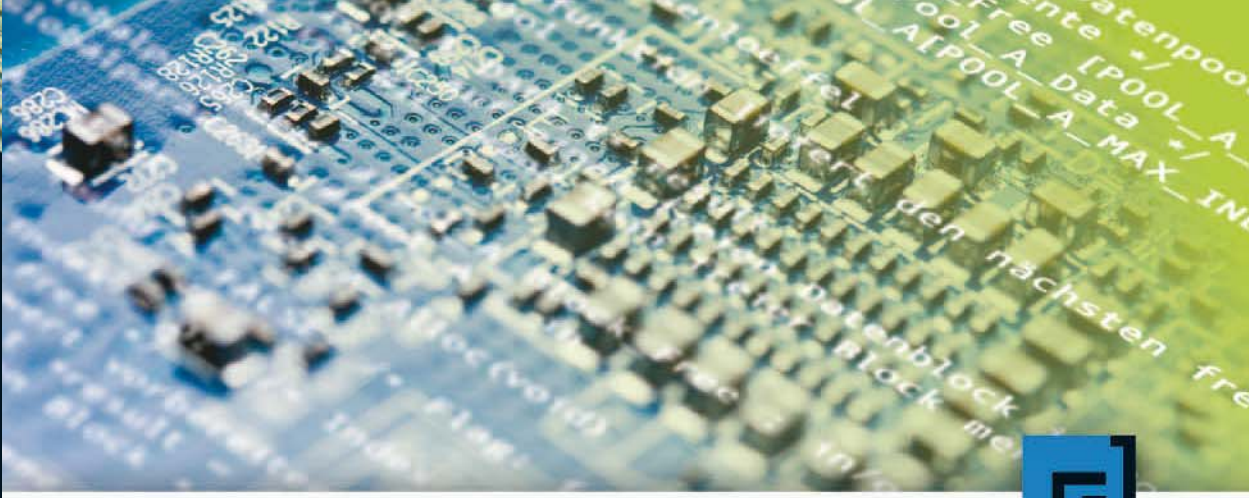


Thomas Eißenlöffel

Embedded-Software entwickeln

Grundlagen der Programmierung eingebetteter Systeme – Eine Einführung für Anwendungsentwickler

dpunkt.verlag

Dipl.-Ing. Thomas Eißenlöffel studierte Elektrotechnik an der Universität Karlsruhe (TH). Anschließend ging er zu Alcatel in die Software-Entwicklung für Funkmessdatensysteme. Nach seiner Tätigkeit als weltweit verantwortlicher technischer Produktmanager bei Force Computers leitet er bei der MBtech Group u.a. Projekte für Steuergeräte-Software. Er ist SPICE Assessor und Senior Expert für Embedded Software und Safety.

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:
Buch 978-3-89864-727-4
PDF 978-3-86491-099-9
ePub 978-3-86491-100-2

1. Auflage 2012

Ringstraße 19 B
69115 Heidelberg

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.

Echtzeitbetriebssysteme

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

Entwicklungsprozess

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

Dokumentation

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.

Planung

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.

Qualitätssicherung, Reviews und Tests

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.

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.

Sicherheit

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.

Der Entwicklungsprozess

Das V-Modell

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*

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.

Inhaltsverzeichnis

1 Grundlagen

Zum besseren Verständnis der in den Folgekapiteln beschriebenen Themen und Problemlösungen werden hier notwendige Grundbegriffe erläutert.

Sie lernen Hardwarearchitekturen von Embedded-Systemen kennen und wie sie sich von einer PC-Architektur unterscheiden. Viel Neues erfahren Sie über den Umgang mit Echtzeitanforderungen an die Software, über hardwarenahe Software und robustes Design.

Die einzelnen Abschnitte sind in ihrer Reihenfolge so angeordnet, dass sie auf dem Wissen des Vorgängerabschnitts aufbauen. Sie können aber zum Nachschlagen und Vertiefen eines Themas auch dort direkt einsteigen. Behandelt werden dort:

- Hardwarearchitekturen
- Zeitverhalten
- Betriebssysteme
- Entwicklungsumgebungen
- Entwicklungsprozess
- Konfigurations- und Variantenmanagement
- Qualitätssicherung
- Sicherheit
- Dokumentation

1.1 Hardwarearchitekturen

Anhand einer Gegenüberstellung von typischen Embedded-Hardwarearchitekturen und Embedded-Anwendungen einerseits und der Hardwarearchitektur von sogenannten Mehrzweck-Computern (engl. »general purpose computer«) und ihren Anwendungen für Büro und Freizeit andererseits werde ich Ihnen im Folgenden anschaulich den Unterschied zwischen beiden Welten deutlich machen.

PC-Architektur

Anwendungen wie Browser, Tabellenkalkulationsprogramme oder Textverarbeitungssysteme laufen üblicherweise auf (teilweise mobilen) Geräten wie PCs und Laptops, die über Tastatur und Bildschirm bedient werden. Die Daten, die diese Anwendungen verarbeiten, und die Anwendungen selbst werden auf nichtflüchtigen Speichermedien wie Festplatten oder Flash mit Kapazitäten im Gigabyte-Bereich gespeichert. Der Datenaustausch mit anderen Rechnern erfolgt über ein Datennetz, entweder ein lokales Netzwerk (LAN), das Internet oder Datenträger, wie DVDs oder USB-Sticks. Und die Anwendungen selbst sind auf die Bedienung durch den Menschen zugeschnitten. Deshalb nimmt die Bedienerschnittstelle, besonders die Darstellung der Daten auf einem Bildschirm, breiten Raum ein (siehe Abb. 1–1).

