Wenn Sie neue Module definieren, sollten Sie Funktionen so bündeln, dass das Design so transparent und verständlich wie möglich ist. Dies ist wichtig im Hinblick auf zukünftige Wartung des Designs.

Wenn Sie neue Module erstellen, sollten Sie die in Abschnitt 4.2 beschriebene Vorgehensweise anwenden.

4.2 Softwaredesign bei Neuentwicklung von Embedded-Software

Entwickeln Sie Embedded-Software neu, haben Sie mehrere Möglichkeiten, das Softwaredesign durchzuführen:

- Design der Module nacheinander (oder auch parallel bei mehreren Entwicklern)
- Design der Datenstrukturen aller Module, dann Verfeinerung der Funktionalität

Bewährt hat sich, das Design der einzelnen Module nacheinander durchzuführen und nur die wichtigsten, vorab aufgrund von Anforderungen festlegbaren Datenstrukturen vor dem Design der Modulfunktionalität zu entwerfen.

Die Reihenfolge, in der Sie die Module entwerfen, sollte sich nach deren Priorität richten. Die Priorität der Module bestimmen Sie aus der Priorität der Anforderungen, die durch die jeweiligen Module umgesetzt werden.

Die parallele Entwicklung von Modulen bei größeren Projekten, an denen mehrere Entwickler mitarbeiten, hat mehrere Vorteile:

- Entwickler können bei der Fokussierung auf ein spezielles Thema ihre spezifischen Erfahrungen besser einsetzen. Wer beispielsweise viel Erfahrung in der Entwicklung hardwarenaher Software hat, kann das Design der Basissoftware-Module übernehmen, und Entwickler mit mehr Erfahrung in der Anwendungsentwicklung können sich mit den Modulen beschäftigen, welche die Regel- oder Datenverarbeitungsfunktionen enthalten.
- Entwickler können sich wechselseitig unterstützen und ihre Designs wechselseitig verifizieren und zum Beispiel als unvoreingenommene Reviewer Vorschläge zur Verbesserung des Designs machen.
- Durch paralleles Entwickeln sparen Sie auch Zeit.

Allerdings müssen Sie darauf achten, dass die Schnittstellen der parallel zu entwickelnden Module bereits definiert wurden. Ist dies nicht der Fall, müssen Sie bei paralleler Bearbeitung mit zusätzlichem Abstimmungsaufwand rechnen.

Thomas Eißenlöffel, Embedded-Software entwickeln, dpunkt.verlag, ISBN 978-3-89864-727-4

Beim Design der Module selbst schlage ich folgende Vorgehensweise vor:

- Design der Datenstrukturen
- Strukturierung der Funktionalität

Vorab die funktionsübergreifenden Datenstrukturen zu entwerfen ist oft schwieriger, als sich den einzelnen Funktionen zuzuwenden und immer nur gerade die Datenstrukturen zu entwickeln, die mit der jeweiligen Funktion entworfen werden.

Wenn Sie zuerst die Datenstrukturen festlegen, hat das aber folgende Vorteile:

- Die Daten orientieren sich klarer an den Anforderungen.
- Es entstehen weniger, dafür große Datenstrukturen.

In den wenigen, aber großen Datenstrukturen sind alle Daten einer Funktionalität gebündelt. Das erleichtert die Erweiterung einer Funktionalität, da nur an einer Datenstruktur Änderungen vorgenommen werden müssen.

Werden die Datenstrukturen aus Sicht einer Einzelfunktion entwickelt, dann fallen mehrere Datenstrukturen an, die genau die Anforderungen dieser Einzelfunktion erfüllen. Dies führt zwar zu effizienten Einzelfunktionen, macht aber Wartung und Erweiterung der Software aufwendig, da der Entwickler bei Änderungen viele einzelne Datenstrukturen im Blick haben muss.

Im folgenden Abschnitt 4.2.1 erfahren Sie mehr über das Design von Datenstrukturen in Embedded-Software und in Abschnitt 4.2.2, was Sie bei dynamischen Datenstrukturen berücksichtigen sollten.

Eine Vorgehensweise zum Design von Funktionen stelle ich Ihnen in Abschnitt 4.2.3 vor.

4.2.1 Design der Datenstrukturen des Moduls

Normalerweise versucht man, Daten möglichst zu kapseln und Zugriff nur innerhalb der Funktionen zu erlauben, die Zugriff haben müssen. Neben systemweit globalen Daten, die von jedem Modul gelesen und geschrieben werden können, existieren auch globale Daten, auf die nur die innerhalb eines Moduls definierten Funktionen Lese- und Schreibzugriff haben, auf die aber die Funktionen anderer Module nicht zugreifen können. Ganz unten in der Hierarchie der Daten stehen die lokalen Variablen, die nur innerhalb einer Funktion bekannt sind.

Ob globale Daten nur innerhalb eines Moduls »sichtbar« sein sollen, lässt sich meist aus den Anforderungen an das Modul ableiten. Die

Anforderungen geben auch über die Initialwerte dieser Daten Auskunft. Falls nicht, müssen Sie spätestens zu diesem Zeitpunkt die Initialswerte der globalen Variablen herausfinden oder festlegen.

Werden lokale Variablen innerhalb einer Funktion initialisiert, so ist die Festlegung, wo globale Variablen initialisiert werden, nicht so einfach zu treffen. Diese Daten können ja keiner einzelnen Funktion zugeordnet werden. Damit diese Variablen nicht verstreut in verschiedenen Funktionen (oder fälschlicherweise überhaupt nicht oder mehrfach) initialisiert werden, sollten Sie in jedem Modul eine Initialisierungsfunktion für die globalen Variablen des Moduls definieren. Die Initialisierungsfunktionen aller Module müssen beim Systemstart aufgerufen werden, bevor die Module selbst aktiviert und ihre Funktionen aufgerufen werden.

