Schlechte Strukturierung und Dokumentation der Software kann zukünftige Erweiterungen so verteuern, dass die langfristige Wettbewerbsfähigkeit des Produktes gefährdet wird und die Wahrscheinlichkeit von Folgeaufträgen sinkt.

6.1 Teststrategie

Beim Test von Embedded-Software ist die Teststrategie entscheidend. Sie betrifft Modultest, Softwareintegration, Integrationstest und den Software-Anforderungstest. Je nach Aufgabenstellung werden auch Software-spezifische Themen für HW-/SW-Integrationstest, Komponententest und Systemtest in der Teststrategie berücksichtigt.

In der Teststrategie werden folgende Aspekte des Testens betrachtet:

- Sowohl Kunden als auch firmeninterne Qualitätsvorgaben stellen Anforderungen an den Test und beeinflussen Art und Umfang der Tests (Abschnitt 6.1.1).
- Vor der Entwicklung der Tests selbst sollten Sie die Voraussetzungen für die Tests klären, die Tests planen, die Integrationsstrategie festlegen und bestimmen, was genau getestet wird. Legen Sie Struktur und Inhalt der Testspezifikationen und Testprotokolle fest (Abschnitt 6.1.2).
- Definieren Sie auch die Testumgebung für die Embedded-Software sowie die Testwerkzeuge und legen Sie fest, wie geprüft wird, dass die Testeinrichtung zuverlässig und fehlerfrei arbeitet (Abschnitt 6.1.3).
- Zur Durchführung der Tests gehören neben individuellen Testvorbereitungen (Abschnitt 6.1.4), dem Test selbst (Abschnitt 6.1.5) auch Kriterien zum Beenden bzw. Abbruch der Tests, die Prüfung der Testergebnisse und die Freigabe der Software (Abschnitt 6.1.6).

Bei der Entwicklung von Büroanwendungen ist die Teststrategie oft überschaubar und von der Entwicklungsumgebung vorgegeben. Bei Embedded-Software müssen Sie viele zusätzliche Aspekte beim Erstellen der Teststrategie berücksichtigen. Einige dieser Themen sind:

- Hardwareabhängigkeiten
- Echtzeit-Randbedingungen
- Einschränkungen bei Testumgebungen
  - Funktionalität
  - Verfügbarkeit

- Störungen
  - Bitfehler in digitalen Signalen
  - Unterbrechung von Signalen

6.1.1 Anforderungen an den Test

Oft werden vom Kunden viele Anforderungen an die Verifikation der Software gestellt. Gerade bei Embedded-Software, die in Investitionsgüter oder teure Massenverbrauchsgüter integriert wird, ist die Qualität der Software wichtig. Meist kann die Software nur mit erheblichem Kostenaufwand ausgetauscht werden, wenn Geräte bereits an Kunden ausgeliefert wurden.

Teststrategie mit dem Kunden abstimmen

Manche Kunden von Embedded-Systemen bevorzugen eine Abstimmung der Teststrategien mit den Lieferanten. Diese Kunden haben erkannt, dass eine Abstimmung der Teststrategien von Embedded-Systemen und den Systemen oder Anlagen, in die Embedded-Systeme integriert werden, hilft, Aufwände beim Test insgesamt zu reduzieren und damit (meist beim Kunden) Kosten zu sparen. In diesem Zusammenhang stehen auch die Forderungen der Kunden, Testprotokolle als Teil der Produktdokumentation zu liefern. Einige Kunden beschränken sich auch darauf, die Testprotokolle bei Bedarf einsehen zu können.

Viele Kunden machen teilweise detaillierte Vorgaben zur Verifikation von Embedded-Systemen. Diese Vorgaben betreffen dabei nicht nur die Verifikationsmaßnahmen an sich, sondern oft auch die Prozesse, also die Vorgehensweise beim Test insgesamt. Die Prozessvorgaben orientieren sich meist an Normen und Standards, wie den Sicherheitsnormen IEC 61508, ISO 26262 oder EN 50128, aber auch an Qualitätsstandards wie SPICE oder CMMI.

Qualitative Anforderungen an die Tests

Qualitative Anforderungen betreffen die Vorgehensweise beim Erstellen der Testunterlagen, die Vorgehensweise beim Testen und die Dokumentation dazu und werden gestellt an:

- Testspezifikationen
- Testfall-Implementierung
- Testanleitungen
- Testwerkzeuge
- Testpersonal

Quantitative Anforderungen

Aus diesen Vorgaben leiten sich dann auch quantitative Anforderungen an die Tests ab. Diese Anforderungen an den Umfang der Tests betreffen:

- Testabdeckung
- Testtiefe
- Regression

Die Testabdeckung macht eine Aussage darüber, in welchem Umfang der Quellcode getestet wurde (vgl. Kap. 1), die Testtiefe betrifft die Anzahl der Testfälle pro Funktion, und Regression bezieht sich auf die Anzahl der Wiederholungen, die dann wichtig sind, wenn Zeitbedingungen, Unterbrechungen oder Hardwareeinflüsse eine Rolle spielen.

6.1.2 Testvoraussetzungen

Machen Sie sich beim Erstellen der Teststrategie Gedanken, welche Vorbereitungen getroffen und welche Bedingungen erfüllt sein müssen, damit die Softwaretests erfolgreich und gemäß den Vorgaben durchgeführt werden können.

Testplanung

Wie alle Prozess-Schritte innerhalb des Entwicklungszyklus sollten Sie auch die Testphase planen. Auch wenn Sie für ein Embedded-System nur den Software-Modultest für eine einfache Funktion durchführen müssen, sollten Sie sich Gedanken über den Ablauf machen. Planen Sie eine komplette Testphase, müssen Sie viele Details im Zeitplan berücksichtigen:

- Überarbeiten und Freigeben aller nötigen Testspezifikationen, Anleitungen und Protokollvorlagen
- Prüfen und Freigeben aller Testwerkzeuge und Testumgebungen
- Abstimmen der Entwicklung der Testspezifikationen und Testfälle mit der Design- und Implementierungsphase
- Zusammenstellen der zu testenden Software (Baseline)
- Zeit zum Einrichten der Testumgebung vor dem Test
- Reservieren kritischer Ressourcen wie
  - Messgeräte
  - Testeinrichtungen
  - Lizenzen für die Testsoftware
- Schulen oder Einweisen des Testpersonals

- Nachbereiten der Tests
  - Review der Testprotokolle
  - Dokumentieren gemachter Erfahrungen
  - Initiieren von Prozessverbesserungen

Integrationsstrategie

Eine Integrationsstrategie ist wichtig, wenn sich Embedded-Software aus mehreren Modulen zusammensetzt. Theoretisch können Sie alle Module einzeln testen, dann Modul um Modul zusammenzusetzen und jeweils die teilintegrierte Software testen, bis die gesamte Software integriert ist.

In der Realität orientiert sich die Reihenfolge der Tests an den Gegebenheiten des jeweiligen Projekts. Ausgangspunkt ist ein fertiggestelltes Modul, das einem Modultest unterzogen wird. Danach ist zwar die Funktion des Moduls verifiziert, ob aber das Modul mit anderen in der gewünschten Weise zusammenarbeitet, stellt sich erst beim Integrationstest heraus, der genau dieses Zusammenwirken prüft. Die Module, die integriert und dem Integrationstest unterzogen werden, müssen vorher geprüft worden sein. Dadurch ist eine teilweise Parallelisierung der Testaktivitäten möglich.

Bei der Integration mehrerer Module ist es schwierig, eine überschaubare Testmenge zu bestimmen, welche die Anforderungen ausreichend prüft. Versuchen Sie, aus den Anforderungen und den Einsatzszenarien der Software entsprechende Testfälle für die teilintegrierte Software abzuleiten.