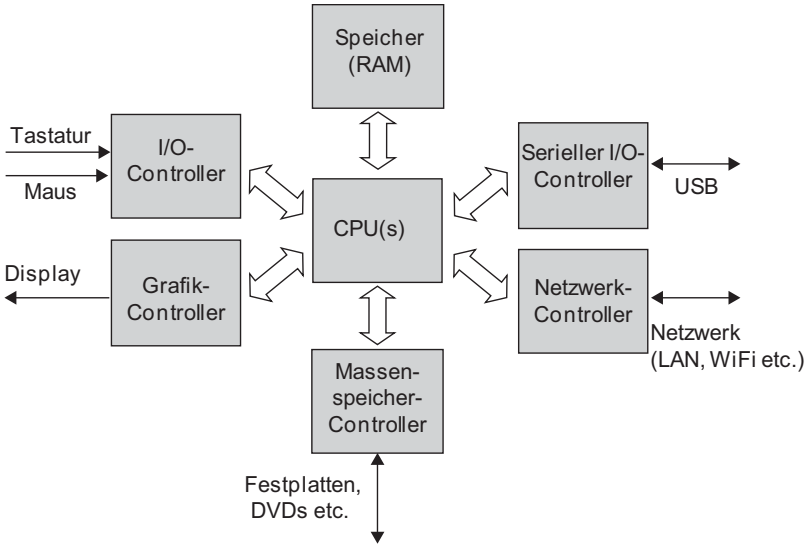


Abb. 1–1
Typische Hardwarearchitektur eines Mehrzweck-Computers für Büroanwendungen (z. B. PC)

Embedded-Systeme

Embedded-Systeme werden, im Gegensatz zum Mehrzweck-Computer, für eine bestimmte Anwendung entworfen und programmiert. Ein Embedded-System ist oft Teil einer Maschine oder Anlage (wie etwa einer Waschmaschine, eines Autos oder Kernkraftwerks) und enthält meist neben der Computer-Hardware und der Embedded-Software auch elektromechanische Komponenten.

Anwendungen von Embedded-Systemen überwachen, steuern und regeln Prozesse. Dabei werden oft zyklisch Datenquellen ausgewertet und Ausgangsdaten erzeugt. Embedded-Anwendungen werden nach Einschalten der Embedded-Systeme aktiv und müssen nicht wie PC-Programme über Tastatureingaben oder Mausklicks separat gestartet und beendet werden.

Embedded-Systeme haben deshalb meist keine Anschlussmöglichkeit für die vom Arbeitsplatzrechner gewohnten Bediengeräte Bildschirm, Tastatur und Maus und werden auch oft als *headless systems* bezeichnet. Oft fehlen auch weitere PC-typische Schnittstellen, wie der *Universal serial bus* (USB). Festplatten sind eher die Ausnahme, dafür ist ein nichtflüchtiger Speicher die Regel. Dieser Speicher (heute meist Flash-Speicher) verfügt aber selten über ein Dateisystem.

Embedded-Systeme bestehen im Wesentlichen aus einem Prozessor (CPU) und zusätzlicher Spezialhardware. Diese zusätzliche Hardware wandelt die Eingangsdaten für die Verarbeitung im Prozessor um und bereitet die Ausgangsdaten auf, beispielsweise zur Steuerung einer Maschine. Da Embedded-Systeme oft in großen Stückzahlen hergestellt werden und klein und preisgünstig sein müssen, wird Hardware mit hohem Integrationsgrad verwendet. Je mehr Peripherie auf dem Prozessorchip integriert ist, desto weniger Bauelemente sind dann auf der Platine und desto einfacher und billiger wird das Hardwaredesign. Häufig verwenden Embedded-Systeme deshalb andere Prozessorarchitekturen als die im Desktop- und Serverbereich verbreiteten Chips auf Basis der x86-Architektur.

Im Embedded-Bereich werden Prozessoren unterschiedlichster Wortbreite (8, 16 oder 32, selten 64 Bit) eingesetzt. Die Chips verfügen meist über nur einen Prozessorkern. In Anwendungen, die viel Rechenzeit benötigen, kommen zunehmend auch Mehrkernprozessoren zum Einsatz. Der Vorteil von mehreren Rechenkernen niedrigerer Taktfrequenz gegenüber einem einzigen Kern mit höherer Taktrate liegt im insgesamt geringeren Stromverbrauch dieser Designs bei vergleichbarer Rechenleistung. Intel berichtet beispielsweise, dass das Absenken der Taktfrequenz um 20 Prozent die Leistungsaufnahme um die Hälfte reduziert, die Rechenleistung sich dabei aber nur um 13 Prozent verringert (siehe [INT08]).

Embedded-Systeme verwenden oft Prozessoren mit einer Kombination unterschiedlicher Kerne, wobei meist ein Standardrechenkern etwa auf Basis der ARM-Architektur für Verwaltungs- und Netzwerkkommunikation eingesetzt wird und der oder die anderen Kerne für die Datenverarbeitung. Diese Kerne sind etwa digitale Signalprozessoren (DSPs), Gleitkommaprozessoren (FPUs) oder Grafikprozessoren (GPUs).

