programmcode. Die Relokation erfordert Informationen, welche nicht Inhalt des Programmocdes sind. Um die Information zu erhalten und diese zu verwalten, ist die Aufwendung einer Vielzahl von Ressourcen erforderlich. Die Vorgehensweisen aus den Punkten b) und c) sind hingegen vergleichsweise einfach umzusetzen. Abbildung 5.8 veranschaulicht das jeweilige Prinzip. Die Verwendung eines Puffers stellt die deutlich schlechtere Möglichkeit dar.

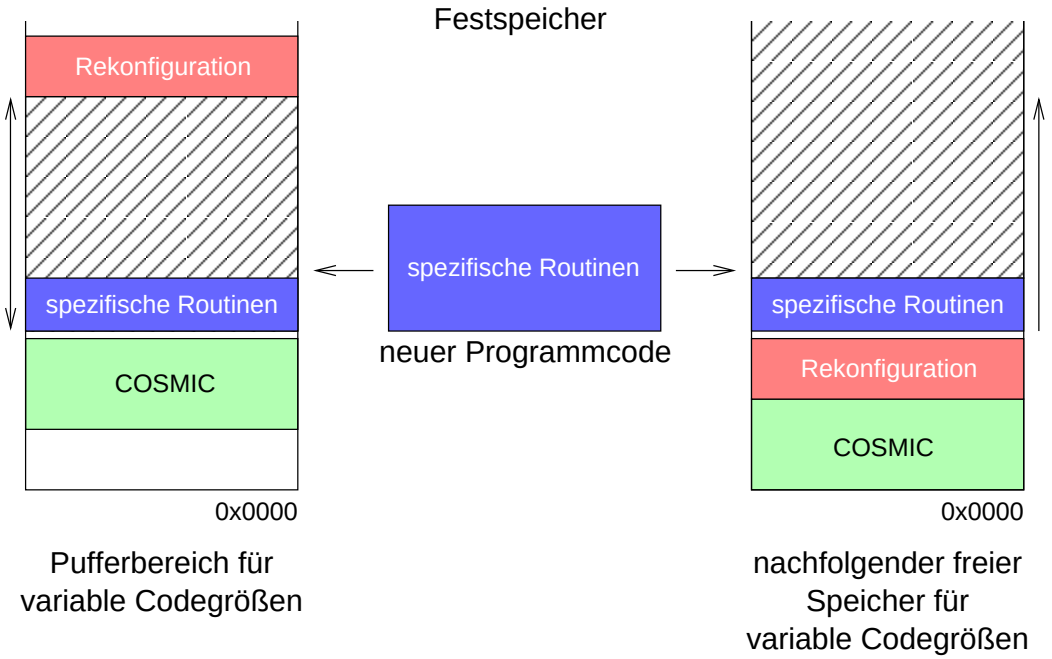


Abbildung 5.8: Varianten zur Behandlung von Größenveränderungen beim Ersetzen der anwendungsspezifischen Routinen

Hier wird Festspeicher unabhängig vom tatsächlichem Bedarf reserviert, zudem müssen einzelne Bestandteile explizit platziert werden. Hinzu kommt, dass die Möglichkeit zum Überschreiben von Programmcode noch immer gegeben ist. Übersteigt der Umfang des einzuspielenden Programmcodes den Puffer, wird in nachfolgende Bestandteile eingegriffen. Wie in Abschnitt 5.2 beschrieben, betrifft die Rekonfiguration lediglich einen Bestandteil der Anwendung. Einer ständigen Platzierung dieses Bestandteils an das Ende des Anwendung steht somit nichts im Wege. Für den entsprechenden Programmcode kann frei auf Speicher aus dem nachfolgenden, ungenutzten Bereich zurück gegriffen werden.

5.2.3. Auswirkungen auf den Entwicklungsprozess

Für die Rekonfiguration der anwendungsspezifischen Routinen müssen die notwendigen Bestandteile bereits auf dem Mikrocontroller vorliegen. Im Lebenszyklus einer Anwendung finden dafür zwei unterschiedliche Vorgehensweisen Verwendung. Gemäß dem

Gedanken des IAP, muss die erste Version der erforderlichen Routinen zunächst auf anderem Wege im Mikrocontroller gespeichert werden. Stehen die Routinen zur Rekonfiguration auf dem Mikrocontroller bereit, kann in nachfolgenden Vorgängen sämtlicher Programmcode durch die Anwendung selbst aktualisiert werden. Das letztgenannte Vorgehen entspricht der Rekonfiguration im Sinne dieser Arbeit.

Für beide Möglichkeiten bietet es sich an, zugleich eine Version der anwendungsspezifischen Routinen einzuspielen. Mit hinreichend funktionierendem Programmcode kann auf diese Weise bereits die Bearbeitung der Aufgabe der Anwendung eingeleitet werden. Die Alternative besteht, wie in Abschnitt 5 aufgeführt, aus dem Verzicht auf die anwendungsspezifischen Routinen. Unabhängig davon erfordert die Möglichkeit zur Aktualisierung eines Teils der Anwendung die Unterscheidung zweier Fälle:

1. Bleiben die Routinen zur Rekonfiguration unverändert, muss der neue Programmcode gegen deren Symbole gelinkt werden. Neben der Anpassung an das vorhandene Speicherlayout stellt der Linkprozess sicher, dass Fragmente der unveränderten Bestandteile nicht Inhalt der Rekonfiguration sind. Mehrfach definierte Symbole und mehrfach verwendeter Speicherplatz führen zu einem Abbruch des Linkens. Durch sein Vorgehen muss der Anwender zudem sicher stellen, dass der auf diese Weise referenzierte Programmcode mit dem Speicherinhalt des Mikrocontrollers überein stimmt. Veränderte Platzierungen des Programmcodes, ob als Produkt von selbstständigen Entscheidungen des Linkers oder des Anwenders, resultieren in neuen Zuweisungen von Adressen. Wird für das Linken eine andere Adressierung verwendet, als auf dem Mikrocontroller vorliegt, passt der erzeugte Programmcode nicht zu den vorliegenden Bestandteilen. Bei der Ausführung kommt es zur Verwendung von Adressen, die eventuell anders belegt sind, als beim Linken angenommen wurde. Die Ausführung der gesamten Anwendung bleibt in diesem Fall unbestimmt.

2. Ist eine Modifikation der Routinen der Rekonfiguration erforderlich, muss eine der Vorgehensweisen zur Rekonfiguration der gesamten Anwendung aus Abschnitt 5.1 zum Einsatz kommen. Der gebräuchliche Entwicklungsprozess, bestehend aus dem Übersetzungsvorgang aus Abbildung 3.1 sowie dem Transfer an den Mikrocontroller, kommt unverändert zum Einsatz. Obwohl die anwendungsspezifischen Routinen möglicherweise unverändert bleiben, werden sie somit neu eingespielt. Eine denkbare Alternative besteht darin, die unveränderten anwendungsspezifischen Routinen gegen die Symbole der modifizierten Routinen der Rekonfiguration zu linken. Ein Einspielen der unveränderten Bestandteile kann somit zunächst unterbleiben, im weiteren Verlauf ist aber eine Anpassung notwendig. Aufgrund der Unantastbarkeit des betroffenen Codes ist dennoch zwingend weiterer Festspeicher erforderlich. Da der zu ändernde Programmcode gerade ausgeführt wird, muss der Inhalt der Rekonfiguration temporär oder permanent an anderen Adressen gespeichert werden.

Je nach Modifikation der Bestandteile muss der Anwender für die richtige Wahl des Vorgehens Sorge tragen. Eine Anpassung des gewohnten Entwicklungsprozesses ist somit unumgänglich.

5.2.4. Bewertung

Eine Rekonfiguration nur der anwendungsspezifischen Routinen bietet mehrere Vorteile. Die Aktualisierung der Bestandteile zur Rekonfiguration unterbleibt. In der Folge reduziert sich der Aufwand an Kommunikation, an Energie und nicht zuletzt an Zeit. Der Festspeicher wird von der Durchführung von Schreiboperationen entlastet. Der Fokus der Entwicklung bleibt auf der Erfüllung der Aufgabe der Anwendung beschränkt. Dies lässt eine Minderung an Fehlerquellen und eine kompaktere Anwendung erwarten. Nachteilig ist die notwendige Anpassung des Entwicklungsprozesses. Deren statische Natur erlaubt aber eine Automatisierung. Einmal entsprechend vorbereitet, können die Abweichungen vom üblichen Vorgehen wiederholt und ohne zusätzlichen Aufwand berücksichtigt werden.

Besonders geeignet erscheint zunächst der Ansatz, den anwendungsspezifischen Bestandteil direkt zu überschreiben. Der volle Festspeicher steht der Anwendung zur Verfügung. Er wird von Schreiboperationen nur im notwendigen Maße beansprucht. Jedoch ist die Ausführung der Anwendung für die Dauer der Rekonfiguration nicht gegeben. Verschiedene Einsatzszenarien akzeptieren einen zwischenzeitlichen Ausfall mehrerer Mikrocontroller, andere dagegen nicht. Die Faktoren des Umfangs der Rekonfiguration und der Kommunikation beeinflussen die Dauer der Unterbrechung. Der Faktor Kommunikation lässt sich zwar durch Anforderungen an die Ereigniskanäle von COSMIC bestimmen. Eine allgemeingültige Aussage zur Dauer lässt sich durch den variablen Umfang der Rekonfiguration aber nicht treffen. Das Zwischenspeichern und abschließende Kopieren des Programmcodes ist eine geeignete Alternative. Der Einfluss der Kommunikation auf die Dauer der Unterbrechung ist nicht mehr vorhanden. Lediglich interne Eigenschaften des Mikrocontrollers und seines Festspeichers sind relevant. Für deren zeitliches Verhalten beim Kopieren des Programmcodes kann angenommen werden, dass lediglich ein Bruchteil der Dauer der Kommunikation notwendig ist. Ein ernst zu nehmender Nachteil ist die Beschränkung des Umfangs der Aktualisierung. Da nur die anwendungsspezifischen Routinen Gegenstand der Rekonfiguration sind, steht dennoch mehr Platz zur Verfügung als im Vorgehen aus Abschnitt 5.1.

5.3. Rekonfiguration von Modulen

Die aufeinanderfolgenden Entwicklungsstände von Software weisen häufig nur geringe Unterschiede zueinander auf, insbesondere im Vergleich zum Umfang der gesamten Software. Kann die Granularität der Rekonfiguration weiter in Richtung der getätigten Änderungen erhöht werden, ist eine erneute Reduzierung des Aufwands und der Voraussetzungen zu erwarten.

Ein erster Schritt zur Strukturierung einer Anwendung ist die Abschnitt 5.2 verfolgte Zerlegung in funktionale Bestandteile. Bei weiterer Unterteilung bilden Module die nächsten Elemente. Die Unterteilung entspricht der vom Programmierer vorgenommenen Strukturierung des Quellcodes in Dateien. Änderungen zwischen verschiedenen Entwicklungsständen drücken sich durch Änderungen des Quellcodes in Dateien aus. Durch die individuelle Übersetzung von Dateien spiegeln sich Änderungen in den jeweils resultierenden Objektdateien wieder. Objektdateien sind mit Modulen gleichzusetzen. Sie beinhalten den zum Quellcode äquivalenten Maschinencode. Brauchen nur die Objektdateien von geändertem Quellcode betrachtet werden, erscheint eine Vereinfachung der Rekonfiguration möglich.

In den Erläuterungen zu bereits existierenden Verfahren aus Abschnitt 4.1 wird beschrieben, dass die Auswirkungen von Veränderungen im Quellcode nicht lokal beschränkt bleiben. Wie die Dateien des Quellcodes, stehen auch die Objektdateien untereinander in Zusammenhang. Zum Erhalt einer ausführbaren Anwendung werden Objektdateien unter Beachtung ihrer Beziehungen verbunden. Veränderungen in den Objektdateien haben jedoch Einfluss auf die Beziehungen. Abbildung 5.9 verdeutlicht dies an einem Beispiel. Die Module A, B und C referenzieren sich über ihre Bestandteile.

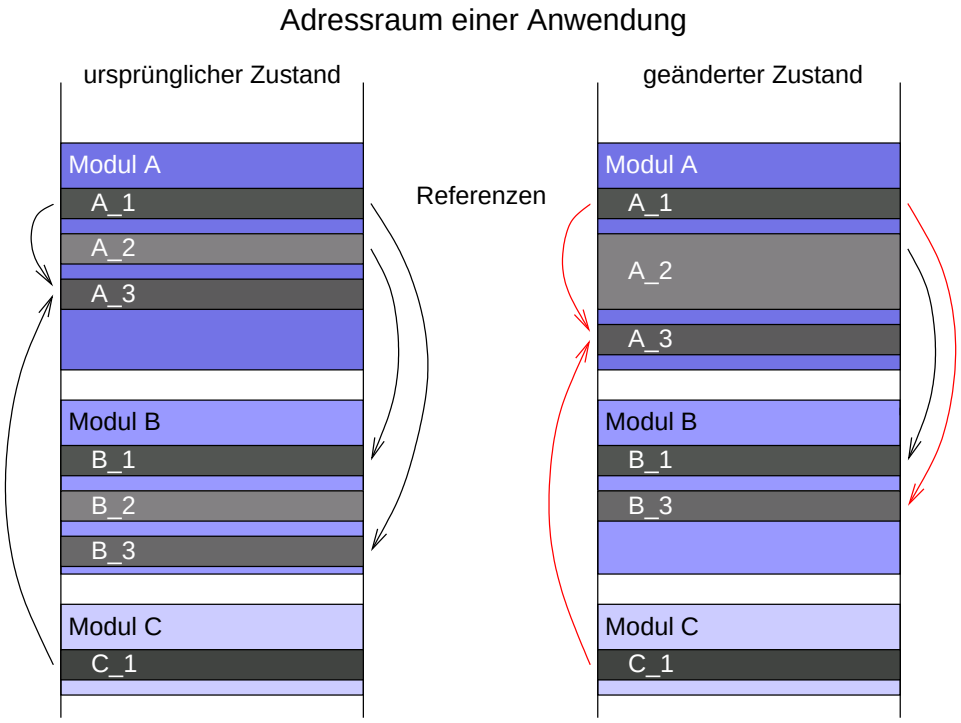


Abbildung 5.9: Beispiel für die Folgen der Veränderung von Modulen

Zusätzlich sind auch Referenzen innerhalb eines Moduls möglich, wie eine beispielhafte Referenz von A_1 auf A_3 zeigt. Welches Detail eines Moduls die einzelnen Bestandteile repräsentieren, ist unerheblich. Typische Vertreter sind Routinen, Konstanten oder Variablen. Änderungen eines Moduls haben Folgen für die über Referenzen ausgedrückten Beziehungen. Im Beispiel legt der Bestandteil A_2 an Größe zu. Die Beziehung zwischen

den Bestandteilen A_1 und A_3 bleibt erhalten. Durch die erforderliche Verschiebung von A_3 muss jedoch das Ziel der Referenz aktualisiert werden. Zusätzlich ist es notwendig, beim Linken der Module die zwischenmodulare Referenz von C_1 auf A_3 neu aufzulösen. Dies gilt auch für die Beziehung von A_1 zu B_3. Durch das Entfernen von B_2 hat sich die Position von B_3 innerhalb des Moduls B geändert. Für den Fall, dass die Änderung nicht in die neu erstellte Anwendung einfließt, verweist A_1 auf eine falsche Adresse. Wie die Referenz von A_2 auf B_1 auch zeigt, brauchen unveränderte Verweise nicht erneuert werden.

Am Beispiel zu sehen ist die Repräsentation der Beziehungen durch Referenzen. Für jedes Modul gilt, dass Referenzen auf andere Module vorhanden sein können. Um ein Modul im Rahmen einer Rekonfiguration in die Menge der vorhandenen Module einzuarbeiten, sind die folgenden Schritte notwendig:

- Auflösen der Referenzen des neuen Moduls auf die bereits vorliegenden Module
- Aktualisieren der Referenzen der vorliegenden Module auf das neue Modul

Beide Schritte sind unter dem Begriff der Symbolauflösung (*Symbol Resolution*) bekannt. Für ihre Durchführung sind mehrere Informationen erforderlich:

1. Benötigt wird die Adresse, auf welche eine Referenz abzielt. Um ein nachfolgendes Linken gegen eine erzeugte Objektdatei überhaupt zu ermöglichen, pflegt der Übersetzer Informationen über referenzierbare Adressen in die Objektdatei mit ein. Für jede der Adressen wird ein abstrakter Name vergeben, so dass für die Adresse ein symbolischer Bezeichner bzw. ein Symbol vergeben ist. Da die Bezeichner nach außen bereitgestellt werden, sind für sie Begriffe wie exportierte Symbole (*Exported Symbols*) oder Einstiegspunkte (*Entry Points*) [37] gebräuchlich.

2. Der Ausgangspunkt einer Referenz muss bekannt sein. Eine Referenz benennt lediglich ein Symbol als Ziel. Ist dem Symbol zum Zeitpunkt des Übersetzens keine Adresse zugeordnet, kann der Übersetzer eine derart referenzierte Adresse nicht in den Maschinencode einarbeiten. Daher versieht er die Objektdatei mit Informationen, um einem nachfolgenden Prozess die zu aktualisierende Position im Maschinencode mitzuteilen. Der nachfolgende Prozess, das Linken, aktualisiert die derart beschriebene Adresse. Ihr Inhalt wird an die Verteilung der Module im Speicher angepasst. Aus diesem Grunde lassen sich die vom Übersetzer eingebrachten Information als Relokationseinträge (*Relocation Entries*) bezeichnen.

Jedes zu linkende Modul bzw. jede zu linkende Objektdatei beinhaltet die Informationen zu den jeweils verwendeten Symbolen in Form einer Tabelle. Gleiches trifft auf die Relokationseinträge zu. Der Informationsgehalt der jeweiligen Tabelleneinträge ist unterschiedlich. Abbildung 5.10 stellt die Einträge gegenüber. Da mehrere Relokationseinträge auf das gleiche Symbol verweisen können, bietet sich die Verwendung eines Zeigers auf einen Symbolnamen an. Dadurch kann Platz gespart und Redundanz verhindert werden. Möglich ist auch ein Verweis auf eine Symbolinformation, da diese per

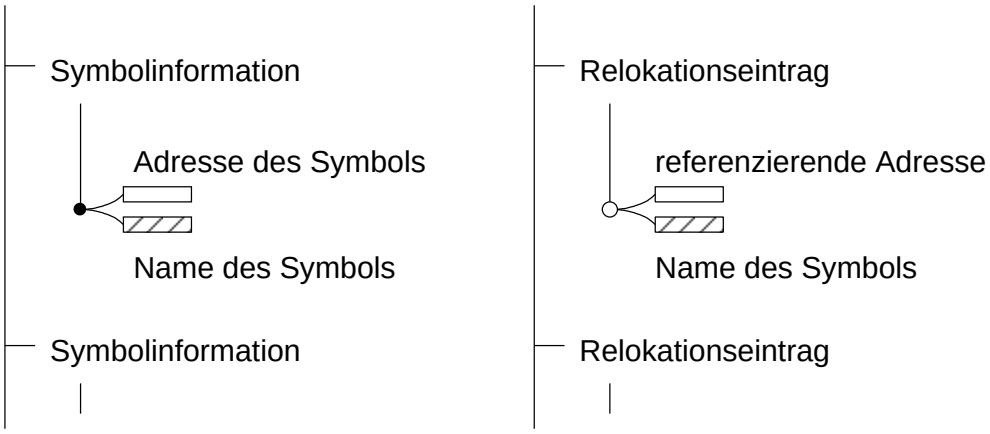


Abbildung 5.10: Attribute von Symbolinformationen und Relokationseinträgen

Definition den Namen eines Symbols bereit hält. In Symbolinformationen kann ebenfalls ein Zeiger auf einen Symbolnamen zur Anwendung kommen. Eine Unterscheidung von definierten, dass heißt mit einer gültigen Adresse versehenen Symbolen, und undefinierten Symbolen muss dabei möglich bleiben. Die jeweiligen Details sind vom Format abhängig, in welchem die Objektdateien vorliegen.

Die in den Symbol- als auch Relokationstabellen vorliegenden Daten sind Metadaten zum Programmcode eines Moduls. Zusammenfassend sind zur Rekonfiguration eines jeden Moduls drei Gruppen von Informationen notwendig:

1. Der Programm- bzw. Maschinencode des neuen Moduls
2. Die Symbole sowohl des neuen als auch der bereits vorliegenden Module
3. Die Relokationseinträge sowohl des neuen als auch der bereits vorliegenden Module

Einzig für die spätere Ausführung bedeutend ist der Programmcode. Die zuletzt aufgeführten Informationen sind lediglich zu dessen Erstellung im Vorfeld notwendig. Das Bereithalten der Gesamtheit der Metadaten auf dem Mikrocontroller ermöglicht die selbstständige Rekonfiguration eines einzelnen Moduls. Jeder Mikrocontroller kennt seinen eigenen Festspeicher sowie dessen Belegung und kann das Linken individuell durchführen. Auf diesem Wege werden aber Ressourcen an Speicher für einen Prozess reserviert, der unter Umständen selten oder gar niemals zur Ausführung kommt. Im Gegenzug verfügt der Host im Regelfall über ausreichende Ressourcen für die Speicherung und für den Gebrauch der Metadaten. Jedoch würde eine Rekonfiguration eines Moduls ein Linken für einen fremden Speicher bedeuten. Über dessen Layout fehlen dem Host bislang die Informationen.

Vorstellbar ist eine Verteilung der Aufgaben. Bei diesem Kompromiss werden die zum Linken notwendigen Informationen getrennt vorgehalten. Zur Rekonfiguration wird explizit von der Kommunikation Gebrauch gemacht. Ein derartiges Vorgehen entspricht dem Gedanken der verteilten Systeme.

5.3.1. Autonomes Linken auf dem Mikrocontroller

Abgesehen von den weitaus beschränkteren Ressourcen, ist jeder Mikrocontroller mit dem Host vergleichbar: Ein Mikroprozessor bearbeitet mit Hilfe von angeschlossenem Speicher und Peripherie eine Aufgabe. Bei Vorlage aller Informationen kann das Verbinden aller Module zu einer ausführbaren Anwendung auch lokal durchgeführt werden. Die Arten der erforderlichen Informationen wurden bereits im vorigem Abschnitt 5.3 aufgeführt.

Durch dieses Vorgehen beschränkt sich die zur Rekonfiguration notwendige Kommunikation auf die Übertragung einer oder mehrerer Objektdateien. In ihnen sind der Programmcode und die Metadaten der jeweils neuen Module vereint. Die möglichen Auswirkungen des Vorgehens auf die Kommunikation hängen vom Umfang der Metadaten ab. Mit dem Verzicht auf die Übertragung unveränderter Module werden zunächst Vorteile verbunden. Die Reduzierung der zu übertragende Module bedeutet eine Verringerung des Datenaufkommens. Entsprechend weniger Ressourcen sind aufzubringen. Nicht zuletzt wird das Kommunikationsmedium geringer beansprucht. Mit einer hohen Anzahl an Metadaten besteht jedoch die Gefahr, dass die Vorteile wieder zunichte gemacht werden. Sind aufgrund der Metadaten mehr Daten zu übertragen, als durch den Verzicht auf Module eingespart werden, hat das Vorgehen entgegengesetzte Folgen.

In Anschluss an eine Übertragung der Module kann der Mikrocontroller das Linken wie der Host durchführen:

- Anhand der Relokationseinträge eines neuen Moduls sind die auf diese Weise ausgedrückten Referenzen aufzulösen. Die dazu notwendigen Symbole müssen sämtlichst auf dem Mikrocontroller definiert sein. Dies schließt neben den Symbolinformationen des neuen Moduls vor allem die Symbolinformationen der vorhandenen Module ein.

- Alle Referenzen, welche auf das aktualisierte Modul gerichtet sind, müssen ebenfalls aktualisiert werden. Dafür sind die Relokationseinträge der installierten Module erforderlich. Da im Vorfeld unbekannt ist, welches Modul rekonfiguriert wird, müssen vorsorglich alle Relokationseinträge aller Module aufbewahrt werden. Die notwendigen Symbolinformationen bringt das neue Modul in seiner Objektdatei mit.

Sind mehrere Module Bestandteil der Rekonfiguration, können Referenzen zwischen diesen bestehen. Daher sind vor jeder Symbolauflösung zunächst die Symbolinformationen zu aktualisieren. Für die Adressen der Symbole und der Relokation besitzt jeder Mikrocontroller durch seine Platzierung des Programmcodes die alleinige Verantwortung. Somit kann sich das Speicherlayout aller Mikrocontroller für die gleiche Anwendung unterscheiden, ohne dass dies für die Rekonfiguration von Bedeutung ist.

Zur Ersetzung von bestehenden Modulen muss vorausgesetzt werden, dass sowohl neue als auch installierte Module eindeutig identifiziert werden können. Aus Gründen der Einfachheit bietet sich der Name der Datei des Moduls als Signatur an. Um eventuelle Größenänderungen von Modulen berücksichtigen zu können, ist das Wissen über

den belegten Speicher notwendig. Die Aussagen zum Problem der Größenänderungen aus Abschnitt 5.2.2 lassen sich auf die Module übertragen. Da nicht jedes Modul an den freien Restspeicher grenzen kann, ist eine Speicherverwaltung erforderlich. Für gänzlich neue Module muss freier Speicher alloziert werden. Module, welche existierende Module aktualisieren, können die vormals verwendeten Adressen erhalten. Liegen Größenveränderungen vor, ist es Aufgabe der Speicherverwaltung, eine entsprechende Startadresse für ein Modul bereitzustellen. Dabei notwendig gewordene Verschiebungen können ein erneutes Linken von existierenden Modulen erforderlich machen.

Entscheidend für die Anwendbarkeit des Vorgehens ist der Umfang des erforderlichen Speichers. Neben dem Programmcode müssen alle Metadaten für ein eventuelles Linken vorgehalten werden. In der Einleitung in [27] dient ein sog. *True-life Example* zur Erläuterung des Linkprozesses. Dies soll exemplarisch dazu verwendet werden, einen Eindruck vom Ausmaß der Metadaten zu erhalten. Für seinen Quellcode sei auf Anhang A verwiesen. Das Beispiel entspricht dem klassischem „Hello-World“ in der Programmiersprache C. Es besteht aus den Dateien `m.c` und `a.c`. Für beide stellt Abbildung 5.11 die Größe der in einem Übersetzungsprozess erzeugten Metadaten dar. Zur Ermittlung der Daten

Quelldatei `m.c`

```
Section Headers:
  [Nr] Name              ...   Size
       ...
  [ 1] .text             ...   000026
  [ 2] .rel.text         ...   000010
  [ 3] .data             ...   000010
       ...
  [ 8] .symtab           ...   0000a0
  [ 9] .strtab           ...   000018
```

Quelldatei `a.c`

```
Section Headers:
  [Nr] Name              ...   Size
       ...
  [ 1] .text             ...   000041
  [ 2] .rel.text         ...   000008
  [ 3] .data             ...   000000
       ...
  [ 8] .symtab           ...   000090
  [ 9] .strtab           ...   00000d
```

Abbildung 5.11: Beispielhafter Umfang der Metadaten

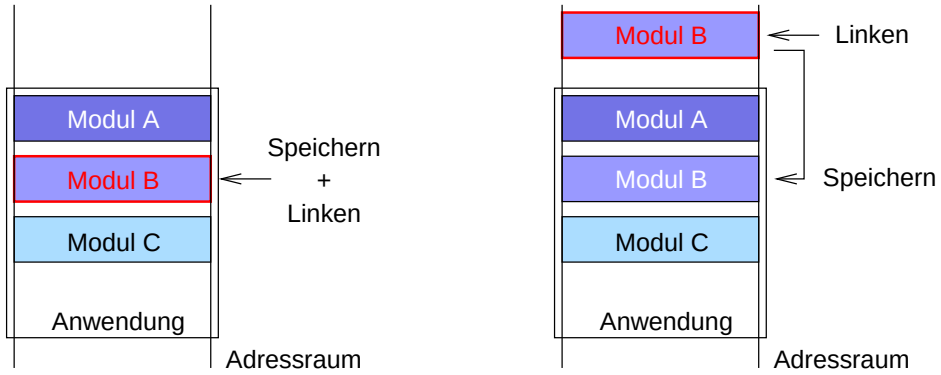
kommt das Tool readelf zum Einsatz, welches Teil der *binutils* [12] ist. Die vorherige Übersetzung des Quellcodes erfolgt wie in [27] mit dem C-Compiler aus der GNU Compiler Collection (GCC) [13] (Version 4.1.1). Wie der Name des Tools schon vermuten lässt, liegen die Objektdateien im ELF vor. Der unveränderliche Programmcode ist jeweils im Abschnitt (*Section*) „.text“ enthalten, zugehörige Daten im Abschnitt „.data.“. Zusammengenommen beträgt der Umfang 54 bzw. 65 byte. Die Relokationseinträge und die Symboltabelle sind in den entsprechend benannten Abschnitten „.rel.text“ und „.symtab“ enthalten. Direkt oder indirekt verwenden beide den Abschnitt „.strtab“, um auf den Namen eines Symbols zu verweisen. Alle drei Abschnitte machen zusammen die Metadaten aus. Ihr Speicherbedarf beträgt mit 200 bzw. 165 Byte ein Vielfaches des Bedarfs des Programmcodes.

