Die richtige Wahl

Die Auswahl von Entwicklungswerkzeugen für Embedded Systeme

Andreas Pfeiffer

Die richtige Auswahl von Werkzeugen für die Entwicklung von Software für Embedded Systeme ist entscheidend für den Ablauf und die Qualität des Entwicklungsprozesses. Darüber hinaus müssen die gewählten Werkzeuge auf bestimmte Entwicklungsstufen optimal abgestimmt sein und können nicht immer universell eingesetzt werden. Dieser Artikel beleuchtet den Entwicklungsprozeß und versucht für die einzelnen Entwicklungsschritte herauszustreichen, welche Werkzeuge genau passen, wie man passende Werkzeuge findet, welche Parameter und Optionen berücksichtigt werden müssen und wie es um die Integration der Werkzeuge steht.

Stufenweiser Entwicklungsprozeß

Der Entwicklungsprozeß kann in einzelne Stufen eingeteilt werden. Man kann die einzelnen Werkzeuge als zusammenhängende Kette betrachten, wobei jedes Glied dieser Kette mit dem nächsten eng verbunden ist. Bild 1 zeigt diese Kette vom Hochsprachencompiler über Assembler, Linker bis zu den verschiedenen Debugger-Varianten.

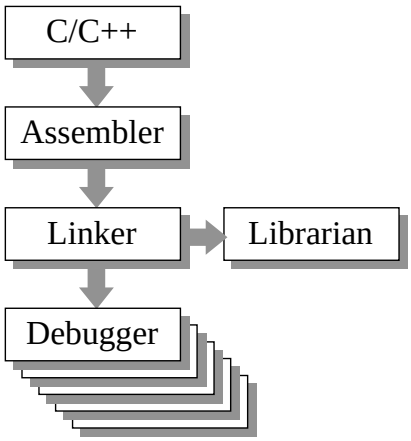

Bild 1: Entwicklungswerkzeuge

In dieser Kette gibt es zwei große Teile:

- Die **Build-Tools**: Werkzeuge zum Übersetzen von Quellkode in ausführbaren Code
- Die **Debug-Tools**: Werkzeuge zum Testen und zur Beseitigung von Fehlern

Die Build-Tools sollen unabhängig von der Entwicklungsumgebung sein und sämtliche Optionen und Parameter müssen zu gleichen Endergebnissen führen. Unterschiedliche Debugger-Varianten dürfen keinen Einfluß auf den Ablauf und das Erstellen der Applikation haben. Es ist daher nicht nötig je nach Umgebung unterschiedliche Bibliotheken ein zu müssen. Notwendige Änderungen am Quellkode um diesen für die verschiedenen Stufen des Entwicklungsprozesses anzupassen sind immer fehleranfällig und daher nicht erwünscht.

Wichtige Eigenschaften des Compilers

Die klassische Embedded-Programmierung erfolgt in C. Mit steigender Popularität von objektorientierter Programmierung kommt immer mehr die Sprache C++ zum Einsatz. Compiler für Cross-Entwicklungen können nicht nach den gleichen Kriterien ausgewählt werden wie Native-Compiler.

- Programmiersprache nach Standard: Um Portabilität zu gewährleisten muß ein Compiler nach ANSI-C-Standard arbeiten. Für ältere Programmteile kann auch Kompatibilität zu K&R-C gewünscht sein. Bei C++ stellt AT&T V2.1 einen weit verbreiteten Standard dar. Neue C++ Compiler werden konform zum Annotated Reference Manual (Stroustrupp) als Vorstufe zu ANSI-C++ entworfen.
- Bibliotheken: Ein Schwachpunkt der ANSI-C-Laufzeitbibliothek ist, daß viele der enthaltenen Funktionen nur bedingt echtzeitfähig sind. Dies kann bei der Entwicklung von Embedded Systemen zu erheblichen Einschränkungen führen. Nachdem an einzelne Bibliotheksfunktionen sehr hardwarenahe Anforderungen gestellt werden (z.B. printf), ist auch der Zugriff auf den Quellkode der Bibliotheksroutinen wünschenswert.
- Objektorientierte Programmierung ermöglicht einmal erstellten Code wiederzuverwenden. Ein guter C++ Compiler sollte durch eine leistungsfähige Standard-Klassenbibliothek ergänzt werden.
- Unterstützung von Prozessorfamilien: Man wird einen Cross-Compiler auswählen, der genau den gewählten Zielprozessor unterstützt. Zukünftige Erweiterungen basieren oft auf Prozessoren, die der gleichen Prozessorfamilie angehören. Ein Compiler für eine Prozessorfamilie stellt eine sichere Investition auch für die Zukunft dar. Es sollten jedoch nicht nur die Gemeinsamkeiten aller Prozessoren unterstützt werden sondern auch die ganz speziellen Eigenschaften der unterschiedlichen Prozessor-Derivate. Der erzeugte Code wird optimal an den gewählten CPU-Typ angepaßt.
- Support: Die Qualität eines Produktes soll auch an der Unterstützung des Herstellers nach dem Kauf gemessen werden.