Für den Test einzelner Module gilt:

- Werden im Projekt Softwaremodule neu erstellt, sollten Sie diese separat testen.
- Sicherheitskritische Module müssen Sie separat testen. Bei sicherheitskritischen Modulen ist im Vergleich zu nicht-sicherheitsrelevanter Software ein höherer Testaufwand erforderlich. Oft müssen Sie dann auch zusätzliche Testwerkzeuge einsetzen, etwa zur Bestimmung der Testabdeckung.
- Bei bereits bestehenden und aus anderen Projekten übernommenen Modulen können Sie oft auf eine erneute komplette Prüfung des Moduls verzichten und sich auf einen Test der Schnittstellen und der tatsächlich verwendeten Funktionalität beschränken.
- Von Dritten beigestellte Software wird meist nicht separat getestet. Im Rahmen der Integration der Software sollten Sie aber die im Projekt benutzte Funktionalität prüfen. Auch sollten Sie die Dokumentation der Software auf Einschränkungen und bekannte Fehler

und die vorgeschlagenen Maßnahmen zur Umgehung dieser Fehler hin durchlesen.

- Hardwarenahe Software können Sie meist nur sinnvoll in Verbindung mit der Hardware testen, also auf dem Zielsystem oder einer dem Zielsystem möglichst ähnlichen Hardware.
- Sind zwei Softwaremodule über viele Schnittstellen miteinander verbunden, ist ein gemeinsamer Test dieser Module, also eine Teilintegration, weniger aufwendig als ein separater Test jedes Moduls.

In welchen Schritten Sie die Integration und den Integrationstest der Module durchführen, hängt vom Zusammenwirken der Module ab.

Hat das Embedded-System viele verschiedene Funktionen, die wenig miteinander zu tun haben (beispielsweise Diagnosefunktion und Regelungsfunktion), empfehle ich, zuerst die Module der Applikation zu integrieren, die für die wichtigste Funktion unbedingt gebraucht werden, dann die restlichen. Dabei sollten Sie in der Richtung des Datenflusses vorgehen, also zuerst das Datenaufnahmemodul mit dem Datenverarbeitungsmodul integrieren, danach das Datenausgabemodul hinzufügen.

Meist können Sie Applikationssoftware und die hardwarenahe Basissoftware (vgl. Kap. 3) getrennt integrieren und testen. Diese Arbeiten lassen sich auch parallel ausführen, wie viele Modultests übrigens auch. In welchem Umfang Sie parallelisieren, hängt meist von der Anzahl der verfügbaren Tester ab. Rechnen Sie aber auch mit weiteren Einschränkungen der Parallelisierung wie:

- Anzahl der Testsoftware-Lizenzen
- Anzahl der Testumgebungen, z.B. bei hardwarenahen Tests

Testobjekte

Aus der Integrationsstrategie können Sie ableiten und festlegen, wie viele und welche Testobjekte (Module, teilintegrierte Software) einem Test unterzogen werden müssen. Sie wissen dann auch, welche Integrationstests von welchen Modultests abhängen und damit nach diesen erfolgen müssen. Aus diesen Abhängigkeiten ergibt sich eine Sequenz von Tests. Je mehr Ressourcen (Testumgebungen und Testpersonal) Sie haben, desto mehr können Sie die in der Sequenz enthaltenen parallelen Pfade nutzen, um Zeit zu sparen und die Testdauer zu minimieren.

Testspezifikationen

Unabhängig vom jeweiligen Testobjekt müssen Sie die Testspezifikation vor Beginn des Tests fertiggestellt haben. Zur Fertigstellung gehört auch, dass diese Spezifikation von den Entwicklern und Testern einem Review unterzogen wurde. Beim Test von Embedded-Software müssen Sie viele Randbedingungen beachten (Hardware, Zeitverhalten, Testumgebung), so dass die Gefahr besteht, beim Spezifizieren den einen oder anderen Aspekt zu übersehen.

Bauen Sie die Testspezifikationen deshalb, wenn möglich, nach einem einheitlichen Schema auf. Beschreiben Sie zu Beginn die Testobjekte und deren Testumgebungen. Denken Sie dabei auch an Kleinigkeiten, wie spezielle Kabel und Geräteeinstellungen. Nur weil Sie beim Konfigurieren einer Testumgebung ein Oszilloskop in einem bestimmten Betriebsmodus vorgefunden haben, können Sie nicht davon ausgehen, dass dies auch beim nächsten Mal so sein wird.

Wenn Sie die einzelnen Testfälle beschreiben, stellen Sie für jeden Testfall diese Informationen bereit:

- Vorbedingungen:
  Was muss vor dem Test alles eingestellt oder verändert werden?
- Nachbedingungen:
  Was muss nach dem Test alles zurückgesetzt oder in den vorherigen Zustand versetzt werden?
- Soll-Ergebnis:
  Welches Ergebnis soll der Test liefern, um als »bestanden« gewertet zu werden?
- Link zu den Anforderungen, die mit dem Testfall verifiziert werden.

Risiken für Testhardware und Testumgebung

Klären Sie auch mögliche Risiken für die Testhardware und die Testumgebungen ab. Wenn Sie etwa Modultests auf der Zielhardware durchführen, muss dazu die Software oft ins Flash geladen werden oder Daten in EEPROMs geschrieben werden. Diese Bausteine lassen nur eine beschränkte Anzahl von Schreibzyklen zu. Zwar wird die Hardware so ausgelegt, dass die maximale Zahl der Schreibzyklen später im realen Betrieb nicht überschritten wird, doch im Testlabor herrschen andere Bedingungen. Rechnen Sie sicherheitshalber nach, in welchem Umfang die Tests die Testeinrichtung belasten und abnutzen.

Schalten Sie in der Testeinrichtung Signale und Spannungen, dann prüfen Sie, ob durch eine falsche Reihenfolge von Schaltvorgängen Schäden an der Ausrüstung verursacht werden könnten. Oft muss eine bestimmte Verbindung zuerst aufgetrennt werden, bevor ein Kontakt

mit einer Testspannung verbunden wird. In diesem Fall sollten Sie in der Dokumentation entsprechende Gefahrenhinweise angeben.

Testfälle und ihr Bezug zu den Anforderungen

Listen Sie bei der Beschreibung der einzelnen Testfälle alle Anforderungen auf, die mit den Testfällen verifiziert werden sollen. Tragen Sie die Testfälle auch in die Anforderungsverfolgungsmatrix ein. Dann können Sie leicht feststellen, ob die Testfälle alle Anforderungen abdecken oder ob noch Testfälle fehlen. Die Anforderungen sind auch eine unabdingbare Voraussetzung zum Beurteilen der Wichtigkeit eines Tests und damit eine Voraussetzung, um die Priorität eines Tests festzulegen.

Sicherheitsanforderungen fallen damit in die Kategorie »unbedingt durchführen«.

Wenn es Anforderungen gibt, die sich auf optionale Features beziehen, müssen diese möglicherweise nicht immer getestet werden, und Tests, die diese Anforderungen verifizieren, können Sie in der Priorität zurückstufen.

Testprotokoll

Meine Erfahrungen haben gezeigt, dass Sie bei Tests von Embedded-Software möglichst viele Details protokollieren sollten. Nur so können Sie aus den Testergebnissen die richtigen Schlüsse ziehen.

Zweitrangig ist, ob Sie die Testergebnisse automatisch, manuell oder in einer Mischung aus beiden Protokollarten festhalten.