Das im Beispiel gegebene Verhältnis zwischen Programmcode und Metadaten lässt sich nicht allgemein auf jeden Quellcode übertragen. Zu groß ist die Einflussnahme des Programmierers durch die Struktur und durch die Verwendung von Verweisen im Quellcode. Zusätzlich beeinflusst der Programmierer die Länge von Symbolnamen durch seine Wahl von Bezeichnern. Hier nimmt ebenfalls der Compiler Einfluss. Im *Name Mangling* genannten Prozess erfolgt die Festlegung der Namen der Symbole. Grundlage sind die vom Programmierer im Quellcode verwendeten Bezeichner. Mit Bestimmung der Symbolnamen ist auch der Platzbedarf festgelegt. Am Beispiel deutlich wird aber, dass bereits sehr einfacher Programmcode zu vergleichsweise vielen Metadaten führen kann. Die Rekonfiguration darf dem Programmierer aber keine Änderung seines Quellcodes aufzwingen, wie auch der Übersetzungsvorgang transparent behandelt werden muss. So ist von einem potentiell hohen Anteil an Metadaten in einer Objektdatei auszugehen. Ihr Vorhalten hat somit bedeutenden Einfluss auf die Anwendung. Große Bereiche an Festspeicher bleiben für die Zwecke der Rekonfiguration reserviert und stehen der Anwendung nicht mehr zur Verfügung. Hinzu kommen die Eingangs des Abschnitts angeführten Auswirkungen umfangreicher Metadaten auf die Kommunikation.

In den in Abschnitt 4.2 vorgestellten Verfahren wird dem Problem unterschiedlich begegnet. Durch umfangreiche Speichermöglichkeiten kann das Problem im Verfahren aus Abschnitt 4.2.1 schlicht ignoriert werden. Im Gegensatz dazu beschreiben die Abschnitte 4.2.2 und 4.2.3 Verfahren, die unter deutlich strengeren Anforderungen an den verfügbaren Speicher entwickelt wurden. Hier wird auf das Problem mit Einschränkungen reagiert. Eigene Objektformate werden vorgeschlagen, um auf diese Weise den notwendigen Speicherbedarf zu reduzieren. Entsprechend lassen sich Algorithmen und Routinen, welche aus der Behandlung der Standardformate bekannt sind, nur bedingt übertragen. Bei Verwendung des Standardformats ELF beschränkt das Verfahren aus 4.2.2 die Möglichkeit zum Linken auf vorgegebene Symbole und verringert so die erforderlichen Informationen.

Analog zu den Betrachtungen aus Abschnitt 5.2.3, muss auch die Möglichkeit von Fehlern untersucht werden. Vorstellbar ist, dass ein Linken eines Moduls aufgrund eines fehlenden Symbols nicht erfolgreich ist. Als nahe liegende Ursache sind Module auszumachen, welche für eine andere Ausgabe der Anwendung entwickelt wurden als auf dem Mikrocontroller vorliegend. Anders als im Vorgehen nach Abschnitt 5.2, in welchem ein funktionaler Bestandteil ungeprüft eingearbeitet wird, besteht durchaus die Möglichkeit zur Einflussnahme. Die einzelnen Vorgehensweisen hängen davon ab, ob das betreffende Modul bereits vor dem Linken in den Kontext der Anwendung einbezogen wurde. Abbildung 5.12 skizziert die folgenden bekannten Varianten:

- Ist das Modul bereits einbezogen, liegt als Folge eines Fehlers eine inkonsistente Anwendung vor. Ihre Ausführung muss unterbunden werden. Ein Modul kann durch Überschreiben des Vorgängers einbezogen sein, oder im Falle eines neuen Moduls durch bereits eingearbeitete Referenzen von anderen, ebenfalls neuen Modulen. Abbildung 5.12a verdeutlicht die Situation. Mit Stopp der Anwendung fällt der Mikrocontroller für das verteilte System aus. Bleibt der Mikrocontroller jedoch für weitere Rekonfigurationen empfänglich, kann der Fehler mit passenden



(a) im Adressraum der Anwendung (b) in einem Zwischenspeicher

Abbildung 5.12: Platzierung eines zu linkenden Moduls

Modulen behoben werden. Aus diesem Grund müssen die Routinen der Rekonfiguration ihre Ausführung bei erkannten Fehlern nicht einstellen, sondern einzig aktiv bleiben.

- Liegen sämtliche neuen Module an anderen Adressen, als sie relokiert werden, bleibt die Anwendung von Fehlern unbeeinflusst. Wie in Abbildung 5.12b beispielhaft dargestellt, werden neue Module zunächst zwischengespeichert. Infolgedessen verändert das Linken lediglich Speicherinhalte außerhalb des Adressraums der Anwendung. Im Falle eines Fehlers kommt daher die Anwendung weiterhin zur Ausführung. Erst im Erfolgsfall werden die komplett gelinkten neuen Module in den Adressraum der Anwendung eingespielt. Ihre Ausführung ist sichergestellt.

Beide Vorgehen hängen somit davon ab, an welchen Adressen die zu linkenden Module vorliegen. Ersteres ist eine Fortführung des Schreibens in den Adressraum der Anwendung, wie in Abschnitt 5.2 angesprochen. Letztgenanntes Vorgehen entspricht der Verwendung eines Zwischenspeichers entsprechend Abschnitt 5.1.3.

5.3.2. Vollständiges Linken auf dem Host

Von den an einer Rekonfiguration beteiligten Systemen nimmt der Host eine Sonderstellung ein. In Abschnitt 3.1 wird er als das System bezeichnet, welches die Daten einer Rekonfiguration verbreitet. Oftmals stimmt er aber auch mit dem System überein, auf dem die Anwendungen für Mikrocontroller entworfen und implementiert werden. Dies gilt insbesondere im Forschungs- und Entwicklungsbereich. Im Folgenden soll daher angenommen werden, dass der Host ebenfalls mit den Aufgaben der Softwareentwicklung betreut ist. Findet die Entwicklung auf einem anderen System statt, gelten die nachfolgenden Aussagen entsprechend.

Wie in Abschnitt 3.3 erläutert, ist das Linken ein Bestandteil des Entwicklungsprozesses. Der Ort der Softwareentwicklung setzt voraus, dass die notwendigen Ressourcen auf

dem Host vorhanden sind. Dazu zählen adäquate Rechenleistung, ausreichender Speicher und verfügbare Software. Die Rekonfiguration eines einzelnen Moduls erfordert das Linken gegen andere Module. Der Vorgang unterscheidet sich nicht vom Linken im Rahmen der Softwareentwicklung, in dem ebenfalls Module gegeneinander gelinkt werden. Allein durch die Äquivalenz der Prozesse bietet sich der Host für die Rekonfiguration eines Moduls an. Der Blick auf die vorhandenen Ressourcen, welche zumeist ein Vielfaches von denen der Mikrocontroller betragen, untermauert die Eignung des Hosts.

Ein Linken auf dem Host kehrt die Aufgabenverteilung aus dem vorigen Abschnitt 5.3.1 ins Gegenteil um. Demzufolge sind auch die Schritte der Rekonfiguration vertauscht:

1. Das neu einzuspielende Modul muss auf die bereits vorhandenen Module gelinkt werden

2. War das Linken erfolgreich, kann das derart vorbereitete Modul an die Mikrocontroller übertragen und in den Adressraum der Anwendung geschrieben werden

Die Anwendung kommt nicht auf dem Host zur Ausführung, sondern auf einem anderen System mit eigenem Speicher. Um das Linken durchzuführen, ist der Host daher auf Informationen über die Belegung des fremden Speichers durch die Anwendung angewiesen. Benötigt werden die in Abschnitt 5.3 aufgeführten Informationen zur Symbolauflösung, bezogen auf die von der Anwendung verwendeten Adressen im Speicher der Mikrocontroller. Zum Auflösen der Referenzen müssen die Symbolinformationen der auf dem Mikrocontroller installierten Module verfügbar sein. Der aktualisierte Programmcode kann gegen diese gelinkt werden. Zur Aktualisierung von Referenzen auf die neue Modulen sind die Relokationseinträge der installierten Module erforderlich. Auf ihren Programmcode kann verzichtet werden, da lediglich einzelne Adressen angepasst werden müssen. Stets berücksichtigt werden muss die Platzierung der Module.

Es ist Aufgabe des Hosts, an die erforderlichen Informationen zu gelangen. Zwei Möglichkeiten sind bekannt:

1. Der Host definiert und pflegt die Adressen der Anwendung und ihrer Module über mehrere Rekonfigurationen hinweg. Diese Möglichkeit entspricht dem in Abschnitt 4.1.5 vorgestellten Verfahren. Basierend auf einem bekannten Speicherlayout, wird eine Rekonfiguration durchgeführt. Die dabei verwendeten Symbolinformationen, Relokationseinträge und Speicherbelegungen dienen als Eingangsinformationen für eine nachfolgende Rekonfiguration. In deren Verlauf werden die aufbewahrten Informationen aktualisiert, um wiederum für folgende Rekonfigurationen zur Verfügung zu stehen. Abbildung 5.13 skizziert das Vorgehen. Positiv ist die minimale Kommunikation. Lediglich neue Inhalte von Speicheradressen werden übertragen. Von großem Nachteil ist jedoch die bloße Annahme zum Speicherlayout der Adressaten. Eine Berücksichtigung des aktuellen Zustands erfolgt nicht. Der Empfänger kann die Daten einer Rekonfiguration nicht überprüfen, da es sich lediglich um Inhalte von Speicheradressen handelt. Daher besteht keine Möglichkeit zur Korrektur im Falle eines abweichenden Layouts auf Seiten des Empfängers. Da das

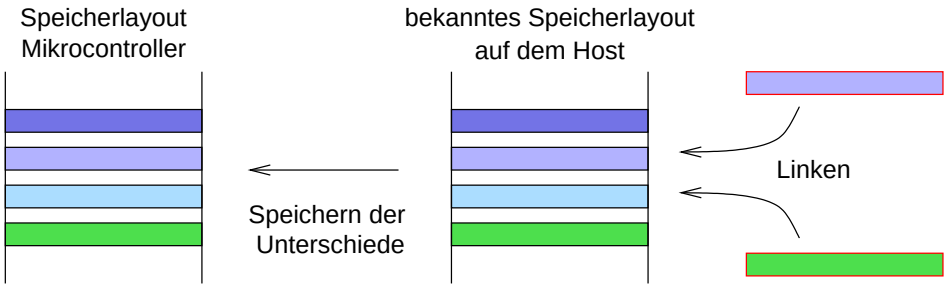


Abbildung 5.13: Linken auf dem Host bei Pflege des Speicherlayouts

Vorgehen diesbezüglich mit dem Vorschlag zur Rekonfiguration aus Abschnitt 5.2 verwandt ist, gelten die dort getroffenen Aussagen entsprechend.

2. Der Host überträgt die Aufgabe der Adressierung der Module an die Mikrocontroller. Zwar hält er sämtliche Module einer Anwendung vor, erfragt aber deren Platzierung im Speicher von den Adressaten der Rekonfiguration. Unter der Voraussetzung, dass die Metadaten der Module auf die Startadressen bezogen sind, kann der Host die Symbolauflösung für die erhaltenen Adressen durchführen. Abbildung 5.14 verdeutlicht den Ablauf des Vorgehens. Die Beteiligung der Mikrocontroller

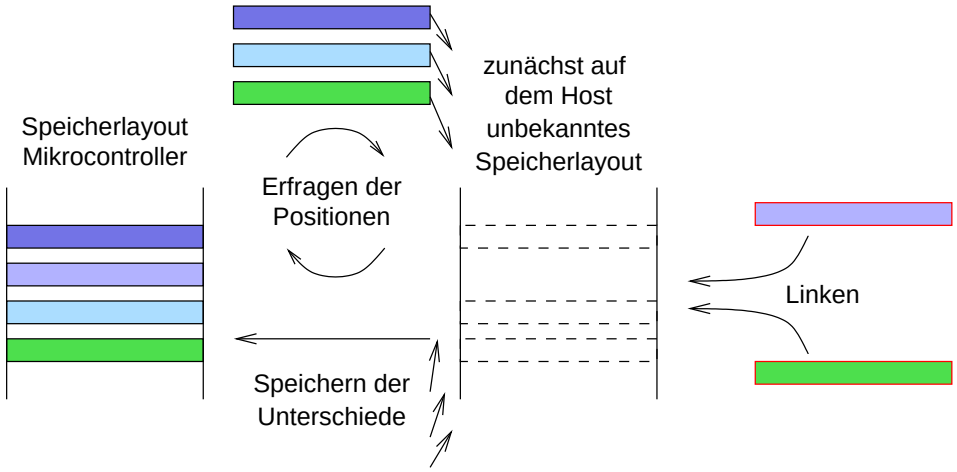


Abbildung 5.14: Linken auf dem Host bei Anfrage des Speicherlayouts

erhöht die Sicherheit. Das Linken erfolgt mit bestätigten Adressen, so dass die übertragenen Daten der Rekonfiguration zum Speicherinhalt der Mikrocontrollers passen. Auf diese Weise können auch verschiedene Speicherlayouts behandelt werden. Bei erkannten Unterschieden zwischen einzelnen Mikrocontrollern wird dazu das Linken mehrfach mit den jeweils verschiedenen Adressen durchgeführt. Alternativ kann die Rekonfiguration in einem solchen Fall auch abgebrochen werden. Damit wird der Aufwand für das Linken und für die Übertragung an verschiedene Mikrocontroller vermieden.

Durch die Anfragen des Hosts erhöht sich das Kommunikationsaufkommen. Ein möglicher Einfluss auf die Anwendbarkeit des Vorgehens hängt aber vom Einsatzgebiet und deseen Anforderungen ab. Ein dagegen konzeptionelles Problem ist die notwendige Adressierung der Empfänger, soll die Behandlung unterschiedlicher Speicherlayouts vorgenommen werden. Es muss sichergestellt sein, dass der Inhalt einer Rekonfiguration genau die Mikrocontroller anspricht, für welche die im Linkprozess verwendeten Daten zutreffen. Die Zuordnung der Empfänger einer Rekonfiguration widerspricht aber dem Grundgedanken des P/S-Prinzips.

Das Linken auf dem Host hat den Vorteil, dass Fehler im Linkprozess keine Auswirkungen auf die Arbeit des verteilten Systems haben. Können Module nicht erfolgreich gelinkt werden, findet keine Übertragung von Programmcode an die Mikrocontroller statt. Ursachen für Fehler im Linkprozess können, analog wie in Abschnitt 5.3.1, unterschiedliche Entwicklungsstände der Anwendung sein. Vermieden wird so auch der Kommunikationsaufwand für offensichtlich ungeeignete Daten.

5.3.3. Linken auf dem Mikrocontroller mit Hilfe von Anfragen an den Host

Zur Beseitigung des Widerspruchs zwischen unzureichenden Ressourcen auf Seiten der Mikrocontroller und unzureichenden Informationen auf Seiten des Hosts bietet sich eine Verteilung der Aufgaben an. Bleibt der Mikrocontroller mit der Bearbeitung der Relokationseinträge beauftragt, so bleibt ihm die Kontrolle über die Rekonfiguration erhalten. Unter Benutzung der Kommunikation kann der Host Aufgaben zur Speicherung und Verwaltung der zum Linken benötigten Informationen übernehmen. Im Gegensatz zum Vorschlag aus Abschnitt 5.3.1 sind die Mikrocontroller somit auf die Zusammenarbeit mit dem Host beim Linken von Modulen angewiesen. Im Gegenzug kann der Aufwand zur Speicherung von Metadaten reduziert werden.

Für die Aufgaben des Linkens sind, wie in Abschnitt 5.3 beschrieben, Relokationsinformationen erforderlich. Die Abbildung von Relokationsinformationen auf Adressen erfolgt mit Hilfe von Symbolen und Symbolinformationen. Wie in Abschnitt 5.3.1 angedeutet, ist der notwendige Speicherplatz für einen Symbolnamen keine feste Größe. Somit muss auch für die in der Symboltabelle zusammengefassten Symbolinformationen von umfangreichem Speicherbedarf ausgegangen werden. Der Bedarf für die Relokationseinträge hängt von der Repräsentation des referenzierten Symbols ab. Verweisen Relokationseinträge lediglich auf einen Eintrag in einer Symboltabelle, besitzen sie eine feste und im Vergleich zu den Symbolinformationen geringe Größe. Verschiedene Möglichkeiten zur Aufteilung der Metadaten zwischen Host und Mikrocontroller sind vorstellbar.

Eine erste Möglichkeit zur Verteilung der Aufgaben liegt darin, sämtliche Metadaten auf dem Host zu belassen. Im Unterschied zum Verfahren aus Abschnitt 5.3.2 führt aber der jeweilige Mikrocontroller das Linken aus. Zur Relokation der Module ist die wiederholte Ausführung der folgenden Schritte notwendig:

1. Der Mikrocontroller fragt den Host nach weiteren Relokationseinträgen.

2. Hat der Host bereits alle Relokationseinträge versendet, signalisiert er dies mit einer entsprechenden Nachricht. Andernfalls verbreitet er einen Relokationseintrag. Da der Mikrocontroller das zugehörige Symbol nicht selbstständig auflösen kann, ist ebenfalls das Ziel der Relokation erforderlich. Für beide Informationen ist die Zuordnung zu einem Modul notwendig. Die Identifizierung der Module kann wie in Abschnitt 5.3.1 über den Dateinamen des Moduls erfolgen. Insgesamt sind die folgenden Informationen erforderlich:

   - referenzierende Adresse
   - Name des Moduls der referenzierenden Adresse
   - referenzierte Adresse bzw. referenziertes Symbol
   - Name des Moduls der referenzierten Adresse bzw. des referenzierten Symbols

   Vorausgesetzt werden müssen relative Adressangaben. Die Ermittlung von absoluten Adressen ist Aufgabe des Linkens. Mit Hilfe der Namen der Module kann der Mikrocontroller deren Startadressen ermitteln. Dementsprechend kann er Adressangaben, welche sich auf den Start eines Moduls beziehen, in absolute Adressen umrechnen. Bezieht sich die Adressangabe nicht auf den Beginn, sondern auf eine festgelegte Position in einem Modul, so ist eine weitere Information zur Beschreibung dieses Offsets erforderlich.

Abbildung 5.15 zeigt eine schematische Darstellung. Die darin verwendete Symbolik ist

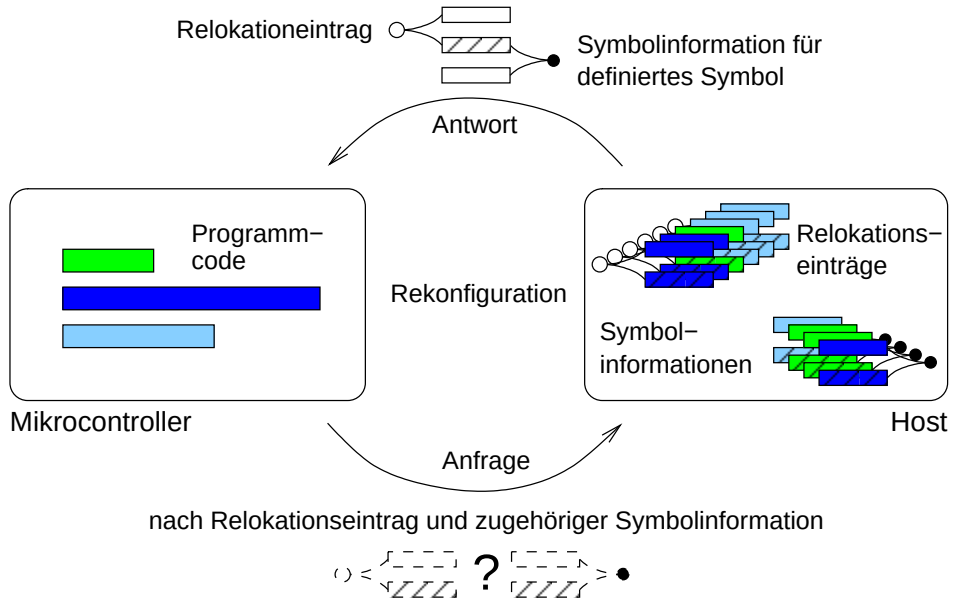


Abbildung 5.15: Linken bei Speicherung aller Metadaten auf dem Host

bereits aus Abbildung 5.10 bekannt. Die beschriebene Vorgehensweise bedeutet Aufwand für die Kommunikation und für die Bearbeitung durch den Host. Letztgenannter

besitzt jedoch ausreichende Möglichkeiten. Jeder Mikrocontroller wird von der Speicherung aller Metadaten befreit. Von Nachteil ist die notwendige Terminierung der Abfragen. Der Host muss die abgeschlossene Bearbeitung aller Relokationseinträge durch einen Mikrocontroller erkennen. Dazu sind mehrere Möglichkeiten denkbar:

- Der Host kann die Anfragen eindeutig den einzelnen Mikrocontrollern zuordnen. Auf diese Weise kann er selbstständig den Versand der Relokationsinformationen an jeden Empfänger überwachen. Jedoch entspricht eine Zuordnung nicht dem Gedanken der Kommunikation nach dem P/S-Prinzip. Die Vorgehensweise gleicht in diesem Aspekt dem Vorschlag aus Abschnitt 5.3.2.

- Der Mikrocontroller teilt dem Host die bereits bearbeiteten Relokationsinformationen mit. Dies kann im Rahmen der Anfrage des Mikrocontrollers erfolgen. Der Host kann mit der empfangenen Information noch nicht versandte Relokationsinformationen bestimmen und übermitteln. Ist keine weitere Übertragung erforderlich, erfolgt das Signal zum Abschluss der Bearbeitung. Neben der nicht notwendigen Zuordnung ist die Berücksichtigung von Relokationsinformationen, die aus Anfragen anderer Mikrocontroller resultieren, von Vorteil. Eine Reduzierung der Anfragen ist zu erwarten. Hier kann eine Beschreibung durch einen Bitvektor wie im Verfahren aus Abschnitt 4.1.2 zum Einsatz kommen. Damit ein Mikrocontroller die einzelnen Relokationsinformationen unterscheiden und vermerken kann, ist für jede ein Bezeichner erforderlich. Sowohl der Bezeichner als auch die Angaben zu bereits bekannten Relokationsinformationen erhöhen das zu übertragende Datenvolumen.

- Gibt der Mikrocontroller in seiner Anfrage die Anzahl der bearbeiteten Relokationsinformationen kund, kann ebenfalls auf eine Zuordnung verzichtet werden. Dazu ordnet der Host sämtliche Relokationseinträge sequentiell an. In Reaktion auf eine Anfrage versendet er den nachfolgenden Eintrag mit den bereits erwähnten zusätzlichen Daten. Stimmt die Anzahl aus der Anfrage mit der Anzahl der Relokationseinträge überein, signalisiert der Host das Ende des Linkprozesses. Die Anzahl der bearbeiteten Relokationsinformationen dienst somit als Sequenznummer, die durch die Bearbeitung inkrementiert wird. Ein derartiges Vorgehen ist einfach zu realisieren. Eine Kennzeichnung der übertragenen Informationen ist auch hier notwendig. Die Beschränkung auf eine Zahl lässt eine moderate Erhöhung des Kommunikationsaufkommens erwarten. Empfangene Relokationsinformationen, welche aus Anfragen von anderen Mikrocontrollern resultieren, können ohne weiteres jedoch nicht verwendet werden. Übertragungen, deren Sequenznummern sich an die jeweils aktuelle Sequenznummer anschließen, können von einem Mikrocontroller jederzeit berücksichtigt werden. Um aber bereits empfangene Relokationseinträge überspringen zu können, ist eine Verwaltung der entsprechenden Sequenznummern notwendig. Mit einer solchen Verwaltung kann auch bei diesem Vorgehen von einer Reduzierung der Anfragen an den Host ausgegangen werden.

Zusammengefasst erfordern alle Möglichkeiten zusätzlichen Aufwand. Die Verwendung von Sequenznummern stellt eine allgemein anwendbare Lösung dar, welche gleichzeitig sehr einfach ist.

Eine zweite Möglichkeit der Verteilung besteht aus der Speicherung der Relokationseinträge auf den Mikrocontrollern. Eine Vereinfachung der Kommunikation ist die Folge. Jeder Mikrocontroller erhält Zugriff auf die Relokationseinträge der Module seiner Anwendung. Jeder Mikrocontroller kann somit auch eigenständig feststellen, ob im Rahmen des Linkprozesses bereits alle Einträge verwendet wurden, und gegebenenfalls die Relokation beenden. Für jeden Relokationseintrag ist die Ausführung der folgenden Schritte erforderlich:

1. Der Mikrocontroller sendet die Daten des Eintrags an den Host. Wie im vorherigem Vorschlag, in dem sämtliche Daten auf dem Host verbleiben, ist auch hier die Zuordnung der Daten zu einer Datei oder gar einer Position innerhalb einer Datei erforderlich.

2. In Reaktion auf eine Anfrage löst der Host das erforderliche Symbol auf. Er verbreitet das Ergebnis mit Bezug auf ein Modul und eventueller weiter notwendigen Positionsangaben.

Den beschriebenen Ablauf skizziert Abbildung 5.16. Die Zuordnung der Antwort des

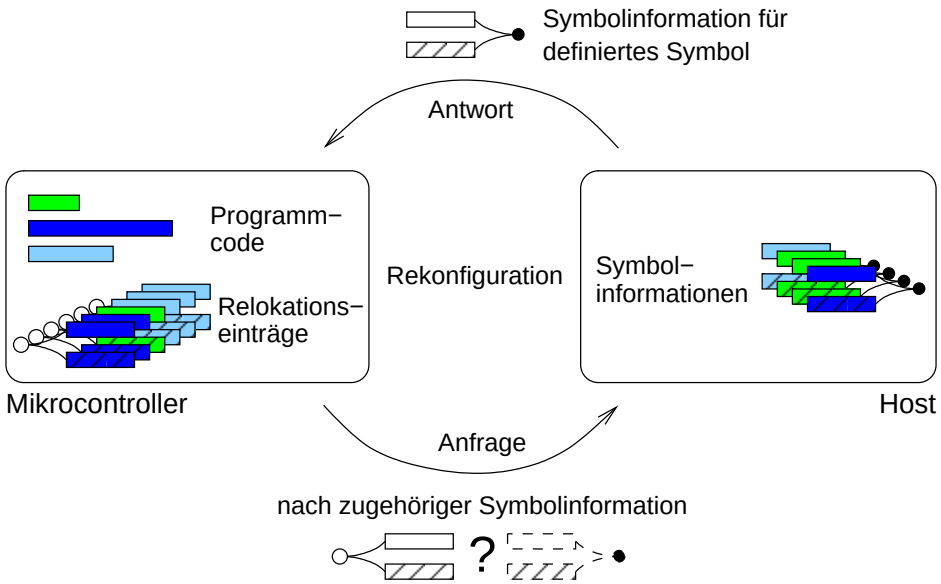


Abbildung 5.16: Linken des Programmcodes bei Speicherung der Relokationseinträge auf dem Mikrocontroller

Hosts kann auch hier auf unterschiedlichem Wege erfolgen:

- Nimmt der Host in seiner Antwort Bezug auf die Anfrage, erfolgt die Zuordnung zu einem Relokationseintrag durch die Antwort selbst. Eine Kennzeichnung der Anfrage durch eine Nummer ist nicht vorteilhaft. Jeder Mikrocontroller kann seine

Relokationseinträge individuell anordnen. Mit gleichen Nummern für ungleiche Relokationseinträge ergibt sich die Gefahr der fehlerhaften Zuordnung der Antwort. Besser ist es, den Inhalt der Anfrage in die Antwort des Hosts aufzunehmen. Alle Mikrocontroller können mit beiden Informationen die entsprechende Relokation durchführen. Somit ist eine Verringerung der Anfragen durch die Berücksichtigung der Ergebnisse fremder Anfragen möglich. Im Gegenzug nimmt der Umfang der übertragenen Daten zu. Auch erhöht sich der Bearbeitungsaufwand für die Mikrocontroller. Um bearbeitete von unbearbeiteten Relokationseinträgen zu unterscheiden, ist eine Verwaltung notwendig. Lokale Einträge müssen gesucht und markiert werden. Auf diesem Wege wird zugleich eine beschränkte Prüfung der vom Host gesendeten Daten möglich. Existiert ein Relokationseintrag aus einer Antwort nicht, ist die gesamte Rekonfiguration in Frage zu stellen.

- Kann der Host den Absender einer Anfrage eindeutig identifizieren und zugleich adressieren, ist eine Zuordnung der Antworten gegeben. Ein Mikrocontroller wartet in diesem Szenario auf eine an ihn gerichtete Antwort auf eine Anfrage. Eine Zuordnung durch den Inhalt der Antwort selbst kann unterbleiben. Damit lässt sich der Umfang der übertragenen Daten reduzieren. Die Identifizierung, Adressierung und implizite Kontrollübergabe ist jedoch nicht im Sinne von Kommunikationsmodellen wie P/S. Zudem kann angenommen werden, dass ein derartiges Vorgehen eine vergleichsweise geringe Leistung aufweist. Da Antworten an andere Mikrocontroller unberücksichtigt bleiben, geht jeder Mikrocontroller selbstständig vor. Als Resultat steigt die Anzahl der Anfragen linear mit der Anzahl der beteiligten Systeme. Die in kurzen Zeitspannen auftretende Menge an Anfragen belastet sowohl den Host als auch das Kommunikationsmedium.

Im Vergleich überwiegen die Vorteile einer Zuordnung durch den Inhalt einer Antwort. Für die Aufbewahrung der Relokationseinträge ist Speicher erforderlich. Über die Struktur der Relokationseinträge kann Einfluss auf den notwendigen Umfang genommen werden. Verweisen die Relokationseinträge lediglich auf einen Eintrag in der Symboltabelle, ist ihr Platzbedarf vom Namen eines Symbols unabhängig. Alle Relokationseinträge besitzen somit eine feste und im Vergleich zu den Symbolinformationen geringe Größe. Dennoch lässt sich die Menge des benötigten Speichers nicht beliebig reduzieren. Für die Anwendbarkeit des Vorgehens ist somit stets eine Berücksichtigung der Ressourcen der Mikrocontroller erforderlich.
Verglichen mit dem vorherigen Vorgehen, fallen die Vorteile einer Speicherung der Relokationseinträge bescheiden aus. Der Umfang der übertragenen Nachrichten nimmt geringfügig ab, da keine Informationen über den Zustand eines Mikrocontrollers versendet werden müssen. Zugleich lassen sich Antworten an andere Mikrocontroller leicht verwenden. Eine Beschleunigung der Rekonfiguration ist daher zu erwarten. Erkauft wird sich dies mit dem Verwaltungsaufwand, den jeder Mikrocontroller für die gespeicherten Relokationseinträge betreiben muss.