Erweiterungen für Embedded Systeme

Ein Cross-Compiler ist wesentlich komplexer als ein native Compiler. Bei der Entwicklung des Compilers können sehr wenige Aussagen darüber getroffen werden, wie das endgültige Zielsystem aufgebaut sein wird. Der Hersteller muß zur Anpassung flexible Einstellmöglichkeiten implementieren.

Die Speicherkonfiguration unterscheidet sich bei Embedded-Hardware wesentlich von einer klassischen PC-Umgebung. Einfachste Systeme bestehen zumindest aus ROM für Programm-Code und RAM für Variablen. Ein Embedded-Compiler muß in jedem Fall Rom-fähigen Code generieren können, der streng vom Datenbereich getrennt ist. Anforderungen dieser Art liegen jedoch noch meist viel höher (ROM, RAM, FLASH, EEPROM, NOVRAM,...siehe Bild 2).

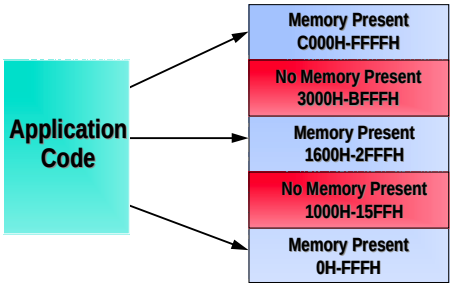

Bild 2: Aufteilung einer Applikation im Speicher

Ein wichtiger Punkt betrifft ein Sprachkonstrukt von C. In C vergibt der Programmierer einen Initialisierungswert an eine statische Variable. Dadurch vermeidet man komplexe Initialisierungsroutinen für jene Variablen, die bereits zur Übersetzungszeit auf bekannten Adressen liegen. In der Programmdatei werden diese Variablen auf ihren Startwert gesetzt und samt Programm in den Speicher des Systems geladen. In einem System, indem das Programm bereits fix im ROM steht, kann dieser Mechanismus nicht funktionieren. Man könnte diese Situation auf drei Arten umgehen:

- Statische Variablen werden nicht initialisiert
- Statische Variablen werden als Konstanten im ROM abgelegt.
- Beim Programmstart werden die Initialisierungswerte aus dem ROM ins RAM kopiert.

In C und C++ ist der direkte Zugriff auf den Speicher erlaubt. Darum sind diese Sprachen für hardwarenahe Entwicklungen sehr populär. Hier darf ein Compiler zu keinerlei Einschränkungen führen.

Speicher ist oft nur sehr begrenzt verfügbar. Der Cross-Compiler muß in diesem Bereich effizient beeinflußt werden können. Mit dem Schlüsselwort *packed* kann man z.B. Felder und Strukturen platzsparender definieren. Diese Einsparungen führen andererseits oft zu längeren Zugriffszeiten.

Das Schlüsselwort `volatile` ist eine Erweiterung zum ANSI-Standard. Eine derart definierte Variable kann sich während des Programmablaufes beliebig ändern. Der Compiler darf den Zugriff auf diese Variable daher nicht optimieren. Volatile Variablen können z.B. für den Zugriff auf I/O-Register definiert werden. Die Sprache C++ beinhaltet dieses Schlüsselwort nicht. Manche Compiler bieten diese Spracherweiterung trotzdem an und sind daher für Embedded-Entwicklungen besser geeignet.

Trotz Hochsprache sollte es auch möglich sein Programmteile in Assemblersprache zu implementieren. Assemblercode verwendet man meist um die letzten Nanosekunden aus einem Prozessor zu holen. Er bietet aber auch einfachen Zugriff auf Prozessor-Resourcen, die nicht direkt durch C-Konstrukte unterstützt werden (z.B.: Interrupt enable/disable). Aus Effizienz- und Portabilitätsgründen sollte natürlich soweit wie möglich auf den Einsatz von Assembler-Code verzichtet werden. Ein Compiler sollte aber so flexibel sein, auch diese Möglichkeit der Programmierung zu unterstützen und erlauben, beliebige Assemblerzeilen in das Hochsprachenprogramm einzufügen.