Auf den in Embedded-Systemen eingesetzten Chips sind häufig RAM, Flash-Speicher und Peripheriebausteine integriert. Diese sog. Mikrocontroller (MCUs) enthalten beispielsweise CAN-Controller, Zeitgeber(*Timer*), universelle Ein-/Ausgabebausteine (GPIO), Analog-/Digitalwandler (ADC), Digital-/Analogwandler (DAC) und spezielle

Überwachungslogik (sog. *Watchdogs*). Diese Chips gibt es oft in mehreren Varianten, die auf bestimmte Anwendungen zugeschnitten sind.

Wenn Daten von einem 12-Bit-A/D-Wandler geliefert werden, reicht meist ein 16-Bit-Prozessor aus, um diese zu verarbeiten. Würde man statt einem 16-Bit-Prozessor nun einen 32-Bit-Prozessor verwenden, so würden die zusätzlichen 16 Bit nur wenig Vorteile bringen, da die Genauigkeit der Daten ja auf 12 Bit beschränkt ist. Geringe Vorteile können durch einen 32-Bit-Prozessor entstehen, wenn sehr viele Multiplikationen und Divisionen in der Signalverarbeitung erforderlich sind. Um eine ausreichende Genauigkeit sicherzustellen, müssen diese oft mit 32 Bit Genauigkeit ausgeführt werden. Auf einem 16-Bit-Prozessor benötigen 32-Bit-Operationen viele Prozessorzyklen und damit mehr Rechenzeit als 16-Bit-Operationen.

Die Leistungsfähigkeit solcher Chips ist häufig deutlich geringer als die von PC-Prozessoren, sowohl was die Rechenleistung angeht als auch den adressierbaren Speicherbereich.

Speicher in Embedded-Systemen kostet Geld, also wird bei großen Stückzahlen an Speicher gespart. Insbesondere Speicher, der nicht auf dem Mikrocontroller selbst integriert ist, kostet Platinenfläche für die RAM-Chips. Dies verlängert die Entwicklungszeit der Hardware, erhöht den Testaufwand der Hardware (je mehr Chips, desto mehr Tests) und damit die Hardwareentwicklungskosten. Dazu verringert jeder zusätzliche Baustein die Zuverlässigkeit der Hardware, und das Risiko eines Ausfalles eines Bausteins und damit des Embedded-Systems steigt. Meist sind wenige hundert Kilobyte RAM ausreichend. Viele 16-Bit-Prozessoren können nur einen kleinen Speicherbereich direkt adressieren. Benötigen Software und Daten mehr Speicher, kann dieser über sogenannte Segmentregister in den Adressraum des Prozessors eingeblendet werden. Das kostet zwar Prozessorzyklen für die Segmentumschaltung und damit Rechenzeit, die geringere Anzahl von Adressleitungen und die einfachere Logik auf dem Chip sparen aber letztlich Chipfläche und machen den Chip billiger als einen Prozessor, der über mehr Adressleitungen und Logik größerer Wortbreite für die direkte Adressierung des gesamten Speichers verfügt.

Die Taktfrequenzen liegen oft bei wenigen zehn bis zu mehreren hundert MHz. Auch hier gilt: Je weniger Stromverbrauch und Abwärme, je geringer die Chipfläche und der Preis, desto besser. Und allgemein gilt bei hohen Stückzahlen: So einfache Hardware wie möglich und nur so viel Ressourcen, wie die jeweilige Anwendung unbedingt braucht (siehe Abb. 1–2).

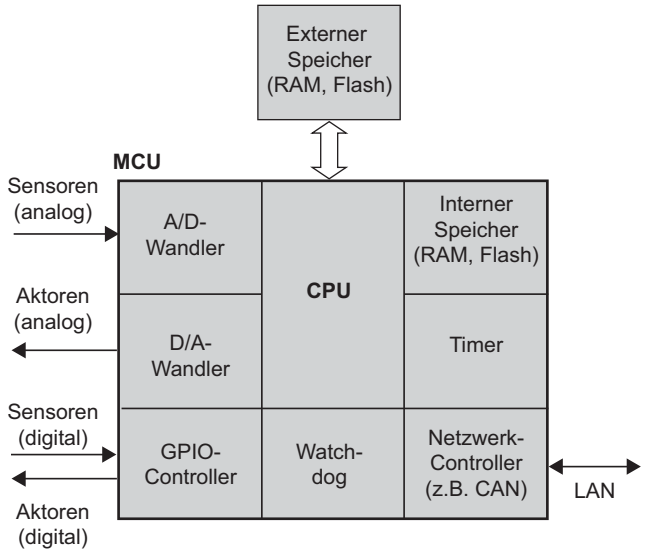


Abb. 1–2
Mikrocontroller – CPU für Embedded-Systeme

Vernetzte Rechner

Auch im Embedded-Bereich können Anwendungen über mehrere Rechnerknoten verteilt sein. Dabei ist jeder Rechnerknoten ein eigenständiges Embedded-System, wobei einige Embedded-Systeme für andere Systeme Dienstleistungen erbringen. Steuert z.B. ein Embedded-System die Motoren der Kühlerventilatoren, so kann es von einem anderen Rechnerknoten, der für die Energieverteilung zuständig ist, Aufträge annehmen und die Motordrehzahl ändern.