In Abschnitt 5.3 wurde das Format von Symbolinformationen erläutert. Unter der Voraussetzung der dort angedeuteten Möglichkeit, anstatt des Symbolnamens einen

Verweis auf eine Zeichekette zu verwenden, ergibt sich eine letzte Variante der Aufgabenverteilung zwischen Host und Mikrocontroller. Sowohl Relokationseinträge als auch Symbolinformationen werden Inhalte des Speichers der Mikrocontroller. Lediglich die Namen der Symbole verbleiben auf dem Host. Motivation für eine derartige Aufteilung ist der Speicherbedarf von Symbolnamen. Wie bereits mehrfach angesprochen, ist dieser im Gegensatz zu anderen Metadaten nicht fest, sondern von mehreren Faktoren abhängig.
Ein naiver Ansatz zur Bearbeitung eines einzelnen Relokationseintrags umfasst die folgenden Schritte:

1. Die im Relokationseintrag referenzierte Symbolinformation wird ermittelt. Ist dieser Information bereits eine Adresse zugeordnet, handelt es sich um ein definiertes Symbol. In diesem Fall kann die Relokation umgehend durchgeführt werden. Andernfalls erfolgt eine Anfrage an den Host mit der Beschreibung der Symbolinformation.

2. Der Host liefert in Reaktion auf die Anfrage den Namen des Symbols. Mit dem Namen versucht der Mikrocontroller, das Symbol aufzulösen.

Da jedoch sämtliche Symbolinformationen lediglich Verweise auf einen Symbolnamen darstellen, ist das Auflösen des Symbols nicht möglich. Zu jedem Symboleintrag muss der entsprechende Name des Symbols bekannt sein. Da im Vorfeld unbekannt ist, welches Symbol erforderlich ist, kommt dies einem kompletten Übertragen aller Namen gleich. Somit ist der Vorteil der Einsparung von Speicher durch die Aufgabenverteilung dahin, das Vorgehen entspräche dem aus Abschnitt 5.3.1.
Zur erfolgreichen Relokation muss der Host statt des Namens eines Symbols die Symbolinformation liefern, in welcher das Symbol definiert ist. Die zu übertragenden Daten setzen sich demnach wie folgt zusammen:

- Definition des Symbols
- Name des Moduls, in dem das Symbol definiert ist

Die Definition bezeichnet die Adresse, welche durch das Symbol ausgedrückt wird. Die in den anderen Vorgehensweisen dieses Abschnittes getätigten Angaben zum Bezug der Adresse gelten analog.
Um die Definition bereitzustellen, muss der Host das angefragte und undefinierte Symbol auf ein definiertes Symbol abbilden. Dementsprechend muss auch der Host noch Symbolinformationen vorhalten. Abbildung 5.17 verdeutlicht die Aufteilung der Metadaten und die beschriebenen Schritte. Mit der Symbolinformation kann jeder Mikrocontroller die Relokation durchführen. Die Ermittlung der referenzierten Adresse ist mit Aufwand verbunden, da die Adresse des bezeichneten Moduls und gegebenenfalls eines Unterabschnitts des Moduls aufgefunden werden muss.
Wie bei den bereits vorgestellten Möglichkeiten der Aufgabenverteilung, ist erneut eine Zuordnung der Antworten des Hosts erforderlich. Die getätigten Aussagen aus dem vorherigen Vorgehen zur Zuordnung der Relokationseinträge lassen sich uneingeschränkt

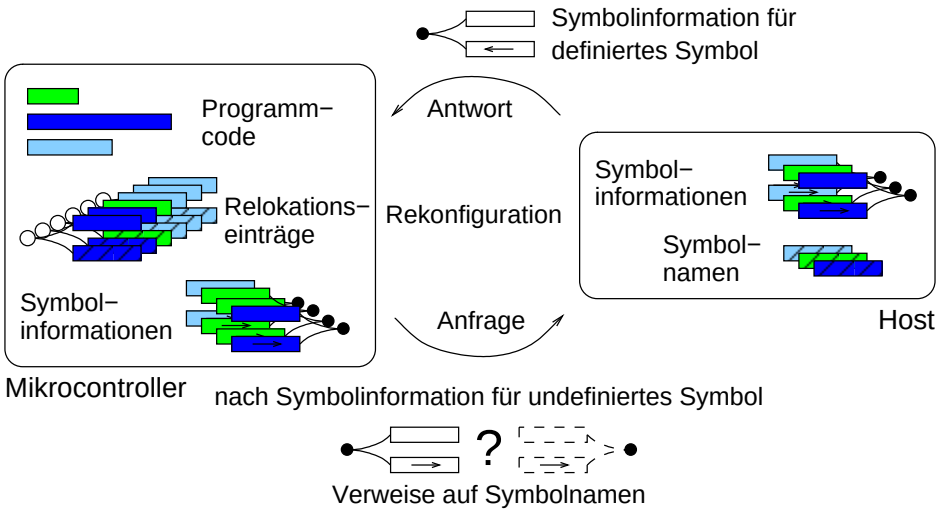


Abbildung 5.17: Linken des Programmcodes bei Speicherung auch der Symbolinformationen auf dem Mikrocontroller

auf die Zuordnung der Symbolinformationen übertragen. Dem Nachteil des weiter erhöhten Speicherbedarfs bei der beschriebenen Vorgehensweise stehen nur wenige Vorteile gegenüber. Der Aufwand für den Host verringert sich geringfügig. Jedoch konnte bisher stets angenommen werden, dass der Host über nahezu unbegrenzte Ressourcen und Möglichkeiten verfügt. Möglich wird die selbstständige Relokation für Einträge, welche direkt auf definierte Symbole verweisen. Dies trifft für Relokationen zu, welche innerhalb eines Moduls verbleiben und keine Adresse aus einem anderen Modul referenzieren. Das Kommunikationsaufkommen mit dem Host verringert sich somit.

5.3.4. Bewertung

Durch die Betrachtung von Modulen lässt sich der Overhead einer Rekonfiguration vermindern. Der Anteil unveränderter Daten, welche trotzdem Bestandteil der Rekonfiguration sind, ist im Vergleich zu den Vorschlägen aus Abschnitt 5.1 und 5.2 gering. Fehler in der Rekonfiguration können durch den unumgänglichen Schritt des Linkens erkannt werden. Mit dem bloßen Verzicht auf einzelne Module ist dennoch keine unbedingte Reduzierung des Kommunikationsaufkommens und Entlastung des Festspeichers verbunden. An Stelle der eingesparten Module treten die Metadaten, welche für das Einbinden der übertragenen Module erforderlich sind. Ihre Verfügbarkeit im Linkprozess führt, mit Blick auf die dazu notwendigen Ressourcen, zu unterschiedlichen Vorgehensweisen.

Das Vorgehen zum autonomen Linken verleiht den Mikrocontrollern weitreichende Freiheiten. Einmal angestoßen, ist die Rekonfiguration eine für jeden Mikrocontroller selbstständig durchführbare Aufgabe. Zugleich werden aber hohe Anforderungen sichtbar. Insbesondere der Speicherbedarf der Metadaten schränkt die Anwendbarkeit des Vorgehens deutlich ein. Ein Linken auf dem Hostsystem kann dessen weitreichenden Ressourcen und Möglichkeiten für die Rekonfiguration nutzbar machen. Problematisch

sind die Annahmen, die über das Speicherlayout der Adressaten einer Rekonfiguration gemacht werden müssen. Die Einbeziehung der Mikrocontroller in den Vorgang des Linkens zur Beseitigung der Annahmen erfordert wiederum neue Annahmen. Diese können wahlweise das Kommunikationssystem oder die Heterogenität der Speicherlayouts betreffen. Eine Trennung zwischen Speicherung und Bearbeitung einzelner Inhalte der Rekonfiguration erscheint viel versprechend. In Abhängigkeit von der Aufteilung sind Host und Mikrocontroller auf unterschiedliche Art und Weise am Prozess des Linkens beteiligt. Wie in Abschnitt 5.3.3 beschrieben, lässt sich eine zunehmende Verschiebung der Verantwortung an die Mikrocontroller realisieren. Die sich davon erhofften Vorteile fallen jedoch gering aus. Mit Anstieg der Verantwortung nimmt auch der Speicherbedarf zu. Im Einzelfall ist somit ein Vergleich von Kosten und Nutzen erforderlich, um eine Entscheidung über die Übertragung von Aufgaben an die Mikrocontroller treffen zu können. Die Aufteilung, welche alle Metadaten auf dem Host belässt, stellt eine solide Möglichkeit dar. Der im Vergleich zu anderen Aufteilungen erhöhte Aufwand zur Kommunikation wird durch die Einfachheit des Vorgehens ausgeglichen.

5.4. Integration von COSMIC

Bereits in Abschnitt 5 wird die Verwendung von COSMIC dargelegt. Wie beschrieben, muss die Middleware unter den gegebenen Bedingungen ein funktionaler Bestandteil einer Anwendung sein. Aus Sicht des Mikrocontrollers stellt die laufende Software die Anwendung dar. Die Software wendet die Möglichkeiten und Ressourcen des Mikrocontrollers an, um ihre Aufgabe zu erfüllen. Bestandteile einer Anwendung, welche COSMIC nutzen, sind ihrerseits Anwender. Anstatt des Mikrocontrollers stellt COSMIC die angewendeten Möglichkeiten und Ressourcen bereit. In diesem Sinne soll der Begriff „Anwendung“, abweichend von seiner Bedeutung in den anderen Abschnitten, nachfolgend einzelne funktionelle Bestandteile einer Software bezeichnen.

Mit der Nutzung von COSMIC bilden die Routinen zur Rekonfiguration eine Anwendung. Wie jede Anwendung der Middleware, muss auch sie sich an das definierte Kommunkationsmodell halten. Wie schon in Abschnitt 2.2 erläutert, basiert die Kommunikation auf der Verbreitung von Ereignissen. Verschiedenartige Ereignisse werden durch die Zuordnung zu Ereigniskanälen unterschiedlich betrachtet. Informationen werden demnach in Form von Ereignissen über Ereigniskanäle ausgetauscht. Während Ereignisse frei in Ereigniskanäle publiziert werden können, ist für den Empfang eine Registrierung erforderlich. Mit der Registrierung für einzelne Arten von Informationen wird eine bedarfsorientierte Kommunikation ermöglicht.

Ein Teil der in den vorigen Abschnitten 5.1, 5.2 und 5.3 vorgeschlagenen Vorgehensweisen verlangt eine bidirektionale Kommunikation. Nicht allein der Host überträgt Informationen. Auch die angesprochenen Mikrocontroller versenden Informationen wie beispielsweise Adressen oder Anfragen. Für eine derartige Kommunikation mit COSMIC existieren zwei Möglichkeiten:

- Sämtliche Informationen werden in gleich gekennzeichneten Ereignissen verbreitet. Die Anwendung selbst identifiziert die Informationen bei Empfang und entscheidet so über die nachfolgende Nutzung und Reaktion.

- Für alle Informationen erfolgt eine Klassifizierung. Entsprechend werden unterschiedliche Informationen in verschieden gekennzeichneten Ereignissen übertragen. Durch die individuelle Reaktion auf Ereignisse ist eine Identifizierung gegeben.

Die zweite Möglichkeit folgt dem Gedanken des P/S. Die Anwendung wird von der Aufgabe der Identifikation entlastet und benutzt statt dessen eine Dienstleistung der Middleware. Aufgrund dieses Vorteils bietet sich die Verwendung mehrerer Arten von Ereignissen für die Rekonfiguration an. Die vom Host zu übertragenden verschiedenen Informationen sind dafür zu kategorisieren und disjunkten Bezeichnern zuzuordnen. Gleiches gilt für die Mikrocontroller.

Für die verschiedenen Vorschläge aus den vorigen Abschnitten lässt sich die folgende Einteilung vornehmen:

- Ohne zusätzliche Maßnahmen kann in den Vorschlägen aus den Abschnitten 5.1 und 5.2 eine einzige Art von Ereignis Verwendung finden. Der Host überträgt ausschließlich Ereignisse, welche den neuen Programmcode zum Inhalt haben.

- Die modulare Rekonfiguration aus Abschnitt 5.3 erfordert verschiedene Ereignisse. Auch hier ist zunächst der neue Programmcode als Ereignis einzuordnen. Die Anfragen der Mikrocontroller stellen eine weitere Klasse von Ereignissen dar. Zuletzt bilden die Antworten des Hosts eine eigene Kategorie von Ereignissen.

Die Einteilung gilt unter der Voraussetzung, dass die zu verbreitenden Daten für alle Mikrocontroller gleich sind. Sie lässt damit die inhärente Heterogenität eines verteilten Systems unberücksichtigt. Auch für diesen Aspekt bietet die Klassifizierung von Informationen Vorteile:

- Spezielle Details von Verfahren können unterschieden werden. Ein Beispiel sind die verschiedenen Varianten der modularen Ersetzung aus Abschnitt 5.3. Eine Identifizierung der übertragenen Metadaten lässt sich ebenso durch eine Klassifizierung realisieren wie auch die Unterscheidung der verschiedenen Anfragen und Antworten.

- Ein Vorhalten mehrerer Verfahren der Rekonfiguration und ein Wechsel ist möglich. Damit kann die Möglichkeit zur vollständigen Aktualisierung für Ausnahmefälle bereit gehalten werden, gleichzeitig aber für den Regelfall auf eine feinere Rekonfiguration gesetzt werden. Auch ist es denkbar, für besondere Zustände, wie einem Detektieren von Fehlern, auf eine Ausweichmöglichkeit im Laufe einer Rekonfiguration zu wechseln.

- Mit verschiedenen Bezeichnern ist eine Unterteilung der erreichbaren Mikrocontroller möglich. Rekonfigurationen lassen sich so auf eine Teilmenge beschränken.

Auf diese Weise kann auf die Spezifika der Hard- und Software der beteiligten Systeme reagiert werden.

Verglichen mit Sensordaten, erfordert das Verbreiten von Programmcode die Behandlung großer Datenmengen. Ihre Übertragung ist von COSMIC unter Berücksichtigung des verwendeten Kommunikationsmediums sicherzustellen. Von Nachteil für die Zusammenarbeit von COSMIC mit Varianten aus Abschnitt 5.3 ist die nicht vorhandene Identifizierung von Sender und Empfänger. Eine Zuordnung muss somit über den Inhalt eines Ereignisses selbst erfolgen. Bereits beschrieben wurde jedoch, dass auf eine derartige 1:1-Beziehung zwischen Host und Mikrocontroller verzichtet werden kann.

Die zu tätigenden Aufgaben der Routinen zur Rekonfiguration beschränken sich zusammenfassend auf die Kommunikation in Form von Ereignissen. Über die Verwendung entsprechender Ereigniskanäle lassen sich an Ereignissen gebundene Informationen übertragen und empfangen. Neben der Entlastung von Details der Kommunikation unterstützt COSMIC die Rekonfiguration damit auch bei der Identifizierung der Informationen.

6. Entwurf

Für den Entwurf werden einzelne Konzepte aus Abschnitt 5 ausgewählt. Die Anzahl an möglichen Varianten verlangt eine Beschränkung auf allgemein anwendbare und zugleich vorteilhafte Vertreter. Eine Unterteilung der Konzepte ist mit der Gliederung des Abschnitts 5 bereits vorgegeben. Anhand der jeweils vorhandenen Bewertung wird die Auswahl möglich:

- Das Konzept zum Ersetzen der gesamten Anwendung aus Abschnitt 5.1 bildet die Grundlage der Rekonfiguration. Allein mit ihm können auch die Routinen der Rekonfiguration auf einen neuen Stand gebracht werden. Das Vorgehen ist unkompliziert und verzichtet auf Details. Dem Prinzip der Einfachheit entspricht auch die Verwendung eines Zwischenspeichers.

- Durch das Konzept zur Rekonfiguration der anwendungsspezifischen Bestandteile aus Abschnitt 5.2 wird eine Reduzierung der zu betrachtenden Daten erreicht. Das Vorgehen bietet einen guten Kompromiss zwischen Aufwand und Nutzen. Die Änderungen im Vergleich zum Konzept der vollständigen Ersetzung beschränken sich auf den Entwicklungsprozess und damit auf den Host. Für die Mikrocontroller kann der Entwurf des vollständigen Ersetzens übernommen werden.

- Eine Rekonfiguration auf Ebene der Module ist, wie in Abschnitt 5.3 beschrieben, mit vielen Hindernissen verbunden. Die in Abschnitt 5.3.3 als erstes beschriebene Aufteilung, in welcher die Metadaten auf dem Host verbleiben, bietet das beste Verhältnis zwischen aufgewendeten Ressourcen und erreichbarem Gewinn. Dennoch bleibt der Nutzen des im Vergleich zu den anderen Konzepten höheren Aufwands ungewiss. Daher wird auf einen ausführlichen Entwurf der modulbasierten Rekonfiguration verzichtet. Einzelne Details der Rekonfiguration anwendungsspezifischer Bestandteile können jedoch auch für die Rekonfiguration einzelner Module genutzt werden.

Die Ergebnisse des Entwurfs sollen als Implementierungsvorlage dienen. Entsprechend sind zunächst die gegebenen Anforderungen und Bedingungen zu präzisieren. Anschließend kann die Gestaltung der Details erfolgen.

6.1. Entwicklungsumgebung

Die vorgegebenen Anforderungen bilden die Grundlage des Entwurfs. Unterschieden werden kann zwischen Hard- und Software-Anforderungen. Bereits in Abschnitt 2 wird

mit der Beschreibung von Mikrocontrollern auf die Anforderungen der Hardware eingegangen. Für eine Implementationsvorgabe, wie sie das Ziel des Entwurfs ist, sind die dort getroffenen Aussagen zu allgemein und bedürfen weiterer Konkretisierung. Hinzu kommen die bisher nicht genannten, durch den Host vorgegebenen Anforderungen. Die hier stets ausreichenden Ressourcen lassen die Anforderungen der Software in den Vordergrund treten. Die Anforderungen der Verbindung zwischen Host und Mikrocontroller umfassen sowohl die Hardware als auch die zu deren Verwendung benötigte Software.

6.1.1. Mikrocontroller

Schon in der in Abschnitt 1 gegebenen Einleitung und Motivation wird die Heterogenität eines aus Mikrocontrollern bestehenden verteilten Systems beschrieben. Wie in Abschnitt 2.2 aufgeführt, macht COSMIC die Kommunikation für Anwendungen transparent. Im Gegenzug ist eine Anpassung von COSMIC an die Details der verschiedenen Kommunikationsschnittstellen erforderlich. Darüber hinaus muss der jeweilige Mikroprozessor berücksichtigt werden. Die individuelle Anbindung der Speicher ist für die Routinen der Rekonfiguration von Interesse. Somit sind für den Entwurf einzelne Details aller Komponenten eines Mikrocontrollers zu beachten.

Die Berücksichtigung der Spezifika kann in Form von verschiedenen Konfigurationen erfolgen [9]. Die Basis für die notwendige Grundarchitektur soll der Entwurf für Mikrocontroller des Typs AT90CAN128 der Firma Atmel bilden [2]. In der Einteilung der Mikrocontroller aus Abschnitt 2.1 lässt sich der AT90CAN128 den eigenständigen 8-Bit-Mikrocontrollern zuordnen. Seine technischen Daten fasst Tabelle 6.1 zusammen. Die

Merkmal	Wert
Taktfrequenz	maximal 16MHz, skalierbar
flüchtiger Speicher	4 Kilobyte SRAM
nicht flüchtiger Speicher	4 Kilobyte EEPROM, 128 Kilobyte Flash
Kommunikationsschnittstellen	USART SPI, CAN, TWI bzw. I^2C

Tabelle 6.1: Technische Daten der AT90CAN128 Mikrocontroller

Auswahl der einzelnen Merkmale entspricht denen aus Abschnitt 2.1. Deutlich wird, dass der AT90CAN128 bereits umfangreiche Möglichkeiten bietet. Für die Rekonfiguration von Bedeutung sind seine Fähigkeiten zum ISP und IAP. Für letztgenannte Fähigkeit kann jede Kommunikation zum Einsatz kommen, sei es über einen der beiden *Universal Synchronous Asynchronous Receiver Transmitter* (USART), über das *Serial Peripheral*

Interface (SPI), via CAN oder über *Two Wire Interface* (TWI) bzw. *Inter-Integrated Circuit* (I^2C).

Wie alle Mitglieder der AVR-Familie, weist auch der AT90CAN128 einen getrennten Adressraum für Befehle und Daten auf. Damit folgt er der in Abschnitt 2.1 erläuterten Harvard-Architektur. Der SRAM bildet den gebräuchlichen Speicher für variable Operanden von Instruktionen. Als Befehlsspeicher dient der vergleichsweise umfangreiche Flash. Für die Rekonfiguration ist ein Beschreiben des Flash mit Hilfe von Instruktionen aus dem Flash notwendig. Abschnitt 5 führt die damit verbundenen Probleme auf. Die Entwickler des Mikrocontrollers entschieden sich für eine Beschränkung des Schreibens in den Befehlsspeicher. Lediglich Befehlen, welche innerhalb eines definierten Adressbereichs liegen, ist das Schreiben des Flash möglich. Entsprechend der Beschreibung aus Abschnitt 3.1, ist der Adressbereich mit dem Namen „*Boot Loader Flash Section*“ (BLS) bezeichnet. Wie Abbildung 6.1 zeigt, befindet sich die BLS am Ende des Flash. Ihre

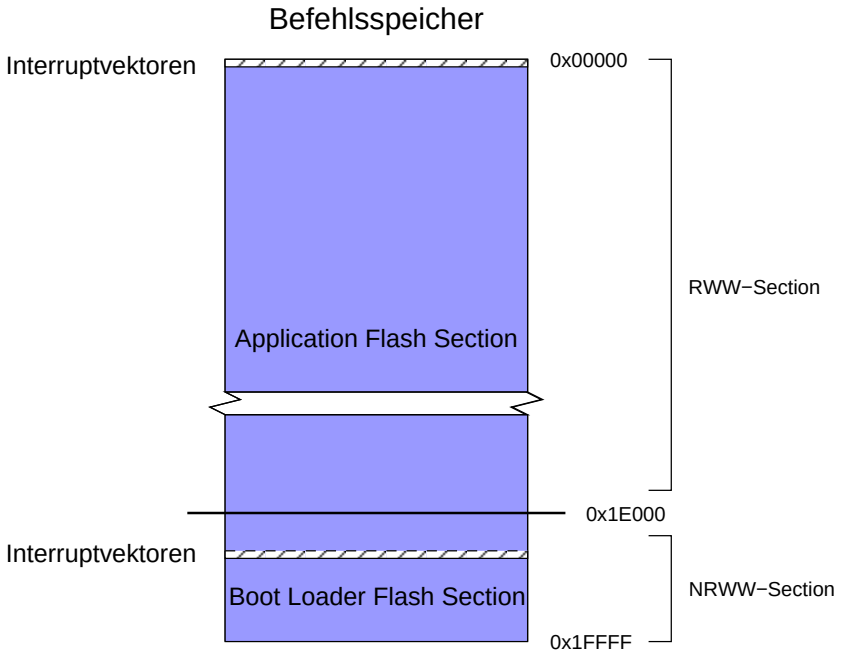


Abbildung 6.1: Unterteilung des Flash

Größe kann im Rahmen eines ISP-Prozesses in vier Stufen festgelegt werden. Die BLS kann maximal 8 Kilobyte umfassen. Für diesen Fall füllt sie den *Non-Read While Write Section* (NRWW Section) genannten Abschnitt des Flash vollständig aus. Schreibvorgänge in der NRWW Section führen zu einem Blockieren des Mikroprozessors. Erst nach Abschluss der Schreiboperation kann wieder auf den Flash zugegriffen werden. Im Gegensatz dazu steht die *Read While Write Section* (RWW Section), welche den überwiegenden Teil des Befehlsspeichers ausmacht. Schon vor Beendigung von Schreiboperationen in der RWW Section kann der Mikroprozessor weitere Instruktionen auf dem Befehlsspeicher anfordern, dekodieren und ausführen. Die Instruktionen müssen jedoch in der NRWW Section vorliegen.

Die Beschränkungen zum Lesen des Befehlsspeichers bei Schreibvorgängen betreffen insbesondere die Behandlung von Interrupts. Die Adressen von Interruptvektoren befinden sich standardmäßig am Beginn des Befehlsspeichers. Aus Abbildung 6.1 wird deutlich, dass sie sich somit in der RWW Section befinden. Durch den asynchronen Charakter von Interrupts kann demnach das Lesen der Interruptvektoren mit Schreibvorgängen in die RWW Section kollidieren. Eine nahe liegende, aber auch sehr grobe Lösung ist die Unterbindung der Interruptbehandlung durch den Mikroprozessor. Eine weitere Möglichkeit bietet der zweite Speicherbereich für Interruptvektoren in der BLS. Die BLS befindet sich in jedem Fall innerhalb der NRWW Section. Da sie somit auch während des Schreibens in die RWW Section lesbar bleibt, steht einer Behandlung von Interrupts durch den Mikroprozessor nichts im Wege. Alle Instruktionen, welche im Verlauf einer Interruptbehandlung verwendet werden, müssen entsprechend ebenfalls in der NRWW Section vorgefunden werden können. Die Entscheidung, welche der beiden Speicherbereiche die Interruptvektoren beinhalten, kann durch die Anwendung getroffen und variiert werden. Das Schreiben in die NRWW Section führt, wie erläutert, zu einem Stopp des Mikroprozessors für die Dauer der Operation. Eine Behandlung von Interrupts kann somit nicht erfolgen. Dementsprechend erübrigt sich die Problematik unerlaubter Speicherzugriffe.

6.1.2. Host

Die Erläuterungen aus Abschnitt 3.1 machen die besondere Position des Hosts im Verlauf einer Rekonfiguration deutlich. Fast ausnahmslos setzen die in Abschnitt 4 beschriebenen, bereits existierenden Verfahren auf eine strikte Trennung von Host und Zielsystem. Grundlegende Annahme für diesen Ansatz ist, das die Rekonfiguration im Regelfall einen nicht alltäglichen Vorgang darstellt. Daher können für den Host andere Anforderungen als für die Zielsysteme geltend gemacht werden. Der Ansatz einer Trennung von Host und Zielsystemen soll auch im Rahmen der vorliegenden Arbeit verfolgt werden.

Indem der Host nicht den Beschränkungen der Zielsysteme unterliegt, kann er mit dem Rechnersystem zur Softwareentwicklung gleichgesetzt werden. Die entsprechende Betrachtungsweise aus Abschnitt 5.3.2 wird somit fortgesetzt. Zum Einsatz kommt ein handelsüblicher Personalcomputer. Als Betriebssystem findet Linux Verwendung. Für die Entwicklung von Software für den im vorigem Abschnitt 6.1.1 genannten Mikrocontroller steht mit *avr-gcc* ein Compiler zur Verfügung (Version 4.1.1). Dieser ist Teil der GCC. Mit dem Linker *avr-ld*, dem Assembler *avr-as* sowie weiteren Tools ist die Unterstützung durch die binutils gegeben. In Zusammenarbeit der Werkzeuge kann der Übersetzungsvorgang gemäß Abbildung 3.1 durchgeführt werden. Beide Komponenten, sowohl die GCC als auch die binutils, sind Standardkomponenten in der Softwareentwicklung unter Linux. Die von ihnen für den Mikrocontroller bereitgestellten Werkzeuge sind lediglich angepasste Varianten bei ansonsten gleicher Verwendung. Die aus der Entwicklung unter Linux bekannten Vorgehensweisen können daher unverändert angewandt werden.

Mit dem verfügbaren Compiler sind Vorgaben zur Programmiersprache verbunden. Vom Übersetzer unterstützt werden C sowie C++. Die Verwendung von Assembler als

Zwischencode erlaubt entsprechende Konstrukte auch in dieser Sprache. Somit besteht die Möglichkeit zur imperativen Programmierung. Bei Verwendung von C++ kann zusätzlich objektorientiert programmiert werden. Zur Unterstützung der Softwareentwicklung existiert mit der *avr-libc* eine Adaption der C-Standardbibliothek [10]. Sie beinhaltet neben den Standardroutinen verschiedene Konstrukte für Mikrocontroller der AVR-Familie. Von alltäglicher Bedeutung sind die Definitionen von Interrupt Service Routinen und die Zugriffe auf Register der Mikroprozessoren. Das Einbinden von Assemblercode ist ebenfalls über eigene Konstrukte realisiert.

Für die Übertragung von Programmcode ist die Programmiersoftware *avrdude* (Version 5.3) auf dem Host installiert [11]. Sie akzeptiert die in Abschnitt 3.1 angesprochenen Standardformate. Avrdude unterstützt verschiedene Möglichkeiten der Kommunikation zwischen Host und Mikrocontroller. Die Kommunikation mit verdrahteten Algorithmen im ursprünglichen Sinne des ISP ist ebenso vorgesehen wie die Kommunikation mit verschiedenen Ausführungen von Bootloadern. Im Entwicklungsprozess zum Einsatz kommt die Kommunikation mit Algorithmen, welche durch die Hardware eines Mikrocontrollers bereitgestellt werden. Diese sehen eine Anbindung an ein Bussystem nach SPI vor. Die notwendige Verbindung des Hosts mit den dafür vorgesehenen Pins des Mikrocontrollers erfolgt über einen Adapter.

6.1.3. Verbindung von Host und Mikrocontroller

Die Rekonfiguration mit Hilfe von Mechanismen, welche die Hardware bereitstellt, bildet die Grundlage für das IAP. Erst nachdem eine Anwendung auf die Mikrocontroller übertragen wurde, stehen deren Algorithmen für nachfolgende Rekonfigurationen zur Verfügung. Das Verhalten der Rekonfiguration kann komplett geändert werden, indem man die Anwendung austauscht.