Echtzeitanforderungen

Viele Embedded Anwendungen müssen echtzeitfähig sein. Auch die Build-Tools müssen dies unterstützen. In Echtzeitsystemen werden zumeist Interrupt Service Routinen verwendet, die in C bzw. C++ z.B. über ein Schlüsselwort (*interrupt*) zu markieren sind. Die Funktion wird dann mit Codeteilen zum Sichern von relevanten Registern versehen und ein spezieller Rückkehrbefehl wird am Ende angefügt.

Codeteile in Echtzeitsystemen können mehrfach in verschiedenen Programmteilen verwendet werden. Dieser Code muß wiedereintrittsfähig (reentrant) geschrieben sein. C und C++ ermöglichen es derartigen Code zu generieren, da Daten- und Programmteile zusammengehörend definiert werden können. Manche Standard-Bibliotheksfunktionen sind jedoch nicht wiedereintrittsfähig definiert und müssen daher mit besonderer Vorsicht verwendet werden.

Optimierungstechnologie

Moderne Compiler bauen auf Optimierungstechnologien zum Erzeugen von qualitativ hochwertigem Code auf und können durchaus mit handoptimierten Assemblerprogrammen konkurrieren was Größe und Effizienz betrifft. Optimierungen können sich dabei lokal auf eine Funktion oder global auf ein Modul beziehen.

Viele Optimierungen können prozessorunabhängig eingesetzt werden. Interessanter sind jedoch jene Optimierungsarten, die Vorteile aus den ganz speziellen Eigenschaften eines Zielprozessors ziehen. Dies betrifft z.B. die Reihung von Instruktionen, Inline-Funtionen und die switch-Anweisung.

Die Reihenfolge, in der ein Prozessor Instruktionen abarbeitet kann großen Einfluß auf Ausführungszeit und Auslastung des Prozessors haben. Grundlegend ist dies bei RISC-Architekturen, es gilt aber auch bei CISC-Prozessoren.

Unter In-Lining versteht man das direkte Einfügen von kleineren Funktionsblöcken ohne diese über Unterprogrammaufrufe zu aktivieren. Die Ausführungszeiten werden dadurch verkürzt. Komfortable Compiler wählen Funktionen automatisch für In-Lining aus.

Bei der `switch`-Anweisung von C hängt es sehr von den Werten und der Abfolge der Verzweigungen ab, wie optimaler Code zu generieren ist. Explizite Ausführung von Testabfragen, Implementierung von Lookup-Tables oder indizierte Sprungtabellen können dabei zielführend sein. Auch Dummy-Werte für jene Verzweigungen die nicht explizit angegeben wurden können zur effizientesten Lösung führen. Der Compiler kann bei Änderungen den passenderen Mechanismus verwenden und sorgt so für höhere Effizienz als die direkte Programmierung in Assembler.

Die Möglichkeit der Feinabstimmung des Compilers auf die Zielplattform ist eine Grundvoraussetzung für brauchbare Entwicklungsergebnisse. Vom Benutzer muß auf jeden Fall die Zielrichtung (Optimale Speichereffizienz bzw. optimale Laufzeit) beeinflußbar sein.

Eigenschaften des Debuggers

Nachdem ein Programm fehlerfrei durch den Compiler übersetzt wurde, wird im nächsten Schritt überprüft, ob sich die Applikation auch wie geplant verhält. Dies gilt generell und nicht nur für die Entwicklung von Embedded Systemen. Hier gibt es jedoch besondere externe Einflüsse die zu speziellen Anforderungen bei der Auswahl der Werkzeuge führen.

Die wichtigste Forderung ist, daß ein Debugger voll optimierten Code verarbeiten können muß. Obwohl das naheliegend ist, sind noch nicht alle Debugger dafür geeignet. Es muß oft noch gewählt werden ob man optimierten Code oder vollüberprüften Code ausliefern will. Ein Mikroprozessor wird auf Grund seiner Leistungsfähigkeit ausgewählt und der Entwickler verläßt sich auf den Compiler, daß dieser die gewünschte Leistung auch gewährleistet. es ist jedoch nicht zu akzeptieren, wenn der Debugger in diesem Punkt zu Einschränkungen führt.