Abgrenzung zwischen PC und Embedded-Systemen

Embedded-Systeme steuern in der Regel zwar keinen Monitor an wie PCs, aber manche Embedded-Systeme verfügen über eine Ansteuereinheit für Anzeigeelemente. Diese Anzeigen reichen von ein paar Leuchtdioden (LEDs) bis zu großen LCD-Einheiten, die neben der klassischen Punktmatrix auch ansteuerbare Felder mit anwendungsspezifischen Symbolen enthalten können. Eingesetzt werden sie beispielsweise zur Anzeige von Mess- und Regelwerten, Programmwahl, Systemstatus und Fehlermeldungen.

Uneinigkeit herrscht darüber, ob Anzeigesysteme, die oft mehrere Grafikprozessoren und/oder Multicorechips enthalten, auch zu den Embedded-Systemen zählen und wo man die Grenze zieht. Meines Erachtens gibt es keine klare Abgrenzung.

Smartphones, wie etwa das iPhone von Apple, sind einerseits Mehrzweck-Computer, andererseits aber auch Embedded-Systeme. Systemprogrammierer müssen mit begrenzten Ressourcen auskommen, wenn sie Anwendungen (sog. Apps) für Smartphones als Mehrzweck-Computer entwickeln. Beim Telefonieren mit Smartphones steuert Embedded-Software den Verbindungsaufbau und die digitale Sprachverarbeitung.

Embedded-Systeme gibt es in sehr unterschiedlichen Ausprägungen, und einige sind einem PC ähnlicher als einer Waschmaschinensteuerung. Man kann aber Embedded-Systeme dadurch charakterisieren, dass sie immer eine eng begrenzte Anzahl von Aufgaben erfüllen – und sich nicht für beliebige Aufgaben umprogrammieren lassen.

In diesem Buch konzentriere ich mich auf Embedded-Systeme, deren primäre Aufgabe darin besteht, wenige klar definierte Aufgaben zu erfüllen, und die in Geräte integriert werden.

1.2 Zeitverhalten

Anwendungsprogramme, die Eingaben eines Benutzers verarbeiten, wie eine Tabellenkalkulation auf einem PC, sind in der Regel datengetrieben, d.h., ohne Eingaben werden sie nicht aktiv und liefern keine Ausgabe (siehe Abb. 1–3).

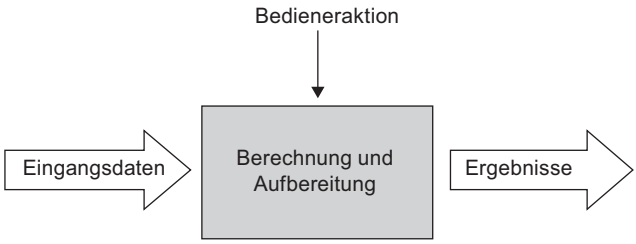


Abb. 1–3
Zeitverhalten eines PC-Anwendungsprogramms

Embedded-Systeme sind programmgetrieben, nicht datengetrieben

Der signifikante Unterschied von Anwendungen auf Embedded-Systemen gegenüber Anwendungsprogrammen auf PCs besteht darin, dass erstere in der Regel programmgetrieben sind, also automatisch, ohne Bedieneraktion aktiv werden und nicht durch Eingabe von Daten. Die Aktivierung der Funktionen von Embedded-Systemen erfolgt meist zyklisch und unabhängig davon, ob sich die von den Datenquellen oder Sensoren eingelesenen Daten seit der letzten Aktivität geändert haben. Die Ausgangsdaten werden von der programmierten Funktion aus den Eingangsdaten neu berechnet, unabhängig davon, ob sich diese Eingangsdaten seit der letzten Berechnung verändert haben oder nicht. Der Datenfluss durch Embedded-Systeme ist im Wesentlichen konstant, kennt aber immer eine feste Obergrenze des Datenvolumens, welches pro Programmzyklus verarbeitet wird (siehe Abb. 1–4).

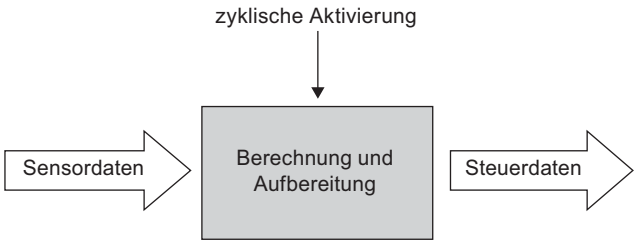


Abb. 1–4
Zeitverhalten der Applikation eines Embedded-Systems

Aus diesem Unterschied resultiert eine andere Softwarearchitektur des Embedded-Systems. Stehen bei PC-Anwendungen neben der eigentlichen Funktion, beispielsweise Seitenlayout-Berechnungen, die Benutzerschnittstelle und grafische Repräsentation der Daten (das GUI) und der Datenfluss im Mittelpunkt, so ist es in Embedded-Systemen die Ablaufsteuerung, welche die Aktivitäten des Systems in der zeitlich richtigen Reihenfolge startet und überwacht. Die Steuerung der Funktionen von Embedded-Systemen folgt meist einem festen zeitlichen Rahmen.

Echtzeitsysteme

Müssen von Embedded-Systemen fest vorgegebene Zeitbedingungen unbedingt eingehalten werden, spricht man auch von Echtzeitsystemen (engl. *real-time systems*).

Bei PC-Anwendungen hängt die Antwortzeit wesentlich von der zu verarbeitenden Datenmenge und der Anzahl der gerade aktivierten Programme ab.