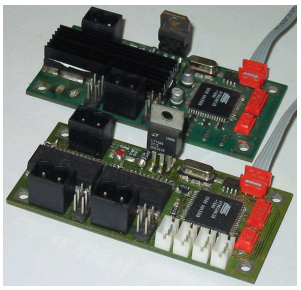
Die im vorigen Abschnitt genannten Verbindungen zwischen Host und einem Mikrocontroller entsprechen klassischen Verbindungen des ISP. Gemäß Abschnitt 3.2 sind diese Verbindungen für die Rekonfiguration in verteilten Systemen ungeeignet. Eine alternative Möglichkeit existiert in Form einer Verbindung über CAN. Als Bussystem konzipiert, erlaubt CAN den gleichzeitigen Anschluss mehrerer Teilnehmer. Die zur Rekonfiguration notwendige Verbindung zwischen Host und Mikrocontroller erfordert auf diese Weise keinen zusätzlichen Aufwand. Sämtliche Mikrocontroller sind über den Bus für den Host erreichbar. Im Gegensatz zu anderen Bussystemen, basiert die Kommunikation im CAN jedoch nicht auf der Adressierung eines Teilnehmers. Statt dessen werden Nachrichten gekennzeichnet. Eine auf dem CAN-Bus versandte Nachricht erreicht entweder alle oder keinen Teilnehmer. Anhand der Kennzeichnung einer Nachricht kann sich jeder Teilnehmer für eine weitere Bearbeitung entscheiden. Die Kennzeichnung einer Nachricht ist in diesem Sinne keine Adressierung, sondern eine Beschreibung des Inhalts. Die aktuelle CAN-Spezifikation sieht eine unterschiedliche Anzahl von Bits für einen Bezeichner (*Identifier*) vor:

- CAN2.0A spezifiziert einen *Standard-Identifier* mit 11 Bit Länge.

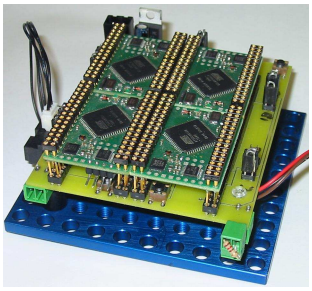
- CAN2.0B erweitert den Identifier auf 29 Bit. Entsprechend wird vom *Extended-Identifier* gesprochen.

Der Inhalt der Nachricht besteht aus maximal acht Byte Daten. Es kann auch auf die Nutzlast verzichtet werden. Insgesamt setzt sich eine CAN-Nachricht somit aus den Bits des Identifiers, den maximal 64 Bits der übertragenden Daten sowie aus weiteren Bits zur Steuerung der Kommunikation zusammen. Die Arbitrierung, dass heißt die Steuerung des Zugriffs auf das Kommunikationsmedium, erfolgt durch eine Priorisierung der Identifier. Niedrige Identifier haben Vorrang. Indem der Sender einer Nachricht den Zustand des Mediums überwacht, kann er selbstständig Kollisionen auf dem Medium auflösen. Verbreitet ein Teilnehmer einen Identifier, liegt aber zeitgleich auf dem Bus ein niedrigerer Identifier an, wird die Übertragung der Nachricht zunächst zurückgestellt. Entsprechend dieser Merkmale, lässt sich die Arbitrierung in CAN mit *Carrier Sense Multiple Access/Collision Detection and Arbitration on Message Priority* (CSMA/CD+AMP) bezeichnen.

Die Kommunikationsschnittstelle von CAN zeigt die Einbindung eines Mikrocontrollers in seine Umgebung. Die der externen Kommunikation dienenden Pins sind nicht kompatibel zu den Steckern und Buchsen der entsprechenden physikalischen Verbindung. Die jeweiligen Anschlüsse müssen daher herausgeführt sein. Hinzu kommen unterschiedliche Signalpegel, welche den Einsatz von weiteren elektronischen Bausteinen notwendig machen. In Abschnitt 2.1 ist die Einbettung eines Mikrocontrollers in ein Board beschrieben. Üblicherweise sind die Anschlüsse für die externe Kommunikation auf dem Board befestigt. Sie sind mit entsprechenden Pins des Mikrocontrollers verbunden, gegebenenfalls über zusätzliche Bausteine. Abbildung 6.2 veranschaulicht die zwei während der Entwicklung verfügbaren Boards. Bei den in Abbildung 6.2a dargestellten



(a) Zwei Controllerboards „mega128CAN“



(b) Clusterboard mit vier „Crumb128-CAN“

Abbildung 6.2: Für die Entwicklung zur Verfügung stehende Boards

Boards handelt es sich um von der Firma KTB mechatronics [26] vertriebene Produkte. Abbildung 6.2b zeigt vier Boards der Firma chip45 [6], welche auf einer eigens entwickelten Platine zusammengefasst sind. Für die Kommunikation der einzelnen Boards nach außen sowie untereinander stellt die Platine Verbindungen über CAN bereit. Durch die

Bündelung mehrerer Boards bzw. Mikrocontroller auf einer Platine verringert sich der Aufwand zur Verkabelung und Behandlung einzelner Bauteile.

Die Anbindung des Hosts an einen gemeinsamen Bus vervollständigt die zur Rekonfiguration notwendige Verbindung. Wie im vorigem Abschnitt 6.1.2 beschrieben, ist der Host ein handelsüblicher Personalcomputer. Eine Kommunikationsschnittstelle für CAN ist standardmäßig nicht vorgesehen. Die entsprechend notwendige Erweiterung des Hosts ist über Produkte der Firma PEAK-System Technik [31] realisiert. Vielfach vorrätig, leicht austauschbar und daher üblicherweise zum Einsatz kommt ein Dongle für den Parallelport. Durch vom Hersteller angebotene Treiber ist eine Unterstützung für das auf dem Host laufende Betriebssystem Linux gegeben. Zusammen mit den Treibern ist eine mit *libpcan* bezeichnete Bibliothek verfügbar (Treiber-Version 5.12). Mit Hilfe der Bibliothek können Anwendungen die erweiterten Kommunikationsmöglichkeiten des Hosts nutzen. Dabei bleibt die jeweils für die Erweiterung verwendete Hardware, ob beispielsweise Dongle, Steckkarte oder USB-Anschluss, für die Anwendung transparent.

6.2. Anpassung des Entwurfs von COSMIC

Bereits in [21] sind, neben der Vorstellung des Konzepts, Vorschläge zum Entwurf von COSMIC unterbreitet. Insgesamt wird eine aus mehreren Ebenen bestehende Architektur vorgeschlagen:

- Die unterste Ebene vollzieht die Anbindung an die Kommunikationsschnittstelle.
- Mit der nächsten Schicht wird die Abstraktion von der Hardware durchgeführt. Entsprechend ist diese Ebene als *Abstract Network Layer* bezeichnet.
- Die dritte Ebene bietet die Schnittstelle für die Anwendungen. Erst hier erfolgt eine Übersetzung von abstrakten Daten zu Ereignissen und ihre Zuordnung zu Ereigniskanälen. Die Benennung *Event Layer* entspricht den Aufgaben der Ebene.
- Als oberste Ebene lassen sich die Anwendungen charakterisieren.

Die Architektur soll die Grundlage für bereits vorliegende Implementationen von COSMIC bilden. Aufgrund von unübersehbaren Differenzen zur Implementation, fehlender Funktionalität und eines Fehlers ist eine Modifikation und Konkretisierung der Architektur erforderlich. Erst mit einem hinreichenden Entwurf samt nachfolgender Implementation kann COSMIC die Grundlage für die Rekonfiguration darstellen.

6.2.1. Korrektur der Abbildung von Ereignissen

Indem eine Anwendung Daten in einen Ereigniskanal schreibt, produziert sie Ereignisse. Jedes Ereignis wird durch ein *Subject* klassifiziert. Ein Subject umfasst 64 Bit.

Nicht jedes Übertragungsprotokoll ist in der Lage, mit einer derartigen Wortbreite umzugehen. Als Beispiel sei CAN genannt, welches maximal 64 Bit Nutzlast und maximal 29 Bit an Metainformationen vorsieht. Für die Verbreitung von Ereignissen ist

entsprechend eine Berücksichtigung der Eigenheiten der Übertragung notwendig. Die Unabhängigkeit vom Übertragungsweg muss jedoch gewahrt bleiben. Daher ist eine Zuordnung der Subjects auf ein für die Übertragung geeignetes Gegenstück durchzuführen. Genau dies ist Aufgabe des *Binding Protocols*. Allein der Abstract Network Layer besitzt Informationen über den verwendeten Übertragungsweg. Somit muss es Aufgabe des Abstract Network Layers sein, das Binding Protocol zu realisieren. In diesem Punkt ist eine Korrektur der in [21] vorgeschlagenen Architektur erforderlich. Bisher ist das Binding Protocol dem Event Layer zugeordnet, dessen Aufgaben jedoch unabhängig vom Übertragungsweg sind. Abbildung 6.3 zeigt das überarbeitete Schichtenmodell. Auf die Berücksichtigung von qualitativen Anforderungen, im Speziellen zur zeitlichen

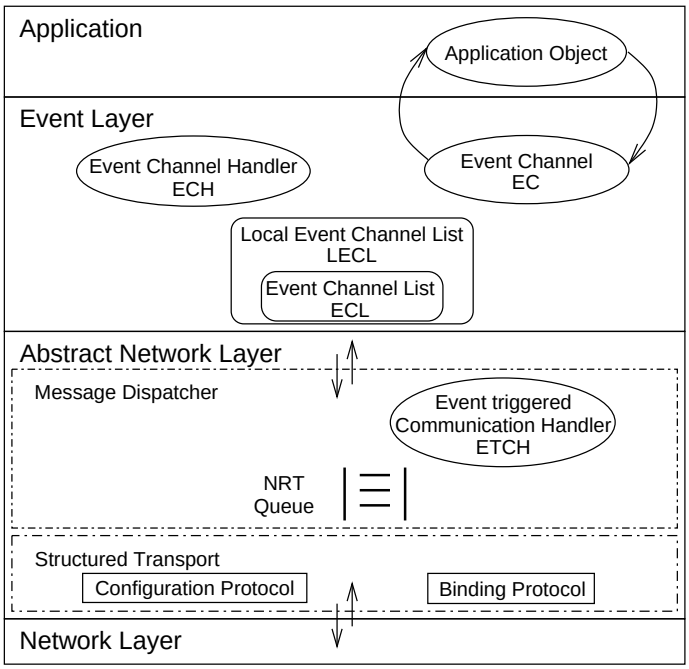


Abbildung 6.3: Schichtenmodell von COSMIC (in Modifikation zu [21])

Vorhersehbarkeit, kann in der vorliegenden Arbeit verzichtet werden. Innerhalb welcher Zeitschranken die Ereignisse und damit die Daten der Rekonfiguration verbreitet werden, ist unerheblich.

6.2.2. Konkretisierung des Entwurfs

Die bereits vorliegende Implementierung von COSMIC berücksichtigt nur in geringem Maße das in Abschnitt 6.2 angesprochene Schichtenmodell. Eine Trennung von Event Layer und Abstract Network Layer findet nicht statt. Aufbauend auf der Kommunikation über einen CAN-Bus, setzen sich entsprechende Konstrukte bis in die Ereigniskanäle fort. Ein Beispiel ist das Binding Protocol, welches durch die direkte Bindung eines Subjects sowie dessen Abbildung auf CAN (*Etag*) an einen Ereigniskanal realisiert ist. Damit ist eine Unabhängigkeit vom Kommunikationsprotokoll nicht vorhanden.

Hingegen ist die Unabhängigkeit vom Kommunikationsprotokoll ein Schwerpunkt der vorliegenden Arbeit. Die vorliegende Implementation folgt, wie beschrieben, nur in Grundzügen und nur in Ansätzen dem Entwurf des Schichtenmodells. Ein darüber hinaus gehender, deutlich detaillierterer Entwurf ist nicht bekannt. Für die Beseitigung der Abhängigkeit von der Kommunikation steht somit kein konkreter Entwurf zur Verfügung. Ein solcher kann zwar aus der Implementation erarbeitet werden. Auf diese Weise bleibt jedoch eine zu keinem Modell konforme, nach einem Einblick zudem mäßig dokumentierte Implementation im Entwurf berücksichtigt. Existierende, aber nicht dokumentierte Bedingungen erschweren nachfolgende Erweiterungen und das Auffinden von Fehlern. Eine derartige Grundlage für die vorliegende Arbeit wird als fragil eingeschätzt. Daher fällt die Entscheidung für eine komplett eigene Konkretisierung des Schichtenmodells. Somit kann der Entwurf frei und vollkommen unvoreingenommen getätigt werden. Die Kenntnis der Implementation erleichtert im Zusammenspiel mit der Dokumentation die Korrektur etwaiger Fehler. Eigene Erweiterungen lassen sich einfach tätigen. Der Aufwand zur Strukturierung einer als unzureichend eingestuften Software kann unterbleiben. Jedoch geht dies mit dem Verlust nahezu sämtliche Vorarbeiten einher. Ein entsprechend hoher Aufwand an Ressourcen ist in der nachfolgenden Implementation zu erwarten.

Dem Entwurf liegen folgende Gedanken zugrunde:

- Potentiell können mehrere auf einem System laufende Anwendungen gleiche Ereignisse verbreiten oder an gleichen Ereignissen interessiert sein. Eine Identifizierung der Anwendungen durch die Middleware ist nicht möglich. So ist eine Zuordnung von Operationen nicht gegeben. Signalisiert eine Anwendung ein Desinteresse am Empfang weiterer Ereignisse, kann die Middleware den entsprechenden und zu entfernenden Handler nicht ermitteln. Daher werden Ereigniskanäle als Stellvertreter verwendet. Der Event Layer gibt die Stellvertreter heraus. Mit Ihnen kann er eine Zuordnung der Operationen durchführen.

- Die Ereigniskanäle bieten die bereits in [21] definierten Schnittstellen. Als Stellvertreter können mehrere Kanäle das gleiche Subject verfolgen. Aufgabe des *Event Channel Handlers* (ECH) ist es, eine Zuordnung zu den Subjects durchzuführen. Innerhalb der *Local Event Channel List* (LECL) sind dazu entsprechende Ereigniskanäle unter dem jeweils gleichen Subject zusammengefasst. Weiterhin hängt die Bearbeitung eines Ereigniskanals von dessen Zweck ab. Ein Kanal, der für den Empfang von Ereignissen vorgesehen ist, besitzt und verwendet zusätzliche Informationen zur Benachrichtigung der Anwendung. Ein Kanal, der lediglich für die Verbreitung von Ereignissen herausgegeben wurde, ist einfacher zu behandeln. Daher beinhaltet die LECL für jedes Subject zwei Listen, die *Event Channel Lists* (ECL). Jede der Listen enthält Verweise auf registrierte Ereigniskanäle für den Empfang bzw. den Versand. Ein Kanal für die Verbreitung unter einem Subject steht somit in der LECL im gleichen Eintrag wie ein Kanal für den Empfang entsprechender Ereignisse.

- Der Entwurf des Abstract Network Layers folgt den Vorschlägen aus [21]. Der zur Anbindung an den Event Layer dienende *Message Dispatcher* kann unabhängig von der Kommunikation entworfen werden. Die Verbindung mit dem jeweiligen Kommunikationsprotokoll erfolgt im unteren Teil, dem *Structured Transport* genannten Sublayer. Gemäß der Entwicklungsumgebung aus Abschnitt 6.1.3, erfolgt ein Entwurf des Sublayers zur Kommunikation über CAN. Vom Message Dispatcher übergebene Strukturen werden in CAN-spezifische Strukturen abgebildet und umgekehrt.
  Das Binding Protocol ist, wie in Abschnitt 6.2.1 beschrieben, abhängig vom Kommunikationsprotokoll. Entsprechend ist es ein Bestandteil des Structured Transport Sublayers. Das Verfahren des Binding Protocols für CAN ist in [22] beschrieben, lässt sich aber auch aus der bereits vorliegenden Implementation ableiten. Aufgabe des Binding Protocols ist das Erarbeiten sowie die Verwaltung und Realisierung von Abbildungen zwischen Subjects und Etags. Der Entwurf folgt daher allgemeinen Strukturen zur gegenseitigen Abbildung von Werten.

Das Schichtenmodell aus Abschnitt 6.2 legt eine lose Kopplung zwischen einzelnen Ebenen nahe. Einfach zu realisieren ist eine handlerbasierte Verbindung. Übergeordnete Komponenten initialisieren ihre Bestandteile und registrieren sich bei diesen. Wie auch im vorherigen Abschnitt 6.2.1, bleiben qualitative Anforderungen an die Kommunikation unberücksichtigt. Ein Aspekt des Entwurfs ist aber das Bestreben, die Strukturen allgemein zu halten. Daher wird erwartet, dass sich die genannten Anforderungen im Bedarfsfall integrieren lassen.

6.2.3. Fragmentierung von Ereignissen

Nur unter Einhaltung protokollspezifischer Bedingungen kann davon ausgegangen werden, dass ein Ereignis in einem Stück über eine Kommunikationsschnittstelle übertragen wird. Ein Beispiel zeigt der bisherige Einsatz von COSMIC zur Verbreitung von Sensordaten. Sowohl die Bedingungen als auch der Speicherbedarf von Ereignissen sind aufeinander abgestimmt. Im Allgemeinen sind lediglich wenige Bytes für die Beschreibung von Sensordaten notwendig. Diese lassen sich bequem in einer Nachricht über CAN versenden.

Die Rekonfiguration erfordert die Übertragung umfangreicher Datenmengen. Entsprechend umfangreiche Ereignisse sind zu verbreiten. Bedingt durch die Anzahl der Daten, können protokollspezifische Bedingungen nicht berücksichtigt werden. Für die Übertragung größerer Datenmengen speziell über CAN existiert bereits ein Entwurf und eine entsprechende, prototypische Implementation [5]. Neben dem einfachen Kopieren von Daten ist auch das Streaming, das heißt das Senden kontinuierlicher Daten spezifiziert. Die zu übertragenden Daten werden zunächst unterteilt, die jeweiligen Fragmente anschließend übertragen und von den Empfängern wieder zusammengesetzt.

Als nicht zu akzeptierender Nachteil wird jedoch die Verlagerung der Fragmentierung in die Anwendung erachtet. Indem ein Publisher oder ein Subscriber Annahmen über das Kommunikationsprotokoll macht, geht die durch COSMIC gebotene Transparenz

verloren. Für den Erhalt der Trennung von Anwendung und Kommunikation ist daher ein neuer Entwurf der Fragmentierung erforderlich.

Jedes zur Anwendung kommende Kommunikationsprotokoll spezifiziert eine mit einer Nachricht übertragbare Nutzlast. Diese bestimmt die Unterteilung eines Ereignisses. Die Fragmentierung von Ereignissen ist daher abhängig von der verwendeten Kommunikation. Anhand Abbildung 6.3 wird ersichtlich, dass lediglich zwei Bestandteile von COSMIC für die jeweilige Kommunikation spezifisch sind. Der Structured Transport Sublayer ist für die Durchführung der Fragmentierung prädestiniert. Er adaptiert für das jeweilige Kommunikationsprotokoll, ist zugleich aber nicht auf die Hardware einer Kommunikationsschnittstelle festgelegt. Möglich wird so auch der Einsatz auf verschiedener Hardware für das gleiche Kommunikationsprotokoll. Als Beispiel kann die in Abschnitt 2.1 angesprochene Anbindung von Kommunikationsbausteinen an die Peripherie genannt werden. Jene erfordert vielfach unterschiedlichste Zugriffsmethoden.

Gemäß Abschnitt 6.1.3 ist für die Kommunikation der CAN-Bus vorgesehen. Somit erfolgt der Entwurf für ein Fragmentierungsprotokoll nur für diesen Bus. Mehrere Anforderungen sind bekannt:

- Ein Rückkanal ist nicht vorgesehen. Bei Verlusten von Fragmenten infolge von Kommunikationsfehlern kann ein Teilnehmer keine fehlenden Fragmente nachfordern. Ein Entwurf mit einem Rückkanal liegt außerhalb des Aufgabenbereichs der vorliegenden Arbeit.

- Das prominente Beispiel des Internet Protocols (IP) zeigt die Fragmentierung mit Hilfe von Daten im Header der Nachrichten. Indem der Header sowohl den Umfang der gesamten Nachricht als auch die Position eines Fragments angibt, kann die Defragmentierung erfolgen. Diese Möglichkeit ist in der beabsichtigten Variante von COSMIC nicht gegeben. Der maximal verfügbare Header einer CAN-Nachricht, der Extended Identifier, ist mit Angaben zur Priorität, zur Kennzeichnung des Senders und durch das Etag eines Ereignisses bereits ausgefüllt.

- Jedes Ereignis erfordert Speicher. Ereignisse mit einem hohem Speicherbedarf können leicht zu Problemen auf Mikrocontrollern führen. Für komplette Ereignisse kann noch der Festspeicher als Ort der Speicherung vorgesehen werden. Fragmentierte Ereignisse setzen sich jedoch erst im Laufe der Übertragung zusammen. Eine stetige Aktualisierung im Festspeicher ist aufgrund der Speichercharakteristik mit hohem Aufwand verbunden. Bedeutend einfacher gestaltet sich die Verwendung des flüchtigen Speichers, welcher aber weitaus geringere Kapazitäten bietet.

Unter Beachtung der Anforderungen wird sich für das folgende Protokoll entschieden:

1. Entspricht ein Ereignis einer Nutzlast von 64 Bit oder weniger, wird es wie bisher innerhalb einer Nachricht übertragen.

2. Umfangreichere Ereignissen erhalten für die Übertragung eine dedizierte Priorität. Auf diese Weise sind fragmentierte Ereignisse für die Empfänger markiert. Zugleich entfällt eine Kennzeichnung für nicht fragmentierte Ereignisse.

3. Jede Nachricht, welche Teil eines fragmentierten Ereignisses ist, beinhaltet eine eindeutige Sequenznummer. So lässt sich jede Nachricht in die Daten eines Ereignisses einordnen.

4. Die erste Nachricht mit der Sequenznummer Null trägt als einzige Information den Umfang des Ereignisses. Obwohl unter Umständen noch Platz in der Nachricht bleibt, wird der Einfachheit halber auf weitere Daten verzichtet.

Mit Hilfe der Sequenznummern bleibt jedem Empfänger die Möglichkeit, ein unvollständiges Ereignis zu erkennen. Dazu muss er den Empfang von Fragmenten protokollieren. Eine platzsparende Möglichkeit stellt die Verwendung eines Bitvektors dar, welcher für jedes Fragment ein Bit enthält. Damit ist auch eine komplette Umordnung (*Reordering*) der Fragmente durchführbar. Einzelne Teile brauchen nicht in der Reihenfolge empfangen werden, in der sie versendet wurden. Dies ist für Controller von Bedeutung, welche mehrere Puffer für empfangene Nachrichten bieten. Der zum Einsatz kommende Mikrocontroller AT90CAN128 aus Abschnitt 6.1.1 besitzt einen derartigen Controller mit 15 Puffern.

Ungelöst bleibt das Problem der Flusskontrolle. Erfordert der Empfang einer Nachricht mehr Zeit als der Versand, stehen jedem Controller in absehbarer Zeit keine freien Puffer zur Verfügung. Eine Flusskontrolle mit mehreren Empfängern ist nicht trivial. Üblicherweise fordern die Empfänger den Sender zur Unterbrechung der Übertragung auf. Die Empfänger können aber zu einem Zeitpunkt sowohl auf eine Fortsetzung als auch auf eine Unterbrechung der Übertragung drängen. Eine denkbare Vorgehensweise ist die Verwendung einer Verzögerungszeit, eines so genannten *Backoffs*. Nach jeder Aufforderung, die Übertragung zu unterbrechen, reduziert der Sender die Übertragungsleistung auf ein Minimum. Für nachfolgende Übertragungen beginnt er, die Leistung schrittweise zu erhöhen. Erfolgt eine erneute Aufforderung zur Unterbrechung, wiederholt sich der Vorgang. Neben dem Aufwand für ein solches Vorgehen ist für die Flusskontrolle stets ein Rückkanal erforderlich. Für die vorliegende Arbeit liegt der Entwurf einer Flusskontrolle abseits der Aufgabenstellung. Daher wird davon ausgegangen, dass der Sender von Ereignissen die Dauer des Empfangs berücksichtigt.

Das beschriebene Protokoll setzt voraus, dass unter einem Subject nur jeweils ein fragmentiertes Ereignis pro Sender publiziert wird. Durch gleiche Identifier in Priorität, Sender und Etag besteht für Empfänger andernfalls keine Möglichkeit, mehrere sich überlappende Ereignisse zu unterscheiden. Dennoch können im Netzwerk zu jedem Zeitpunkt mehrere fragmentierte Ereignisse eines Subjects verbreitet werden. Grundlage ist die vorausgesetzte, stets eineindeutige Beziehung zwischen Ereignis und Sender. Durch die unterschiedliche Kennung der Sender können die Nachrichten unterschieden und zugeordnet werden. Gleiches gilt für unterschiedliche Subjects. Mit ihrer Entsprechung in Form unterschiedlicher Etags ermöglichen sie eine Identifikation der Nachrichten. Somit kann auch ein System parallel mehrere fragmentierte Ereignisse verbreiten, jedoch nur für unterschiedliche Subjects.

Wie in den Anforderungen beschrieben, ist das Vorhalten sämtlicher Fragmente eines Ereignisses im flüchtigen Speicher vorteilhaft. Ist das Ereignis komplett, kann es in

einen anderen Speicher ausgelagert werden. Dementsprechend muss Platz für komplette Ereignisse bereitstehen.

Eine Beschränkung des Speicherbedarfs für Ereignisse ist jedoch unumgänglich. Dieser muss sich an den beschränkten Ressourcen und an der Anzahl der von einem System gleichzeitig empfangbaren Ereignissen orientieren. Für die Rekonfiguration geeignet erscheint eine an die Charakteristik des nicht flüchtigen Speichers angepasste Größe. Wie in Abschnitt 6.1.1 beschrieben, ist dieser Speicher im beabsichtigten Mikroprozessor als Flash ausgeführt. Die Sektorengröße für Schreibzugriffe (*Pagesize*) beträgt 256 Byte. Abgesehen von Fällen mit sehr wenig Programmcode, ist ein Ereignis von derartigem Umfang nicht für die Beschreibung einer Rekonfiguration ausreichend. Jedoch bietet eine Ausweitung auf den kompletten nichtflüchtigen Speicher diesbezüglich keine Lösung. Ein Ereignis ist somit stets nur ein einzelnes Fragment. Um ein Zusammensetzen der Fragmente bzw. Ereignisse zu ermöglichen, können daher weitere Informationen erforderlich werden. Ihr Bedarf an Speicher verlangt gegebenenfalls eine Erhöhung des Umfangs von Ereignissen, damit weiterhin jeweils ein kompletter Sektor (*Page*) übertragen wird.

6.3. Entwurf für den Mikrocontroller

Die nach Abschluss der Rekonfiguration im Festspeicher vorliegende Anwendung muss in Inhalt und Struktur eine Ausführung ermöglichen. Die Hardware nimmt sowohl auf die Struktur als auch auf den Inhalt des Festspeichers Einfluss:

- Das Ausführen der Anwendung erfordert vorbereitende und abschließende Schritte. Der Mikrocontroller AT90CAN128 verwendet den Festspeicher ausschließlich als Befehlsspeicher. Entsprechend der Harvard-Architektur liegen die Operanden in einem anderen Speicher. Der verwendete SRAM besitzt keine Möglichkeit, vorgegebene Operanden dauerhaft vorzuhalten. Zur Vorbereitung ist daher ein Kopieren der initialen Werte aus dem Festspeicher an die Adressen im SRAM notwendig. Das Ausführen von Konstruktoren ist ein Schritt, der vor dem Ausführen einer objektorientiert erstellten Anwendung notwendig ist. In diesem Fall sind nach Beendigung des Programmablaufs auch die jeweiligen Destruktoren aufzurufen. Hinzu kommt die Initialisierung einzelner, für den Programmablauf benötigter Register wie beispielsweise des Stackpointers.

- Durch die Hardware fest vorgegebene Adressen machen eine Berücksichtigung durch die Anwendung notwendig. Bei gleichzeitiger Verwendung als Befehlsspeicher ist auch der Festspeicher betroffen. Die Interruptvektoren des AT90CAN128 sind nach Abbildung 6.1 Teil des nichtflüchtigen Speichers. Entsprechend müssen die Sprungbefehle und -adressen in den vorgegebenen Plätzen abgelegt werden. Gleiches gilt für die Routinen, welche den Festspeicher modifizieren. Gemäß Abschnitt 6.1.1 sind sie in der BLS zu positionieren.

Die zum Ausführen genannten Schritte sind, abgesehen von speziellen Anwendungen, stets notwendig. Programmcode, welcher die Tätigkeiten realisiert, bietet somit eine

Laufzeitumgebung für Anwendungen. Die avr-libc stellt eine Laufzeitumgebung in Form der *gcrt* (GNU C Runtime) bereit. Ergänzt wird die gcrt durch die vom avr-gcc im Rahmen der *libgcc* bereitgestellten Routinen. Im Zusammenspiel mit dem Linker erfolgt die an die Hardware angepasste Strukturierung der Anwendung. Vom Linker werden dazu so genannte Linkerskripte (*Linker Scripts*) abgearbeitet, welche den Inhalt der gcrt sowie den vom Programmierer erstellten Code anordnen.