In Embedded-Systemen benötigen Sie häufig Zugriff von außerhalb eines Moduls auf die globalen Daten im Innern eines Moduls oder die lokalen Daten einer Funktion. Das ist oft beim Software- und Systemtest der Fall. Ungeachtet dieser Anforderung sollten Sie versuchen, Daten so gut wie möglich zu kapseln. In Kap. 5 und 6 zeige ich einige Tricks, wie diese Abschottung für Tests ohne allzu große Eingriffe in den Code aufgehoben werden kann.

Tipp: Vorsicht bei Verwendung identischer Namen:

Sie sollten lokalen und globalen Variablen nie gleiche Namen geben. Identische Namen erschweren Test und Fehlersuche und machen Programme schlecht wartbar und erweiterbar. Mit klar unterscheidbaren Namen nehmen Sie Rücksicht auf Ihre Kollegen und zukünftige Bearbeiter der Software. In einigen Sprachen-Untermengen wie z.B. MISRA-C, ist eine Namensgleichheit sogar verboten.

Wenn Sie folgende Regeln bei der Festlegung globaler Datenstrukturen einhalten, erhöhen Sie die Qualität und insbesondere die Wartbarkeit und Verständlichkeit Ihres Designs.

Gruppierung von Daten

Versuchen Sie, zueinandergehörende Variablen in Datenstrukturen zu gruppieren, um Übersicht zu bekommen. Außerdem reduzieren Sie so die Anzahl der Variablen und ggf. auch die Anzahl der globalen Variablen oder Übergabeparameter in Funktionsaufrufen. Wenn Sie später eine Datenstruktur erweitern, wird die Änderung nur bei der Deklaration der Struktur und in den Programmteilen sichtbar, die auf das neue Element zugreifen. Alle anderen Programmstellen bleiben unberührt.

Thomas Eißenlöffel, Embedded-Software entwickeln, dpunkt.verlag, ISBN 978-3-89864-727-4

Datenübergabe

Beim Datenaustausch zwischen Tasks oder Funktionen werden in Embedded-Systemen im Gegensatz zu betrieblichen Anwendungen möglichst wenig Daten kopiert, da dies Speicherplatz und Rechenzeit spart. Deshalb werden in der Embedded-Software Speicheradressen übergeben oder Feldindizes, also Offsets auf Basisadressen von Speicherbereichen.

Einige Embedded-Systeme verwenden Datenschnittstellen, um die Applikationen von den Datenströmen, Protokollebenen und Treibern komplett abzuschirmen. Fungieren diese zusätzlichen Softwarelayer als Zwischenspeicher für die Daten, wird dies bei Echtzeitanwendungen problematisch, da die verzögerte Datenausgabe durch den Layer das Echtzeitverhalten des Embedded-Systems signifikant beeinflussen kann. Sie sollten also prüfen, ob Daten ungepuffert und verzögerungsfrei ausgegeben werden müssen oder welche Verzögerungszeit bei der Bereitstellung der Daten für die Applikation erlaubt ist.

Beispiel: Schnelle Echtzeit-Regelung

Ein Embedded-System steuert den Durchfluss von Flüssigkeiten in einem Gerät. Die Durchflussmenge pro Zeiteinheit wird über Sensoren erfasst, der Durchfluss selbst über ein Ventil gesteuert. Dabei müssen vorgegebene Reaktionszeiten im Millisekundenbereich eingehalten werden. Wird der Regelalgorithmus beispielsweise alle zwei Millisekunden aufgerufen, dann kann der Softwarelayer, der ebenfalls alle zwei Millisekunden abgearbeitet wird, eine Verzögerung von weiteren zwei Millisekunden bei der Datenausgabe verursachen. Der Regler muss also mit Daten arbeiten können, die um bis zu vier Millisekunden verzögert sind.

Datenorganisation

Mehrere Tasks bzw. Funktionen teilen sich oft einen Speicherbereich, auf den sie gemeinsam zugreifen. Bietet das Betriebssystem keine Dienste an, die gemeinsame Speichersegmente bereitstellen und verwalten, dann können Sie diese Speichersegmente als globale Datenstrukturen realisieren. Dies ist einfach, wenn der im Embedded-System verwendete Mikrocontroller keinen Speicherschutz unterstützt oder diese Schutzfunktion nicht aktiviert wird. Bei globalen Variablen sollten Sie unbedingt deren Verwendung in den jeweiligen Funktionen dokumentieren. Informationen zu Initialisierung globaler Variablen finden Sie in Abschnitt 5.6.7.

Für die Datenstrukturen oder Speichersegmente bieten sich mehrere Organisationsformen an, etwa Wechselpuffer, Ringpuffer oder verkettete Listen.

Wechsel- und Ringpuffer

Ringpuffer bestehen aus einer festen Anzahl von gleich großen Segmenten, die fortlaufend beschrieben werden. Ist das letzte gefüllt, beginnt das Schreiben wieder beim ersten Segment. Der Wechselpuffer ist ein Sonderfall des Ringpuffers mit zwei Segmenten. Der Zugriff erfolgt entweder über die Segmentnummer und damit den Index des Segments oder über die Segmentadresse (siehe auch Kap. 3).

