FROM EXPERIENCE
Time-Driven Development of Software in Manufactured Goods

Tomlinson G. Rauscher and Preston G. Smith

Microprocessors are being incorporated into an increasingly wide range of products. However, many of the companies that manufacture such products are not effectively managing software development for these embedded systems. Despite the current focus on concurrent engineering and cross-functional teams, software engineering is often poorly integrated with the rest of the product development effort. The result is usually a costly delay in the product's introduction to the market.

Tomlinson G. Rauscher and Preston G. Smith describe several practices that have proved helpful for accelerating the development of products that incorporate embedded software. Managerial and economic opportunities for accelerating development of hardware-software systems involve planning for dramatic growth in products that include embedded software, cultivating in-house software knowledge, recognizing the financial effects of project decisions, and measuring project progress. Improving time to market requires hiring and developing software engineering staff and managers with the requisite knowledge of the application, ensuring that they understand the techniques for specifying requirements and design, and providing them with clear guidelines for evaluating the trade-offs between project duration, project cost, and product performance. Progress should be measured in terms of the number of components completed, rather than the number of lines of code that are written.

During the development process, emphasis should be placed on managing the scheduling links between hardware and software development, obtaining user feedback about the system as early as possible, and using a flexible, ongoing review process. Development groups should establish software requirements and design parameters before they start coding, and testing should commence early in the system design process. By creating a working prototype of the user interface, developers can obtain user feedback and thereby sharpen the design specification.

Effective, timely software development requires focusing greater energy and resources on development of the requirements specification. By expending this effort in the first phase of a project, the development team can minimize its use of the time-consuming code-and-debug approach to software development. In addition to breaking down a complex system into understandable pieces, a modular design supports efforts to accelerate product development. With a modular design, work on various modules can be assigned concurrently to relatively independent teams. A modular design facilitates testing of the product as well as reuse of software that was developed and deployed in previous projects

.

Address correspondence to Preston G. Smith, New Product Dynamics, 3493 NW Thurman Street, Portland, OR 97210.

J PROD INNOV MANAG 1995;12:186–199

655 Avenue of the Americas, New York, NY 10010

0737-6782/95/$9.50
SSDI 0737-6782(95)00024-N

Introduction

Today's product development professionals are applying many techniques to reduce the time consumed in delivering manufactured goods to market. These include cross-functional teams, quality function deployment, concurrent engineering, and many other techniques [10,11]. Such techniques typically focus on hardware activities, however, without considering the peculiarities of the software subsystems that often delay the overall project.

The difficulties in managing software development might be relegated to a small group of software specialists if it weren't for the fact that software is becoming an increasingly large part of our products. Products such as telephones, cash registers, thermostats, automotive control systems, copiers, and alarm systems—which had been electromechanical devices—now incorporate a microprocessor. Because these microprocessors require software to drive them, the effective management of software engineering is something we can no longer ignore if we wish to get our new manufactured products to market quickly.

Many of us have recognized that software development requires some new management approaches and thus have turned to the considerable literature on software development processes [6] and software project management [1,13] for answers. These sources of information have been of limited help for software-driven manufactured goods for two reasons. First, the software management literature is generally written for software professionals, thus it often seems as unfathomable to hardware people as the software itself. Secondly—and more subtly—the type of data processing software usually covered in such literature has some essential differences from the type of software that drives the microprocessors embedded in manufactured products. Rauscher and Ott [9] are among the very few authors who address managing the development of the distinctive software used to drive machines in real time, and as far as we know, no one has dealt specifically with techniques for developing such hardware-software products more quickly.

Such techniques are the subject of this article. We explore the subject by describing ten tools used by those who have accelerated the development of hardware-software systems. As outlined in Exhibit 1, the tools cover three areas: management and economics, the development process, and product design.

BIOGRAPHICAL SKETCHES

Tomlinson G. Rauscher has almost two decades of experience managing product development projects. He received his B.S. from Yale University, his M.S. in Computer Science from the University of North Carolina, his Ph.D. in Computer Science from the University of Maryland, and his M.B.A. from the University of Rochester. He has worked for NCR, Amdahl, and GTE in computer systems R&D and for the past 15 years has been a product development manager at Xerox Corporation. Rauscher holds the CDP, the CCP, and two patents. He is the author of two books and more than 25 papers on technical and management subjects. He had lectured on these subjects at numerous conferences, professional meetings, and universities. Rauscher is also listed in *Who's Who*. He is currently managing a product development project that is attempting to set new benchmarks for software productivity and time to market.