Von den binutils werden bereits verschiedene Linkerskripte bereitgestellt. Mit ihrer Bearbeitung erstellt der avr-ld die in Abbildung 6.4 dargestellte Struktur des Programmcodes. Die Initialisierungsroutinen der gcrt folgen dem Programmcode für die

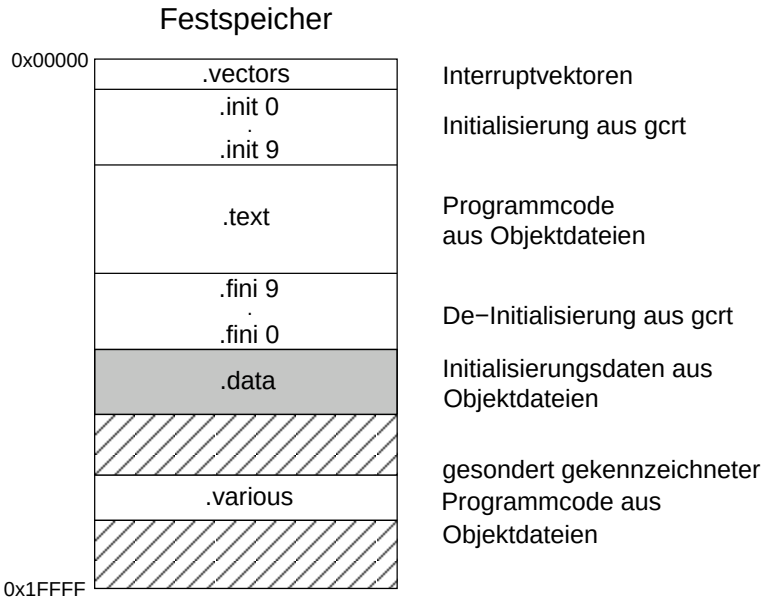


Abbildung 6.4: Speicherlayout im Festspeicher

Interruptvektoren. Je nach Aufgabe sind die Routinen in den einzelnen, mit *.init* bezeichneten Abschnitten (*Sections*) enthalten. Innerhalb der letzten Section der Initialisierung erfolgt der Einstieg in den Programmcode der Objektdateien. Bei Beendigung des Programmcodes schließen die in den *.fini*-Abschnitten enthaltenen Routinen die Ausführung der Anwendung ab. Die Initialisierungsdaten für Operanden sind nach dem Befehlscode angeordnet. Ihr Kopieren in den Datenspeicher erfolgt durch die Instruktionen einer *.init*-Section. Vom Linkerskript nicht behandelte Abschnitte aus Objektdateien folgen standardmäßig dem Programmcode. Somit kommt es zu Überschneidungen mit der .data-Section. Ein Linken ist in diesem Fall nicht möglich. Solche, in der Abbildung beispielhaft mit *.various* bezeichneten Abschnitte können jedoch dazu dienen, Programmcode an gewünschte Adressen zu platzieren. Zu diesem Zweck ist dem Linker die Adresse für einen Abschnitt explizit mitzuteilen.Bei geeigneter Adresswahl treten keine Überlappungen mit Abschnitten aus dem Linkerskript auf.

Im Konzept der Ersetzung von Bestandteilen aus Abschnitt 5.2 erfolgt kein einzelnes Linken. Statt dessen werden zwei Linkprozesse für jeweils einen Bestandteil der Anwendung durchgeführt. Keiner der Linkprozesse besitzt zunächst Informationen über die

Ergebnisse des anderen Prozesses. Die Strukturierung und Adressvergabe erfolgt vollkommen unabhängig. Weder der Programmcode noch die gegebenenfalls vorhandene Laufzeitumgebung eines Teils nimmt Bezug auf den anderen Teil. Somit lassen sich die Resultate nicht als eine Anwendung ausführen. Zusätzlicher Aufwand für das Linken muss aber auch im Konzept der vollständigen Ersetzung aus Abschnitt 5.1 betrieben werden. Dies liegt in der Position der BLS begründet, die eine Zusammenarbeit mit dem limitierten Zwischenpuffer erschwert.

Bei einer denkbaren Rekonfiguration von Modulen gemäß Abschnitt 5.3.3 kann auf ein Linken der Module mit der Laufzeitumgebung verzichtet werden. Es liegt somit auch in der Verantwortung des Mikrocontrollers, den Inhalt der einzelnen Module in die Laufzeitumgebung zu integrieren. Zusätzlicher Verwaltungsaufwand ist die Folge. Das Linken, wie es für die Ersetzung von Bestandteilen nach Abschnitt 5.2 notwendig ist, lässt sich jedoch auch auf Module anwenden. Damit können durch einen Entwurf beide Konzepte abgedeckt werden.

6.3.1. Integration der Routinen zum Beschreiben des Festspeichers

Das Beschreiben des Festspeichers ist ein zentrales Element der Rekonfiguration. Unabhängig vom Konzept, müssen stets einzelne Bereiche des Festspeichers mit neuen Daten gefüllt werden. Nach Abschnitt 6.1.1 müssen die notwendigen Instruktionen in einem speziellen Adressbereich des Fest- bzw. Befehlsspeichers vorliegen.

Eine Rekonfiguration der Anweisungen zum Beschreiben des Festspeichers birgt erneut die Gefahr von Inkonsistenzen, analog zu den Ausführungen in Abschnitt 5. Dem Umfang der Rekonfiguration werden jedoch durch die Verwendung eines Zwischenpuffers gemäß Abschnitt 5.1.3 Grenzen gesetzt. Wie aus Abbildung 6.5 ersichtlich, bleibt die Boot Loader Flash Section im Verlauf einer Rekonfiguration stets unangetastet. Indem der Zwischenpuffer an eine niedrigere Adresse verschoben wird, können mögli-

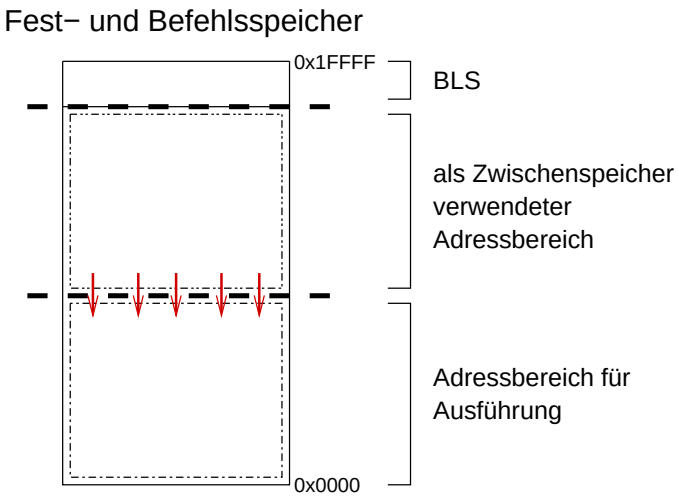


Abbildung 6.5: Beziehung von Zwischenspeicher und Boot Loader Flash Section

che Änderungen keine nachfolgenden Speicherbereiche betreffen. Somit ist auch eine Rekonfiguration der Anweisungen zum Schreiben des Festspeichers, die zur Ausführung in der BLS vorliegen müssen, ausgeschlossen.

Eine Möglichkeit, auch nachfolgende Bereiche zu berücksichtigen, bietet sich mit einer überarbeiteten Aufteilung des Speichers. Der Zwischenpuffer kann lediglich in eine Richtung verschoben werden, entweder an niedrigere oder an höhere Adressen. Da die BLS im Befehlsspeicher die höchsten Adressen einnimmt, bleibt allein die Möglichkeit des Verschiebens in Richtung höherer Adressen. Vornehmlich beinhaltet der Zwischenpuffer aber Anweisungen, welche für den zur Ausführung vorgesehenen Adressbereich vorgesehen sind. Nimmt der Zwischenpuffer niedrigere Adressen als dieser Bereich ein, kann ein Verschieben an höhere Adressen auch dessen Inhalt modifizieren. In Abbildung 6.5 tauschen somit der Zwischenpuffer und der Adressbereich für die Ausführung die Positionen. Durch den Tausch ist der Inhalt des Zwischenpuffers allein für nachfolgende Adressen vorgesehen und kann zusammenhängend verschoben werden. Einzige Ausnahme sind die Interruptvektoren. In Abbildung 6.1 ist deren feste Position am Beginn des Befehlsspeichers dargestellt. Alternativ kann auf die zweite Position der Interruptvektoren ausgewichen werden, welche sich am Beginn der BLS befindet. Zu deren Verwendung ist jedoch ein Modifizieren der Einstellungen der Mikrocontroller erforderlich. Weiterhin ist in diesem Fall eine Berücksichtigung durch den Linker geboten. Die in Abbildung 6.4 dargestellte Struktur muss an die veränderte Position der Interruptvektoren am Beginn der BLS angepasst werden.

Angesichts des statischen Charakters von Routinen zum Beschreiben des Fest- bzw. Befehlsspeichers wird jeglicher Aufwand zu deren Rekonfiguration als unverhältnismäßig bewertet. Einmal in die BLS eingebracht, können die Routinen unverändert in verschiedensten Konzepten zum Einsatz kommen. Die Möglichkeit ihrer Rekonfiguration erfordert neben dem bereits beschriebenem Aufwand für die Interruptvektoren eine Anpassung der gesamten Anwendung. Die aus Abbildung 5.2 bekannten Bestandteile COSMIC, Rekonfiguration und anwendungsspezifische Routinen sind, bedingt durch die geänderte Platzierung des Zwischenpuffers, sämtlichst an neuen Adressen zu positionieren. Allerdings benötigt lediglich der für die Rekonfiguration verantwortliche Bestandteil die Routinen der BLS. Eine explizite Verschiebung des anwendungsspezifischen Bestandteils aufgrund von nicht verwendeten Routinen ist nicht zu begründen. Dies gilt insbesondere mit Blick auf die getrennte Rekonfiguration einzelner Bestandteile der Anwendung gemäß Abschnitt 5.2. Daher wird auf die Möglichkeit zur Rekonfiguration der Routinen zum Beschreiben verzichtet. Entsprechend sorgfältig hat die Implementation zu erfolgen. Von einer geringen Häufigkeit von Fehlern kann aufgrund der begrenzten Komplexität der Routinen aber ausgegangen werden. Jede Korrektur oder Änderung ist mit dem Einsatz eines anderen, aufwendigeren Verfahrens verbunden und muss möglichst ausgeschlossen werden.

Eine Anwendung, welche irrtümlich Code für die BLS beinhaltet, kann nicht installiert werden. Die Bearbeitung des Zwischenpuffers kann dies gewährleisten, indem der Zustand des Zwischenpuffers überwacht wird. Läuft der Zwischenpuffer über, verweist der Programmcode auf Adressen außerhalb des ihr zugedachten Bereichs. Die Rekonfiguration ist daraufhin zu ignorieren. Die Bearbeitung des Zwischenpuffers ist Aufgabe

der Rekonfiguration. Dafür verwendet sie Routinen aus der BLS und muss somit auf diese verweisen können. Da ein Beschreiben der BLS nicht unterstützt wird, darf sie ihr Einbinden in den Programmcode jedoch nicht erfordern. Um trotzdem ein Auflösen der entsprechenden Symbole und damit einen erfolgreichen Linkprozess zu ermöglichen, werden dem Linker explizit Symbole vorgegeben. Zunächst nahe liegend sind die Adressen der jeweiligen Routinen aus der BLS. Weiterhin müssen Symbole für Inhalte des SRAM vorgegeben werden, welche von der Rekonfiguration verwendet, deren Adressen jedoch durch die Schreibroutinen festgelegt werden. Die Vorgabe der Symbole kann beim Aufruf des avr-ld erfolgen. Für eine einfache Anwendung ist die Definition im Rahmen eines Linkerskripts von Vorteil. In diesem Fall sind die Informationen zentral gespeichert und eine Angabe in jedem Projekt kann unterbleiben. Notwendig wird statt dessen einzig eine Aufforderung an den Linker, das angepasste Skript anstelle eines der vorgegebenen Skripte zu bearbeiten. Da die Symbole vorgegeben werden, müssen die Routinen an den beschriebenen Adressen innerhalb der BLS platziert sein. Eine Überprüfung durch den Linker kann ebenso wenig erfolgen wie zur Laufzeit. Der Programmierer des Bestandteils der Rekonfiguration hat für die Übereinstimmung von Linkerskript und Speicherinhalt der BLS Sorge zu tragen. Aufgrund des üblicherweise nur einmal zu tätigenden Aufwands wird dies als ein hinreichend geeignetes und zugleich einfaches Vorgehen angesehen.

6.3.2. Verbindung des Programmcodes

Mit einer Implementation des im vorigen Abschnitt 6.3.1 beschriebenen Entwurfs ist eine komplette Rekonfiguration nach Abschnitt 5.1 möglich. Der Entwurf für die Ersetzung von Bestandteilen gemäß Abschnitt 5.1 erfordert weitere Schritte. Hier müssen die Resultate unterschiedlicher Linkprozesse zusammengeführt werden.

Für jeden Speicher gilt, das Adressen nicht mehrfach vergeben werden können. Während gleiche Symbole bei einem Linkvorgang auf genau eine Adresse abgebildet werden, ist die bidirektionale Abbildung bei mehreren Linkvorgängen nicht gewährleistet. Jedem Linkvorgang fehlen die Informationen über bereits vergebene Adressen. Eine vollständig gelinkte Objektdatei als Endprodukt eines Linkprozesses lässt sich als Eingabe für einen weiteren Linkvorgang verwenden. Unbekannt bleibt die Zuordnung von Adressen im zweiten Linken. Sie obliegt allein dem Linker. Denkbar ist eine mögliche direkte Aneinanderreihung oder gar Verzahnung der Adressbereiche, welche eine unabhängige Änderung eines Bestandteils erschwert. Eine Möglichkeit besteht in der Vorgabe der Adressbereiche durch den Programmierer. Wie in Abschnitt 6.3 beschrieben, strukturiert der Linker den Programmcode durch die Bearbeitung von Linkerskripts. Indem verschiedene Linkerskripte disjunkte Adressbereiche verwenden, werden gleich benannte Abschnitte der einzelnen Bestandteile auf unterschiedliche Adressen positioniert. Abbildung 6.6 skizziert die Zuweisung von verschiedenen Speicherbereichen an zwei Bestandteile. Eine getrennte Positionierung im Fest- bzw. Befehlsspeicher bildet die Grundlage für die spätere Ausführung. Indem beide Bestandteile unterschiedliche Adressen im Befehlsspeicher einnehmen, ist die Konsistenz gewährleistet. Zusätzlich notwendig ist eine Trennung im Datenspeicher. Die Instruktionen der einzelnen Bestandteile nutzen Verweise auf bei-

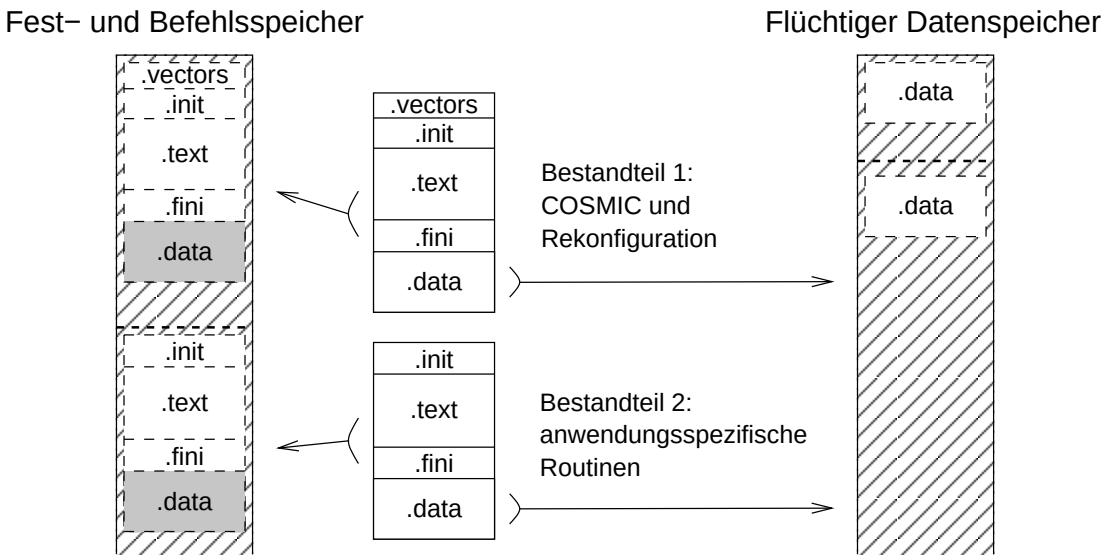


Abbildung 6.6: Platzierung von zwei Bestandteilen im Speicher

spielsweise Variablen. Die Zuweisung der Adressen erfolgt im Zuge der Symbolauflösung beim Linken. Daher ist durch modifizierte Linkerskripte auch eine geeignete Aufteilung des Datenspeichers zu realisieren.

Wenn sich der Umfang des ersten Bestandteils nicht oder nur gering ändert, stellt die Aufteilung durch vorgegebene Linkerskripte ein geeignetes und zugleich überschaubares Vorgehen dar. Der Programmcode von COSMIC und der Rekonfiguration bildet diesen ersten Bestandteil. Es kann angenommen werden, dass sich beide Komponenten nur in geringem Umfang im Leben der Anwendung ändern. Um begrenzte Größenänderungen zu tolerieren, wird das Konzept eines Puffers aus dem in Abschnitt 4.1.3 vorgestelltem Verfahren aufgegriffen. Der erste Bestandteil kann auf diese Weise variieren, ohne stets eine Änderung des nachfolgenden Bestandteils erforderlich zu machen. Nachteilig ist, dass auf diesem Wege Platz im Befehlsspeicher reserviert wird und somit nicht mehr zur Verfügung steht. Problematisch ist ebenfalls, dass dem Programmierer die Verantwortung für eine geeignete Aufteilung des Speichers übertragen wird. Eine Berücksichtigung des auf einem Mikrocontroller vorliegenden Speicherlayouts erfolgt nicht. Durch Verwendung eines geeigneten Linkerskripts kann jedoch das Platzangebot für den ersten Bestandteil vom Linker überprüft werden. Übersteigt der Programmcode von COSMIC und der Rekonfiguration den reservierten Bereich, schlägt das Linken fehl. In einem solchen Fall wird eine Überarbeitung der Aufteilung des Speichers zwischen den verschiedenen Bestandteilen der Anwendung notwendig, zusammen mit einer kompletten Rekonfiguration gemäß Abschnitt 5.1.

6.3.3. Verbindung der Laufzeitumgebungen

Abbildung 6.6 aus dem vorigem Abschnitt 6.3.2 macht weitere Aspekte der Zusammenarbeit von Bestandteilen deutlich.

1. Jeder Bestandteil verwendet potentiell Interrupts. Die Adressen der Interruptvektoren sind vorgegeben. Mehrere Bestandteile können nicht selbstständig Interruptvektoren definieren, ohne einen anderen Bestandteil zu beeinflussen. Lediglich eine Definition der Vektoren ist möglich.

2. Die in der gcrt zusammengefassten Routinen müssen für jeden Bestandteil ausgeführt werden. Das betrifft die Routinen zum Initialisieren von Operanden im Datenspeicher sowie zum Aufruf von Konstruktoren und Destruktoren. Der mehrfache Aufruf von Tätigkeiten wie der Initialisierung von Registern oder des Stackpointers hat jedoch die Zerstörung eines existierenden Zustands zur Folge.

Die eindeutige Definition der Interruptvektoren nach Punkt 1 lässt sich mit der in Abschnitt 5.1.2 beschriebenen Umleitung realisieren. Ein Bestandteil der Anwendung definiert alle Interruptvektoren. Indem als Sprungziele Einträge einer Tabelle im Datenspeicher verwendet werden, können sich mehrere Bestandteile für Interrupts interessieren. Anstatt der direkten Verwendung der Adressen der Interruptvektoren sind nun die Adressen von Routinen notwendig, welche einen Eintrag in der Sprungtabelle vornehmen. Gleichzeitig muss eine erneute und somit irrtümliche Definition eines jeden Interruptvektors verhindert werden. Geeignet dazu ist die Nutzung eines angepassten Linkerskripts. Wird das Einbinden von Interruptvektoren explizit verhindert, bleiben die ursprünglichen Einträge im Befehlsspeicher unangetastet.

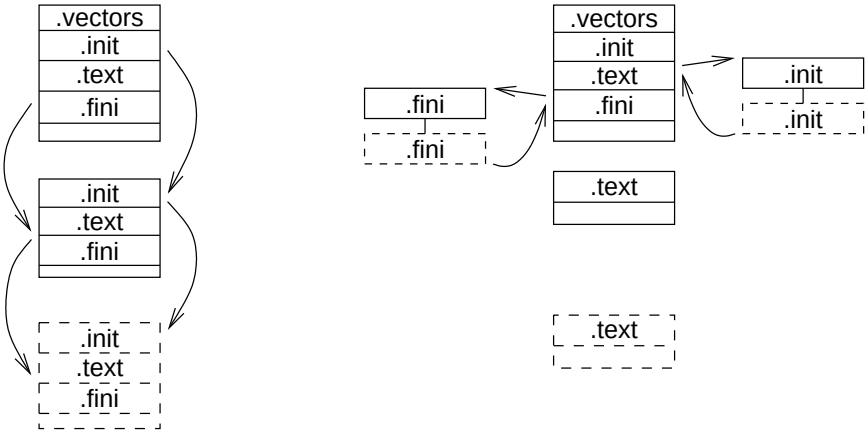
Punkt 2 bedarf unterschiedlicher Varianten der Laufzeitumgebung. Um einen initialisierten Zustand von Registern zu bewahren, dürfen ausgewählte Sections nur einmal ausgeführt werden. Entsprechende Beispiele sind bereits mit Punkt 2 aufgeführt. Derartige Abschnitte dürfen daher kein Teil der Laufzeitumgebung mehrerer Bestandteile sein. Durch ein angepasstes Linkerskript besteht die Möglichkeit, die betreffenden Sections zu ignorieren. Sie werden somit kein Teil des auszuführenden Codes.

In Zusammenfassung der Lösungsvorschlägen für die Punkte 1 und 2 kommen in der Softwareentwicklung somit zwei unterschiedliche Linkerskripte zum Einsatz. Für die Basis, bestehend aus den Komponenten COSMIC und Rekonfiguration, kann ein ursprüngliches Linkerskript aus den binutils verwendet werden. Zu berücksichtigen ist die in Abschnitt 6.3.1 beschriebene Anpassung für die Schreibroutinen des Befehlsspeichers. Für weitere Bestandteile ist ein anderes Linkerskript zu verwenden. Dessen Bearbeitung hat den Ausschluss von Initialisierungscode zur Folge. Wie bereits in Abbildung 6.6 dargestellt, beinhalten zusätzliche Bestandteile somit keine Interruptvektoren. Das Bewahren eines einmal initialisierten Zustands ist nur für einen Rücksprung in vormals aufgerufenen Programmcode notwendig. Ein solcher Sprung ist nicht vorgesehen, die Routinen von COSMIC und der Rekonfiguration laufen allein in Reaktion auf Interrupts. Die betreffenden Teile der Laufzeitumgebung brauchen daher nicht ausgeschlossen werden. Die Verwendung von Interrupts für die Mechanismen der Rekonfiguration erlaubt die zeitgesteuerte Ausführung der anwendungsspezifischen Routinen. Wie in Abschnitt 5 beschrieben, ist keine nebenläufige Ausführung möglich. Eine rein zeitgesteuerte Ausführung der Rekonfiguration führt daher zu einem Blockieren anderer zeitgesteuerter Komponenten. Möglich bleibt auch die Ausführung der anwendungsspezifischen Routinen in Reaktion auf Interrupts. Die Verwendung von Interrupts zur Steuerung der

Rekonfiguration vermeidet somit eine Beschränkung der zur Ausführung kommenden Anwendung.

Die Aufteilung der Anwendung entsprechend Abbildung 5.2 beinhaltet den Aufruf unterschiedlich gebundener Komponenten. Ihre jeweilige Ausführung umfasst zwei Bereiche:

- Die vorbereitenden und abschließenden Maßnahmen aus der Laufzeitumgebung müssen durchgeführt werden. Wird lediglich ein Bestandteil vorbereitet, befinden sich weitere Bestandteile bei ihrer Ausführung in einem undefinierten Zustand. Beispielsweise sind Variablen nicht initialisiert und Konstruktoren nicht ausgeführt. Sämtliche Schritte finden vor dem Einstieg in den eigentlichen, vom Programmierer spezifizierten Programmcode statt, oder sie schließen sich an dessen Ausführung an. Im allgemeinen Fall fällt es daher den einzelnen Laufzeitumgebungen zu, weitere Bestandteile zu berücksichtigen. Abbildung 6.7 skizziert mögliche Vorgehensweisen.



(a) Verkettung mehrerer Instanzen

(b) Bearbeitung aller Abschnitte durch eine Instanz

Abbildung 6.7: Zusammenarbeit von Laufzeitumgebungen

  - Vergleichsweise einfach ist die in Abbildung 6.7a dargestellte Verkettung von Laufzeitumgebungen. Vor Programmeintritt springt eine Umgebung nachfolgende Instanzen an. Zum Zeitpunkt des Linkens ist noch unbekannt, an welcher Adresse sich der nächste Bestandteil anschließt. Das Ende des eigenen Bestandteils ist hingegen bekannt. Damit kann der nachfolgende Speicher sequentiell durchsucht werden. Bei Annahme eines jeweils gleichen Speicherlayouts gemäß Abbildung 6.6 braucht schlicht nach der ersten Anweisung einer weiteren Laufzeitumgebung gesucht werden. Andernfalls ist eine explizite Kennzeichnung eines Bestandteils notwendig.
  - Um mehrere Umgebungen zu vermeiden, können die jeweiligen Abschnitte zentral zusammengefasst werden. Abbildung 6.7b stellt diese Variante dar.

Während der Initialisierung des ersten Bestandteils erfolgt die Berücksichtung weiterer Bestandteile. Gleiches gilt für die Maßnahmen, welche nach Beendigung der Anwendung erforderlich werden.

Beide Vorschläge erfordern eine Modifikation der gcrt. Für eine Verkettung sind Anweisungen zum Suchen im nachfolgenden Befehlsspeicher notwendig. Wie beschrieben, ist gegebenenfalls eine Kennzeichnung notwendig. Hier bietet sich der Code einer illegalen Anweisung an (*Illegal Opcode*). Dieser wird vom Übersetzer nicht erzeugt und liegt dementsprechend nicht im Befehlsspeicher von Anwendungen vor. Eine solche Kennzeichnung kann damit nicht irrtümlich als Beginn einer Laufzeitumgebung interpretiert werden. Das Zusammenfassen stellt zusätzliche Anforderungen an die Verwaltung der jeweiligen Abschnitte. Zum Zeitpunkt des Linkens muss die zentrale Laufzeitumgebung mindestens die Adressen der Abschnitte für einen ersten Bestandteil kennen. Zusätzlich müssen Abschnitte weiterer Bestandteile erkannt werden können. Hinzu kommen zusätzliche Informationen wie zum Beispiel die Adressen im flüchtigen Speicher, an welche Daten aus dem Festspeicher kopiert werden müssen.

Das Vorgehen zur Verknüpfung der Laufzeitumgebungen ist auch für die modulbasierte Rekonfiguration nach Abschnitt 5.3.3 denkbar. Modifikationen einzelner Module verlangen unter Umständen eine Berücksichtigung beim Vor- und Nachbereiten der Programmausführung. Die Möglichkeit der zentralen Behandlung erfordert die explizite Kennzeichnung der zu betrachtenden Abschnitte eines Moduls. Als Beispiel sei die Section .data genannt, deren veränderter Umfang oder Position eine geänderte Initialisierung des SRAM erfordert. Mit einer Verkettung kann die Kennzeichnung unterbleiben. Jedes Modul beinhaltet seine eigene Laufzeitumgebung, welche für seinen Inhalt zutreffend ist. Verwendet die Umgebung ihrerseits Symbole, ist eine eindeutige Symbolbenennung notwendig. Andernfalls kommt es zu Überschneidungen mit anderen Modulen, welche gleich benannte Symbole definieren. Hier ist positionsunabhängiger Code vorteilhaft, der die Initialisierung bzw. Nachbereitung relativ zur Platzierung eines Moduls erlaubt. Eigene Symbole können somit schon beim Linken des Moduls mit seiner Umgebung aufgelöst werden. Nachteilig sind notwendige Änderungen an der gcrt, welche globale und absolut definierte Symbole einsetzt.

- Ziel des umgebenden Codes ist die Vor- bzw. Nachbereitung der Ausführung des vom Programmierer unmittelbar spezifizierten Programmcodes. Für die Ausführung ist ein entsprechender Sprung an den Beginn des Programmcodes notwendig. Dieser kann, wie in Abschnitt 6.3 beschrieben, Teil der Initialisierung sein.

Das Konzept der Middleware COSMIC unterstützt die Verwendung durch mehrere Bestandteile bzw. Anwendungen. Beschränkt sich deren Inhalt auf die Reaktion auf Ereignisse, können mehrere Anwendungen genutzt werden. Bislang nicht möglich ist die Ausführung von Anwendungen, welche unabhängig von Ereignissen ablaufen. Diese arbeiten ihren Programmcode ab und kehren an den Aufrufer zurück. Bei fehlender Nebenläufigkeit gemäß Abschnitt 5 ist die Ausführung einer Anwendung daher mit einer

Kontrollübergabe verbunden. Es obliegt einer jeden ausgeführten Anwendung, die Ausführung weiterer Anwendungen zu ermöglichen. Im Fall eines expliziten Aufrufs erfolgt ein Start, mit dem Ende der Abarbeitung die Fortsetzung der aufrufenden Anwendung.

Der Bestandteil der Rekonfiguration, welcher selbst eine Anwendung von COSMIC darstellt, kann sich auf die Reaktion auf Ereignisse beschränken. Wie beschrieben, ist eine Möglichkeit für mehrere nebenläufige Anwendungen nicht vorgesehen. Daher begrenzt sich der Entwurf auf die Unterstützung der Ausführung nur einer weiteren Anwendung. Für das Konzept der Ersetzung von Bestandteilen nach Abschnitt 5.2 liegen damit lediglich zwei Bestandteile vor. Durch diese Limitierung kann auf die gesonderte Strukturierung der vorbereitenden und abschließenden Maßnahmen verzichtet werden. Abweichend von den Vorgehen aus Abbildung 6.7, ist ein Sprung an den Beginn des zweiten Bestandteils ausreichend. Abbildung 6.8 stellt den derartigen Ablauf der Programmausführung dar. Zusätzlich verbindet die Abbildung die bereits beschriebene

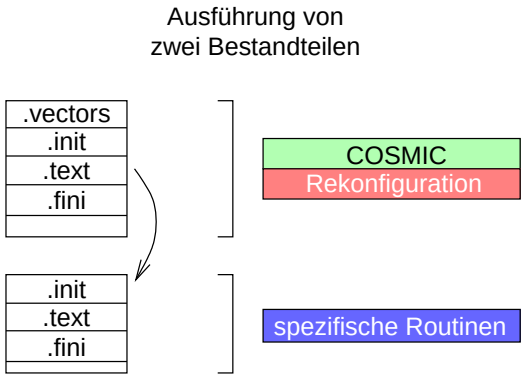


Abbildung 6.8: Ausführung von zwei Bestandteilen im Speicher

Aufteilung der Anwendung mit der Ausführung. Die Komponenten COSMIC und Rekonfiguration bilden den ersten Teil. Im Rahmen ihrer Ausführung erfolgt der Sprung an eine vorgegebene Adresse. An dieser müssen die Instruktionen der anwendungsspezifischen Routinen vorliegen. Durch die Verwendung von Interrupts kann die Rekonfiguration bei Empfang entsprechender Ereignisse weiterhin durchgeführt werden.