Diese Informationen sollten mindestens im Testprotokoll zu finden sein:

- Informationen über das Testobjekt
  - Bezeichnungen der getesteten Softwaremodule
  - Versionsnummern der getesteten Software oder die Bezeichnung der Baseline des Konfigurationsmanagementsystems, mit dem die Software verwaltet wird
- Informationen über die Testumgebung
  - Welche Testumgebung wurde verwendet?
  - spezielle Einstellungen der Geräte, soweit diese nicht in der Testspezifikation dokumentiert sind
- Informationen über die Testprogramme
  - Testsoftware inkl. Version oder ggf. Bezeichnung der Baseline
- Informationen über die Testdurchführung
  - Datum, ggf. Ort
  - Name der Person, die den Test durchgeführt hat
  - Ort der Ablage der Testprotokolle (URL)

- Informationen zu jedem einzelnen Test
  - Test-ID
  - Ist-Ergebnis
  - Bewertung
  - Bemerkungen

Bewertung der Testergebnisse

Zur Bewertung verwenden Sie *bestanden*, *nicht bestanden* oder *nicht durchgeführt*. Sind Sie nicht sicher, ob der Test bestanden wurde, weil das Testergebnis zwar vom Soll-Ergebnis abweicht, andererseits die Funktion Ihrer Ansicht nach i.O. ist, dann kommentieren Sie das Testergebnis in Feld »Bemerkungen« entsprechend.

Gerade die Spalte »Bemerkungen« ist in einem Testprotokoll wichtig. Die dort notierten Auffälligkeiten liefern Hinweise auf:

- Lücken in der Testspezifikation
- unvollständige Testbeschreibung
- nicht eindeutig beschriebene Soll-Ergebnisse
- allgemeine Mängel des Tests wie:
  - Zeitverhalten ist von Test zu Test verschieden.
  - fehlerhafte oder abgebrochene Datenübertragungen von Code auf das Testsystem

Besonders beim Testen sicherheitsrelevanter Software dient das Testprotokoll auch zum Nachweis, dass Sie alle erforderlichen Anstrengungen unternommen haben, um die Software zu prüfen. Dies könnte bei Fragen der Produkthaftung wichtig werden.

Testfälle

Automatische Tests

Erstellen Sie Testfälle für automatische Tests, so unterziehen Sie diese Testsoftware einem Review. Eventuell ist auch ein Kurztest sinnvoll, mit dem der Testablauf geprüft wird. Idealerweise ersetzen Sie die zu testenden Module durch leere Dummy-Module und weisen damit nach, dass alle Tests als Ergebnis *nicht bestanden* liefern.

Manuelle Tests

Leider lassen sich nicht alle Softwaretests automatisieren. Spielt die Hardware eine Rolle, wird die Automatisierung schwierig. Die Schnittstellen des Prozessors können Sie oft mit einer steuerbaren Testumgebung prüfen, welche die Schnittstellen mit geeigneten Testsignalen beschickt. Wollen Sie Software testen, die Hardwarefehler, etwa im RAM, erkennen soll, bleibt Ihnen nur der Einsatz eines Emulators, Debuggers oder anderer zusätzlicher Werkzeuge, um in den Programmablauf einzugreifen.

Beschreiben Sie den Ablauf dieser manuellen Tests detailliert. Falls diese Werkzeuge über Skripte steuerbar sind, nutzen Sie diese Möglichkeit. Je mehr Sie implementieren und automatisch ausführen, desto weniger anfällig ist der Test für Fehlbedienungen und desto schneller ist er durchgeführt. Dazu gehört auch, Messgeräte nicht von Hand einzustellen, sondern die Einstellungen ins Messgerät zu laden und auch die Messergebnisse, wenn möglich, auszulesen und automatisch auszuwerten und zu speichern.

Robuste Tests

Achten Sie darauf, dass Sie bei Tests, an denen Hardware beteiligt ist, die Testsignale und deren Timing so wählen, dass Sie einerseits die Funktion des Testobjekts eindeutig auf Korrektheit prüfen können, andererseits aber an die Testsignale nicht zu hohe Anforderungen stellen müssen. Das würde die Testumgebung erheblich verteuern. Je robuster die Tests, desto besser.

Ist ein Test nicht robust genug und liefert unterschiedliche Ergebnisse, obwohl die getestete Funktion korrekt ist, dann prüfen Sie alle Vorbedingungen des Tests. Meist waren diese bei der Testfallentwicklung nicht in vollem Umfang bekannt oder haben sich zwischenzeitlich geändert.

6.1.3 Testumgebung

Oft wird für einen Test die Testumgebung je nach Verfügbarkeit aus verschiedenen Werkzeugen und Geräten zusammengestellt. In einer solchen Umgebung reproduzierbare und vergleichbare Testergebnisse zu erzielen ist schwierig.

Auch für eine Testumgebung gilt: Beschreiben Sie diese ausführlich. Dazu gehören auch Gerätetypen, Hardwareversionen und Vorgaben zu Software-Werkzeugen inklusive deren Versionsnummern, Konfigurationsdateien.

Bei Embedded-Software spielen spezielle Hardwareschnittstellen, Sensoren und Aktoren eine wichtige Rolle. Versuchen Sie, die Originalteile zu bekommen. Manchmal ist dies nicht möglich, da die Original-Aktoren zu teuer sind, zu groß für das Labor oder weil sie ohne zusätzliche Geräte (etwa zur Kühlung) nicht betrieben werden können. In diesen Fällen ist beim Test besondere Vorsicht geboten. Versuchen Sie, eine möglichst gute elektrische Nachbildung der Sensoren oder Aktoren zu erhalten. Mit einem Ersatzwiderstand statt der Spule eines Ventils kann sich die Regelungs-Software ganz anders verhalten als erwartet. Die fehlende Induktivität des Ersatzwiderstands kann zu einem anderen zeitlichen Verhalten des Steuerstroms und damit der Regelgröße führen.

Spezielle Testumgebungen für Embedded-Software

Zum Testen von Embedded-Software werden unterschiedliche Testumgebungen eingesetzt.

Für den Test der Applikationen bieten sich Simulationsumgebungen an. Damit können insbesondere auch Regelsysteme ausführlich getestet werden. Für einfache Applikationen können Sie ein Simulationsprogramm natürlich selbst schreiben. Bei komplexeren Anwendungen sollten Sie auf Simulationsumgebungen zurückgreifen, wie etwa MATLB/Simulink. Bauen Sie die Simulationsumgebungen bei Regelungen so auf, dass Sie diese als geschlossene Regelschleife (*closed loop*) und als offene Regelschleife (*open loop*) betreiben können. Die offene Regelschleife können Sie mit Messdaten aus der realen Umgebung beschicken und so die Reaktion der Software prüfen. In einer Simulationsumgebung wie MATLAB/Simulink können Sie Softwaremodelle prüfen, aus denen Sie später Code generieren (Model in the Loop = MIL). Wenn Sie den generierten Quellcode statt des Modells einbinden, können Sie in der Simulationsumgebung dessen Verhalten mit dem des Modells vergleichen (Software in the Loop = SIL, nicht zu verwechseln mit dem Safety Integrity Level, der genauso abgekürzt wird). Damit prüfen Sie gleichzeitig die Qualität des generierten Codes. Speziell wenn der Code mit anderer Rechengenauigkeit arbeitet als das Modell, ist dieser Vergleich interessant. Sie können statt generiertem Code auch Handcode in diese Simulationsumgebung einbinden und so testen.

Mit MIL und SIL können Sie komplexe Einsatzszenarien für die Software testen, aber auch einzelne Funktionalitäten der Software untersuchen.