Verkettete Listen

Verkettete Listen eignen sich gut zum Sortieren von und Suchen nach Daten. Jedes Listenelement besteht dabei aus einer Speicherzelle für die Daten und der Adresse des nächsten Listenelements. Für bestimmte Such- und Sortierverfahren ist es sinnvoll, eine Liste in beide Richtungen zu verketten, also neben der Adresse des nächsten auch noch die Adresse des vorhergehenden Listenelements einzufügen. Im Gegensatz zum Ringpuffer kann die Reihenfolge der Elemente in der Liste beliebig geändert werden ohne die Daten mehrerer Elemente umzukopieren. Elemente können Sie etwa mitten in der Kette einfügen oder herausnehmen. Damit ist das Sortieren oder Priorisieren von Daten leicht möglich.

4.2.2 Dynamische Datenstrukturen

Dynamische Datenstrukturen sind immer dann hilfreich, wenn die genaue Anzahl der Datensätze, die verarbeitet werden sollen, nicht bekannt ist. Im Gegensatz zu Feldern mit fester Länge und damit auch einem festgelegten Speicherverbrauch belegen dynamische Datenstrukturen je nach Größe und Anzahl der Elemente eine variable Menge Speicher. Verkettete Listen beispielsweise sind hilfreich, um komplexe Sortier- und Suchalgorithmen zu implementieren. Dynamische Datenstrukturen sind auch deshalb so beliebt, weil die Speicherkonzepte von Betriebssystemen wie Linux und Windows den Anwendungen auf diese Weise vergleichsweise riesiger Mengen Speicherplatz zur Verfügung stellen. Der Speicherplatz besteht dabei nicht nur aus RAM. Falls dieser nicht reicht, werden die Daten unsichtbar für das Anwendungsprogramm auf die Festplatte ausgelagert. Das Verwalten frei werdender Speicherblöcke und die Beseitigung der Fragmentierung des Speichers (garbage collection) erfolgt im Hintergrund. Das Anwendungsprogramm wird davon ebenfalls nicht beeinträchtigt.

Die Verwendung dynamischer Datenstrukturen vereinfacht das Design in bestimmten Anwendungsfällen deutlich. Deshalb erlauben einige Normen trotz des Gefahrenpotenzials den Einsatz von dynamischer Speicherverwaltung in Embedded-Systemen mit Einschränkungen und unter genau definierten Bedingungen.

Thomas Eißenlöffel, Embedded-Software entwickeln, dpunkt.verlag, ISBN 978-3-89864-727-4

Beispiel: Entwurf grafischer Benutzeroberflächen

Beim Entwurf grafischer Benutzeroberflächen (graphical user interfaces = GUI) werden oft dynamische Datenstrukturen für Bild- und Bedienelemente wie Fenster und Buttons eingesetzt, da diese je nach Anwendung und Datenaufkommen in unterschiedlicher Anzahl generiert werden. Dynamische Datenstrukturen erlauben eine flexiblere und einfachere Programmierung des GUI.

In den meisten Embedded-Systemen ist der Speicher allerdings begrenzt, oft sogar extrem knapp. Dazu kommt häufig noch die Anforderung, Daten in Echtzeit zu bearbeiten.

Um mit dynamischen Objekten umzugehen, werden Zeiger (Pointer) verwendet, welche die Adressen der Objekte speichern und über die auf den Speicherplatz der Objekte zugegriffen wird. Der Einsatz von Pointern birgt Risiken.

Mit folgenden Risiken werden Sie konfrontiert, wenn Sie dynamische Speicherverwaltung in Embedded-Software einsetzen wollen:

- Einer Anwendung kann der angeforderte dynamische Speicher in der angefragten Größe nicht zur Verfügung gestellt werden. Ursachen dafür sind:
  - zu starke Fragmentierung
  - nicht ausreichender dynamischer Speicher insgesamt
- Der dynamische Speicher kann nicht zeitgerecht reorganisiert werden, eine zu starke Fragmentierung nicht zeitgerecht beseitigt werden.
- Software-Fehler in der Pointer-Arithmetik haben in Systemen ohne Speicherschutz möglicherweise verheerende Auswirkungen, können ggf. schwer entdeckt werden und sind schwer eingrenzbar, wenn sie sporadisch auftreten.

In Normen und Standards wird diesen Risiken Rechnung getragen:

- Bestimmte Formen der Pointer-Verwendung sind in Untermengen von Programmiersprachen wie MISRA-C verboten oder stark eingeschränkt.
- Einige Sicherheitsnormen verlangen, abhängig von der Sicherheitsintegritätsstufe, die Verwendung von Pointern in sicherheitskritischen Anwendungen zu beschränken.
- In der ISO 26262 wird für Systeme der Stufe ASIL-C dringend davon abgeraten, dynamischen Speicher vom Heap zu verwenden.

Die Risiken führen dazu, dass viele Embedded-Systeme keine dynamische Speicherverwaltung einsetzen.

Wenn Sie dynamische Speicherverwaltung einsetzen wollen

Planen Sie dennoch, dynamische Speicherverwaltung in Embedded-Systemen zu verwenden, dann tragen Sie gegenüber einer Softwareentwicklung für PC-Anwendungen viel mehr Verantwortung. Folgende Ratschläge helfen Ihnen, die Risiken zu verringern:

- Stellen Sie sicher, dass Sie die Anforderungen der gültigen Normen einhalten und damit die Randbedingungen, unter denen Sie dynamische Speicherverwaltung verwenden dürfen.
- Setzen Sie dynamische Speicherverwaltung nur in Funktionen ein, die nicht sicherheitskritisch sind.
- Verwenden Sie keine Bibliotheksfunktionen zur Speicherverwaltung, deren Quellcode Ihnen nicht zur Verfügung steht (Beispiel: die Funktionen malloc() und free() aus einer Funktionsbibliothek der Programmiersprache C).
- Wenn Sie Software für ein Echtzeitsystem entwickeln, prüfen Sie, ob die Bibliotheksfunktionen ein deterministisches Zeitverhalten haben bzw. eine max. Ausführungszeit. Berücksichtigen Sie diese Zeiten bei der Planung bzw. Simulation des Echtzeitsystems.
- Wenn Sie dynamische Speicherverwaltung parallel in mehreren Tasks einsetzen, dann müssen diese Funktionen berücksichtigen, dass ihre Ausführung unterbrochen werden kann und mehrere Tasks sie quasi-gleichzeitig verwenden.
- Falls Sie den Quellcode von Funktionen zur Speicherverwaltung in Ihr Projekt integrieren, prüfen Sie diesen Code gründlich.
- Prüfen Sie die Möglichkeit, den dynamischen Speicher, den Sie zur Laufzeit benötigen, bereits in der Initialisierungsphase des Systems zu belegen. Dies hilft, Allokationsprobleme einzugrenzen und frühzeitig zu finden.
- Berücksichtigen Sie den Fall, dass kein freier dynamischer Speicher mehr vorhanden ist. Bei korrekter Berechnung sollte das zwar nicht vorkommen, aber a) es könnte eine Betriebssituation eintreten, die Sie nicht vorhersehen konnten, oder b) es ist ein Hardware- oder Softwarefehler aufgetreten.
- Prüfen Sie die Adressen von dynamischen Speicherblöcken, die freigegeben werden sollen, auf Korrektheit, denn die Adresse eines freizugebenden Blocks könnte ungewollt verändert worden sein und außerhalb des dynamischen Speicherbereichs liegen.

Thomas Eißenlöffel, Embedded-Software entwickeln, dpunkt.verlag, ISBN 978-3-89864-727-4

Die sichere und robuste dynamische Speicherverwaltung

Für eine sichere und robuste dynamische Speicherverwaltung schlage ich vor, für jeden Datentyp einen eigenen dynamischen Speicherbereich anzulegen. Diese Speicherbereiche (Pools) unterteilen sich in eine Anzahl Datenblöcke. Die Pools können eine unterschiedliche Anzahl Datenblöcke enthalten (siehe Abb. 4–3).

Achten Sie darauf, dass die Länge eines Datenblocks ein Vielfaches der Maschinenwortbreite des Mikroprozessors ist. Platzieren Sie notfalls Füllbytes am Ende des Datenblocks. Falls der verwendete Cross-Compiler die Daten auf Maschinenwortgrenzen platziert, müssen Sie bei der Adressrechnung oder Portierung berücksichtigen, dass ein Compiler möglicherweise Füllbytes automatisch einfügt oder ein Versatz bei der Adressrechnung auftritt (vgl. Beispiel in Abschnitt 5.1.5).

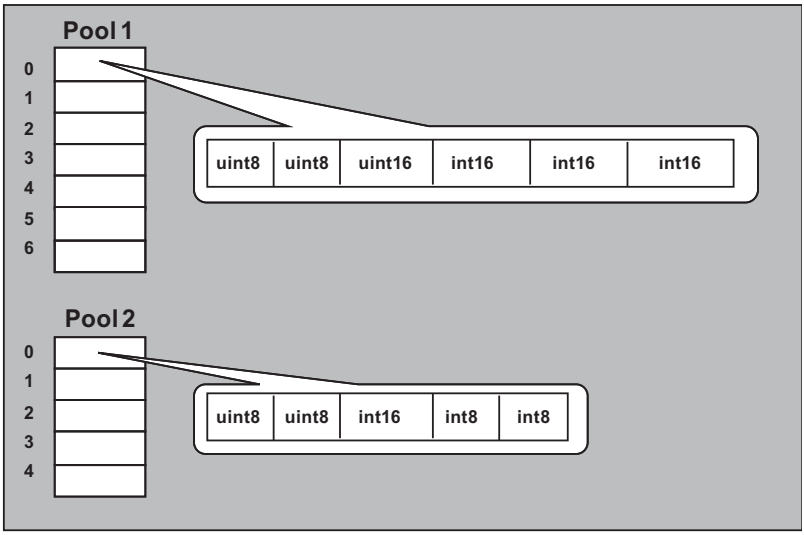


Abb. 4–3
Daten-Pools fixer Länge

Diese Vorgehensweise bietet mehrere Vorteile:

- Die Pools werden nicht fragmentiert, da jeweils Datenblöcke gleicher Länge entnommen und zurückgegeben werden.
- Ein Pool kann als Array von Datenstrukturen angelegt werden. Das vereinfacht die Zugriffsfunktionen, da diese intern über Indizes auf die Feldelemente des Arrays zugreifen, und dies ist leichter zu programmieren als der Zugriff über Pointer.
- Das Verwalten freier und belegter Datenblöcke über Indexlisten ist einfacher, als verkettete Listen mit Pointern zu bilden.