Bei Echtzeitsystemen ist diese vorhersagbar. Einerseits sind die Anzahl der Programme und der zeitliche Ablauf in einem Echtzeitsystem festgelegt, andererseits wird diese Art von Embedded-Systemen auch so ausgelegt, dass die Antwortzeit unabhängig von der Datenmenge ist. Der Zeit, bis zu der ein Ereignis unbedingt bearbeitet worden sein muss (WCET = worst case execution time), kommt in Echtzeitsystemen eine zentrale Bedeutung zu. Eine Überschreitung von bestimmten Zeitvorgaben ist in Echtzeitsystemen nicht erlaubt und würde bestenfalls zu unerwünschten Ergebnissen, schlechtestenfalls zur Katastrophe führen.

Aus dem Echtzeitaspekt ergeben sich vielfältige Anforderungen sowohl an die Vorgehensweise beim Entwickeln der Software, die Entwicklungsumgebung als auch die zu entwickelnde Software selbst.

Beispiel: Apollo 11 Guidance Computer

Der Bordcomputer der Mondlandekapsel von Apollo 11 wurde beim Landeanflug durch Einschalten eines (eigentlich überflüssigen) Rendezvous-Radars mit Daten überfrachtet. Der Datenerfassung wurden von der Hardware zu viele Ressourcen zugeteilt. In der Folge konnte das Echtzeitbetriebssystem Tasks nicht zeitgerecht ausführen, da es die dafür benötigten Ressourcen nicht zuteilen konnte. Die Fehlermeldungen wurden vom Bodenpersonal korrekt interpretiert, und durch entsprechende Anweisungen an die Crew war eine Landung dann doch noch möglich. Der Fehler hätte ohne menschliches Eingreifen höchstwahrscheinlich zum Abbruch der Mission geführt [APO09].

1.3 Betriebssysteme

PCs werden im Allgemeinen mit Windows, MacOS oder einer der vielen Linux-Varianten bestückt. Workstations und Server verwenden häufig eine Unix-Variante oder Windows.

Für Embedded-Systeme sind eine Vielzahl von Betriebssystemen verfügbar. Der Markt für diese Betriebssysteme ist sehr stark zersplittert, und viele Betriebssysteme sind auf ein spezielles Marktsegment fokussiert. Das Angebot reicht von einfachen Betriebssystemkernen, die im Wesentlichen Funktionen für Task-Management bereitstellen, bis zu Komplettsystemen, die einen ähnlichen Funktionsumfang wie ein Unix-System bieten.

Die Betriebssysteme, die in Embedded-Systemen eingesetzt werden, teilen sich in zwei Gruppen auf:

- Betriebssysteme, die keine Echtzeitbedingungen erfüllen
- Echtzeitbetriebssysteme (Real-time Operating Systems, RTOS)

Betriebssysteme ohne Echtzeitverhalten, Windows Embedded, Embedded Linux

In Anwendungsbereichen, in denen ein vorhersagbares Zeitverhalten nicht zwingend erforderlich ist, wird häufig *Windows Embedded* oder *Embedded Linux* eingesetzt, sofern diese Betriebssysteme für die ausgewählte Hardware verfügbar sind oder mit akzeptablem Aufwand portiert werden können.

Die Firma Microsoft bietet seit einigen Jahren ein Betriebssystem für Embedded-Systeme an, das über eine Windows-kompatible Programmierschnittstelle verfügt (*Windows Embedded*). Dieses wird primär in Handhelds, Palmtops und Smartphones eingesetzt. Für harte Echtzeitanwendungen, deren Aktivitäten im Millisekundenbereich oder darunter gesteuert werden, ist es nicht geeignet.

Einige Linux-Derivate für Embedded-Systeme verfügen auch über Echtzeiteigenschaften. Linux benötigt mindestens einen 32-Bit-Prozessor und deckt damit nicht alle Anwendungsbereiche ab.

Auf andere Betriebssysteme weicht man meist aus, wenn Speicher, Rechenleistung, Lizenzgebühren, Lizenzbedingungen oder Wartungsfragen eine Rolle spielen.

Echtzeitbetriebssysteme

Echtzeitbetriebssysteme hingegen werden in Anwendungsbereichen eingesetzt, die ein vorhersagbares Zeitverhalten des Embedded-Systems zwingend voraussetzen, in denen also die Antwort des Embedded-Systems auf ein Ereignis innerhalb einer exakt definierten Zeitspanne erfolgen muss. Diese Echtzeitbetriebssysteme gibt es in vielen Variationen für nahezu jeden Prozessortyp und viele verschiedene Systemarchitekturen. Die Bandbreite reicht von einfachen Betriebssystemkernen für 8-Bit-Prozessoren bis zu 32-Bit-Echtzeitbetriebssystemen mit POSIX-kompatibler Programmierschnittstelle und Unterstützung für Virtualisierung.

Kombination Linux und RTOS

Mehrkernprozessoren ersetzen in einigen zukünftigen Embedded-Systemen Mehrprozessor-Architekturen. Auf einem Prozessorkern läuft etwa Linux oder ein ähnliches Betriebssystem, das über TCP/IP, UDP und andere Netzwerkprotokolle mit dem Internet verbunden ist, und auf den anderen Prozessoren ein Real-Time-Operating System (RTOS). Die Lösung auf Basis von Mehrkernprozessoren ist nicht nur günstiger im Design als solche mit einzelnen Prozessorchips, sie verbraucht auch weniger Energie. Der Einsatz von Linux auf einem Core ermöglicht Echtzeitsystemen auf den anderen Cores den Zugriff auf das Internet.