Eine Reihe von Herstellern bietet auch Testumgebungen an, die auf Embedded-Software zugeschnitten sind. Sie können aber auch auf Open Source zurückgreifen. Für C-Code bietet sich hier das Werkzeug CUnit an (vgl. Abschnitt 6.2). Diese Testumgebungen sind für den Modultest geeignet, wobei einige Werkzeuge sowohl den Test auf dem Entwicklungsrechner (PC) als auch auf dem Zielsystem unterstützen.

Gerade bei hardwarenaher Software, wie Schnittstellentreibern, können einige Funktionalitäten nur in Verbindung mit der Hardware getestet werden. Dazu muss die Schnittstellenhardware mit Signalen stimuliert werden. Die Testaufbauten werden meist aus handelsüblichen Geräten zusammengestellt. Trotzdem sind oft einige Teile konfektioniert, etwa spezielle Kabel. Sorgen Sie deshalb dafür, dass diese Testaufbauten und ihre Verkabelung vollständig dokumentiert werden. Geräte werden ausgeliehen oder dringender woanders gebraucht und der Aufbau zerlegt. Ohne Dokumentation müsste der Aufbau aus

dem Gedächtnis rekonstruiert werden. Speziell angefertigte Kabel sollten Sie detailliert beschreiben, damit sie neu hergestellt werden können, wenn sie nicht mehr auffindbar sind.

Nach der Integration der Software und der Hardware können Sie das Embedded-System in einem Testbett prüfen, das alle Eingangssignale liefert und alle Ausgangssignale aufnimmt. Dabei simuliert das Testbett das Zielsystem, in welches das Embedded-System integriert wird. Ist dieses Testsystem programmierbar und berechnet aus den Ausgangssignalen und einem Modell des Gesamtsystems die Eingangssignale für das Testobjekt, spricht man auch von einem Hardware-in-the-Loop-System (HIL).

Testwerkzeuge

Neben den üblichen Software-Entwicklungswerkzeugen wie Compiler und Debugger werden für die Prüfung von Embedded-Software weitere Werkzeuge eingesetzt.

Codeanalyse-Werkzeuge

Der Code selbst wird mit Analysewerkzeugen untersucht. Diese Werkzeuge können nicht nur die Syntax des Codes prüfen, wie ein Compiler, sondern helfen, Codezeilen zu identifizieren, die zu möglichen Fehlern im Verhalten der Software führen können. Dazu zählen Wertebereichsüberschreitungen und Datenverluste bei Zuweisungen, Fehler bei der Adressrechnung und das Auffinden von Rekursionen.

Werkzeuge für den Modultest

Für den Modultest kommen neben Werkzeugen, die helfen, Testvektoren zu bestimmen, auch solche zum Einsatz, welche die Testabdeckung bestimmen. Damit erhalten Sie eine wichtige Aussage darüber, in welchem Umfang und in welcher Tiefe der Code durch die vorhandenen Testfälle geprüft wird.

Weitere Werkzeuge unterstützen den Test des Codes auf dem Zielsystem (Lade- und Kommunikationssoftware).

Signalquellen und Analysegeräte

Zu einer Testumgebung für hardwarenahe Software gehören neben der Zielhardware auch Signalgeneratoren und Anzeigegeräte, wie Multimeter, Oszilloskop, Logikanalysator oder Busanalysator.

Oft scheitert der Softwareentwickler an so einfachen Dingen wie der Polarität von Signalen. Mit einem Multimeter wissen Sie sofort, welche Spannungen wo anliegen. Mit Schaltplan oder ausführlicher Hardwarebeschreibung wird dann auch schnell erkennbar, wie der Signalwert an die Software weitergereicht wird.

Den zeitlichen Verlauf von Ein- und Ausgangssignalen des Embedded-Systems können Sie mit einem Oszilloskop untersuchen.

Logikanalysatoren helfen Ihnen weiter, wenn Sie den Datenfluss auf parallelen Schnittstellen beobachten wollen oder den zeitlichen

Zusammenhang zwischen mehreren digitalen Signalen, welche die Software auswertet, untersuchen müssen.

Busanalyse-Werkzeuge verwenden Sie, wenn nicht klar ist, wann welche Datenpakete vom Embedded-System über einen Datenbus geschickt werden oder empfangen werden. Für Ethernet ist eine Vielzahl freier Werkzeuge verfügbar. Wollen Sie den Datenverkehr auf Bussen wie CAN oder FlexRay beobachten, werden Sie für gute Werkzeuge einige Tausend Euro investieren müssen.

Dokumentieren Sie in jedem Fall die oft umfangreichen Einstellungen der Messgeräte und Analysatoren, die für die jeweiligen Tests relevant sind. Wenn ein Gerät ausgeliehen wurde, kommt es meist in anderer, für die Tests nicht verwendbarer Konfiguration wieder zurück. Falls Sie Ihre Einstellungen im Gerät speichern, auf Datenträger kopieren oder an den Bedienrechner der Testeinrichtung übertragen können, nutzen Sie diese Möglichkeit unbedingt. Manche Geräteeinstellungen werden aus Versehen gelöscht, manchmal auch bei einem Rücksetzen des Geräts in den Lieferzustand, etwa im Rahmen einer Kalibrierung.

Denken Sie daran, dass Messgeräte auch regelmäßig kalibriert werden müssen. Nur dann sind die abgelesenen Werte verlässlich. Als Softwareentwickler müssen Sie sich darum oft nicht selbst kümmern, aber bei der Vorbereitung von Tests sollten Sie darauf achten, dass die verwendeten Geräte alle kalibriert sind. Diese Information können Sie meist Aufklebern auf den Geräten entnehmen.

Prüfen der Testumgebung

Soweit vorhanden, führen Sie Tests, mit denen Sie die Testeinrichtung auf ihre Funktionstüchtigkeit prüfen können, vor Beginn der Software- oder Systemtests durch. Viele Fehler gehen oft auf Fehler in der Testumgebung zurück und nicht auf Fehler in der Software oder der Implementierung des Tests selbst.

Ich empfehle, die Testeinrichtung vor jedem Test eines neuen Release der Software zu prüfen. Führen Sie häufig Tests durch, ist meist die Prüfung der Teile der Testeinrichtung ausreichend, die beim letzten Mal Schwächen gezeigt haben.

Kümmern Sie sich auch um Wartung und Support der Testsoftware und Testhardware. Oft sind Testgeräte auch geliehen, oder die Lizenz der Testsoftware läuft nur eine begrenzte Zeit. Wenn kurz vor der Auslieferung der Software die Testeinrichtung nicht benutzt werden kann, weil eine Lizenz ungültig geworden ist, ist dies äußerst ärgerlich.

6.1.4 Testvorbereitung

Wenn Sie Tests durchführen wollen, prüfen Sie, ob alle Voraussetzungen für die Testdurchführung erfüllt sind:

- Ist die Testumgebung funktionstüchtig?
  Sind alle Komponenten (Geräte, Kabel, Software usw.) vorhanden und einsatzbereit?
- Ist die Software, die getestet werden soll, für die Tests freigegeben?
  Wurde verifiziert, dass in der Software alle Anforderungen umgesetzt sind, und wurden bereits bekannte Fehler und Einschränkungen dokumentiert?
- Sind die Testspezifikationen und Protokollvorlagen freigegeben?
- Wurde der Testumfang gemäß der Teststrategie festgelegt?
  Je nach Aufgabenstellung sind die Testumfänge verschieden. Für interne Versuche im Zielsystem sind vermutlich weniger Tests erforderlich als für die Freigabe zur Serienproduktion.

Legen Sie fest, welche Tests zu welchem Zweck durchgeführt werden müssen und dokumentieren Sie dies in der Testspezifikation. Markieren Sie bei jedem Test, für welche Teststufe dieser durchgeführt werden muss.