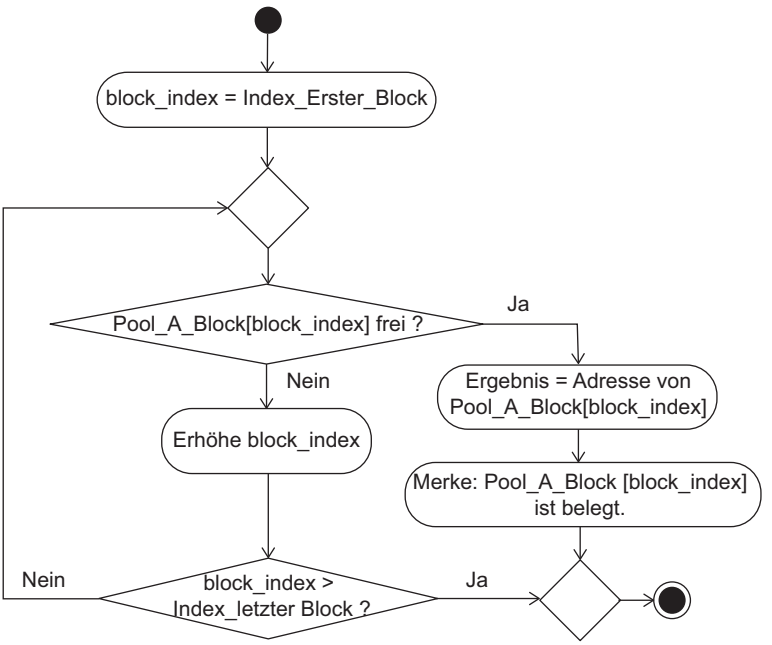
- Das Suchen von Datenblöcken bestimmten Inhalts in einem Pool ist schneller, wenn über Indexlisten adressiert wird statt verkettete Listen zu durchlaufen.
- Das Anordnen von dynamischen Daten nach bestimmten Kriterien (Sortieren) wird einfacher.

Dadurch entstehen nur wenige Nachteile:

- Der Speicher kann weniger flexibel genutzt werden. Unterschiedlich lange Datenblöcke können in einem einzigen Pool nicht verwaltet werden. Dazu wären mehrere Pools nötig.
- Sind alle Datenblöcke eines Pools belegt, ist kein weiterer dynamischer Speicher für Blöcke dieses Datentyps verfügbar, obwohl insgesamt noch unbenutzter Speicher in anderen Pools verfügbar ist.

Verwaltungsfunktionen zur Allokation und Freigabe dieser Blöcke lassen sich dann so beschreiben:

Abb. 4–4
Allokationsfunktion



Ein Beispiel (in der Programmiersprache C) zur Allokationsfunktion aus Abbildung 4–4 finden Sie in Abschnitt 5.3.3.

Die Freigabefunktion ähnelt der Allokationsfunktion. Auch hier muss die Liste durchforstet werden, um den freizugebenden Block zu finden.

Thomas Eißenlöffel, Embedded-Software entwickeln, dpunkt.verlag, ISBN 978-3-89864-727-4

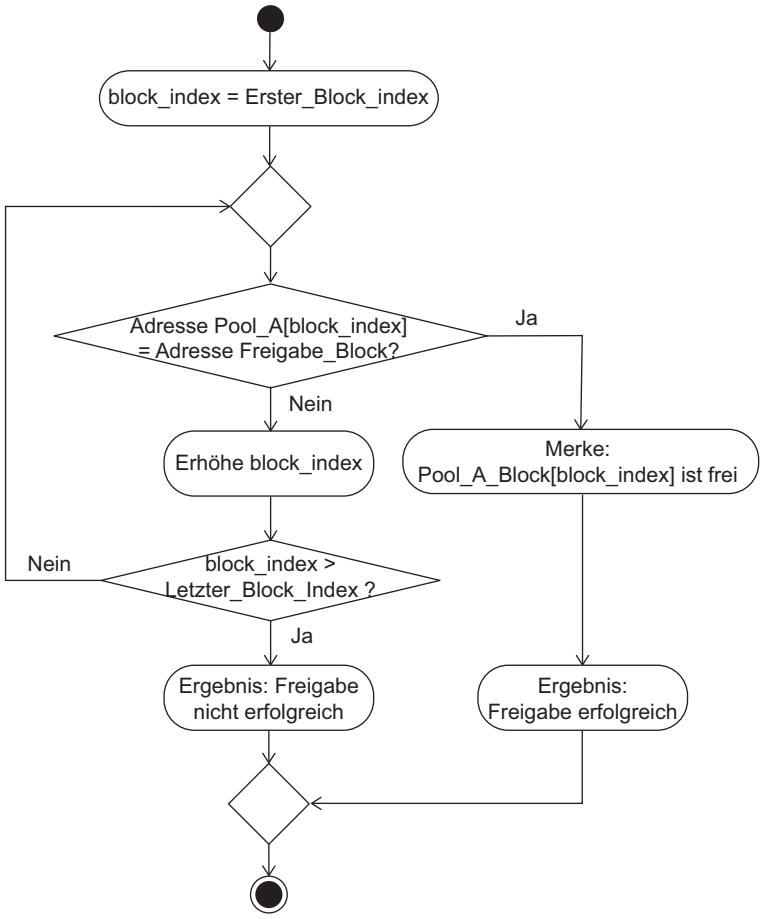


Abb. 4–5
Freigabefunktion

Ein Beispiel (in der Programmiersprache C) zu dieser Freigabefunktion aus Abbildung 4–5 finden Sie in Abschnitt 5.3.3.

Die Dauer des Suchens nimmt im Beispiel oben linear mit der Länge der Liste zu. Gerade beim Freigeben können Sie aber Adressrechnung anwenden und so auf das Suchen verzichten. Pointer-Arithmetik wird allerdings als gefährliche Operation eingestuft und sollte möglichst selten benutzt werden, wenn man beispielsweise die ISO 26262 Teil 6 zurate zieht. Sofern sich die arithmetischen Operationen auf den Wertebereich der Feldindizes beschränkt, ist dies aber nach den Regeln von MISRA-C:2004 erlaubt. Die alternative Freigabefunktion unter Verwendung von Adressrechnung sieht dann so aus:

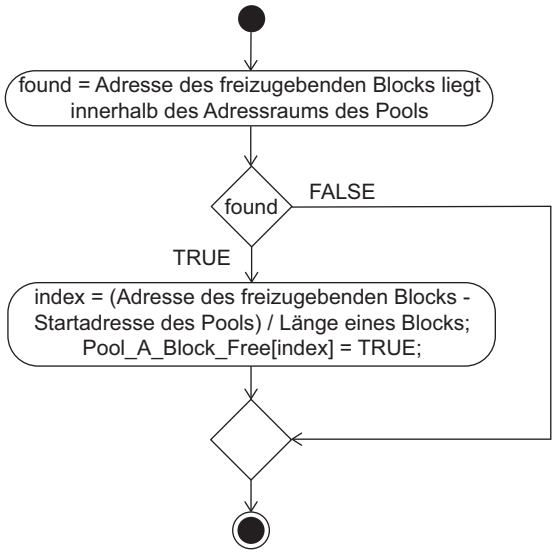


Abb. 4–6
Alternative Freigabefunktion

Ein Beispiel (in der Programmiersprache C) zur alternativen Freigabefunktion aus Abbildung 4–6 finden Sie in Abschnitt 5.3.3.