Es gibt auch Betriebssysteme, die einerseits Echtzeitbetriebssysteme sind, andererseits aber auch eine POSIX-kompatible Programmierschnittstelle anbieten, wie LynxOS [LYN11] und QNX [QNX11].

1.4 Entwicklungsumgebungen

Die Entwicklungsumgebungen für PC- und serverbasierte Anwendungssoftware einerseits (Abschnitt 1.4.1) und Software für Embedded-Systeme andererseits (Abschnitt 1.4.2) unterscheiden sich teilweise grundlegend.

1.4.1 Entwicklungsumgebungen für PC- und Serveranwendungen

PC-Software wird fast ausschließlich auf PCs entwickelt. Die Werkzeuge, die Sie als Entwickler benutzen, laufen alle auf dem PC, angefangen von Entwurfswerkzeugen zur modellbasierten Entwicklung beispielsweise mit der *Unified Modeling Language* (UML), über Editoren, Compiler, Debugger und Analysewerkzeuge bis hin zu Testhilfsmitteln. Auch die Tests selbst laufen in der Zielumgebung ab. Dies gilt auch für Client-Server-Anwendungen. Auch von den meisten Server-

anwendungen gibt es Versionen, die auf dem PC lauffähig sein, allen voran Datenbankprogramme. Die Softwareentwicklung auf dem Zielsystem hat viele Vorteile:

- Es muss in der Regel keine zusätzliche Hardware angeschafft werden.
- Der Bedarf der zu entwickelnden Anwendung an Rechenzeit und Speicherplatz kann einfach bestimmt werden.
- Die Untersuchung von Softwarefehlern erfolgt direkt durch Ausführung der Anwendung auf dem Entwicklungssystem unter Kontrolle eines Debuggers. Dabei können leicht Einsatzbedingungen erzeugt werden, die der realen Anwendungsumgebung entsprechen, da Hardware, Betriebssystemsoftware, Prozessorarchitektur und Peripherie identisch oder vergleichbar sind.
- Ein Test der Software für mehrere Systemkonfigurationen (z.B. unterschiedliche Betriebssystemversionen, verschiedene Browser als Benutzerschnittstelle) kann direkt auf dem Entwicklungssystem erfolgen.

1.4.2 Software-Entwicklungsumgebungen für Embedded-Systeme

Die Softwareentwicklung für ein Embedded-System gestaltet sich im Vergleich zu einer PC-Anwendung deutlich schwieriger. Entwurfswerkzeuge wie Editoren laufen üblicherweise auf einem PC entweder unter Windows oder Linux. Da der Zielprozessor meist eine andere Architektur hat als die auf der Intel-Architektur basierenden Prozessoren in den PCs, ist für die Übersetzung des Quellcodes ein Cross-Compiler nötig, der den Maschinencode für die andere Prozessorarchitektur erzeugen kann.

Cross-Compiler

Cross-Compiler verfügen gegenüber Standard-PC-Compilern über eine Reihe besonderer Fähigkeiten, wie etwa:

- Unterstützung von Optimierungsvarianten:
  - möglichst laufzeiteffizienter Code
  - möglichst speichereffizienter Code
- gezielte Platzierung von Code und Daten in bestimmten Speicherbereichen
- Bestimmung des statischen Stack-Verbrauchs
- Erweiterung der Programmiersprache durch zusätzliche Schlüsselwörter zur Optimierung
- Optionen zur Festlegung der Interpretation von Sprachelementen

Prozessorsimulatoren

Der von Cross-Compilern und -Linkern erzeugte Maschinencode ist nicht auf dem Prozessor des PC lauffähig. Es gibt Simulatoren für Mikrocontroller, die auf dem PC laufen, den Maschinencode des Mikrocontrollers interpretieren und so das Verhalten des Mikrocontrollers nachbilden.

Ladeprogramm

Wenn Sie den Code aber im Zusammenspiel mit der Hardware und echten Ein- und Ausgangssignalen prüfen wollen, muss der erzeugte Maschinencode auf dem Zielprozessor ausgeführt werden. Dazu wird er mit einem Ladeprogramm in den Speicher des Zielprozessors transferiert. Dies hört sich einfach an, ist aber mit allerlei Tücken behaftet. Das Ladeprogramm benötigt nämlich einen Kommunikationspartner auf dem Zielprozessor, der die Daten entgegennimmt. Dies ist entweder ein Monitorprogramm, das bereits im Speicher des Zielprozessors liegt und vor dem Download aktiviert werden muss, oder es gibt eine Kommunikationshardware, die mit dem Ladeprogramm Daten austauschen kann und teilweise in den Zielprozessor integriert ist.

Der Aufbau einer solchen Entwicklungsumgebung ist in Abbildung 1–5 dargestellt. Die Kommunikationshardware wird üblicherweise über USB an den PC angeschlossen. Die Kommunikationshardware selbst ist über eine spezielle Schnittstelle direkt mit dem Prozessor auf der Zielhardware verbunden. Neben herstellerspezifischen Schnittstellen wie dem Background-Debug-Monitor (BDM) des Star-S12-Prozessors gibt es auch Schnittstellenstandards wie JTAG (joint test action group, ein Synonym für die IEEE-1149.1-Schnittstelle), die beispielsweise auf Prozessoren von Texas Instruments implementiert sind.