Die anwendungsspezifischen Routinen müssen eine korrekte Ausführung gewährleisten. Als problematisch muss auch der Sprung an eine vorgegebene Adresse eingeschätzt werden. Für den Fall, dass kein anwendungsspezifischer Bestandteil vorliegt, muss dennoch in ausführbaren, gültigen Programmcode gesprungen werden. Hinzu kommen Fehler, welche auf Probleme beim Beschreiben des Festspeichers zurückzuführen sind. Deren Berücksichtigung geht jedoch über die Zielstellung der vorliegenden Arbeit hinaus. Jeder Fehler birgt die Gefahr eines sich unablässig wiederholenden Reset des Mikrocontrollers. Damit geht die Möglichkeit für nachfolgende Korrekturen verloren. Mit einem Platzhalter, welcher an die Sprungadresse platziert wird, kann ein definierter Ausgangszustand geschaffen werden. Durch seine Präsenz werden gültige Anweisungen aufgerufen, falls noch keine spezifischen Routinen installiert sind. Indem er eine Endlosschleife beinhaltet, bleibt eine Rekonfiguration möglich. Gleiches gilt für eine Schleife nach dem Sprung an die vorgegebene Adresse. Auf diese Weise können weitere Aktualisierungen auch nach

einem Ende der Bearbeitung der anwendungsspezifischen Routinen durchgeführt werden.

Unbeachtet bleibt der Fall von fehlerhaften anwendungsspezifischen Routinen. Einmal installiert, befindet sich die Ausführung auf dem Mikrocontroller stets in einem inkonsistenten Zustand. Im Rahmen eines Recovery kann ein funktionierender Ausgangszustand installiert werden. Schwierigkeiten bestehen in der dazu notwendigen Erkennung des fehlerhaften Zustands. Mit jedem Reset gehen vorherige Informationen verloren. Zwei Mechanismen sind denkbar:

1. Dem Nutzer wird die Möglichkeit gegeben, bei Bedarf einzugreifen.
2. Der Mikrocontroller erkennt selbstständig einen unvorhergesehenen Reset.

Punkt 2 setzt entsprechende Möglichkeiten der Hardware voraus. Ein Eingriff des Nutzers ist eine plattformunabhängige Lösung. Vor Beginn der Ausführung der anwendungsspezifischen Routinen muss der Mikrocontroller dazu auf besondere Nachrichten achten. Im Falle von Ereignissen auf dem Notfallkanal wird das Recovery durchgeführt. Nachteilig für das Startverhalten ist die Zeit, die dem Empfang entsprechender Ereignisse eingeräumt werden muss. Auf einen Einsatz eines der beiden Mechanismen wird aufgrund der Zielstellung der Arbeit verzichtet.

6.4. Entwurf für den Host

Die in Abschnitt 6.1.2 genannten Bedingungen auf dem Host lassen einen im Vergleich zu den Mikrocontrollern vereinfachten Entwurf erwarten.

Die Programmiersoftware bildet die Schnittstelle zwischen dem menschlichen Nutzer und den Algorithmen der Rekonfiguration. Avrdude ist unter der GPL (*GNU Public Licence*) veröffentlicht (Version 2). Aufgrund des damit vorliegenden Quellcodes ist eine Erweiterung für eigene Zwecke möglich. Lizenzrechtliche Probleme für die mit der GPL verbundene Offenlegung des erweiterten Quellcodes bestehen nicht. Gemäß Abschnitt 6.1.3 ist der Host mit Erweiterungen versehen, um die Kommunikation über CAN zu realisieren. Über Funktionen aus der libpcan ist der Zugriff auf die zusätzliche Hardware gegeben. Als Vermittler zwischen avrdude und entsprechenden Funktionen wird eine Instanz von COSMIC gewählt. Auf diese Weise kann eine explizite Unterstützung des P/S-Modells durch avrdude entfallen. Für die Kommunikation verwendet die Programmiersoftware die von COSMIC angebotenen Schnittstellen. Damit ist sie lediglich ein weiterer Nutzer von COSMIC. Sie braucht sich nicht mit Details der Kommunikation beschäftigen und kann aufgrund der gebotenen Transparenz unverändert auf anderen Schnittstellen oder Plattformen zum Einsatz kommen.

6.4.1. Erweiterung der Programmiersoftware

Aus Sicht des Benutzers bietet eine Erweiterung von avrdude den Vorteil, bekannte Vorgehensweisen fortzusetzen. Wie in Abschnitt 6.1.2 beschrieben, ist avrdude im Entwicklungsprozess bereits als Programmiersoftware im Einsatz. Bleibt die Schnittstelle

zum Benutzer auch mit Unterstützung der Rekonfiguration über COSMIC stabil, fällt eine Anpassung der gewohnten Abläufe minimal aus.

Für den Entwurf von Vorteil ist die von avrdude bereit gestellte Umgebung. So muss sich nicht mit Details der Interpretierung von Eingabeformaten auseinander gesetzt werden, wie sie in Abschnitt 3.1 aufgeführt sind. Für die Implementation einer neuen Methode zur Rekonfiguration sind ausschließlich bereits vorgegebene Schnittstellen zu implementieren. Abbildung 6.9 gibt einen Überblick. Jede Implementierung einer

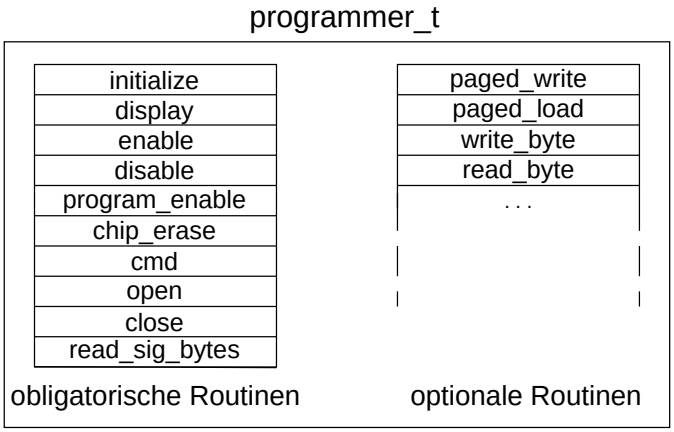


Abbildung 6.9: Gegebene Schnittstellen von avrdude

Rekonfiguration muss festgelegte Routinen beinhalten. Sie dienen dem von avrdude vordefinierten Ablauf einer Rekonfiguration. Die optionalen Routinen bieten einzelnen Implementierungen die Möglichkeit, individuelle Eigenschaften bekannt zu geben. Alle Routinen werden von avrdude in einem Konstrukt verwaltet.

Für die verschiedenen Konzepte aus Abschnitt 5 wird sich für jeweils unabhängige Entwürfe entschieden. So können Eigenheiten trotz ähnlicher Abläufe detailliert berücksichtigt werden. Inhaltlich setzt jeder Entwurf die von COSMIC angebotenen Schnittstellen im von avrdude vorgegebenen Rahmen um. Abbildung 6.10 stellt die Verwendung der Schnittstellen schematisch dar. Als Beispiel dient das Konzept zur kompletten Rekonfiguration aus Abschnitt 5.1. Da kein Rückkanal vorgesehen ist, braucht lediglich ein einzelner Ereigniskanal alloziert und annonciert werden. Anschließend kann das Publizieren von Ereignissen beginnen. Mit dem Veröffentlichen von Ereignissen wird der Programmcode übertragen. Nach Abschluss der Bearbeitung erfolgt die Freigabe des Ereigniskanals. Für das beschriebene Vorgehen müssen die Routinen von COSMIC zur Verfügung stehen. Eine einfache Möglichkeit besteht in der Bereitstellung als statische Bibliothek. Aus ihr werden beim Übersetzen von avrdude die erforderlichen Module hinzu gelinkt.

Nachteilig ist die strenge Orientierung von avrdude am klassischem ISP. Falls nicht anders vom Benutzer angegeben, wird ein *Verify* durchgeführt, welches die installierten Daten mit dem Original vergleicht. Hinzu kommen Sicherheitsüberprüfungen zur Signatur und zu den Einstellungen des Mikrocontrollers. Bereits Abbildung 6.10 lässt die Diskrepanz mit den Konzepten aus Abschnitt 5 vermuten. Keines der Konzepte ist

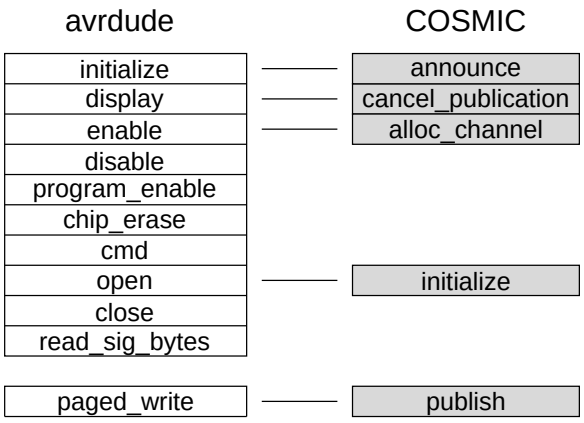


Abbildung 6.10: Verwendung der Schnittstellen von COSMIC in avrdude

für die aufgezählten Funktionen vorgesehen. Die Sicherheit der Rekonfiguration liegt in den Händen von COSMIC und den Routinen zum Beschreiben des Befehlsspeichers. Eine Anpassung von avrdude an die geänderten Voraussetzungen ist erforderlich.

Eine weitere Voraussetzung ist das in Abschnitt 6.2.3 beschriebene Fehlen einer Flusskontrolle. Avrdude als Sender von Ereignissen muss die Dauer des Empfangs berücksichtigen. Zu diesem Zweck wird der Versand von Ereignissen durch Vorgabe einer Zeit künstlich verzögert. Genaue Angaben zur Empfangsdauer können nur abgeschätzt werden. Unnötige Wartezeiten sind die Folge. Kommen neue Empfänger mit anderen Eigenschaften hinzu, wird eine erneute Schätzung der Verzögerung notwendig.

Für die Durchführung einer Rekonfiguration von Bedeutung ist die Eigenheit von avrdude, zu übertragende Daten auf Adresse Null zu beziehen. Die Daten werden von dort bis zum Ende des Programmcodes betrachtet und verbreitet. Grund ist die Annahme, dass bei der Übertragung keine Adressierung erfolgt. Beginnt der Programmcode erst an einer späteren Adresse, ruft avrdude die Übertragungsroutinen der jeweiligen Implementation dennoch mit einem Start bei Adresse Null auf. Sämtliche Daten vor dem Programmcode sind zwar ohne Bedeutung, werden aber als Teil der Rekonfiguration angesehen. Für das Konzept der kompletten Ersetzung aus Abschnitt 5.1 ist dies ohne Bedeutung. Hier können existierende Speicherinhalte auch mit bedeutungslosen Daten überschrieben werden. Anders verhält es sich im Konzept der funktionalen Ersetzung aus Abschnitt 5.2. Dort dürfen zusätzliche Daten kein Teil der Rekonfiguration sein, da sie sonst benötigte Inhalte im Befehlsspeicher ersetzen.

Durch Bekanntgabe der Startadresse des Programmcodes während der Rekonfiguration wird es den Mikrocontrollern möglich, lediglich hinzugefügte und somit unbedeutende Daten zu erkennen und zu ignorieren. Eine selbstständige Erarbeitung der Startadresse durch avrdude erfordert weitreichende Änderungen. Weitaus einfacher und flexibler ist dagegen die Vorgabe durch den Nutzer. Im Rahmen der Einstellungen von avrdude kann der entsprechende Wert dauerhaft hinterlegt werden. Eventuell notwendige Änderungen beschränken sich auf das Editieren der Einstellungen von avrdude. Wie beschrieben, ist eine Spezifizierung der Startadresse nur für das Vorgehen der funktionalen Ersetzung notwendig.

Durch das Verhalten von avrdude geht auch jede Möglichkeit zur Reduzierung der übertragenen Datenmenge verloren. Auch bei der Übertragung nur eines Bestandteils entsprechend Abschnitt 5.2 werden stets die Daten von Beginn des Festspeichers bis zur letzten Adresse des Programmcodes verbreitet. Mit einer weiteren Anpassung von avrdude kann eine derartige Vergeudung von Ressourcen verhindert werden. Wie beschrieben, wird jede von avrdude unterstützte Methode der Rekonfiguration nicht über die Startadresse des Programmcodes informiert. Mit Vorgabe der Startadresse können vor dem Programmcode befindliche Daten bereits vor der Übertragung erkannt und ignoriert werden.

Die Entscheidung für die Spezifizierung der Startadresse fällt mit Blick auf die ebenfalls manuell zu tätigenden Vorgaben aus Abschnitt 6.3.2. Wie auch dort besteht durch das Vorgehen die Gefahr, durch fehlerhafte Angaben einen inkonsistenten Zustand im Befehlsspeicher der Mikrocontroller zu erzeugen. Geschieht dies, ist der Mikrocontroller von weiteren Rekonfigurationen ausgeschlossen und muss durch ein anderes Verfahren mit einem funktionierenden Grundsystem versehen werden. Entsprechend sorgfältig ist die Einstellung von avrdude zu wählen und zu prüfen.

6.4.2. Anpassung von COSMIC

Mit der Vorlage des Entwurfs von COSMIC aus Abschnitt 6.2 sind nur geringe Anpassungen für den Host notwendig. Die Unabhängigkeit der einzelnen Schichten erlauben ein einfaches Vorgehen mit begrenzten Auswirkungen.

Nahe liegend ist der Entwurf eines speziellen Network Layers für die auf dem Host vorhandenen Adapter für CAN. Aufgrund der Unterstützung durch die libpcan braucht der Network Layer lediglich die Rolle eines Vermittlers zwischen den Schnittstellen dieser Bibliothek und dem darüber liegenden Abstract Network Layer einnehmen.

Bedingt durch die vorliegende Unterstützung für die Hardware, ist zusätzlich eine Anpassung des Abstract Network Layers erforderlich. Er muss, da keine Interrupts signalisiert werden können, die Hardware auf das Vorliegen von Nachrichten überprüfen. Das Pollen darf nicht die Ausführung von COSMIC oder der aufrufenden Anwendung blockieren. Eine nebenläufige Ausführung ist notwendig. Die unter Linux verfügbare Implementation der POSIX Threads (*Pthreads*) [40] bietet dafür eine Möglichkeit. Indem die Abfrage auf neue Nachrichten in einen eigenen Thread verlagert wird, kann die Ausführung von COSMIC bzw. der Anwendung ungehindert erfolgen.

6.5. Entwurf des Ereignisprotokolls

Die Daten, welche im Verlauf einer Rekonfiguration übertragen werden, unterscheiden sich für jedes der zu entwerfenden Konzepte. Das in Abschnitt 5.1 beschriebene Konzept der kompletten Ersetzung beschränkt sich auf den Programmcode. Im Konzept aus Abschnitt 5.2 kann eine weitere Information hinzukommen, welche die Adresse des zu ersetzenden funktionalen Bestandteils darstellt. Das zusätzlich betrachtete Konzept aus

Abschnitt 5.3.3 zur Rekonfiguration von Modulen erfordert mehrere Informationen. Zusätzlich zum Programmcode, werden Anfragen der Mikrocontroller und die Antworten des Hosts übertragen. In den einzelnen Konzepten stellen die neben dem Programmcode übertragenen Daten die Metadaten der Kommunikation dar.

Bereits Abschnitt 5.4 beschreibt, dass Informationen allein über Ereignisse verbreitet werden. Wie jedoch im vorigem Abschnitt 6.2.3 festgestellt wird, sind der Nutzlast von Ereignissen Grenzen gesetzt. Wie ebenfalls festgestellt wird, ist die Nutzlast im Allgemeinen nicht für den Programmcode ausreichend. Jeder Vorschlag zur Rekonfiguration baut auf einer Übertragung von Programmcode auf. Für alle Konzepte ist daher der Entwurf eines Protokolls notwendig, mit dessen Hilfe Daten durch mehrere Ereignisse ausgedrückt werden können. Für die eingangs erwähnten Metadaten der Konzepte kann ein begrenzter Speicherbedarf angenommen werden. Bleibt der Bedarf unter dem maximalen Platzangebot von Ereignissen, kann auf den Einsatz des Protokolls bei der Übertragung dieser Daten verzichtet werden.

6.5.1. Fragmentierung des Programmcodes

Die Anforderungen an die Fragmentierung des Programmcodes entsprechen denen des Fragmentierungsprotokolls aus Abschnitt 6.2.3. Erneut müssen mehr Daten übertragen werden als an Nutzlast zur Verfügung steht. Anstelle des Kommunikationsprotokolls sind es nun die Eigenschaften der verbreiteten Ereignisse, welche die Anforderungen vorgeben. Indem ein Protokoll befolgt wird, lassen sich größere Datenmengen durch mehrere Ereignisse übertragen. Der Programmcode ist aufgrund seines Umfangs eine derartige Datenmenge.

Unter zwei Bedingungen kann auf ein Protokoll verzichtet werden:

1. Es geht kein Ereignis verloren.

2. Die Ereignisse werden in der Reihenfolge empfangen, in der sie gesendet werden.

Es ist Aufgabe von COSMIC, die Übertragung von Ereignissen sicherzustellen. Gehen Ereignisse verloren, hat COSMIC die beteiligten Anwendungen in Kenntnis zu setzen. Für ein Verfahren, mit dem die Middleware Verluste erkennt, liegen bisher keine Entwicklungen vor. Im Falle von fragmentiert übertragenen Ereignissen ist eine Auswertung der Sequenznummern denkbar. Bei Ereignissen, welche nicht fragmentiert übertragen werden, besteht diese Möglichkeit hingegen nicht. Die Entwicklung und Realisierung entsprechender Vorgehensweisen ist jedoch keine Aufgabe der vorliegenden Arbeit. Daher wird sich auf ein Erkennen von fehlenden Fragmenten während der Rekonfiguration beschränkt. Im Falle eines Fehlers erfolgt der Abbruch. Auf ein Protokoll zur Behandlung von verlorenen Ereignissen wird somit verzichtet. Die Reihenfolge der Ereignisse ist ebenfalls nicht gewährleistet. Wie in Abschnitt 6.2.1 beschrieben, wird auf die zeitliche Vorhersehbarkeit von Ereignissen verzichtet. Obwohl das Konzept von COSMIC eine derartige qualitative Anforderung explizit vorsieht, sind dafür weder ein Entwurf noch eine Implementation bekannt. Für die Zwecke der Rekonfiguration wird der Aufwand zur

Realisierung von qualitativen Anforderungen zudem als unverhältnismäßig angesehen. Mit einem Protokoll auf Anwendungsebene bietet sich eine einfache Alternative.

Wie beschrieben, kann sich lediglich die Reihenfolge des Empfangs von der des Sendens unterscheiden. Ereignisverluste müssen durch COSMIC signalisiert werden. Für die Beschreibung von Daten durch mehrere Ereignisse bedeutet der Verlust eines Fragments bzw. Ereignisses eine fehlerhafte Übertragung. Zusätzliche, über die Aufgabe dieser Arbeit hinausgehende Protokolle sind zur Korrektur notwendig. Im folgenden wird daher von der Atomarität der Übertragung ausgegangen. Entweder alle Ereignisse werden korrekt übertragen oder als Reaktion auf den Verlust eines Ereignisses verworfen.

Analog zum Vorgehen aus Abschnitt 6.2.3 wird sich für die Verwendung von Sequenznummern entschieden. Eine sequentielle Nummerierung der Ereignisse lässt sich einfach realisieren. Die Reihenfolge der Ereignisse ist leicht zu erkennen und erfordert daher nur geringen Bearbeitungsaufwand. Zudem sind für das Verfahren nur wenige zusätzliche Daten zu übertragen. Abbildung 6.11 skizziert die Zerlegung von Daten in Ereignisse. In

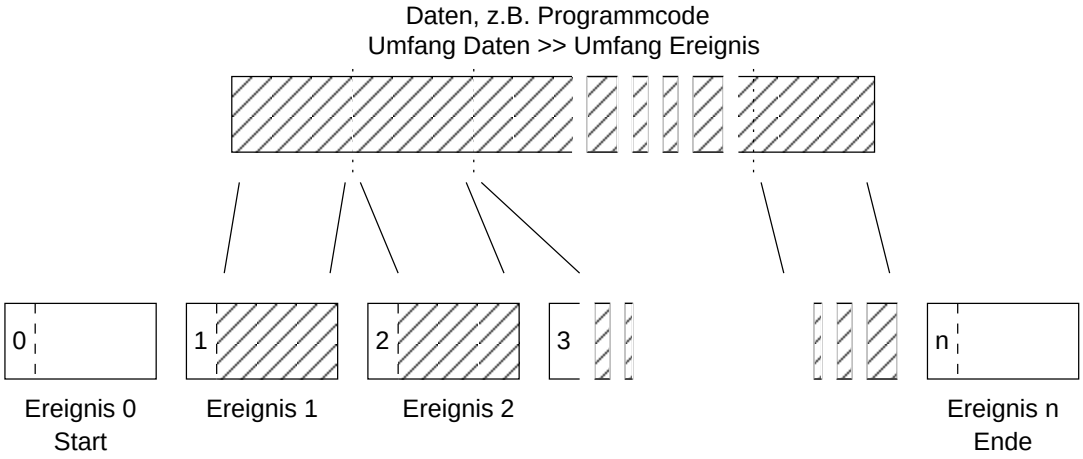


Abbildung 6.11: Fragmentierung von Daten durch Ereignisse

der Abbildung dargestellt ist der Inhalt der Ereignisse. Bestandteil eines jeden Ereignisses ist die Sequenznummer. Abgesehen von zwei Ausnahmen, stellen die zu übertragenden Daten den restlichen Inhalt dar. Damit fehlt die Zuordnung eines Fragments zu den übrigen Daten. Dies kann zu Problemen führen, falls mehrere Anwendungen fragmentierte Daten publizieren. Im Falle von gleichen Subjects kann ein Empfänger die Fragmente nicht identifizieren. Wie schon der Entwurf aus Abschnitt 6.2.3, gilt somit auch dieses Protokoll nur unter einer gegebenen Voraussetzung: Zu jedem Zeitpunkt dürfen unter einem Subject nur von genau einer Anwendung fragmentierte Daten in Form von Ereignissen verbreitet werden. Um die Einschränkung zu vermeiden, ist eine Zuordnung des Fragments bzw. des Ereignisses zu den gesamten Daten erforderlich. Im verteilten System sind dazu jederzeit eindeutige Bezeichner notwendig. Deren Verwaltung ist mit weiterem Aufwand verbunden. Für die Rekonfiguration im Sinne dieser Arbeit stellt die Bedingung hingegen keine Einschränkung dar. Als einzige Anwendung verbreitet die auf dem Host laufende Anwendung umfangreichere Daten mit Hilfe von verketteten Ereignissen.

Wie aus Abbildung 6.11 ersichtlich, nehmen zwei Ereignisse eine Sonderstellung ein:

- Das zuerst vom Publisher verbreitete Ereignis kündigt die zu übertragenden Daten an. Auch dieses Ereignis kann bereits ein Fragment der Daten beinhalten. Unter Umständen sind für eine Verarbeitung des Datenstroms aber beschreibende Informationen notwendig. Aus diesem Grunde bleibt das restliche Platzangebot des Ereignisses zunächst ungenutzt. Abgesehen von der Bearbeitung eines zusätzlichen Ereignisses, stellt der Verzicht keinen Nachteil dar. Durch das Fehlen von Inhalt brauchen auch weniger Daten übertragen werden.

- Sind alle Daten in Form von Ereignissen versendet worden, signalisiert ein zusätzliches Ereignis den Abschluss der Daten. Da das letzte Datenfragment möglicherweise exakt ein Ereignis ausfüllt, ist eine explizite Kennzeichnung der finalen Sequenznummer erforderlich. Eine Alternative besteht in der Übermittlung der Anzahl nachfolgender Ereignisse mit dem ersten Ereignis. Dieses Vorgehen wird schon bei der Fragmentierung auf Ebene des Kommunikationsprotokolls gemäß Abschnitt 6.2.3 angewendet. Wie beschrieben, ist das Startereignis aber anderen Zwecken vorbehalten. Hinzu kommt der bereits erläuterte geringe Aufwand für ein zusätzliches Ereignis. Daher wird der einheitlichen Gestaltung aller Ereignisse der Vorrang gegeben.

Um ein Vorliegen aller Ereignisse zu erkennen, muss ein Empfänger die Sequenznummern registrieren. Bereits Abschnitt 6.2.3 erläutert die Verwendung eines Bitvektors für diesen Zweck. Sind noch nicht alle Ereignisse übertragen worden, muss ein Empfänger die Bearbeitung der Daten zunächst zurückstellen. Die einzigen Möglichkeiten, diesen Zustand zu beenden, sind der Empfang der ausstehenden Ereignisse oder das Signal für den Verlust eines Ereignisses. Letzteres resultiert, wie beschrieben, im Verwerfen aller bereits empfangenen Fragmente.

6.5.2. Struktur der Metadaten

Abschnitt 6.5 erläutert die notwendigen Metadaten der einzelnen Konzepte. Für das Konzept der kompletten Ersetzung kann auf die Übertragung von Metadaten verzichtet werden. Dagegen kommen bei den anderen Konzepte verschiedenste Metadaten zum Einsatz.

Im Konzept zur Ersetzung funktionaler Bestandteile muss die Übertragung einer Adresse vorbereitet werden. Der Entwurf für den Mikrocontroller kann auf der Übertragung der Zieladresse des Programmcodes bestehen. Auf diese Weise braucht die Adresse nicht schon zum Zeitpunkt des Übersetzens bekannt sein. Das im vorigen Abschnitt 6.5.1 entworfene Übertragungsprotokoll macht die Übertragung der Adresse einfach. Das erste Ereignis stößt bei weitem noch nicht an die maximale Größe. Die Zieladresse kann sich unmittelbar an die Sequenznummer anschließen. Für den Mikrocontroller AT90CAN128 ist die Übermittlung einer 32-Bit-Adresse stets ausreichend. Wird der Festspeicher von 128 Kilobyte entsprechend Abschnitt 5.1.3 halbiert, genügt auch eine Adresse von 16 Bit Breite. Aus Gründen der Flexibilität, beispielsweise für eine andere

Aufteilung, wird auf die Betrachtung dieses Spezialfalls verzichtet. Mit 32 Bit ist die übertragene Adresse für die Mehrzahl der derzeit eingesetzten Rechnersysteme zutreffend. Die Plattformunabhängigkeit des Konzepts bleibt somit berücksichtigt.

Die Kommunikation im Konzept der modularen Ersetzung aus Abschnitt 5.3.3 bietet mehrere Besonderheiten, die einen ungleich höheren Aufwand erfordern.

1. Die verbreiteten Module müssen identifizierbar sein.
   Im Konzept beziehen sich die übertragenen Relokationsinformationen auf Module. Für ihre Zuordnung ist eine Identifizierung der Module notwendig. Zusätzlich müssen Host und Mikrocontroller die gleichen Bezeichner verwenden. Indem der Host die Bezeichner vorgibt, ist die Übereinstimmung sichergestellt. Die Übertragung der Module erfolgt mit dem Protokoll aus Abschnitt 6.5.1. Für die Übermittlung der Bezeichner nahe liegend ist die Nutzung des Startereignisses des Protokolls.

   Auf dem Host setzt sich die Kennzeichnung eines Moduls üblicherweise aus dem Pfad und aus dem Dateinamen zusammen. Für die Übertragung und Bearbeitung auf den Mikrocontrollern ist eine derartige Kennzeichnung von Nachteil. Pfad- und Dateiname übersteigen leicht das Platzangebot im Startereignis. Im Mikrocontroller erfordert ein derartiger Bezeichner viel Speicher. Weiterhin müssen im Linkprozess häufig Bezeichner ermittelt werden. Dazu sind Vergleichsoperationen notwendig. Die beschränkte Rechenleistung der Mikrocontroller lassen für lange Bezeichner eine schlechte Performance erwarten. Ein abstrakter Bezeichner ist für das Linken der Module ausreichend. Geeignet erscheint ein Hashwert des Pfads und des Dateinamens. Da ein Hashwert einen definierten und im Allgemeinen weitaus kürzeren Umfang als seine Eingabe besitzt, werden die Anforderungen der Kommunikation und der Mikrocontroller berücksichtigt. In der Kommunikation mit den Mikrocontrollern verwendet der Host stets den Hashwert zur Bezeichnung der Module. Die Mikrocontroller erfahren das Äquivalent auf dem Host nicht. Für sie genügt der abstrakte Bezeichner eines Moduls. Nachteilig ist dies für die Zwecke des Debugging.

   Für die in Abschnitt 6.1.2 beschriebene Umgebung des Hosts ist der Einsatz der Hashfunktion *MD5* [36] geläufig. Der Bezeichner erscheint mit 128 Bit bzw. 16 Byte umfangreich. Jedoch ist kein gebräuchliches Verfahren bekannt, welches einen kürzeren Wert liefert. Im Gegensatz zu umfangreichen Pfad- und Dateinamen passen 128 Bit stets in ein Startereignis. Mit *OpenSSL* [41] besteht eine etablierte Bibliothek, welche die Berechnung der Hashwerte anbietet.