Diese Beispiele sind natürlich nicht optimal, weder was die Verwaltungsstrategie der freien Pool-Elemente betrifft noch den Speicherplatzverbrauch. Außerdem kann in diesen Beispielen der Rechenzeitverbrauch noch optimiert werden.

4.2.3 Funktionsaufbruch

Zergliedern Sie die Funktionalität des Moduls auf folgende Weise (siehe Abb. 4–7):

Der in Abbildung 4–7 skizzierte Ablauf lässt sich bei allen Programmtypen anwenden. Er unterscheidet sich zumindest auf dieser Abstraktionsebene nicht von der Vorgehensweise, die beim Programmieren betrieblicher Anwendungen eingesetzt wird.

Bei der Programmierung in einer Hochsprache wie beispielsweise C erhalten Sie dann eine Sequenz von Funktionsaufrufen. Die Sequenz kann möglicherweise auch Schleifen beinhalten, etwa könnten so lange Daten eingelesen werden, bis plausible Werte vorliegen, oder Datenerfassung und Berechnung werden so lange ausgeführt, bis das Ergebnis der Berechnung bestimmte Bedingungen erfüllt.

Thomas Eißenlöffel, Embedded-Software entwickeln, dpunkt.verlag, ISBN 978-3-89864-727-4

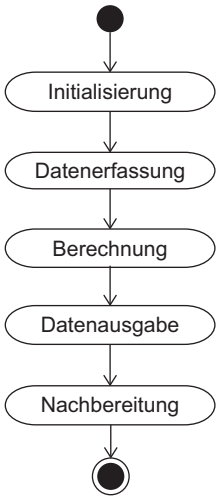


Abb. 4–7
Funktionsgliederung

Warum sollen Funktionen nach fünf Schritten gegliedert werden?

Die oben angegebene Gliederung einer Funktion mag Ihnen vielleicht trivial erscheinen, und Sie fragen sich, warum ich einem Programmierer diese Struktur vorschlage, insbesondere dann, wenn er bereits über mehrere Jahre Erfahrung verfügt. Nun, die Antwort ist folgende:

Der Struktur einer Funktion wird meiner Erfahrung nach häufig zu wenig Beachtung geschenkt. Ich habe viele Softwarefunktionen gesehen, die nicht klar strukturiert waren. Beispielsweise habe ich mitten in einer Funktion Variablen gefunden, die kurz vor Gebrauch initialisiert wurden, oder ein gerade errechnetes Ergebnis, das in den Ausgabewert umgewandelt wird, obwohl danach damit noch weitergerechnet wird.

Eine klare Strukturierung hat folgende Vorteile:

- Werden alle Initialisierungen am Anfang einer Funktion durchgeführt, ist leicht überprüfbar, ob auch alle Variablen initialisiert wurden. Bei Wartung und Änderung der Software können Sie sich schnell einen Überblick über die Startwerte in einer Funktion informieren.
- Die Trennung von Datenerfassung, Datenverarbeitung und Datenausgabe hat den Vorteil, dass die Algorithmen zur Datenverarbeitung klar getrennt von Datenein- und -ausgabe vorliegen. Dies dient der Verständlichkeit und erleichtert damit die Wartung der Algorithmen. Auch kann so ein Algorithmus dann einfach gegen einen anderen getauscht werden.
- Wenn Sie die Software wie o.a. strukturieren, erhalten Sie in den einzelnen Schritten Blöcke zusammengehörender Anweisungen. Dann merken Sie sofort, ob Sie (zu) viel Funktionalität in einer

Funktion formulieren. Wenn dies so ist, sollten Sie überlegen, ob Sie diese Funktion nicht in mehrere Einzelfunktionen aufteilen und so die Software stärker modularisieren.

Wenn Sie das Design wie oben angegeben detaillieren, fallen in den einzelnen Schritten des oben beschriebenen Ablaufs folgende Aktivitäten an:

Initialisierung

Beschaffen Sie die zur Berechnung nötigen Größen, wie

- Parameter
- Konstanten
- im letzten Rechenzyklus bestimmte systeminterne (Zwischen-) Ergebnisse

Dies geschieht beispielsweise durch:

- Zugriff auf globale Werte
- Verwendung von Übergabeparametern
- Aufruf von Funktionen, die solche Werte liefern

Bei Systemstart müssen Sie alle statischen Daten mit sinnvollen Werten (nicht notwendigerweise mit Null) initialisieren.

Variablen, in denen Signalwerte gespeichert werden, sollten am Anfang auf einen Wert gesetzt werden, der signalisiert, dass das Signal nicht gültig ist. Meist ist das ein Wert, bei dem alle Bits gesetzt sind. Für diesen Wert wird auch oft die Abkürzung SNV (Signal nicht verfügbar) bzw. SNA (signal not available) verwendet.