Zur Vorbereitung des Tests gehört auch das Ausfüllen eines Protokolls, in dem die Testdurchführung dokumentiert wird. Was alles im Testprotokoll festgehalten werden sollte, finden Sie in Abschnitt 6.1.2.

6.1.5 Testdurchführung

Wenn Sie einen Softwaretest durchführen, verwenden Sie die aktuellen und freigegebenen Unterlagen. Halten Sie sich an die dokumentierten Arbeitsschritte. Dokumentieren Sie das Ergebnis eines einzelnen Testschritts eindeutig mit *bestanden* oder *nicht bestanden*. Nicht verwenden sollten Sie Häkchen, Striche, Kreuzchen oder dergleichen. Fehlinterpretationen bei der Auswertung der Protokolle sind die Folge.

Notieren Sie Messergebnisse, damit später festgestellt werden kann, wo innerhalb der Grenzen des von der Testspezifikation vorgegebenen Soll-Ergebnisses das Testergebnis liegt.

Markieren Sie Tests, die Sie ausgelassen haben, mit *nicht durchgeführt*. Notieren Sie die Gründe dafür.

Wenn Sie feststellen, dass sich der Test so nicht durchführen lässt, ist das bei Embedded-Software, insbesondere wenn Hardware und Signalquellen verwendet werden, keine Seltenheit. Oft wird etwas übersehen, oder manchmal wurde auch eine Einstellung an den Geräten oder

der Hardware verändert: Ein Jumper sitzt nicht korrekt, oder eine Adresse wurde nicht korrekt eingestellt.

Mein Tipp:

Ändern Sie die Einstellungen ab, so dass Sie den Test durchführen können, und dokumentieren Sie Ihre Änderungen im Testprotokoll. Werten Sie nach dem Test das Protokoll aus und optimieren Sie die Testspezifikationen.

Dokumentieren Sie alle Auffälligkeiten. Notieren Sie auch zusätzliche oder geänderte Einstellungen an Instrumenten, die Sie vorgenommen haben. Keine Spezifikation ist vollständig, weder die Testspezifikation noch die Vorgaben, die der Softwareentwickler hatte.

6.1.6 Ende des Tests

Kriterien für das Testabbruch

Legen Sie fest, wann ein Test abgebrochen wird. Eine Testsequenz bis zum Ende durchzuführen, wenn die Fortsetzung der Testsequenz nicht mehr sinnvoll ist und die Testergebnisse keine weiteren verwertbaren Informationen liefern, ist unrentabel und ineffizient. Mögliche Abbruchkriterien sind:

- Anzahl der fehlgeschlagenen Tests
- Fehlschlag eines (sicherheits-)kritischen Tests

Testauswertung

Werten Sie am Ende der jeweiligen Testphase (Modultest, Software-Integrationstest, HW-/SW-Integrationstest, Systemtest) die Testprotokolle auswerten.

Automatisierte Protokollbewertung

Protokolle von automatisierten Tests können meist auch automatisch ausgewertet werden. Ob sich die automatisierte Auswertung lohnt, hängt vom Aufwand im Verhältnis zu der Anzahl der durchzuführenden Tests ab.

Protokollbewertung mit Checklisten

Erstellen Sie eine Checkliste, in der Sie festlegen, wonach Sie in den Testprotokollen schauen. Die Testauswertung bekommt dadurch den Charakter eines Reviews. Dokumentieren Sie die Ergebnisse der Auswertung.

Die Ergebnisse der Auswertung sind eine wichtige Quelle zur Verbesserung der Testspezifikationen, der Tests und der Testumgebung.

Auswertung fehlgeschlagener Tests

Sind Tests fehlgeschlagen, sollten Sie zuerst die Ist-Ergebnisse mit den Soll-Ergebnissen vergleichen. Liegt kein Spezifikationsfehler vor und auch kein Fehler in der Testumgebung, haben Sie höchstwahrscheinlich einen Softwarefehler gefunden. Zur Sicherheit sollten Sie überprüfen, wie die ursprüngliche Anforderung an die Software lautet und ob diese und der Testfall zusammenpassen. Die Anforderungen, welche mit dem Testfall verifiziert werden sollen, finden Sie über die Anforderungsverfolgungsmatrix. Spiegelt der Testfall die Anforderung korrekt wider, kann aus dem Testergebnis oft schon geschlossen werden, welcher Fehler in der Software vorliegt.

Wurden Tests nicht durchgeführt, prüfen Sie die Begründungen. Tests können etwa wegen fehlgeschlagener anderer Tests übersprungen worden sein.

Wurden Fehler in der Testumgebung gefunden und behoben, müssen Sie überlegen, ob Sie nur den Test neu durchführen, bei dessen Ausführung Sie den Fehler in der Testumgebung gefunden haben oder ob Sie alle Tests nochmals durchführen. Die Änderung der Testumgebung kann Auswirkungen auf andere Tests haben.

Am Ende der Auswertung sollte Sie auflisten, welche Tests nach Behebung der gefundenen Fehler in Software, Testumgebung und Testspezifikation nochmals durchgeführt werden.

Teststatistik

Führen Sie eine Statistik und erfassen Sie für jede Testphase, wie viele Tests jeweils *bestanden*, *nicht bestanden* oder *nicht durchgeführt* wurden. Dokumentieren Sie auch die Werte aus der Code-Überdeckungsanalyse der Tests. Dies hilft, einen Eindruck von der Reife der Software und der Tests zu bekommen.

Freigabe

Nach dem Auswerten der Testprotokolle wird die Software freigegeben. Diese kann am einfachsten im Review-Protokoll der Testergebnisse dokumentiert werden. Wichtige Freigaben, etwa bei Übergabe an den Kunden oder an die Serienfertigung, sind aufwendiger. Dabei prüfen Sie meist zusammen mit dem Qualitätssicherungsbeauftragten, ob alle festgelegten Freigabekriterien erfüllt sind. Neben dieser schriftlichen Bestätigung ist die Liste der bekannten Mängel und Einschränkungen das wichtigste Dokument. Dieses unterziehen Sie im Rahmen der Freigabe einem Review. Bei diesem Review muss auch jemand teilnehmen, der die Auswirkungen der Einschränkungen und Mängel auf die weiteren Tests oder Arbeiten im Projekt beurteilen kann.

Thomas Eißenlöffel

Embedded-Software entwickeln

Grundlagen der Programmierung eingebetteter Systeme – Eine Einführung für Anwendungsentwickler

dpunkt.verlag

Thomas Eißenlöffel
E-Mail: thei@gmx.de

Lektorat: René Schönfeldt
Copy-Editing: Annette Schwarz, Ditzingen
Herstellung: Nadine Thiele
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN 978-3-89864-727-4

1. Auflage 2012
Copyright © 2012 dpunkt.verlag GmbH
Ringstraße 19 B
69115 Heidelberg

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

Vorwort

Hier lesen Sie

- für wen dieses Buch interessant ist
- was Sie darin erfahren
- und was Sie darin nicht erfahren

Wer sollte dieses Buch lesen?

Embedded-Einsteiger

Wenn Sie gerade Ihr Informatikstudium abgeschlossen oder sich bisher mit der Anwendungsentwicklung für Arbeitsplatzrechner beschäftigt haben oder nur mit Programmen zu tun hatten, die auf Bildschirm, Tastatur und Dateien zugreifen, und nun Software für Embedded-Systeme entwickeln wollen, stehen Sie vor vielen neuen Herausforderungen.

Denn Software für Embedded-Systeme unterscheidet sich teilweise sehr stark von Anwendungsprogrammen, die auf Arbeitsplatzrechnern (wie PC oder Mac) oder auf Servern im Netz laufen und von Anwendern bedient werden. Embedded-Software muss etwa in einer Aufzugssteuerung Sensoren überwachen und Motoren ein- und ausschalten.