In der Praxis kann das Überprüfen von optimiertem Code zu einer ziemlichen Herausforderung für den Entwickler werden. Die Ergebnisse von so manchen Optimierungen (z.B.: Verschieben von Codeteilen oder Register Coloring) können es extrem erschweren dem Programmfluß zu folgen. Zur allgemeinen Überprüfung wird man deshalb noch nicht alle Optimierungsoptionen voll ausnützen. Für den endgültigen Test muß der Debugger jedoch vollständig optimierten Code verarbeiten können.

Der Debugger muß auf Hochsprachenebene arbeiten. Die Ausführung des Programms muß Befehl für Befehl und nicht nur zeilenweise erfolgen. Der Zugriff auf Datenbereiche muß angepaßt und bequem möglich sein (Berechnung von Ausdrücken, Dekodieren von Strukturen, Verfolgen von Listen). Andererseits muß auch wahlweise ein maschinennaher Zugriff auf Code und Daten möglich sein.

Auch bei Einsatz von C++ muß der Debugger auf Hochsprachenebene arbeiten können. Der vom Präprozessor erzeugte C-Zwischencode muß im Hintergrund verdeckt bleiben. Funktionsnamen müssen immer in der C++ Form erscheinen (unmangled).

Auch ein Debugger muß flexibel an die Zielumgebung anzupassen sein. Bei einem Debugger kann die Funktionalität über eine Makrosprache erweitert werden um I/O-Einheiten zu modellieren, Testläufe zu automatisieren oder Codebereiche zu patchen.

Die Benutzerschnittstelle des Debuggers beeinflußt in hohem Maße die Produktivität des Entwicklungsprozesses. Die Oberfläche muß leistungsfähig, durchgängig in der Bedienung und leicht erlernbar sein (siehe Bild 3). Wenn Debugger-Varianten in unterschiedlichen Umgebungen eingesetzt werden sollen, muß der Umstieg ohne Lernaufwand erreichbar sein. Eine Debugger-Familie für verschiedene Anforderungen bietet hier die beste Lösung.

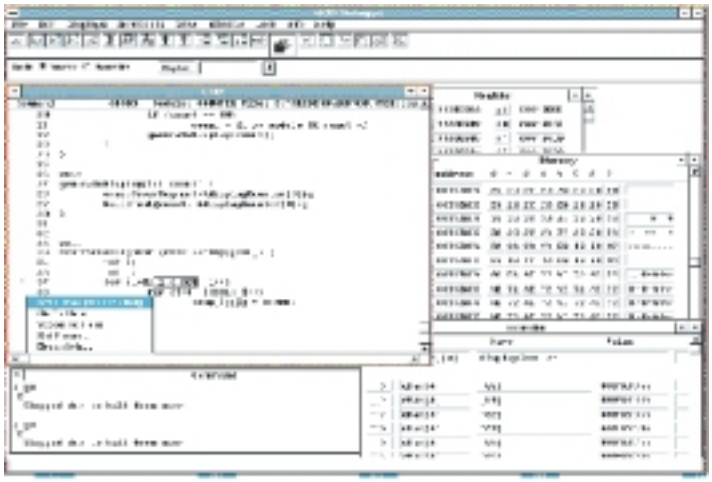

Bild 3: Oberfläche des XRAY-Debuggers von Microtec Research

Entwicklungsphasen

Den Entwicklungszyklus für ein Embedded System kann man in fünf Phasen einteilen (siehe Bild 4). Die Randbedingungen für die Entwicklung ändern sich von Phase zu Phase.

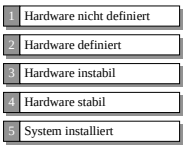

Bild 4: Entwicklungsphasen

In Phase 1 können auch ohne definierte Hardware Softwarealgorithmen und neue Ideen ausgearbeitet werden.

In Phase 2 ist die Hardwarekonfiguration bereits bekannt. Die Softwareentwickler können bereits einen großen Teil der Anwendung im Detail erstellen.

Die Integration von Hardware und Software kann in Phase 3 beginnen, obwohl die Hardware noch nicht 100%ig stabil läuft.

In Phase 4 gibt es stabil lauffähige Hardwareeinheiten zur endgültigen Integration und für Abschlußtests.

Manche Entwicklungsprojekte können bereits vor Phase 5 abgeschlossen werden. Manchmal muß jedoch in Phase 5 ein Feinabgleich bei Inbetriebnahme im Feld vorgenommen werden. Zu einem späteren Zeitpunkt werden ggf. noch Erweiterungen im Feld vorgenommen.