2. Der Host versendet mehrere Blöcke von Daten.
   Anders als in den anderen Konzepten, können sich an einem empfangenen Datenblock noch weitere Daten anschließen. Erst nach der Übermittlung aller Module liegen die Daten der Rekonfiguration vollständig vor. Das Ende der Übertragungen ist mit dem bisherigen Vorgehen aber nicht ersichtlich. Die Empfänger sind auf Mitteilungen des Hosts angewiesen, um den Stand der Datenübertragung beurteilen zu können. Hinreichend ist die Bekanntgabe der Anzahl der Module. Eine

Übermittlung am Beginn der Übertragungen hat den Vorteil, die Empfänger explizit auf eine Rekonfiguration vorzubereiten. Die Bekanntgabe dient in diesem Sinne als Startsignal einer Rekonfiguration. Mit dem Wissen über die Anzahl kann für jedes empfangene Modul entschieden werden, ob weitere Module folgen. Liegen alle Module vor, ist die Rekonfiguration abzuschließen.

Analog zum Vorgehen aus Punkt 1, ist die Bekanntgabe im Startereignis des zuerst übertragenen Moduls möglich. Dies spart den Entwurf und Aufwand für ein separates Ereignis. Gleichzeitig erfolgt eine Abhängigkeit vom Übertragungsprotokoll der Module, da eine feste Struktur der Startereignisse vorausgesetzt werden muss. Variiert die Struktur, können andere Daten irrtümlich als Anzahl der Module interpretiert werden. Unter der Annahme, dass alle Startereignisse der Module gleich strukturiert sind, signalisiert eine zusätzliche Angabe das Startereignis einer Rekonfiguration. Wird weiterhin ein sequentieller Empfang der Module vorausgesetzt, erzwingt dieses Ereignis das Verwerfen etwaiger Module aus laufenden Rekonfigurationen.

3. Die Kommunikation erfolgt in beiden Richtungen.
   In Reaktion auf Anfragen verbreitet der Host auch Relokationsinformationen. Die Anfragen werden durch die Mikrocontroller publiziert. Der Entwurf beider Datenstrukturen kann ohne zusätzlichen Aufwand erfolgen. Gemäß Abschnitt 5.3.3 beinhalten die Anfragen der Mikrocontroller lediglich eine Sequenznummer. Diese lässt sich auch für eine hohe Anzahl an Relokationsinformationen in einem Ereignis unterbringen.

   Ebenfalls in Abschnitt 5.3.3 beschrieben ist der Inhalt der Antworten des Hosts. Sowohl die referenzierte als auch die referenzierende Adresse passen selbst bei hohen Wortbreiten in ein Ereignis. Die Bezeichnung der Module geht einher mit der Identifizierung gemäß Punkt 1. Entsprechend erfolgt die Bezeichnung der Module mit einem Hashwert anstelle eines Namens. Mit den Relokationsinformationen sind die Bezeichner von zwei Modulen zu übertragen. In Punkt 1 angeführt ist der Umfang eines Bezeichners von 128 Bit. Die Übertragung von 256 Bit sowie der zwei Adressen ist innerhalb eines Ereignisses gegeben.

Alle Konzepte basieren auf der Übertragung eines vergleichsweise umfangreichen Programmcodes durch mehrere Ereignisse. Aus Sicht der Rekonfiguration ist ein beliebiger Umfang der Ereignisse von Nachteil. Ein konstanter Faktor, mit dessen Hilfe bereits aus der Sequenznummer auf eine konkrete Adresse geschlossen werden kann, ist nicht mehr vorhanden. Die Sequenznummer ist somit für die Platzierung eines Ereignisses ohne Bedeutung. Wird auf die explizite Übermittlung einer Adresse verzichtet, lässt sich die Position allein aus der Platzierung des vorhergehenden Ereignisses ermitteln. Im Falle einer beliebigen Reihenfolge des Empfangs müssen somit Ereignisse gegebenenfalls zwischengespeichert werden. Erst nachdem alle vorhergehenden Ereignisse eingetroffen sind, kann ein Verschieben an die korrekte Position erfolgen. Mit der Festlegung der Ereignisse auf stets gleiche Größen vereinfacht sich das Vorgehen entscheidend. Unter Verwendung der Sequenznummer kann umgehend die Position eines Ereignisses bestimmt werden. Wird

im Startereignis einer Übertragung der Umfang aller nachfolgenden Ereignisse angegeben, ist eine Berücksichtigung sogar zur Laufzeit möglich. Vorauszusetzen ist jedoch, dass das Startereignis vor allen anderen Ereignissen empfangen wird. Daher bleibt dieses Vorgehen zunächst unberücksichtigt. Sender und Empfänger von Programmcode gehen von einem definierten Umfang der Ereignisse aus. Das Hinzufügen einer weiteren Information in das Startereignis erfolgt nicht.

7. Evaluation

Bestandteil einer jeden Entwicklung ist das Überprüfen der Ergebnisse. Mit einer Evaluation kann zugleich ein Eindruck in die Funktionalität gegeben werden. Die zur Anwendung kommenden Parameter und Ergebnisse der Entwicklung sind ebenso zu betrachten wie das genutzte Szenario.

7.1. Szenario und Parameter

Wesentliche Bestandteile des Szenarios sind bereits in den Umgebungsbedingungen des Entwurfs aus Abschnitt 6.1 genannt. Ein handelsüblicher Personalcomputer stellt den Host. Die vorgestellten Boards mit einem Mikrocontroller des Typs AT90CAN128 repräsentieren die Zielsysteme der Rekonfiguration. Verfügbar sind insgesamt sieben Boards.

Aufgrund der zentralen Funktion besitzen die Parameter von COSMIC bedeutenden Einfluss. Tabelle 7.1 listet die in der Implementation verwendeten Parameter auf. Die

Parameter	**Wert**
verwendete Schnittstelle	CAN
Bitrate	250kBit/s
maximale Anzahl der Ereigniskanäle	4
maximale Anzahl unterschiedlicher Subjects	4
maximale Größe eines Ereignisses	257 Byte
Zugleich fragmentiert übertragbare Ereignisse	1

Tabelle 7.1: Parameter von COSMIC

Vorgaben zu den Ereigniskanälen stehen im direkten Zusammenhang mit dem Speicherbedarf und der Bearbeitungszeit der Implementation. Mit zunehmender Anzahl der zu verwaltenden Subjects erhöht sich der Aufwand an Rechenzeit und Speicher. Eine Vereinfachung besteht in Szenarien, in denen mehrere Ereigniskanäle gleiche Subjects verwenden. Für den allgemeinen Fall einer eineindeutigen Zuordnung hat ein Zuwachs an Ereigniskanälen aber ebenfalls einen höheren Aufwand zur Folge. Jedes der zwei implementierten Konzepte verwendet ein dediziertes Subject und einen daran gebundenen Ereigniskanal. Für die Zwecke der Evaluation sind zwei weitere Subjects bzw. Kanäle vorgesehen. Die zur Übertragung von fragmentierten Ereignissen notwendige Reservierung von Speicher ist bereits in Abschnitt 6.2.3 erläutert. Mit Blick auf den verfügbaren nicht flüchtigen Speicher von 4Kilobyte ist lediglich Platz für genau ein derartiges Ereignis reserviert. Der Umfang des Ereignisses realisiert die Übertragung einer kompletten

Page des Festspeichers. Zu den 256 Byte kommt ein Byte für die Sequenznummer hinzu, welches in der Fragmentierung des Programmcodes nach Abschnitt 6.5.1 Verwendung findet. Die Sequenznummern für die Fragmentierung durch COSMIC gemäß Abschnitt 6.2.3 sind keine Bestandteile eines Ereignisses. Sie dienen lediglich der Beschreibung eines ausgewählten Fragments eines Ereignisses und sind somit den Metadaten zuzurechnen. Die Signalisierung von Ereignissen erfolgt auf Host und Mikrocontroller unterschiedlich. Wie in Abschnitt 6.4.2 beschrieben, erfolgt die Benachrichtigung auf dem Host durch einen Thread. Im Gegensatz dazu erfolgt der Empfang auf den Mikrocontrollern allein in Bearbeitung eines Interrupts.

In Abschnitt 6 ist der Einsatz des Zwischenpuffers im Entwurf aufgeführt. Die Dimensionierung des Zwischenpuffers kann nicht vollständig nach Abschnitt 5.1.3 erfolgen. Für den Fall, dass sich der Zwischenpuffer über die Grenze von 64 Kilobyte erstreckt, treten beim Verschieben des Puffers einzelne fehlerhafte Bits auf. Vermutet werden Probleme durch verschiedene Adressierungen, die vom AT90CAN128 mit seinen lediglich 16 Adressleitungen angewandt werden. Beginnt der Zwischenpuffers bei der Hälfte des Festspeichers, tritt der beschriebene Fall ein. Die Startadresse ist mit 0xF000 bzw. 61440 unterhalb, der Großteil des Zwischenpuffers oberhalb von 64 Kilobyte. Der derzeit beschrittene Weg vermeidet das Problem und lässt den Zwischenpuffer bei Adresse 65536 bzw. 0x10000 beginnen. Durch die BLS, welche kein Bestandteil des Zwischenpuffers ist, verringert sich damit der maximale Umfang einer Rekonfiguration. Mit der verwendeten Größe der BLS von acht Kilobyte beträgt der Umfang insgesamt $0\mathrm{x}1\mathrm{E}000 - 0\mathrm{x}10000 = 0\mathrm{xE}000$ bzw. 57344 Byte. Eine Erweiterung des Zwischenpuffers durch eine Reduzierung der BLS ist aufgrund deren begrenzten Inhalts jedoch möglich. Im Bedarfsfall lässt sich so eine annähernde Halbierung des Festspeichers erreichen.

7.2. Statische Kennwerte

Je nach verwendetem Rechnersystem kommen unterschiedliche Entwicklungen zum Einsatz.

Auf dem Host liegt eine mit dem gcc übersetzte Implementation von COSMIC vor. Da keine Größenbeschränkungen vorliegen, wird die Übersetzung mit mittlerer Optimierung durchgeführt (Option *-O2*). Beschränkungen bestehen dagegen auf dem Mikrocontroller. Entsprechend wird der Compiler aufgefordert, die Übersetzung auf geringen Platzbedarf zu optimieren (Option *-Os*). Auf beiden Systemen kommt im wesentlichen die gleiche Implementierung der Middleware zum Einsatz. Sie ist in der Programmiersprache C geschrieben. Mit den vorhandenen Übersetzern ist die einfache Portierung auf das jeweilige System möglich. Bestehende funktionelle Unterschiede sind bereits in Abschnitt 6.4.2 aufgeführt.

Die Rekonfiguration erfolgt durch die Zusammenarbeit von verschiedenen Anwendungen. Im Zuge einer Rekonfiguration kommt auf dem Host die Programmiersoftware avrdude zur Ausführung. Die in Abschnitt 6.4.1 entworfenen Ergänzungen versetzen avrdude in die Lage, als gewöhnliche Anwendung von COSMIC aufzutreten. Auf Seiten der Mikrocontroller laufen die unter der Bezeichnung *reCOS* zusammengefassten

Komponenten der Rekonfiguration. Gebunden mit den Routinen von COSMIC, stellen sie das Grundsystem dar.

Tabelle 7.2 listet den Speicherbedarf der Implementationen auf dem Host auf. Aufgrund

Rechnersystem	**Komponente**	**Umfang**
Host	COSMIC	Instruktionen 8337 Byte Daten 824 Byte
	avrdude	
	unverändert	Instruktionen 184358 Byte Daten 21720 Byte
	erweitert	Instruktionen 198717 Byte Daten 22644 Byte
	Differenz	Instruktionen +14359 Byte Daten +924 Byte

Tabelle 7.2: Umfang der Implementation auf dem Host

der vorhandenen Ressourcen des Host dient die Auflistung allein dem Zweck, eine Abschätzung der getätigten Implementationen zu ermöglichen. Die Umsetzung von COSMIC ist, wie in Abschnitt 6.4.1 beschrieben, gegen avrdude in Form einer Bibliothek gelinkt. Sie ist der umfangreichste Bestandteil der Erweiterung. Die Routinen zur vollständigen Ersetzung nach Abschnitt 5.1 umfassen 2464 Byte, die der funktionellen Ersetzung aus Abschnitt 5.2 beanspruchen 2576 Byte für Instruktionen. Die restliche Differenz von wenigen hundert Byte entfällt auf Änderungen in avrdude selbst, beispielsweise zum Parsen hinzu gekommener Einstellungen.

Für den Einsatz auf den Mikrocontrollern besitzt der Speicherbedarf einen ungleich höheren Stellenwert. Eine Auflistung für die einzelnen Komponenten der Rekonfiguration gibt Tabelle 7.3. In die Addition der Bestandteile gehen die Instruktionen der Laufzeitumgebung implizit mit ein. Der Mikrocontroller stellt für Daten insgesamt 4 Kilobyte flüchtigen Speicher zur Verfügung. Annähernd ein Fünftel wird durch die Implementation beansprucht. Daran hat der reservierte Speicher für ein maximales Ereignis mit 257 Byte großen Anteil. Trotz der Umleitung aller Interrupts fällt der Speicherbedarf für die verfahrensunabhängigen Routinen gering aus. Im normalen Vorgehen erzeugt der avr-gcc weitaus umfangreicheren Code. Bedingt durch den zur Umleitung notwendigen Aufruf einer Routine, werden sämtliche Register gesichert und wiederhergestellt. Bei insgesamt 36 möglichen Interrupts des AT90CAN128 und der Befehlsbreite von 16 Bit erfordern diese Schritte bereits mehr als 2 Kilobyte Speicher. Sämtliche Schritte sind aber für jeden Interrupt gleich. Daher werden sie im Rahmen einer eigens entwickelten Optimierung zentral vorgehalten. Die Bearbeitung eines jeden Interrupts verzichtet demnach zunächst auf umfangreichen Operationen, sondern springt statt dessen an eine zentrale Adresse. Allein der dort vorgehaltene Code realisiert das Sichern und das

Rechnersystem	Komponente	Umfang
Mikrocontroller	COSMIC	Instruktionen 8300 Byte Daten 625 Byte
	reCOS	
	BLS-Routinen	Instruktionen 704 Byte Daten 0 Byte
	vollständiges Ersetzen	Instruktionen 844 Byte Daten 50 Byte
	funktionelles Ersetzen	Instruktionen 1032 Byte Daten 54 Byte
	verfahrens-unabhängige Routinen	Instruktionen 850 Byte Daten 72 Byte
	gesamt	Instruktionen 12044 Byte Daten 813 Byte

Tabelle 7.3: Umfang der Implementation für die Mikrocontroller

Wiederherstellen der Register. Hinzu kommt gegebenenfalls den Aufruf einer Routine, welche sich für die Behandlung des Interrupts registriert hat.

7.3. Dynamisches Verhalten

Eine Bewertung des Verhaltens des implementierten Entwurfs erfolgt mit Bezug auf die Zeit. Für den Nutzer von Interesse ist die Dauer der Rekonfiguration. Der Zeitrahmen beginnt mit dem Aufruf der Programmiersoftware und endet mit der Ausführung des neuen Programmcodes auf den interessierten Mikrocontrollern. Neben den Voraussetzungen der Hardware besitzt die Kommunikation bedeutenden Einfluss auf die Dauer des Vorgehens.

7.3.1. Fragmentierung durch COSMIC

Die in Abschnitt 6.2.3 beschriebene Fragmentierung der Daten verlängert die Dauer der Kommunikation. Zusätzliche Daten werden verbreitet, um ein Zusammensetzen der einzelnen Fragmente zu ermöglichen. Bei nur geringer Nutzlast pro Nachricht nimmt jede Protokollinformation einen hohen Anteil an der Datenübertragung ein. Entsprechend deutlich verringert sich der Anteil der Nutzdaten mit jeder hinzu kommenden Information. Die Kommunikation über CAN ermöglicht, wie in Abschnitt 6.1.3 beschrieben, lediglich acht Byte Nutzlast pro Nachricht. Die Verwendung einzelner Bytes für Protokollinformationen lässt Auswirkungen auf die Dauer der Kommunikation erwarten.

Der Entwurf des Fragmentierungsprotokolls aus Abschnitt 6.2.3 ist mit folgenden Bedingungen umgesetzt:

- Eine Sequenznummer besteht aus acht Bit.
- Der Umfang der Daten wird durch 16 Bit beschrieben.

Die gewählte Wortbreite der Sequenznummer erlaubt maximal 255 Fragmente, da die Sequenznummer Null für die initiale Beschreibung der Fragmentierung reserviert ist. Mit sieben Byte pro Fragment ergibt sich der maximale Umfang der Daten von $255 \cdot 7 = 1785$ Byte. Die 16 Bit zur Beschreibung des Umfangs sind somit ausreichend, eine Darstellung ist bereits durch 11 Bit gegeben. Mit der gewählten Wortbreite unterstützt COSMIC die Übertragung von Ereignissen bis zu einer Größe von 1785 Byte.

Abbildung 7.1 skizziert den Anteil der Daten bei der Übertragung. Bei weniger als

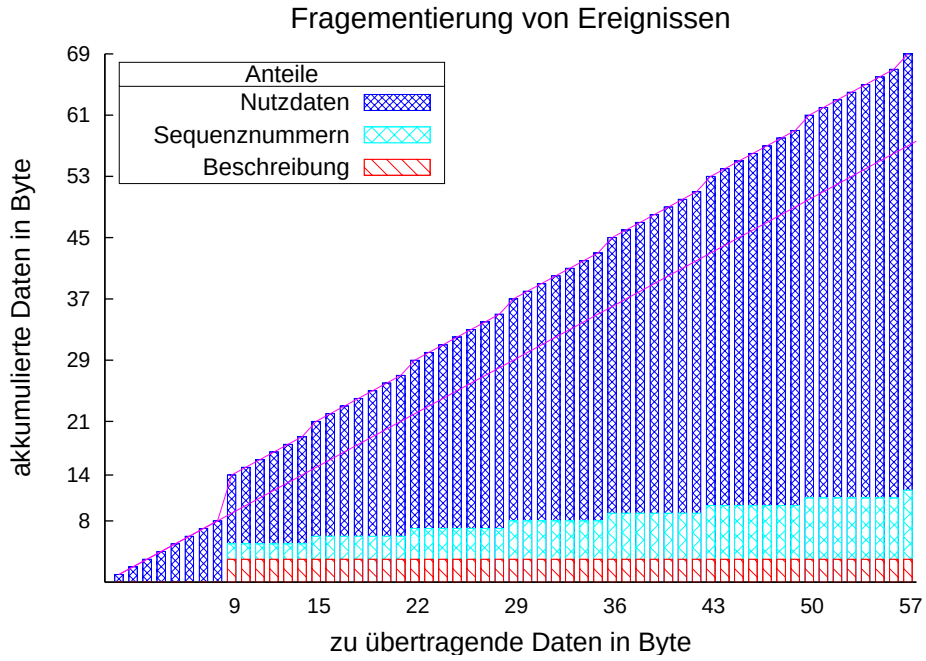


Abbildung 7.1: Anteile der Daten bei der Übertragung

neun Byte ist zunächst keine Fragmentierung erforderlich. Auf zusätzliche Daten kann verzichtet werden. Beim ersten Überschreiten der Nutzlast einer Nachricht liegt zugleich das schlechteste Verhältnis zwischen Protokollinformationen und Nutzdaten vor. Mit der initialen Nachricht, welche die Beschreibung der Fragmentierung beinhaltet, sowie zwei weiteren Nachrichten müssen insgesamt drei Nachrichten verbreitet werden. Nachfolgend verbessert sich das Verhältnis stetig. Nur jedes achte Byte an Nutzdaten erfordert eine weitere Nachricht und somit eine weitere Sequenznummer. Die Sprünge im Datenaufkommen lassen sich somit auf einzelne, dargestellte Mengen an zu übertragenen Daten beziehen. Während die Übertragung von beispielsweise 14 Byte genau drei Nachrichten erfordert, ist bei 15 Byte eine weitere Nachricht notwendig. Sie enthält

die Sequenznummer und lediglich ein Byte an Nutzdaten. Mathematisch lässt sich das Datenaufkommen für $n > 8$ zu übertragene Byte durch Gleichung 7.1 beschreiben.

$$\text{Datenaufkommen} = 3 + 8 \cdot (n/7) + ((n \bmod 8) + (n-1)/7 + 1) \bmod 8 \quad (7.1)$$

Der erste Term der Summe beschreibt den Umfang der initialen Nachricht. Der zweite Term bestimmt die Anzahl an Daten in vollständig gefüllten Nachrichten. Die Nutzlast einer möglicherweise notwendigen, unvollständig gefüllten letzten Nachricht beschreibt der finale Term der Summe. Vorauszusetzen ist die Division ohne Rest. Nicht betrachtet werden weitere Daten, welche mit einer CAN-Nachricht übertragen werden. Header, Checksumme und diverse Steuerbits sind jedoch Bestandteil einer jeden Nachricht. Am Verhältnis der Daten ändert sich nichts, bleiben einzelne Bits zur Synchronisierung unberücksichtigt.

Zur Bestimmung des Einflusses der Fragmentierung werden Messungen der Übertragungsdauer von Ereignissen durchgeführt. Im Rahmen der Messung werden vom Host verschieden große Ereignisse verbreitet. Eine direkte Bestimmung der Latenz, nach der die Ereignisse auf den Mikrocontrollern vorliegen, ist nicht möglich. Dazu bedarf es einer Synchronisierung der Uhren der beteiligten Systeme. Statt dessen wird die *Round-Trip* Zeit von Ereignissen bestimmt. Beim Empfang eines Ereignisses senden die Mikrocontroller umgehend eine Bestätigung in Form eines Ereignisses. Der Host als Urheber des Ereignisses misst die Zeit zwischen Versand und Empfang von Ereignissen. Kann die für eine Antwort notwendige Zeit geschätzt werden, sind Rückschlüsse auf den Zeitrahmen zur Übertragung eines Ereignisses möglich. Da der Umfang einer Antwort stets konstant ist, ist die Schätzung für alle gesendeten Ereignisse gültig. Der systematische Einfluss der Dauer der Antworten ermöglicht auch Aussagen allein durch den Vergleich der Messwerte unterschiedlich großer Ereignisse.

Der Messaufbau besteht aus der in Abschnitt 6.1 beschriebenen Entwicklungsumgebung. Aufgrund der einfachen Handhabung kommt ausschließlich das Clusterboard aus Abbildung 6.2b zum Einsatz. Der Messvorgang beinhaltet die Übertragung von je 1000 Ereignissen eines gegebenen Umfangs. Die gewählten Ereignisgrößen sind bereits in Abbildung 7.1 angedeutet. Gemessen werden die Zeiten für den Versand von neun Byte, 15 Byte, 22 Byte usw.. Zum Vergleich mit nicht fragmentierten Ereignissen erfolgt die Verbreitung von Ereignissen mit einem Umfang von kleiner bzw. gleich acht Byte. Der Inhalt aller Ereignisse ist eine Folgen von Nullen. Bevor der Host ein weiteres Ereignis verbreitet, wartet er die Antworten aller angeschlossenen Mikrocontroller ab. Auf diese Weise kann der Ausfall einzelner Teilnehmer und damit eine Verfälschung des Messergebnisses erkannt werden. Die Mikrocontroller antworten mit einer vordefinierten Nachricht von fünf Byte.

Abbildung 7.2 präsentiert die Mittelwerte der gemessenen Round-Trip Zeiten. Wie ersichtlich, werden verschiedene Anzahlen von Mikrocontrollern betrachtet. Das theoretische Datenaufkommen aus Abbildung 7.1 lässt einen linearer Verlauf erwarten. Umso mehr überrascht die zeitweilig drastische Reduzierung der gemessenen Zeiten, insbesondere bei hohen Teilnehmerzahlen. Zum Vergleich wird eine weitere Messung unter veränderten Parametern vorgenommen. Ihre Ergebnisse stellt Abbildung 7.3 dar. Einzig

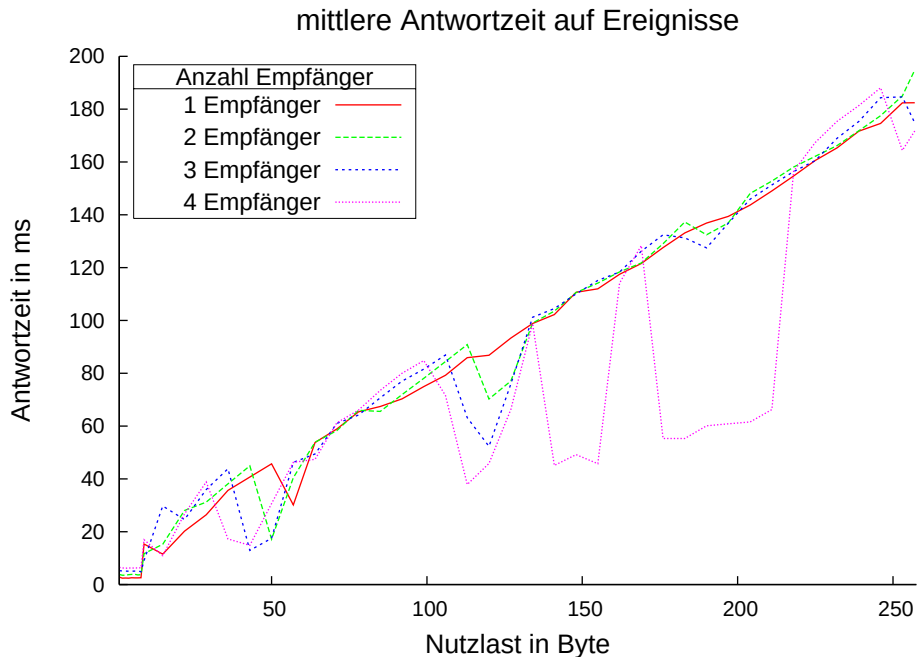


Abbildung 7.2: Mittlere Antwortzeit auf Ereignisse

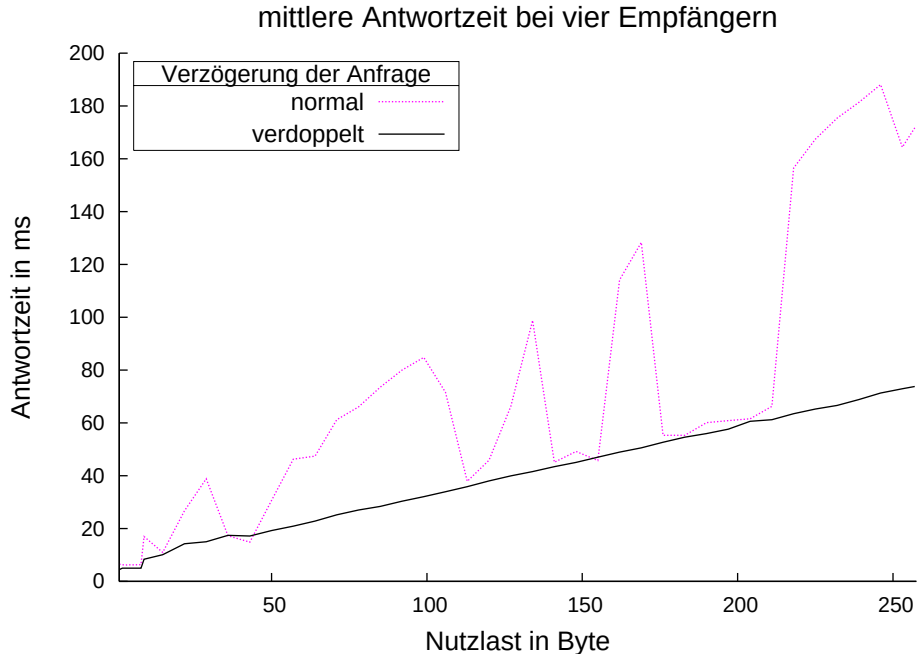


Abbildung 7.3: Mittlere Antwortzeit auf Ereignisse bei verzögerter Übertragung

veränderter Parameter ist die Verzögerung, mit der COSMIC Ereignisse fragmentiert. In Abschnitt 6.2.3 angesprochen, ist keine Flusskontrolle vorhanden. Daher werden einzelne Nachrichten mit einer Verzögerung per CAN übertragen. Die stabilen Messwerte signalisieren die Bedeutung dieser Verzögerung. Sendet der Host die einzelnen Fragmente zu schnell, müssen die Empfänger ständig Empfangspuffer bearbeiten. Dies unterbricht die Auslieferung der Bestätigung an den Host. Bei Erhöhung der Verzögerungszeit können die Empfänger umgehend antworten. Das verminderte Auftreten mehrerer gefüllter Empfangspuffer bei einer zunehmenden Zahl von Empfängern lässt sich mit vermehrten Buszugriffen erklären. Jedem Empfänger bleibt mehr Zeit für die Vorbereitung seiner Antwort, bevor die Übertragung eines weiteren Ereignisses beginnt.

Die ermittelten Zeiten setzen sich je aus drei Bestandteilen zusammen:

1. Bearbeitungszeit auf dem Host beim Senden und Empfang
2. Zugriff und Zeit auf dem Kommunikationsmedium
3. Bearbeitungszeit auf dem Mikrocontroller beim Empfang und beim Senden.