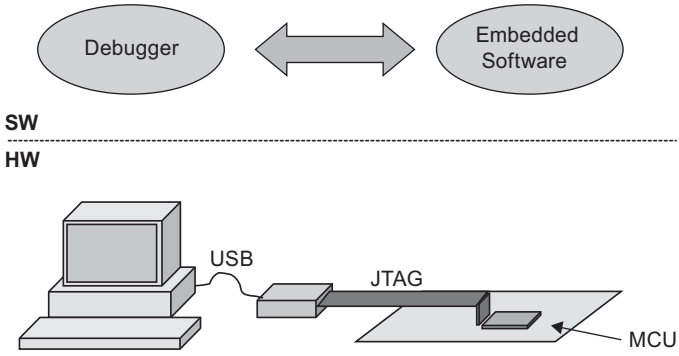


Abb. 1–5
Entwicklungsumgebung für Embedded-Software

Debugger

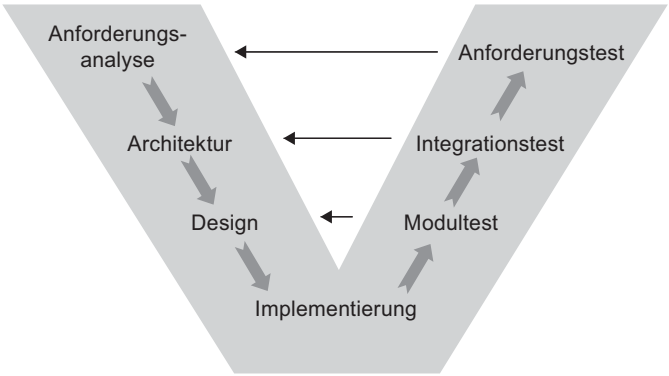
Neben dem Ladeprogramm gibt es auch spezielle Debugger, die mit dem Monitor auf dem Zielprozessor kommunizieren können. Mit einem sogenannten Remote Debugger können Sie Programme auf dem Zielprozessor ähnlich einem Debugger für PC-Software schrittweise ausführen, Speicherzellen lesen, etc.

Der Befehlsumfang solcher Remote Debugger ist gegenüber einer Debug-Umgebung für die Entwicklung von PC-Anwendungsprogrammen eingeschränkt – sie hängt von den Möglichkeiten der Monitorhardware oder -software auf dem Zielprozessor ab. Der Funktionsumfang beschränkt sich oft auf das Lesen und Schreiben von Speicherzellen und das Setzen einiger Haltepunkte (breakpoints). Typische Einschränkungen bestehen auch in der Art und Anzahl der Haltepunkte, die Sie mit einem solchen Debugger setzen können. Auch bedingte Haltepunkte, also solche, bei denen das Programm angehalten wird, wenn z.B. eine bestimmte Variable einen vordefinierten Wert erreicht oder über- bzw. unterschritten hat, sind oft nicht möglich. Bei solchen Einschränkungen ist das Debuggen einer Embedded-Software oft mühsam und langwieriger als das Debuggen eines PC-Programms. Sie sollten diese Einschränkungen bei der Softwareentwicklung berücksichtigen und die Komplexität der Software entsprechend gering halten, sodass das Testen bestimmter Funktionalitäten und das Beobachten bestimmter Abläufe noch möglich ist.

1.5 Entwicklungsprozesse

Wie bei jeder komplexen Aufgabe ist auch bei der Softwareentwicklung eine gewisse Struktur in der Vorgehensweise empfehlenswert. Unter den verschiedenen Vorgehensmodellen ist das V-Modell bei der Entwicklung eingebetteter Systeme am weitesten verbreitet und in vielen Branchen als Standard etabliert (siehe Abb. 1–6).

Abb. 1–6
V-Modell



Die Kapitel 2 bis 6 sind nach der Prozess-Struktur des V-Modells gegliedert. Zu jedem Prozess-Schritt im V-Modell wird dort auf spezifische Themen eingegangen, die in der Softwareentwicklung für Embedded-Systeme besonders wichtig sind.

Das V-Modell (vgl. Abb. 1–6) ordnet die Entwicklungsaktivitäten in einen sequenziellen Ablauf und stellt die zur gleichen Abstraktionsebene gehörenden Entwicklungs- und Testaktivitäten einander gegenüber. Dabei ist der höchste Detaillierungsgrad unten an der Basis des V und der höchste Abstraktionsgrad oben rechts und links an den beiden Spitzen des V zu finden.

Linker Ast:

Ausgehend von der Anforderungsanalyse wird die Softwarearchitektur entwickelt und daraus das Design abgeleitet, das die Software in einzelne Module untergliedert. Die Implementierung des Designs liefert dann den Quellcode.

Rechter Ast:

Nach Test der einzelnen Softwaremodule werden diese in der Integrationsphase Zug um Zug integriert, und nach jedem Integrationsschritt wird das Zwischenprodukt getestet. Die am Ende der Integrationsphase entstandene Gesamtsoftware wird dann noch einem Test unterzogen, der die Umsetzung der Anforderungen verifiziert.

Im Folgenden werden die verschiedenen Prozess-Schritte aus Abbildung 1–6 sowie die Unterstützungsprozesse kurz beschrieben.