Für jede einzelne dieser Entwicklungsphasen sollte eine angepaßte Version des Debuggers existieren.

Native Debugger

Ein native Debugger (ausführen des Codes in der Host-Umgebung) für die Embedded Entwicklung erscheint zunächst unpassend. Während Phase 1 des Entwicklungsprozesses kann man aber bereits mittels native Debugger die Funktionalität der Algorithmen und Programmteile überprüfen. Bietet ein native Debugger darüber hinaus noch die gleiche Benutzerschnittstelle, wie später die Cross-Debugger-Versionen, so kann dieser Debugger eine optimale und stabile Trainingsplattform zum Erlernen der Werkzeuge sein.

Debugger mit Simulator

Die Prozessor-Simulation auf Instruktionsebene am Hostsystem ist in fast allen Phasen nützlich. Speziell in Phase 2 beschleunigt ein Simulator den Entwicklungsprozeß erheblich.

Mittels Simulator kann die Applikation bereits sehr detailliert getestet werden. Trotz reduzierter Geschwindigkeit kann der Simulator bereits genaue Angaben über die Laufzeiten geben. Kritische Programmbereiche können sehr früh optimiert werden. Mit dem Simulator werden aber auch Informationen generiert, die mit anderen Debugger-Varianten nicht verfügbar sind. Die Analyse der Laufzeiten erfolgt exakt und ohne Overhead. Informationen über die prozentuelle Ausnutzung verschiedener Programmbereiche können auch nur in einer Simulatorvariante generiert werden. Ein leistungsfähiger Simulator muß des weiteren in der Lage sein, über den Prozessorkern hinaus auch Interrupts und I/O-Einheiten nachzubilden.

Debugger mit Emulator-Anbindung

Incircuit-Emulatoren sind sehr leistungsfähige Werkzeuge um Software auf noch instabiler Hardware zu integrieren. Die Applikation läuft mit voller Geschwindigkeit und es können sehr komplexe Breakpoints gesetzt werden. Die Bedienoberfläche des Emulators sollte zu jener der anderen im Projekt eingesetzten Debugger-Varianten kompatibel sein. Ein wichtiger Faktor ist hier die Unterstützung namhafter Emulatorhersteller.

Debugger und Monitor

Ist die Hardware stabil lauffähig, braucht man nicht mehr unbedingt den Leistungsumfang eines teuren Emulators, der ja nebenbei nicht für jeden Entwickler im Team zur Verfügung steht. Hier führen Monitor-Debugger zum Ziel, wobei die Zielhardware über eine Schnittstelle (Ethernet, seriell,...) mit dem Hostsystem verbunden ist. Am Zielsystem läuft ein kleines (<10kByte) Monitorprogramm zur Kommunikation mit dem Host. Man erhält eine billige aber sehr leistungsfähige Debugger-Lösung bei voller Ablaufgeschwindigkeit.

Der Monitor am Zielsystem muß einfach an neue Hardwaregegebenheiten anpaßbar sein. Für Standard-Boards müssen Anpassungen ab Lager verfügbar sein.

Ein Monitor-Debugger wird meist in Phase 4 eines Projekts eingesetzt. Auch wenn die Applikation ausgeliefert wurde kann das Monitorprogramm am Zielsystem verbleiben und erlaubt einfache Erweiterungen und Anpassungen im Feld z.B. mittels Notebook-Computer.

Debugger für Echtzeitsysteme

Mit steigender Komplexität der Applikationen werden vermehrt Echtzeitbetriebssysteme eingesetzt. Dafür müssen auch im Debugger funktionelle Erweiterungen vorgesehen werden:

- Das Debuggen muß taskabhängig vorgenommen werden können. Ein Breakpoint muß abhängig von einer Task gesetzt werden können. Dies ist dann wichtig, wenn einzelne Programmteile in mehreren Tasks Verwendung finden. Auch Datenbereiche müssen als zu einer Task gehörend identifiziert werden können.
- Informationen über die Betriebssystemumgebung wie z.B. Task Status oder Kommunikationsmechanismen usw. müssen extrahiert und dargestellt werden können.

Wenn ein selbsterstelltes Betriebssystem verwendet wird, muß auch dieses in eine leistungsfähige Debugger-Umgebung einfach zu integrieren sein.