Dieses Buch hilft Ihnen, sich in der für Sie noch neuen Welt der Embedded-Systeme, insbesondere auch der Echtzeitsysteme zurechtzufinden.

Entwickler mit Embedded-Erfahrung

Haben Sie bereits Software für Embedded-Systeme entwickelt, gibt Ihnen dieses Buch Anregungen, Tipps und beschreibt Vorgehensweisen, die ich in langjähriger Praxis in verschiedenen Branchen kennengelernt habe.

Falls Sie sich zwar in einem bestimmten Teilbereich der Embedded-Softwareentwicklung gut auskennen, wie z.B. der Implementierung, aber bisher mit anderen Entwicklungsschritten wie Architektur, Design oder Test nur wenig zu tun hatten, hilft Ihnen dieses Buch, einen Überblick über die gesamte Bandbreite der Entwicklungstätigkeiten zu bekommen und besser zu verstehen, wie die verschiedenen Aktivitäten zusammenhängen und ineinandergreifen.

Was erfährt der Leser?

Dieses Buch gibt im Grundlagen-Kapitel einen Überblick darüber, was Sie als Softwareentwickler im Embedded-Bereich erwartet und was dort anders ist als im Umfeld der betrieblichen Anwendungsentwicklung.

Die Kapitel nach dieser Einführung beschreiben die Entwicklung von Embedded-Software, die zur Steuerung und Regelung auf vielen Gebieten eingesetzt wird, z.B. in Maschinen und Medizintechnik, in Kraftfahrzeugen und Flugzeugen, in Telefon- und Industrieanlagen. Die Beschreibung stützt sich dabei auf einen Entwicklungszyklus, der von der Analyse der Anforderungen über Architektur, Design und Implementierung bis hin zur Integration und zum Softwaretest reicht und auch Planungs-, Qualitäts- und Sicherheitsaspekte berücksichtigt.

In jedem Prozess-Schritt des Software-Entwicklungszyklus werden die speziellen Anforderungen eines Embedded-Systems bzw. Echtzeitsystems herausgestellt und die Maßnahmen anhand von Beispielen aus der Praxis begründet. Für typische Probleme werden Lösungen vorgestellt und diskutiert. Die dabei vermittelten Tipps werden Ihnen helfen, sichere, schnelle und wartungsfreundliche Software für Embedded-Systeme zu entwickeln.

Die einzelnen Kapitel bauen aufeinander auf, können aber je nach Aufgabenstellung und Interesse auch separat durchgearbeitet werden, setzen allerdings das Know-how aus dem Grundlagen-Kapitel voraus.

Echtzeitsysteme

Da sehr viele Embedded-Systeme Echtzeitsysteme sind, bei denen die rechtzeitige Reaktion oder Abarbeitung einer Aufgabe höchste Priorität hat, beleuchte ich diesen Aspekt in allen Entwicklungsschritten besonders.

Sie werden in diesem Buch viele wertvolle Tipps aus der Praxis finden, etwa zur Vorgehensweise bei Entwicklung und Test. Diese helfen Ihnen, die Robustheit und Antwortzeit Ihres Systems auch dann zu verbessern, wenn Echtzeit für Ihre Aufgabenstellung keine Anforderung ist.

Die Grundlagen

Das Grundlagenkapitel vermittelt die in den darauf folgenden Kapiteln verwendeten Grundbegriffe.

Systemarchitekturen

Wie die Architekturen von Embedded-Systemen und ihrer Software aufgebaut sind und sich von der PC-Architektur unterscheiden, erläutert dieser Abschnitt.

Viele Embedded-Systeme sind Echtzeitsysteme. Echtzeitanwendungen verwenden deshalb spezielle Betriebssysteme. Das Grundlagenkapitel erklärt deren Grundfunktionen und deren spezielle Echtzeitbetriebssystemdienste.

Echtzeitbetriebssysteme

Es folgt die Beschreibung eines Entwicklungsprozesses, an dem sich die Gliederung der Folgekapitel orientiert.

Entwicklungsprozess

Was Sie bei der Entwicklung von Embedded-Software dokumentieren sollten und wie Sie mit möglichst wenig Aufwand Softwaredokumentation erzeugen, wird ebenfalls im Grundlagenkapitel erklärt. Beispielcode zeigt, wie man Dokumentation bereits bei der Implementierung quasi »nebenbei« mitschreibt.

Dokumentation

Auch wenn Sie primär Software entwickeln, werden Sie hin und wieder mit der Planung konfrontiert. Sowohl bei der Zeit- als auch bei der Ressourcenplanung müssen Sie Aufwände speziell für die Entwicklung von Embedded-Software berücksichtigen.

Planung

Qualitätsanforderungen im Embedded-Bereich unterscheiden sich sehr stark von denen für PC- und Server-Software. Mit Fehlern muss in Embedded-Systemen anders umgegangen werden, da eine korrigierte Softwareversion nicht so einfach in die ausgelieferten Geräte geladen werden kann.

Qualitätssicherung, Reviews und Tests

Reviews sind ein unverzichtbares Werkzeug zur Qualitätssicherung von Embedded-Software. Das Grundlagenkapitel beschreibt neben der prinzipiellen Vorgehensweise auch, in welchen Prozess-Schritten Reviews die Qualität von Embedded-Software verbessern.

Welche Qualität die Tests von Embedded-Software haben müssen, wird in diesem Zusammenhang ebenfalls erklärt.

Ein Abschnitt im Grundlagenkapitel beschreibt schließlich das Thema *Sicherheit*. Der Schwerpunkt wird dabei eher auf der Funktionssicherheit (engl. *functional safety*) und weniger auf der Zugriffssicherheit (engl. *security access*) liegen. Der Grund hierfür ist, dass Anforderungen an die funktionale Sicherheit von Embedded-Software enormen Einfluss sowohl auf den Entwicklungsprozess als auch auf die Entwicklungswerkzeuge und die Softwarearchitektur haben. Wer Software für Embedded-Systeme erstellt, deren Fehlfunktion eine Gefahr für Leib und Leben von Menschen darstellen könnte, muss z.B. eine Norm wie die IEC 61508 oder ISO 26262 berücksichtigen.

Sicherheit

Der Entwicklungsprozess

Die auf das Grundlagenkapitel folgenden Kapitel sind nach dem sogenannten *V-Modell* gegliedert, an dem sich der Ablauf einer Softwareentwicklung im Embedded-Bereich häufig orientiert. Das *V-Modell*

Das V-Modell

beschreibt einen Entwicklungsprozess, der in mehrere Phasen unterteilt ist: Anforderungsanalyse, Architektur, Design, Implementierung und verschiedene Test- und Integrationsphasen.

Falls Sie das V-Modell nicht kennen, gibt Ihnen die nachfolgende Kurzbeschreibung der Prozess-Schritte eine Einführung, und Sie erfahren, in welchem Prozess-Schritt welche Aktivitäten ablaufen und in welchem der Folgekapitel welche Themen behandelt werden.

Angepasste Entwicklungsprozesse

Natürlich ist der Ablauf der Softwareentwicklung in einem spezifischen Projekt im Detail von vielen Faktoren abhängig. Wird in Ihrem Projekt etwa nur ein Algorithmus angepasst, werden Sie wahrscheinlich den ein oder anderen Prozess-Schritt nicht (z. B. Architekturentwurf) oder nur verkürzt (z. B. Design) durchführen.

Dann können Sie sich die passenden Kapitel herausgreifen bzw. die nicht relevanten auslassen.