Preston G. Smith has worked exclusively for over 10 years on accelerated product development, initially as a staff consultant within a large conglomerate and for the past 8 years as a management consultant with New Product Dynamics. He has applied the techniques of rapid product development to consumer electronics, home appliances, computers, medical electronics, industrial automation equipment, and telecommunications gear. He is coauthor of *Developing Products in Half the Time* and several articles on fast-cycle product development. In addition to his consulting experience, Smith has held engineering and management positions at IBM, AT&T, GM, and other firms. He holds an engineering Ph.D. from Stanford and is a Certified Management Consultant.

Management and Economics Opportunities

Topic 1: Plan for Dramatic Growth in Software

In the quarter century since the introduction of the first commercial microprocessors, performance capabilities have increased dramatically. For example, 1978's Intel 8086 microprocessor could execute 330,000 instructions per second [5], whereas their current Pentium P54C can perform 150,000,000 instructions per second. Similarly, the amount of memory in microprocessor-controlled systems has increased from tens of kilobytes to megabytes. This increase in hardware performance has enabled the development of powerful yet inexpensive systems for controlling the operation of a wide variety of manufactured products.

As it is software that provides much of the functionality and control of electromechanical products, increasing hardware performance capabilities have generated an increasing demand for software. Accordingly, the amount of software in manufactured products has grown rapidly. For example, Figure 1 illustrates the increasing amount of software in copier products from a major manufacturer. To address this growth, many firms have significantly increased the

Exhibit 1. Ten Tools That Have Accelerated Hardware-Software Development. Each of the 10 topics covered in this article describes effective practices for managing the accelerated development of products incorporating embedded software. Here we list some of these practices.

Topic	Effective Management Practices to Reduce Development Time
Management and economics opportunities	
1. Plan for dramatic growth in software	• Staff and train for rapid growth in software complexity • Learn from companies like Hewlett-Packard, where two out of three engineers are devoted to software
2. Cultivate software knowledge	• Emphasize applications knowledge when recruiting software engineers • Assure that managers overseeing software development understand it
3. Root your decisions in project economics	• Calculate the financial value of a one-month launch slip • Be sensitive to software scheduling implications of working close to hardware performance limits
4. Insist on measurable progress	• Don't use lines of code written as a metric for evaluating progress • Measure progress in terms of the number of software components completed
Development process opportunities	
5. Manage the scheduling links between hardware and software	• Co-plan hardware and software activities so that hardware development supports software testing requirements and vice versa • Encourage the use of modern software techniques, which permits testing to occur later
6. Let users "test drive" the system early	• Insist that programmers get actual user feedback early on • Understand the powerful but limited role of software prototyping
7. Institutionalize a fluid design review process	• Plan on software engineers spending 20%—25% of their time on reviews • Monitor the review process to keep it informal (thus fast) but effective
Product design opportunities	
8. Specify requirements first	• Clearly establish the "what" of the software before addressing the "how" • Ensure that programmers understand total project objectives and scope
9. Break up the software monolith	• Keep software modules as independent as possible to break down the communication burden • Emphasize module reusabiity
10. Factor time into hardware/software tradeoff decisions	• Fully assess the time and cost of making software changes • Simplify software development by making cost-effective tradeoffs with hardware costs.

size of their software engineering staffs. The hardware engineering staff has grown more slowly, if at all, because the increased integration of functionality on microprocessor and other chips simplifies the design of increasingly sophisticated systems.

Whereas hardware capabilities have increased rapidly with the commercialization of a new generation of technology every few years, software development techniques have improved at a slower rate. Even with improved software development tools and processes and widespread software engineering education, the amount of software being developed begs for improved productivity. As a result, development managers who have not planned for dramatic software

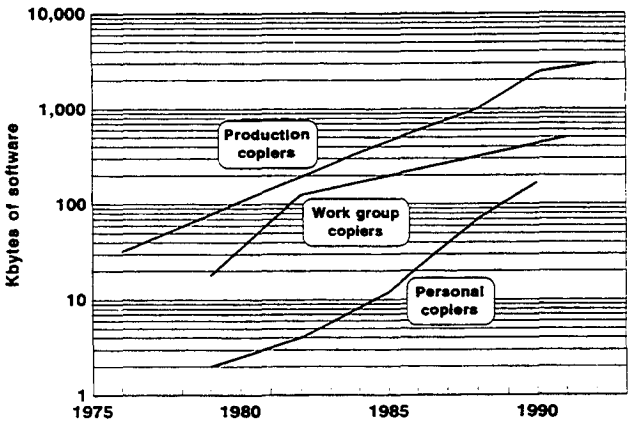


Figure 1. Growth of software in copier products. Note the logarithmic scale; for example, the code in personal copiers has grown over 80-fold in 11 years, or at a 50% compound annual rate of growth.

growth often find that software is delaying the schedules of their products.