Debugger mit Hardwareunterstützung

Der Preis und die Komplexität von Hardware-Emulatoren steigt mit der Leistungsfähigkeit der heutigen Mikroprozessoren. Daher werden von den Halbleiterherstellern immer mehr Funktionsblöcke auf den Prozessorchips integriert, die den Zugriff eines Debuggers in die Interna des Prozessors erleichtern. Das Spektrum reicht hier von der direkten Unterstützung mittels Breakpoint-Register (Adress- und Datenkomparatoren) bis zu speziellen Betriebsmodi um den Prozessor in einen Debug-Mode zu setzen.

Ein gutes Beispiel für den letzten Punkt ist der sogenannte Background Debug Mode (BDM) von Motorola, welcher in den Prozessoren der Serie 683xx (CPU32) implementiert wurde. Wird der Prozessor in diesen Modus gesetzt, so kann der Debugger über eine eigene Schnittstelle Register und Speicherstellen lesen und schreiben. Die Anbindung an ein Host-System wurde einfach gehalten und bedeutet kaum Mehraufwand für die Systementwicklung. Die Hardwareunterstützung bietet emulatorähnliche Möglichkeiten zu sehr geringen Kosten jedoch ohne zusätzlich notwendigem Monitorprogramm am Zielsystem.

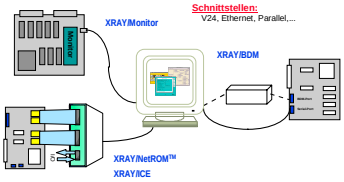

Bild 5: In-Circuit-Varianten des XRAY-Debuggers von Microtec Research

Einflüsse auf die Auswahl

Die Auswahl der Entwicklungswerkzeuge ist natürlich nur einer aus vielen Entscheidungsprozessen beim Entwurf neuer Systeme. Weitere Entscheidungen betreffen den Zielprozessor, den Entwicklungsrechner und das Echtzeitbetriebssystem. Diese Entscheidungen greifen ineinander und sollten nicht isoliert betrachtet werden.

Zielprozessoren

Es gibt viele gute Gründe sich für einen bestimmten Prozessor zu entscheiden. Natürlich sollte nicht außer Acht gelassen werden, wie es um die Unterstützung durch Entwicklungswerkzeuge steht. Selten wird nur ein Hersteller die vollständige Palette zur Entwicklung eines Prozessors anbieten. Sind für einen Prozessor nur sehr eingeschränkt gute Werkzeuge am Markt zu finden, so sollte man diesen Typ eher in Frage stellen.

Entwicklungsrechner

Eigentlich kann man zur Programmentwicklung jeden beliebigen Computer verwenden. Man kann sich auch für jene Werkzeuge entscheiden, die für einen bestimmten Computer portiert wurden. Der Einsatz von standardisierten Systemen bietet jedoch die größte Sicherheit auch in Zukunft mit Werkzeugen und Updates versorgt zu werden. Hersteller, die ihre Werkzeuge auf sehr vielen unterschiedlichen Plattformen anbieten sollten genau bezüglich Support überprüft werden. Oft leidet die Pflege der Portierung auf exotischen Computern unter den Versionen auf populären Plattformen.

Echtzeitbetriebssysteme

Hier wäre ein eigener Artikel angebracht. Der Blick sollte bei der Auswahl durchaus auch auf die Unterstützung durch Entwicklungswerkzeuge gelenkt werden. Ein Betriebssystem sollte in jedem Fall eine flexible offene Architektur bieten. Verschiedene Build-Tools sollen eingesetzt werden können. Passende Debug-Tools müssen natürlich auch zur Verfügung stehen. Oft wird daran gedacht, ein eigenes Betriebssystem zu entwickeln. Man muß jedoch berücksichtigen, daß dann auch für die Anbindung an die Entwicklungswerkzeuge gesorgt werden muß.

Zusammenfassung

Die Auswahl von Software-Entwicklungswerkzeugen für Embedded Systeme ist keine leichte Aufgabe. Viele Anbieter bieten Teillösungen an, manche verfügen über komplett abgestimmte Produktserien. Microtec Research hat eine umfassende Palette an Werkzeugen um diese Entscheidung zu erleichtern. Die Compiler und XRAY Debugger von Microtec Research erfüllen alle oben genannten Forderungen. Die VRTX-Echtzeitbetriebssysteme bieten eine offene, abgestimmte Architektur für härteste Echtzeitanforderungen. Die integrierte Entwicklungsoberfläche XRAY MasterWorks bietet eine Plattform zur Integration aller Werkzeuge, auch jener von Drittherstellern. Die Spectra-Software-Backplane ermöglicht größtmögliche Flexibilität zur Entwicklung von Echtzeitapplikationen durch eine durchgängige offene Architektur. □