Variablen, die Systemzustände speichern, initialisieren Sie mit dem Grundzustand. Bei sicherheitsrelevanten Zuständen sollte dies auch ein sicherer Zustand sein.

Datenerfassung

Sie erhalten die zur Verarbeitung nötigen Eingangsdaten durch:

- Zugriff auf globale Variablen, die von einem Treiber beschrieben wurden
- den Aufruf von Funktionen, die diese Werte aus den Puffern der Datenerfassung abrufen

Meist stellen Betriebssysteme und Treiber die Daten nicht in der von der Applikation gewünschten Form bereit. Bevor die Applikation die Daten verwenden kann, müssen diese vorverarbeitet werden. Führen Sie in dieser Vorverarbeitung folgende Teilaufgaben aus:

Thomas Eißenlöffel, Embedded-Software entwickeln, dpunkt.verlag, ISBN 978-3-89864-727-4

- Prüfen der Daten auf Plausibilität. Physikalische Randbedingungen führen dazu, dass Daten oft nur bestimmte Werte annehmen. Nutzen Sie die Dynamik des Systems. Wenn Sie wissen, um welche Beträge die Eingangsdaten ihre Werte von Verarbeitungszyklus zu Verarbeitungszyklus maximal ändern können, dann eliminieren Sie Werte, die physikalisch zu diesem Zeitpunkt gar nicht auftreten können. Beachten Sie dabei, dass Sie warten müssen, bis das System eingeschwungen ist. Es muss also erst eine längere Sequenz von Eingangswerten verarbeitet haben, bevor Werte aufgrund der Systemdynamik unterdrückt werden. Die Einschwingzeit von Embedded-Systemen ist gewöhnlich recht kurz, sodass Sie diese Methode bei den meisten Embedded-Systemen anwenden können.

Beispiel: Zeitverhalten eines Geschwindigkeitssignals

Ein Geschwindigkeitssensor liefert alle 100 Millisekunden einen neuen Wert. Legt man eine maximale (Brems-)Beschleunigung von 10 m/s^2 zugrunde, können aufeinanderfolgende Werte nicht mehr als 1 m/s = 3,6 km/h voneinander abweichen.

- Vergleichen Sie Eingangsdaten mit ihrem Wertebereich. So erkennen Sie unmögliche oder ungültige Werte. Selten wird der Wertebereich von Variablen nämlich komplett ausgenutzt. Statt den Wertebereich eines Eingangswerts auf den Wertebereich einer 16-Bit-Integer-Variablen vollständig abzubilden, wird so skaliert, dass die Genauigkeit des Eingangswerts erhalten bleibt, aber andererseits der Wert in der Variablen ohne große Umrechnung verständlich bleibt (siehe Beispiel unten). Variablen haben also häufig »unerlaubte« oder »ungültige« Wertebereiche. Dies können Sie ausnutzen, um fehlerhafte Daten zu erkennen.

Beispiel: Abbildung eines Temperatursignals

Ein Temperatursensor liefert Temperaturwerte zwischen -40°C und +60°C mit einer Genauigkeit von 0,005°C.

Würde man diesen Wertebereich komplett auf eine 16-Bit-Integer-Variable T abbilden, so würde der Wert -40°C mit T = -32768 dargestellt und der Wert +60°C durch T= +32767.

In einer Abbildung, welche die Genauigkeit berücksichtigt, entsprechen -40°C dann T = -8000, und +60°C entsprechen T = +12000.

Unerlaubte/ungültige Werte von T liegen in den Intervallen [-32768, -8001] und [+12001, +32767] (siehe Abb. 4–8).

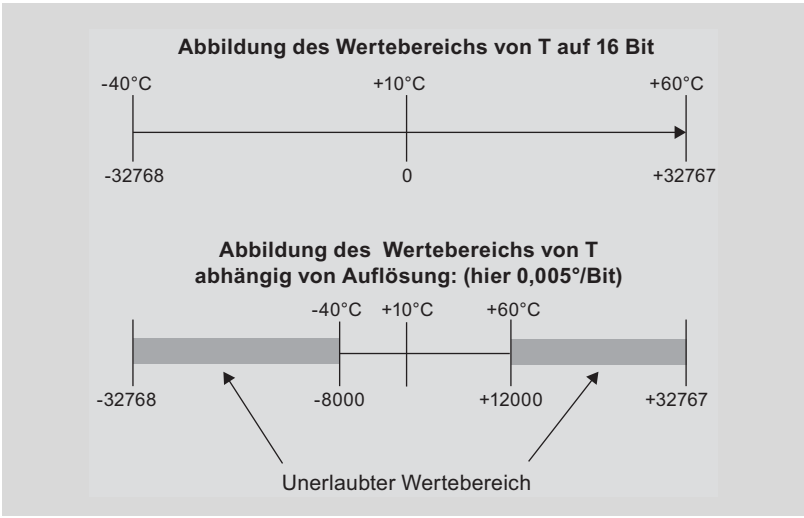


Abb. 4–8
Beispiel: Abbildung von Wertebereichen

- Gibt es physikalische Gesetzmäßigkeiten, über die einzelne Daten zusammenhängen, lässt sich dies zwar nicht immer in einer Umrechnungsformel darstellen (sonst wäre ja auch einer der beiden Werte überflüssig). Oft kann man den Zusammenhang aber in einer Bedingung beschreiben, wie etwa »Wenn Wert A größer Null ist, dann ist auch Wert B größer Null« oder »Wert B ist mindestens doppelt so groß wie Wert A«. Diese Beschreibung können Sie nutzen, um fehlerhafte Daten herauszufiltern.