Andere Prozessmodelle

Auch wenn Sie für Ihre Vorgehensweise nicht das V-Modell verwenden, werden Sie die im V-Modell beschriebenen Tätigkeiten ausführen, nur die Reihenfolge und Zuordnung ihrer Aktivitäten zu Prozess-Schritten ist dann anders.

Anforderungsanalyse

Das Kapitel *Anforderungsanalyse* beschreibt, nach welchen Informationen Sie in erster Linie suchen müssen und welche Quellen dafür in Frage kommen. Besonders wichtig sind Informationen über Zeitverhalten, Qualitätsaspekte und funktionale Sicherheit.

Architektur

Das Kapitel *Architektur* geht auf die unterschiedlichen Betriebsumgebungen von Embedded-Systemen ein, welche die Architektur von Embedded-Software wesentlich bestimmen.

Design

Das Kapitel *Design* erklärt, wie die spezifischen Eigenschaften von Software für Embedded-Systeme Einfluss auf den Softwareentwurf nehmen, und betrachtet sowohl eingebettete (weitgehend hardwareunabhängige) Betriebsfunktionen als auch hardwarenahe Software.

Ist die Vorgehensweise beim Design der Betriebsfunktionen noch sehr ähnlich zur Anwendungsentwicklung für PC und Server, so unterscheiden sich die Strategien und Vorgehensweisen beim Design der hardwarenahen Software beträchtlich. Dies wird an Beispielen verdeutlicht.

Implementierung

Das Kapitel *Implementierung* erläutert, worauf es bei der Erstellung von Quellcode für Embedded-Systeme besonders ankommt und geht auf verschiedene Programmiersprachen und die Generierung von Code ein. Beispiele sind in der Sprache »C« verfasst. Neben Eigenheiten von Cross-Compilern beschreibt das Kapitel auch, nach welchen Gesichtspunkten bei Embedded-Systemen die Software optimiert wird und wie man dabei vorgeht.

Test

Das Kapitel *Test* beschreibt nach der Teststrategie die Testphasen *Modultest*, *Integrationstest* und *Anforderungstest* und wie diese mit den Entwicklungsschritten zusammenhängen. Auch wirken Softwareingenieure beim Hardware-/Software-Integrationstest, Komponententest und Systemtest mit.

Modultest

Der Abschnitt *Modultest* beschreibt das Vorgehen beim Test einzelner Module und wie man die spezifischen Eigenheiten von hardwarenaher Software und Echtzeitsoftware berücksichtigt. Der Begriff der Testabdeckung wird erläutert und es werden Beispiele für verschiedene Abdeckungsgrade gegeben. Verschiedene Werkzeuge werden vorgestellt und ihre Vor- und Nachteile diskutiert.

Integration und Integrationstest

Der Abschnitt *Integration und Integrationstest* stellt Vorgehensweisen bei der Softwareintegration vor und was beim Integrationstest geprüft wird.

Software-Anforderungstest

Der *Software-Anforderungstest* zielt darauf ab, die Umsetzung der Anforderungen an die Software als Einheit nachzuprüfen. Liegt der Fokus bei der Softwareintegration auf der Interaktion der Softwaremodule miteinander, so steht beim Anforderungstest das Zusammenspiel der Software mit seiner Umgebung im Vordergrund.

Hardware-/Softwareintegration, Komponententest und Systemtest

Da ein Gesamttest der integrierten Software oft nicht vollständig ohne die reale Hardware möglich ist, gehen diese Abschnitte auf die hardwarenahen Aspekte des Gesamttests und die betroffenen typischen Funktionen eines Embedded-Systems ein und beschreiben Werkzeuge und Testumgebungen und die Grenzen, an die man damit stößt.

Ausblick

Nach den Kapiteln über den Test der Embedded-Software gehe ich auf die Aufgaben ein, die auf Sie als Softwareentwickler nach Auslieferung Ihrer Software im Zusammenhang mit dem Entwicklungsprojekt zukommen könnten.

Der zweite Abschnitt dieses Kapitels zeigt einige Trends in der Softwareentwicklung für Embedded-Systeme auf.

Glossar

Im Glossar werden alle wichtigen Fachbegriffe und Abkürzungen erläutert. Dort finden Sie auch eine Liste von Internetadressen, auf die im Text verwiesen wird.

Aufgaben des Entwicklers

Dieses Buch beschreibt nicht nur die Codierung von Embedded-Software, sondern alle Arbeiten im Software-Entwicklungsprozess, inkl. zusätzlicher Aufgaben wie Planung und Qualitätssicherung.

Falls Sie noch nicht im Embedded-Bereich gearbeitet haben, sollten Sie berücksichtigen, dass Ihnen je nach Arbeit- oder Auftraggeber nur ein Teil dieser Aufgaben übertragen wird. Bestimmte Aufgaben,

wie etwa das Testen, werden von Großunternehmen oft an darauf spezialisierte Firmen vergeben und nicht selbst durchgeführt.

In größeren Unternehmen werden Sie meist nur eine einzelne Aufgabe ausführen, z.B. Spezifizieren oder Testen, und damit nur die Rolle des Spezifikations- bzw. Testingenieurs übernehmen. In Konzernen werden in Projektteams oft neben eigenen Entwicklern auch Fremdarbeitskräfte von Ingenieurdienstleistern beschäftigt, die als erfahrene Spezialisten bestimmte Aufgaben im Entwicklungsprozess übernehmen und das Team für mehrere Monate oder wenige Jahre verstärken.

In kleinen und mittleren Unternehmen werden Sie viele der Aufgaben als Entwickler selbst bearbeiten und damit verschiedene Rollen übernehmen: Architekt, Designer, Tester. In kleinen Projekten gibt es meist keine bestimmte Person für die Rolle des Systemingenieurs. Stattdessen müssen sich Software- und Hardwareentwickler diese Rolle teilen. Da Sie sich als Softwareentwickler intensiver mit der Funktionsweise der Applikation auseinandersetzen, fällt Ihnen hier meist die Verantwortung zu.

Was erfährt der Leser nicht?

Dieses Buch ist als Einführung in Embedded-Softwareentwicklung gedacht. Es gibt zwar Beispiele aus verschiedenen Branchen, das Buch ersetzt aber nicht das Studium branchenspezifischer Details, etwa Normen und Standards wie der EN 61131, welche die Grundlagen speicherprogrammierbarer Steuerungen (SPS) behandelt.

Der in diesem Buch skizzierte Software-Entwicklungsprozess dient der Strukturierung der vermittelten Informationen. Das Buch erhebt nicht den Anspruch, einen vollständigen Entwicklungsprozess zu beschreiben.

Zwar geht das Buch auf viele für Embedded-Systeme relevanten Testarten ein und vermittelt Tipps zum Test von Embedded-Software, es ist aber kein Fachbuch für Teststrategien und Testkonzepte.

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.

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.

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.

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.

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).

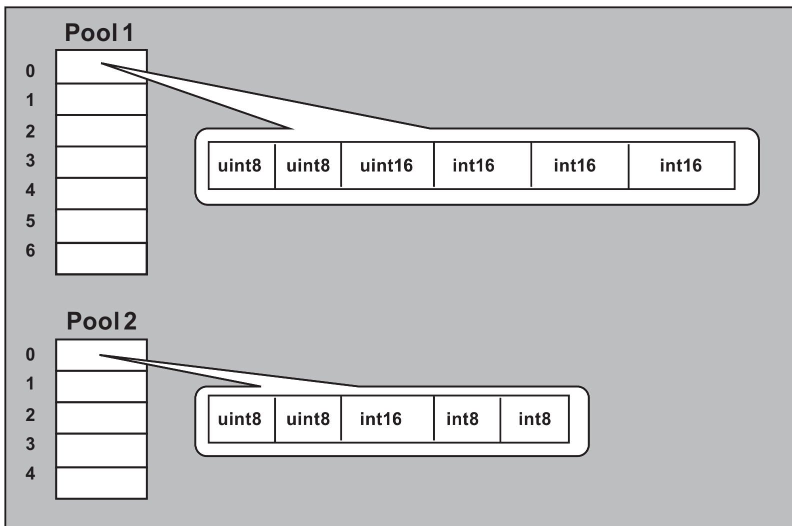


Abb. 4–3
Daten-Pools fixer Länge

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.