Anforderungsanalyse

Ausgangspunkt der Softwareentwicklung ist die Analyse der Anforderungen an die Software. Beachten Sie hier, dass diesem Schritt die Analyse der Anforderungen an das Embedded-System (integrierte Hardware und Software) und die Aufteilung dieser Anforderungen in die Gruppen *System*, *Software*, *Hardware* etc. vorausgeht. Gibt es in einem Projekt keinen Systemingenieur, der diese Aufteilung vornimmt, etwa weil das Projekt zu klein ist, sollten Sie diese Rolle zusammen mit dem Hardwareentwickler wahrnehmen oder zumindest klar abgrenzen, welche Funktion die Software in diesem Projekt erfüllen soll, und dokumentieren, welche Anforderungen der Software zufallen.

Die Anforderungen gliedern sich in zwei Gruppen:

- funktionale Anforderungen
- nichtfunktionale Anforderungen

Funktionale Anforderungen

Funktionale Anforderungen beschreiben die Reaktion der Software auf Ereignisse, ihre Ein- und Ausgangsdaten und die Berechnungsvor-

schrift, nach der die Ausgangsdaten aus den Eingangsdaten erzeugt werden.

Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen sind alle Anforderungen, die, wie der Name bereits nahe legt, nicht die Funktionalität betreffen. Nichtfunktionale Anforderungen betreffen sowohl technische, das Produkt Software betreffende Aspekte (z.B. die Auswahl des Betriebssystems) als auch nichttechnische, den Herstellungsprozess betreffende Aspekte (z.B. den Testumfang).

Zu den nichtfunktionalen Anforderungen gehören:

- die Programmiersprache
- das Betriebssystem
- die Qualität
- der Testumfang
- die Testtiefe
- der Speicherplatzbedarf
- die Verwendung des Speichers (z.B. Grenzen für dynamischen Speicherbereich)
- die Rechenzeit
- das Zeitverhalten bestimmter Funktionen
- die funktionale Sicherheit
- die Dokumentation

Beispiele für Qualitätsanforderungen finden Sie in Abschnitt 1.8 und speziell für Testumfang und Testtiefe in Abschnitt 1.8.3.

Bereits bei der Analyse der Softwareanforderungen sollten Sie sich Gedanken machen, wie die Umsetzung der jeweiligen Anforderung nachgeprüft werden kann.

Architektur

Nach Analyse der Anforderungen an die Software legen Sie die Architektur der Software fest. In diesem Prozess-Schritt teilen Sie die Aufgaben, die das System erfüllen soll, in Module auf, versehen die Funktionen der Softwaremodule mit Prioritäten und teilen sie in voneinander getrennt ausführbare Programme (sog. *tasks*) auf. Bei der Zuordnung berücksichtigen Sie die Hardware. Die Hardware kann mehrere Prozessoren unterschiedlichen Typs enthalten, sodass Sie bei der Zuordnung Vor- und Nachteile (Rechenleistung vs. Programmierbarkeit) abwägen müssen.

Für jede Task spezifizieren Sie das Zeitverhalten. Dabei legen Sie z.B. fest, ob eine Task einmalig nach einem Ereignis oder zyklisch in bestimmten Zeitabständen aktiviert wird. Mit diesen Informationen

können Sie den *Scheduler* des Betriebssystems so konfigurieren, dass jede Task gemäß ihres spezifizierten Zeitverhaltens aktiviert wird und passend Rechenzeit zugeteilt bekommt.

Neben dem Zeitverhalten müssen Sie auch die Kommunikationsarchitektur und damit die Schnittstellen der Tasks untereinander festlegen. Je nach Systemanforderungen kommen verschiedene Kommunikationsformen in Betracht:

- nachrichtenbasierte Kommunikation (*message passing*)
- Datenaustausch über gemeinsamen Speicher (*globale Daten* oder *shared memory*)

Soweit bereits bekannt, sollten Sie hier die Schnittstellen zur Hardware festlegen. Für jede Schnittstelle wird später eine hardwarenahe Software, der sog. *Treiber*, benötigt.

Je nach Anwendungsfall werden auch bereits Detailanforderungen an die Hardware erkennbar. Ist die Hardware vorgegeben, müssen Sie diese Anforderungen mit der Hardware-/Software-Interfacebeschreibung abgleichen. Wird die Hardware extra für das Embedded-System entwickelt, sollten Sie Ihre Anforderungen separat beschreiben und mit dem Hardwareentwickler besprechen. Hardwareentwickler haben andere Prioritäten als Softwareentwickler. Sie wollen Bauteile einsparen und Hardware entwickeln, die sich einfach testen und produzieren lässt. Einfache Programmierbarkeit ist nicht ihr primärer Fokus. Eine intensive Diskussion mit dem Hardwareentwickler führt möglicherweise zu Hardware, die einfacher zu programmieren ist. Auch haben Sie nach der Diskussion die Hardware besser verstanden und können sie leichter programmieren.

Planen Sie, Embedded-Computer-Baugruppen (Katalogware) *zu kaufen*, auch *commercial off-the-shelf* bzw. COTS-Hardware genannt, sollten Sie an dieser Stelle unbedingt prüfen, inwieweit diese Hardware den Anforderungen der Software genügt.

Design

Auf Grundlage der Architekturbeschreibung legen Sie dann die Grobstruktur der Module fest und ordnen die Module den Tasks zu.

In der Designphase definieren Sie auch die globalen Datenstrukturen und die Schnittstellen der Module. Die Aufgabe jedes Moduls sollten Sie in einer Kurzbeschreibung festhalten. In der Modulbeschreibung wird erklärt, wie die Ausgangdaten aus den Eingangsdaten bestimmt werden.