Beispiel: Vergleich eines Temperatursignals mit einer Temperaturänderung

Ein Temperatursensor liefert kontinuierlich eine der Temperatur entsprechende Spannung. Das Ausgangssignal des Sensors wird dem Eingang eines ADC zugeführt und parallel in einen analogen Differenzierer eingespeist, dessen Ausgang ebenfalls in einen ADC eingespeist wird. Die ADC liefern also ein digitalisiertes Temperatursignal und ein digitalisiertes Signal der Temperaturänderung.

Der Wert der Temperaturänderung ist umso größer, je schneller sich die Temperatur ändert, d. h. je mehr sich aufeinanderfolgende Abtastwerte des Temperatursignals unterscheiden. Ist Temperaturänderung etwa nahe Null, dann darf sich die Temperatur auch nicht oder nur wenig ändern.

Weicht die Differenz zweier Temperaturabtastwerte zu weit von dem Wert der Temperaturänderung ab, so liegt ein Fehler vor, und das Signal muss als ungültig bewertet werden.

- Prüfen der Gültigkeit der Daten:
  - Einhalten von Zeitvorgaben für den Datenempfang
  - Gültigkeitsdauer der Daten wird nicht überschritten

Thomas Eißenlöffel, Embedded-Software entwickeln, dpunkt.verlag, ISBN 978-3-89864-727-4

- Verrechnen der Daten redundanter Kanäle nach festgelegten Algorithmen und Bestimmen eines Ergebniswerts. Zu den Algorithmen zählen:
  - der Vergleich redundanter Daten
  - die Mittelung redundanter Daten
  - das Priorisieren von Signalwerten unterschiedlicher Präzision
- Skalieren Sie die Eingangsdaten. Signale werden auf I/O-Kanälen meist mit einem an den Kanal angepassten Wertebereich übertragen. Die Skalierung, die sich aus diesem Wertebereich ergibt, ist für die Weiterverarbeitung in der Applikation meist nicht geeignet. Die Eingangsdaten sollten Sie dann so skalieren, dass der nachfolgende Algorithmus die Werte in der für ihn passenden Auflösung vorfindet.

Berechnung

In der *Berechnung* formulieren Sie den eigentlichen Algorithmus und bestimmen die Ergebnisse aus den Eingangsdaten. Die Skalierung der Ergebnisse des Algorithmus kann sich dabei durchaus von der Skalierung unterscheiden, welche die Eingangsdaten der nachfolgenden Verarbeitungs- oder Übertragungseinheiten haben müssen. Die Wortbreite der Daten auf einer Übertragungsstrecke hängt davon ab, welchen Wertebereich und welche Genauigkeit die Empfänger erwarten.

Datenausgabe

Die Berechnungsergebnisse des Algorithmus werden in diesem Schritt so umgerechnet, dass sie weiterverarbeitet oder an andere Verarbeitungseinheiten übertragen werden können.

Beispiel: Ausgabe von Ergebnissen in ein Signal einer CAN-Botschaft

Intern berechnet ein Algorithmus eine Geschwindigkeit in der Einheit 0,01 m/s. Das Signal in der CAN-Botschaft selbst hat eine Auflösung von 0,01 km/h.

Im Abschnitt *Datenausgabe* wird dann der interne Geschwindigkeitswert mit folgender Formel in den Geschwindigkeitswert für das Signal der CAN-Botschaft umgerechnet:

```
v_can = v_intern * 3,6.
```

Nachbereitung

Dieser Schritt ist oft nicht notwendig. Werden aber zu Beginn der Ausführung der Funktion bestimmte Systemzustände oder Systemgrößen verändert, müssen Sie vor Beendigung der Funktion die Verhältnisse, die vor dem Start der Funktion gegeben waren, wiederherstellen. Oft müssen auch zusätzliche statistische oder der Absicherung dienende Daten vor Beendigung einer Funktion aktualisiert werden.

Beispiel 1: Wiederherstellung ursprünglicher Verhältnisse

Bei Start der Funktion wurden bestimmte Interrupts gesperrt. Diese werden in diesem Schritt wieder freigegeben.

Beispiel 2: Absicherung des Programmablaufs

Um die korrekte sequenzielle Abarbeitung zu verifizieren, wird in einen Programm ein Programmlaufzähler verwendet. Aufgrund der sequenziellen Abarbeitungsreihenfolge kennt jede Funktion ihren Platz in der Reihe. Wird am Ende einer Funktion der Zähler erhöht, kann die nächste Funktion am Anfang prüfen, ob der Zählerstand mit ihrem Platz in der Aufrufreihenfolge übereinstimmt. Sind die Werte verschieden, ist der Programmablauf gestört, und eine Fehlerbehandlung wird eingeleitet, die meist mit dem Reset des Systems endet.

4.3 Anwendungsprogramme

Je nach Problemstellung werden unterschiedliche Methoden zur Anwendungsentwicklung eingesetzt. Dabei richtet sich die Auswahl der Methode auch nach der gewünschten Abstraktionsebene für die Formulierung der Problemstellung.

Mathematische Algorithmen werden oft kompakt durch Rekursion beschrieben (Abschnitt 4.3.1).

Steuerungsaufgaben etwa können Sie gut mit Zustandsautomaten formulieren (Abschnitt 4.3.2).

Für regelungstechnische Aufgaben eignen sich Modellierungswerkzeuge, die für diese Aufgabe vorgefertigte Funktionsblöcke wie z.B. Filter anbieten (Abschnitt 4.3.3).

Viele Aufgaben lassen sich auch gut mit Aktivitätsdiagrammen der UML beschreiben. Die Diagramme können Sie leicht in den Code einer höheren Programmiersprache wie C übersetzen.

Thomas Eißenlöffel, Embedded-Software entwickeln, dpunkt.verlag, ISBN 978-3-89864-727-4