Aufgrund der ungleich höheren Ausführungsgeschwindigkeit des Hosts werden dessen Bearbeitungszeiten als vernachlässigbar angesehen. Der Zugriff auf das Medium ist abhängig von dessen aktueller Beanspruchung. Ebenso kann die Übertragungszeit nicht exakt beschrieben werden. Obwohl alle Mikrocontroller mit gleichem Inhalt antworten, ist eine einheitliche Länge der Nachricht auf dem Medium nicht möglich. Einzelne Bits, welche im Bitstopfen (*Bit Stuffing*) zur Synchronisierung eingebracht werden, betreffen auch den Header einer Nachricht in CAN. Bereits in Abschnitt 6.2.3 angedeutet wird die Unterteilung des Headers in Priorität, Identifier sowie Etag. Lediglich die Priorität kann zum Zeitpunkt des Übersetzens vom Programmcode bestimmt werden. Der Identifier eines Senders wird durch eine zentrale Instanz im Netz vergeben, dem so genannten *Broker*. Beim Start von COSMIC auf einem Teilnehmer wird mit dem Broker der Identifier ausgehandelt. Die externe Vergabe betrifft auch die Etags. Bei Verwendung eines Subjects erfolgt eine Anfrage an den Broker, welcher das Etag für das Netz eindeutig vergibt. Die Vergabe der Identifier erfolgt in absteigender, die der Etags in aufsteigender Reihenfolge. Dennoch ist eine auf das Bit genaue Bestimmung der Länge einzelner Nachrichten im Vorfeld nicht möglich, da einzelne Inhalte des Headers erst zur Laufzeit bestimmt werden. Mit einem Bitstopfen nach fünf gleichen Bits ist für den Header samt umgebender Bits ein Zuwachs von maximal sechs Bits möglich. Für die in Abschnitt 7.1 benannte Bitrate von 250kBit/s beträgt die maximale Variation demnach 24 Mikrosekunden. Die Zeiten auf dem Mikrocontroller können, wie beschrieben, aufgrund der Bearbeitung von weiteren Ereignissen variieren.

Die Übertragung der theoretisch maximal möglichen CAN-Nachricht von 147 Bit nimmt mit der gegebenen Übertragungsrate 588 Mikrosekunden in Anspruch. Bei einer angenommenen Ereignisgröße von 257 Byte sind 37 Nachrichten zu übertragen. Unter idealen Bedingungen dauert deren Übertragung $0.588 \cdot 37 = 21,7$ Millisekunden. Da vorausgesetzt werden kann, dass das Medium bei der Übertragung des Hosts nicht belegt ist, kann der überwiegende Teil der Antwortzeit den Mikrocontrollern zugeschrieben werden. Die Verzögerung durch konkurrierende Buszugriffe ist von der Anzahl

der vorhandenen Kommunikationspartner abhängig. Für n Mikrocontrollern beträgt sie $n \cdot 0.5883$ Millisekunden. Im während der Evaluation aufgetretenen Höchstfall von vier Mikrocontrollern ist für die maximale Nachricht eine Verzögerung von $4 \cdot 0.5883 = 2,353$ Millisekunden zu berücksichtigen.

Mit den Messergebnissen ist eine Abschätzung der Dauer der Rekonfiguration möglich. Für die Übertragung eines Kilobyte wird ein Zeitrahmen von ungefähr 800 Millisekunden erwartet. Eine Berücksichtigung durch eine erhöhte Verzögerungszeit ist nicht notwendig, da die Bearbeitung, wie in 6.3.3 beschrieben, in Bearbeitung eines Interrupts erfolgt. Somit können mehrere Empfangspuffer benutzt und sequentiell bearbeitet werden. Beachtet werden muss aber, dass stets wenigstens ein Puffer frei ist. Andernfalls bleiben Nachrichten unbeachtet, so dass einzelne Fragmente eines Ereignisses fehlen. Das in Abbildung 7.2 verwendete Timing muss als sehr knapp gelten, da für ausgewählte Ereignisgrößen bereits mehrere Puffer in Gebrauch sind.

7.3.2. Dauer der Rekonfiguration

Im Vergleich zum individuellen Vorgehen ist die parallele Rekonfiguration mit zusätzlichem Aufwand verbunden. So wird ein gemeinsames Kommunikationsmedium benötigt, es sind spezifische Kommunikationsmethoden zu verwenden sowie Änderungen im Vorgehen zu beachten. Vorausgesetzt, eine individuelle Behandlung der Einzelsysteme ist möglich, zeigt sich jedoch der Nutzen des Aufwands im direkten Vergleich.

Als wesentliches Vergleichskriterium wird die Ressource Zeit angesehen. Änderungen im Programmcode sollen möglichst schnell auf allen beteiligten Systemen eingespielt sein, um zeitnah auf geänderte Bedingungen reagieren zu können. Damit einher geht eine möglichst geringe Beanspruchung der Zeit des Nutzers. Mit Hilfe von Messungen zur Dauer der Rekonfiguration lassen sich Aussagen zur Anwendbarkeit der entworfenen Verfahren aus Abschnitt 6 treffen.

Für den Messaufbau kommen die in der Entwicklungsumgebung aus Abschnitt 6.1 beschriebenen Komponenten zum Einsatz. Aufgrund der einfachen Handhabung werden ausschließlich die auf dem Clusterboard aus Abbildung 6.2b befindlichen Mikrocontroller verwendet. Sie werden im Rahmen der Messung mit neuem Programmcode bespielt. Als Kommunikationspartner steht auf dem Host eine angepasste Variante von avrdude zur Verfügung. Diese vermerkt den Zeitrahmen der Rekonfiguration:

- Der Start einer Rekonfiguration wird beim Aufruf der ersten Routine der zu untersuchenden Implementationen notiert.

- Den Abschluss der Rekonfiguration stellt der Zeitpunkt dar, an dem der letzte Mikrocontroller den neuen Programmcode ausführt. Zur Bestimmung dieses Zeitpunkts beinhaltet das übertragene Programm die Anweisungen, sofort nach Programmstart ein Ereignis zu versenden. Avrdude registriert sich für den Empfang solcher Ereignisse. Mit dem Wissen über die Anzahl der beteiligten Systeme kann das letzte Ereignis identifiziert und der Zeitpunkt seines Empfangs aufgezeichnet werden.

Zur Verringerung zufälliger Einflüsse wird jede Rekonfiguration zehn mal durchgeführt. Für die Messung werden zudem einzelne Parameter der Rekonfiguration variiert:

- Eine verschiedene Anzahl an Mikrocontrollern steht als Adressaten bereit. Entsprechend der Möglichkeiten des Clusterboards sind ein, zwei, drei oder vier Boards aktiv.

- Der übertragene Programmcode weist einen unterschiedlichen Umfang auf. Ständiger Bestandteil der kompletten Ersetzung ist das Grundsystem aus COSMIC und den Routinen der Rekonfiguration. Hinzu kommt der anwendungsspezifische Bestandteil. Er beschränkt sich auf das beschriebene Verbreiten eines einzelnen Ereignisses. Durch explizite Platzierung einer einzelnen Routine wird der anwendungsspezifische Bestandteil künstlich erweitert.

Im Rahmen der Ersetzung der anwendungsspezifischen Routinen können mit der Platzierung einer Routine vorgegebene Datenmengen zur Übertragung kommen. Die Adresse der Routine wird so gewählt, dass die Gesamtzahl der Elemente mit frei gewählten Zweierpotenzen übereinstimmt. Für die komplette Ersetzung müssen ebenfalls die Routinen des Grundsystems übertragen werden. Da avrdude, wie in Abschnitt 6.4.1 beschrieben, keine Adressierung der Daten vorsieht, sind Lücken in der Übertragung nicht möglich. Daher ist auch der in Abschnitt 6.3.2 beschriebene Freiraum zwischen Grundsystem und anwendungsspezifischen Routinen Teil der Rekonfiguration. Insgesamt werden die Daten von Adresse Null bis zum Ende des anwendungsspezifischen Bestandteils übertragen. Der Umfang der Übertragung zur kompletten Ersetzung lässt sich somit durch die Addition von Startadresse und Umfang der anwendungsspezifischen Routinen berechnen. Als Startadresse gewählt ist die Adresse 0x5000. Zum Umfang der anwendungsspezifischen Routinen sind somit 20480 Byte zu addieren, um das Volumen der kompletten Ersetzung zu erhalten.

Das Messverfahren ist kritisch zu bewerten:

1. Die Bearbeitungszeit von avrdude vor dem Aufruf der ersten Routine eines unterstützten Verfahrens wird nicht erfasst.

2. Variationen im Zeitraum zwischen dem Start eines Mikrocontrollers bis zur Kenntnisnahme auf dem Host bleiben unberücksichtigt. Der Zeitraum unterteilt sich in die Bearbeitung auf dem Mikrocontroller, auf dem Host sowie in die Dauer der Kommunikation.

Für die Bearbeitungszeiten auf dem Host kann aufgrund der hohen Ausführungsgeschwindigkeiten von vernachlässigbar kleinen Zeiträumen ausgegangen werden. Unterschiedliche Zeiten für die Bearbeitung auf dem Mikrocontroller sowie für die Kommunikation sind durch das derzeitige Startprotokoll ausgeschlossen. Jedes System, welches Daten via COSMIC übertragen möchte, benötigt einen Identifier. Der erste Schritt der Initialisierung ist daher der Erhalt eines Identifiers vom Broker. Vom Broker nicht vorgesehen ist die zeitgleiche oder überlappende Behandlung mehrerer Teilnehmer. Daher

starten alle betrachteten Mikrocontroller streng sequentiell gestaffelt. Aufgrund ausreichenden Platzes zwischen einzelnen Startprozessen sind Variationen der Kommunikationsdauer aufgrund von simultanen Zugriffen ausgeschlossen. Gleichzeitig werden Variationen in der Bearbeitungszeit, welche nur durch Interrupts auftreten können, unterbunden. Kein Mikrocontroller sendet Nachrichten zu einem Zeitpunkt, an dem ein anderer Mikrocontroller startet. Somit betreffen die Auswirkungen einer Nachricht in Form eines Interrupts nur bereits gestartete Systeme.

Abbildung 7.4 stellt die gemittelten Zeiten für die Rekonfiguration eines Mikrocontroller dar. Wie beschrieben, variiert der Anteil des anwendungsspezifischen Bestandteils.

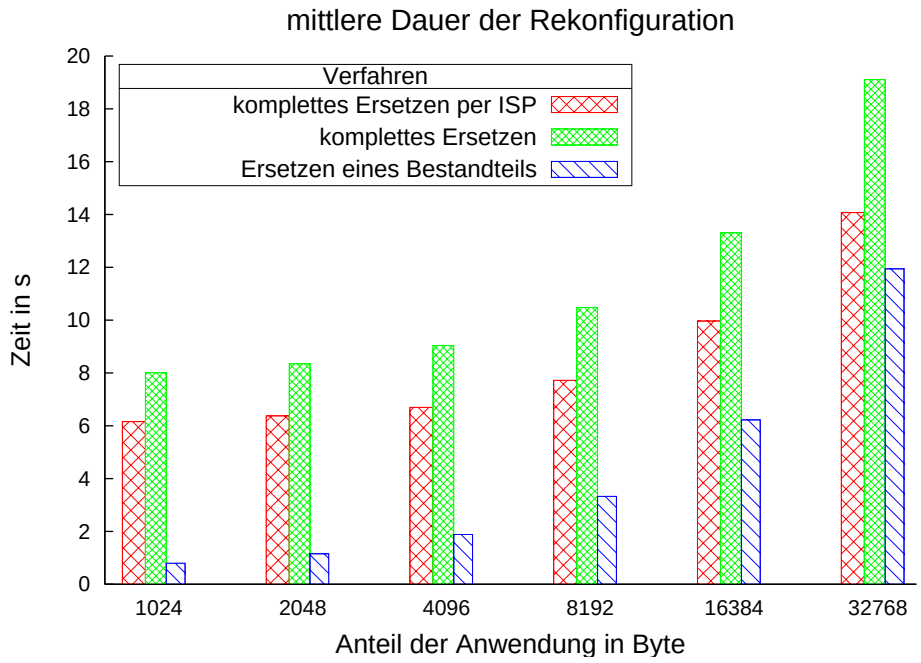


Abbildung 7.4: Dauer der Rekonfiguration für einen Empfänger

Zum Vergleich mit einer individuellen Rekonfiguration werden die Messungen eines Verfahrens des ISP hinzu gezogen. Diese Rekonfiguration erfolgt ebenfalls mit avrdude bei gleichen Parametern. Zur Ausführung kommt lediglich eine Implementation für die parallele Schnittstelle des Hosts, welche über einen gesonderten Adapter (sog. *STK200*) mit fest verdrahteten Algorithmen des Mikrocontrollers kommuniziert.

Der Vorteil bei der Beschränkung auf den anwendungsspezifischen Bestandteil überrascht nicht. Insbesondere bei den geringeren Anteilen ist nur ein Bruchteil an Daten zu übertragen. Nimmt der Umfang der anwendungsspezifischen Routinen zu, schwindet der Vorteil des Verfahrens. Ein gleiches Datenaufkommen weisen die anderen beiden Verfahren auf. Die Rekonfiguration per klassischem ISP nimmt ungefähr ein Viertel weniger Zeit in Anspruch als sein Pendant via COSMIC. Das beschriebene Startverhalten des Empfängers sorgt für eine Verzögerung der Antwortzeit um 786 Millisekunden. Die Verzögerung betrifft die Messwerte aller betrachteten Verfahren und verzerrt diese

gleichmäßig. Das Verhältnis der einzelnen Ergebnisse untereinander bleibt daher unangetastet. Somit behalten die vergleichenden Aussagen ihre Gültigkeit.

Mit gemessenen Zeiten für die Rekonfiguration mehrerer Mikrocontroller ist eine Einschätzung des Aufwands möglich. Abbildung 7.5 stellt die Zeiten von verschiedenen Verfahren gegenüber. Benutzt wird die gleiche Symbolik wie in Abbildung 7.4. Das ISP

Abbildung 7.5: Dauer der Rekonfiguration für mehrere Empfänger

sieht nur die Rekonfiguration von jeweils einem Mikrocontrollers vor. Die zum ISP notwendigen Zeit ist daher das Ergebnis einer Hochrechnung. Dafür wurden die Werte für die Rekonfiguration eines Mikrocontrollers multipliziert. Vernachlässigt bleiben Zeiten für die erneute Verkabelung sowie für den Zugang zum Mikrocontroller. Daher stellt Abbildung 7.5 eine Übervorteilung des klassischen Verfahrens dar. Wie ersichtlich, ist das Network Programming dennoch bereits mit zwei Teilnehmern im Vorteil. Während die Zeit für die individuelle Behandlung linear ansteigt, bleiben die parallel arbeitenden Verfahren von der Anzahl der Teilnehmer weitestgehend unbeeindruckt.

8. Zusammenfassung und Ausblick

Die Nutzung von Mikrocontrollern ermöglicht eingebetteten Systemen ein hohes Maß an Flexibilität. Durch die Veränderung der Software, welche von den Mikrocontrollern ausgeführt wird, lässt sich fehlerhaftes Verhalten korrigieren oder ein gänzlich neues Verhalten vorgeben. Die Vorgehensweisen zur Aktualisierung sind stets mit Aufwand verbunden. Dies hat Auswirkungen, treten mehrere eingebettete Systeme als verteiltes System auf. Dem effizienten Einsatz von Ressourcen steht ein mit der Anzahl der Teilsysteme proportional zunehmender Aufwand der Rekonfiguration gegenüber. Ein weiterer Faktor kommt hinzu, besitzt das verteilte System einen heterogenen Charakter.

Im Rahmen der vorliegenden Arbeit wurde die Möglichkeit zur Rekonfiguration eines verteilten Systems entwickelt. In Kontext von bereits existierenden Verfahren, erfüllt die Entwicklung sämtliche Merkmale des Network Programming. Vorgesehen zur Verbreitung der Daten war die bestehende Kommunikation der eingebetteten Systeme, welche auf der Middleware COSMIC basiert. Aufgrund der besonderen Anforderungen der Rekonfiguration erfolgte eine Weiterentwicklung und Ergänzung des Entwurfs der Middleware. Als Resultat ist ihr Einsatz nicht nur für umfangreichen Programmcode, sondern generell zur Verbreitung beliebiger Datenmengen gegeben. Ein Beispiel sind Daten, mit denen sich einzelne Teilsysteme selbst beschreiben, wie dies in einem Forschungsprojekt zur Anwendung derartiger *Self-Describing Devices* [32] bereits genutzt wurde. Mit Nutzung von COSMIC wurde eine Verbreitung aller Daten unabhängig vom Kommunikationsmedium möglich.

Je nach Vorgehensweise der Rekonfiguration variieren Art und Umfang der zu übertragenen Daten. Verschiedene Konzepte wurden untersucht. Grundlegendes Konzept ist eine vollständige Ersetzung der auf dem eingebetteten System laufenden Software. Obgleich eine Vergeudung von Ressourcen wahrscheinlich ist, kann durch dieses Vorgehen ein definierter Ausgangszustand erreicht werden. Einen effizienteren Umgang mit den Ressourcen bietet die Aktualisierung einzelner Bestandteile. Die Unterteilung der Software in funktionale Bestandteile und in Module wurde betrachtet. Als Ergebnis konnte festgestellt werden, dass die Zunahme der Granularität mit einem höheren Verwaltungsaufwand verbunden ist. Mehrere funktionale Bestandteile erfordern jeweils eine individuelle Unterstützung durch eine Laufzeitumgebung. Hinzu kommt die Berücksichtigung bereits installierter, anderer Bestandteile zum Zeitpunkt des Linkens. In der modulbasierten Rekonfiguration wird diese vom Mikrocontroller selbst überprüft und gewährleistet. Erkauft wird sich der Vorteil durch zusätzlich zu übertragende Informationen.

Insgesamt erkennbar ist, dass der Aufwand für einzelne Verfahren unmittelbar von den Details der individuellen Anwendung beeinflusst wird. Im Rahmen der Arbeit wurde das Konzept der vollständigen Ersetzung umgesetzt. Ebenso erfolgte eine Realisierung

des Konzepts für die Rekonfiguration einzelner Bestandteile, aufgrund der gegebenen Anforderungen jedoch lediglich für einen einzelnen und vorgegebenen Bestandteil. Wie aus der durchgeführten Evaluation ersichtlich wird, ist die Anwendung beider Verfahren bereits bei einem minimalen verteilten System von Vorteil.

Von der Arbeit nicht erreicht wurde eine Flexibilität des Rekonfigurationsprozesses, die über die Aktualisierung eines einzelnen Bestandteils hinausgeht. Die aktuell laufende Software bleibt dabei unberücksichtigt. Es werden zur Laufzeit keine Informationen über existierende Programmteile ausgetauscht noch im weiteren Verlauf zur Einbindung neuer Programmteile genutzt. Die Verantwortung für eine korrekte Verbindung des Bestandteils obliegt weiterhin dem Nutzer durch entsprechende Angaben im Linkprozess. Ein modulbasiertes Vorgehen, welches die angestrebte Flexibilität verspricht, ist mit den Erkenntnissen dieser Arbeit durch einen hohen Aufwand gekennzeichnet. Der sinnvolle Einsatz der Ressourcen für die Umsetzung bleibt von den Details der nachfolgenden Anwendung abhängig. Von einer Realisierung wurde daher Abstand genommen. Die konzipierte Übertragung von Verantwortung an die eingebetteten Systeme bleibt insgesamt jedoch viel versprechend, auch mit Blick auf die zunehmenden Möglichkeiten eingebetteter Systeme. Insbesondere Ansätze, welche speicherintensive Aufgaben auf dem Host der Rekonfiguration belassen, können die Grundlage für zukünftige Entwicklungen darstellen.

Das Erstellen der Arbeit zeigte ebenfalls, dass der Aspekt der Sicherheit einen hohen Stellenwert in nachfolgenden Arbeiten besitzen muss. Unter allen Umstände ist zu vermeiden, dass ein verteiltes System nicht mehr rekonfigurierbar ist. Unvorhersehbare Betriebszustände wie beispielsweise ein Ausfall der Stromversorgung sind zu berücksichtigen. Größere Auswirkungen, aber zugleich unweit schwieriger zu behandeln sind semantische Fehler in der Software, welche weitere Rekonfigurationen verhindern. Fehler des menschlichen Anwenders im Linkprozess, welche in falsch verbundenen Bestandteilen resultieren, sind nur ein kleiner Teil des Problems. Fällt den eingebetteten Systemen die Aufgabe des Verbindens der Bestandteile zu, werden zumindest letztgenannte Fehler ausgeschlossen.

Der Aspekt der Sicherheit betrifft auch die Kommunikation. Die Anwendung der Rekonfiguration hängt von der Qualität der Datenübertragung ab. Mit dem Einsatz von COSMIC betrifft die Übertragung eine zentrale Instanz. Eine Ergänzung um die Möglichkeit zur Flusskontrolle als Bestandteil zukünftiger Entwicklungen kann die Sicherheit und Flexibilität der Übertragung umfangreicher Datenmengen wesentlich erhöhen. Eine Verringerung des Bearbeitungsaufwands speziell bei der Übertragung über CAN verspricht die Nutzung von Filtern auf Nachrichten. Bietet die Hardware bereits Möglichkeiten zur Filterung an, brauchen ausschließlich Nachrichten für registrierte Ereignisse verfolgt werden. Mit reduziertem Bearbeitungsaufwand sinkt die zur Rekonfiguration notwendige Zeit, im Gegenzug steigt die Effizienz der Ressourcennutzung.

Sicherheit als auch Kommunikation bilden die Ausgangspunkte für weitere, vorstellbare Ergänzungen. Das Hinzufügen von Recovery-Mechanismen erscheint geeignet, einen stets definierten Zustand der beteiligten Systeme nach Abschluss einer Rekonfiguration zu gewährleisten. Die Auswirkungen von Fehlern wie dem bereits beschriebenen Ausfall der Stromversorgung während der Rekonfiguration werden so im Voraus bekannt. Denk-

bare Einschränkungen der Sende- und Empfangsreichweiten erfordern eine Abweichung vom Grundsatz der direkten Kommunikation. Ein Ergänzung um ein Vorgehen zum epidemischen Verbreiten der Rekonfiguration ist die Antwort auf derartige Anforderungen. Das Vorgehen geht aber direkt mit den Mechanismen von COSMIC einher. Speziell für das ungerichtete Verbreiten der Rekonfiguration ist der bisherige Verzicht der Kennzeichnung des Programmcodes nachteilig. Jede Rekonfiguration wird akzeptiert, was bei mehreren Wellen der Verbreitung zu unnötigem Aufwand führt. Mit Hilfe einer Versionsverwaltung können bereits vorliegende Aktualisierungen erkannt und ignoriert werden. Weitere Effizienzsteigerungen lässt die Komprimierung von Daten erwarten. Dies muss sich nicht nur auf die Kommunikation beschränken. Denkbar ist ebenfalls die komprimierte Speicherung von Programmcode. Erst im Bedarfsfall, dass heißt vor Ausführung, erfolgt die Dekompression. Beide Vorgehensweisen sind im Rahmen von eigenständigen Ergänzungen vorstellbar.

Anhang A. Quellcode zur Abschätzung des Umfangs von Metadaten

Der Quellcode wurde nahezu unverändert aus [27] übernommen. Einzig korrigiert wurde ein Syntaxfehler in der Definition des Zeilenumbruchs am Ende des auszugebenden Strings.

Datei: `m.c`

```
extern void a(char *); int main(int ac, char **av)
{
  static char string[] = "Hello, world! \n";
  a(string);
}
```

Datei: `a.c`

```
#include <unistd.h>
#include <string.h>
void a(char *s)
{
  write(1, s, strlen(s));
}
```

Literaturverzeichnis

[1] Aho, A. V. ; Sethi, R. ; Ullman, J. D.: *Compilerbau*. 1. Auflage. Addison-Wesley Longman, Inc., Reading, 1988

[2] Atmel Corporation: *8-bit AVR Microcontroller with 32K/64K/128K Bytes of ISP Flash and CAN Controller*. Rev. 7679D-CAN-02/07, http://www.atmel.com/dyn/resources/prod_documents/doc7679.pdf

[3] Barr, M. : *Programming Embedded Systems in C and C++*. 1. Auflage. O'Reilly & Associates, Inc., Sebastopol, 1999

[4] Berger, A. S.: *Embedded Systems Design*. 1. Auflage. CMP Books, Lawrence, 2002

[5] Bernauer, A. : *CAN Framing Protocol*. Mai 2004. – Draft Specification

[6] chip45 GmbH & Co. KG: *Mikrocontroller Module „Crumb128-CAN"*. Website: http://www.chip45.com, 2007. – revision 2007-07-18

[7] Contiki: *The Contiki operating system*. Website: http://www.sics.se/contiki/, 2007. – revision 2007-04-08

[8] Crossbow Technology, Inc.: *Mote In-Network Programming User Reference*. Website: http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf, März 2003. – revision 2007-04-07

[9] Dumke, R. : *Software Engineering*. 4. Auflage. Vieweg Verlag, Wiesbaden, 2003

[10] Free Software Foundation: *AVR C Runtime Library*. Website: http://www.nongnu.org/avr-libc/, 2006. – revision 2006/10/09 21:03:34

[11] Free Software Foundation: *AVR Downloader/UploaDEr*. Website: http://www.nongnu.org/avrdude/, 2006. – revision Fri Sep 23 23:21:06 MET DST 2005

[12] Free Software Foundation: *GNU Binutils*. Website: http://sources.redhat.com/binutils/, 2006. – revision 2007-06-13

[13] Free Software Foundation: *GCC, the GNU Compiler Collection*. Website: http://gcc.gnu.org/, 2007. – revision 2007-06-13

[14] Gauger, M. : *Dynamischer Austausch von Komponenten in TinyOS*, Universität Stuttgart, Diplomarbeit, April 2005

[15] Graf, J. : *Murphys gemeinste Computergesetze*. 1. Auflage. Markt und Technik, Haar bei München, 1998

[16] Heath, S. : *Embedded Systems Design*. 2. Auflage. Newness, Oxford, 2003

[17] Hui, J. W. ; Culler, D. : The dynamic behavior of a data dissemination protocol for network programming at scale. In: *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–879–2, S. 81–94

[18] Ibrahim, D. : *Microcontroller Projects in C for the 8051*. 1. Auflage. Newness, Oxford, 2003

[19] Jeong, J. ; Culler, D. : Incremental network programming for wireless sensors. In: *IEEE SECON 2004: 1st Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, 2004, S. 25–33

[20] Jeong, J. ; Kim, S. ; Broad, A. : *Network Reprogramming*. Website: http://www.tinyos.net/tinyos-1.x/doc/NetworkReprogramming.pdf, August 2003. – revision 2007-04-07

[21] Kaiser, J. ; Brudna, C. ; Mitidieri, C. ; Pereira, C. : COSMIC: A middleware for event-based interaction on CAN. In: *9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2003)*, 2003

[22] Kaiser, J. ; Mock, M. : Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN). In: *2nd. Int. Symp. on Object-Oriented Real-time distributed Computing (ISORC99)*, 1999

[23] Knuth, D. E.: *The Art of Computer Programming*. 3. Auflage. Addison-Wesley Longman, Inc., Reading, 1997

[24] Koshy, J. ; Pandey, R. : Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks. In: *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, 2005, S. 354–365

[25] Koshy, J. ; Pandey, R. : VM*: synthesizing scalable runtime environments for sensor networks. In: *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*. New York, NY, USA : ACM Press, 2005. – ISBN 1–59593–054–X, S. 243–254

[26] KTB mechatronics GmbH: *Controllerboard „mega128CAN“*. Website: http://www.ktb-mechatronics.de, 2005. – revision 2007-07-18

[27] Levine, J. R.: *Linkers and Loaders*. 1. Auflage. Morgan Kaufmann Publishers, San Francisco, 1999

[28] Levis, P. ; Culler, D. : Maté: a tiny virtual machine for sensor networks. In: *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA : ACM Press, 2002. – ISBN 1–58113–574–2, S. 85–95

[29] Liu, T. ; Martonosi, M. : Impala: a middleware system for managing autonomic, parallel sensor systems. In: *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA : ACM Press, 2003. – ISBN 1–58113–588–2, S. 107–118

[30] Pahl, G. ; Beitz, W. ; Feldhusen, J. ; Grote, K.-H. : *Konstruktionslehre*. 7. Auflage. Springer Verlag, Heidelberg, 2007

[31] PEAK-System Technik GmbH: *PCAN Hardware*. Website: http://www.peak-system.com, 2007. – revision 2007-07-18

[32] Piontek, H. ; Kaiser, J. : Self-Describing Devices in COSMIC. In: *10th IEEE International Conference on Emerging Technologies and Factory Automation*, 2005

[33] Predko, M. : *Handbook of Microcontrollers*. 1. Auflage. McGraw-Hill, New York, 1999

[34] Reijers, N. ; Langendoen, K. : Efficient code distribution in wireless sensor networks. In: *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*. New York, NY, USA : ACM Press, 2003. – ISBN 1–58113–764–8, S. 60–67

[35] Robert Bosch GmbH: *CAN Specification Version 2.0*, September 1991

[36] Ronald L. Rivest: *The MD5 Message-Digest Algorithm*. RFC 1321. : MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992

[37] Salomon, D. : *Assemblers and Loaders*. 1. Auflage. Ellis Horwood Limited, Chichester, 1992

[38] Stallings, W. : *Betriebssysteme*. 4. Auflage. Pearson Education Deutschland, München, 2003

[39] Tanenbaum, A. S. ; Goodman, J. : *Computerarchitektur*. 4. Auflage. Pearson Education Deutschland, München, 2001

[40] The IEEE and The Open Group: *The Open Group Base Specifications Issue 6 IEEE Std 1003.1*. 2004

[41] The OpenSSL Project: *OpenSSL*. Website: http://www.openssl.org/, 2007. – revision 2007-08-06

[42] TinyOS: *The TinyOS operating system*. Website: http://www.tinyos.net, 2007. – revision 2007-04-07

[43] TIS Comittee: *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*. 1995

[44] Trampert, W. : *AVR-RISC Mikrocontroller*. 2. Auflage. Franzis' Verlag Gmbh, Poing, 2003

[45] Tridgell, A. : *Efficient Algorithms for Sorting and Synchronization*, The Australian National University, Diss., Februar 1999

[46] VDI-Richtlinie 2221: *Methodik zum Entwickeln und Konstruieren technischer Systeme und Produkte*. VDI-Verlag, Düsseldorf, 1993

[47] VDI-Richtlinie 2222 Blatt 1: *Konstruktionsmethodik - Methodisches Entwickeln von Lösungsprinzipien*. VDI-Verlag, Düsseldorf, 1997