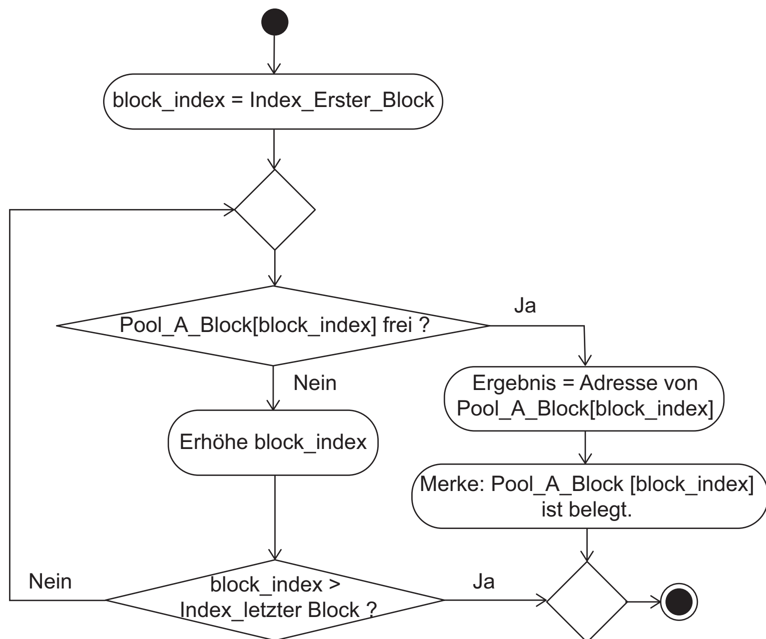
- Das Suchen von Datenblöcken bestimmten Inhalts in einem Pool ist schneller, wenn über Indexlisten adressiert wird statt verkettete Listen zu durchlaufen.
- Das Anordnen von dynamischen Daten nach bestimmten Kriterien (Sortieren) wird einfacher.

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.

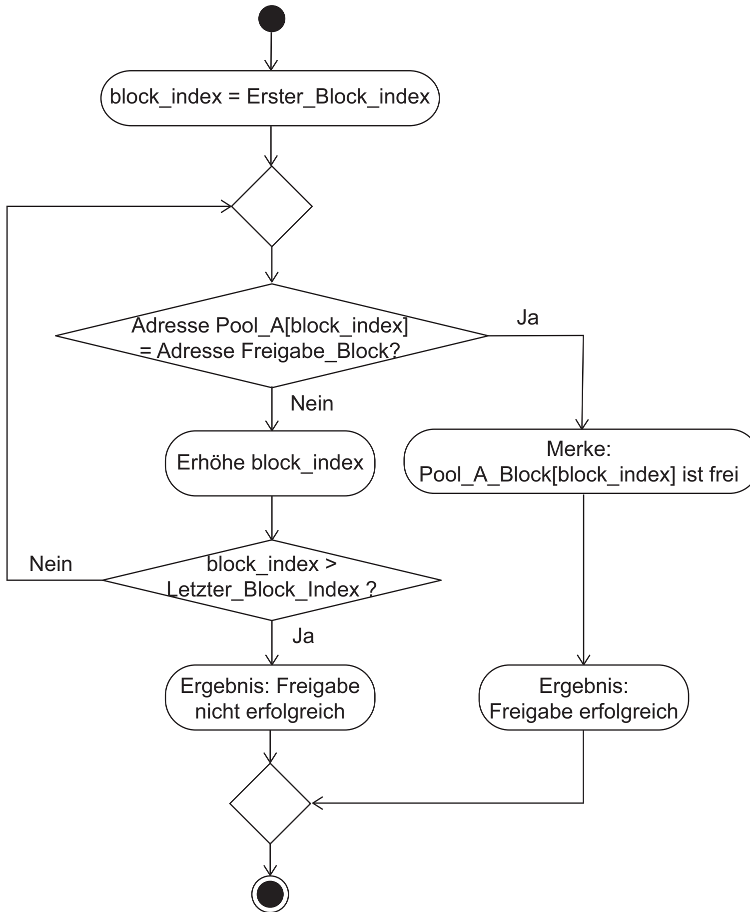
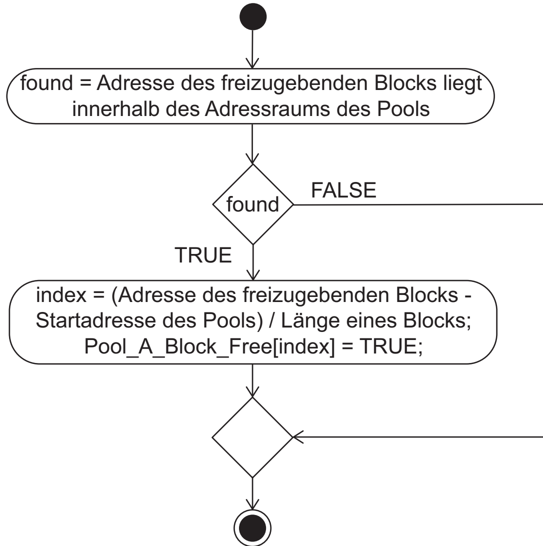


Abb. 4–5
Freigabefunktion

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.

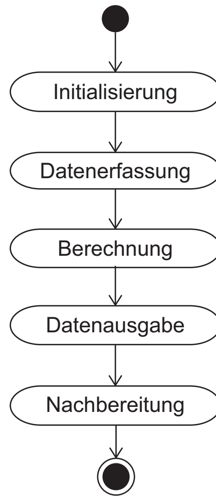


Abb. 4–7
Funktionsgliederung

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:

- 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

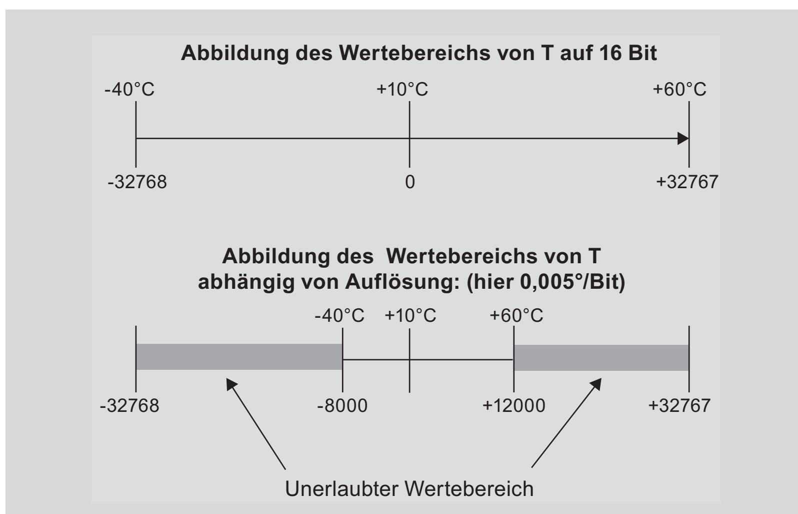
Ein Temperatursensor liefert Temperaturwerte zwischen -40°C und +60°C mit einer Genauigkeit von 0,005°C.

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

- 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.