Exacerbating this situation is the fact that software is more flexible than hardware; it seems easier to change. Thus, product development planners usually allow for some software additions or changes late in a product development cycle to correct hardware problems or add new functionality. With more software to develop for products already delayed by too much software, these planners may realize too late their inadequate plans for managing software growth.

Over the past several years, managers in progressive companies have increased dramatically the percentage of technical staff that develops software in their organizations, with the balance reaching as high as 80% for some software-intensive types of manufactured products. For example, it now stands at 65% companywide at Hewlett-Packard [12].

Thus, to achieve rapid time to market for their products, firms must plan on expanding the software development and management aspects of their businesses. And because the shift to software implementation of functionality is occurring faster than staffing and training often can be done, leading time-to-market companies actually plan *in anticipation* of software growth.

Topic 2: Cultivate Software Knowledge

Because embedded software is a growing component of so many products, a firm must actively expand software knowledge at all levels in its organization just to keep up its pace. In product development teams, the increasing amount of embedded software has created a growing demand for software engineers. At higher levels, product development managers need a greater knowledge of software technology and management techniques to make effective decisions on the direction of their firm.

Development managers should address the growing pains of the software engineering staff through a continuing recruiting program and a concomitant skills development program. The two fundamental approaches to recruiting, acquiring permanent employees, and hiring temporary contract or consultant personnel, offer different advantages and disadvantages to the firm.

In hiring new software engineers, many firms advertise for positions by describing specific qualifications. This may include a programming language (C, C++, Visual Basic, COBOL, etc.), a database system (DB2, Access, Oracle, SQL, etc.), an operating system (UNIX, AIX, Windows, etc.), or even particular hardware. The advantage of this approach is that it may attract people whose particular expertise can help on a project that has been well defined and only requires knowledge of the particular implementation details. Thus if a project is in the coding or testing phase of development, this approach may help accelerate product development.

Unfortunately, projects experiencing delays are seldom well defined and seldom require only coding and testing assistance. More often, projects that experience delays have poorly expressed requirements and design specifications. To achieve rapid time to market, such projects require software engineers with skills in techniques for requirements and design specification and with experience in the general application area. Software engineers who have the knowledge to specify the requirements and design of an embedded system can define the software modules in a way that facilitates implementation by engineers working concurrently with one another on various portions of the software system. Software engineers who have specific knowledge of a particular product application usually have deeper insights into the problem domain, and can perform the high level definitional activities very rapidly. Thus, to improve time to market for most projects, our goal should be to attract software engineers who are experienced in the application area and in the techniques of requirements and design specification. An effective long-term approach to this subject is to train people inside the company in modern software engineering techniques. These people become the knowledge engineers who have the specific infor-

mation needed to develop applications software quickly.

With the demand for increasing the software engineering staff, it may be difficult, or undesirable from a financial commitment viewpoint, to hire permanent employees in a timely way. To fill the gap, many firms use large numbers of contract software developers to help them bring projects to market rapidly. One advantage in using contract employees is that a company can acquire them much more quickly than permanent employees. Another apparent advantage is a cost savings compared to permanent employees; the company usually does not pay for benefits or make a long-term commitment to the temporary employee. However, a closer examination reveals additional costs. In order for contract employees to be productive, the company must provide tools and facilities: office space and furniture, computer systems for development and debug, software tools for program development, etc. The largest cost from an accelerated product development perspective is the time required for contract employees to learn the particular application and product on which they are working. This factor causes problems in two ways:

- New employees require some learning time before they are able to work productively in their assignments.
- Employees who already are working on the product must devote significant time to training the new employees and communicating with them. This can make current employees less productive.

Thus, hiring contract personnel requires a significant investment in time and money; that investment is lost when contract personnel terminate their employment with a company.

The investment in time that must be made to bring new software engineers onto a project is so significant that management should plan employee staffing profiles carefully over time or they will surely lag the increasing demand. For a project experiencing delays, the quick fix of adding new people may be counterproductive. This observation led to the formulation of Brooks' Law: "Adding manpower to a late software project makes it later" [3, p. 25].

Because software development requires a great deal of technical knowledge, skill development of the engineers involved can accelerate product delivery. However, skill development takes time, so skill development must be carefully coordinated with the project schedule. Similarly, using advanced software development tools can accelerate product delivery; however, deploying new tools during a project can be a time-consuming activity. A useful technique for balancing skills development and tools acquisition with the desire to deliver a product to market rapidly is to perform these activities with a group of people while the rest of the team is developing requirements specification and high level design. At the detailed design stage, the two groups formally merge to form a product development team with a requirements specification, high level design, high skills level, and deployed set of tools. Another aspect of rapid product development is the deployment of software technology. On one hand, it is desirable to incorporate new technology into products, as that affords market leadership opportunities. On the other hand, incorporating new technology into a product is more difficult and time-consuming than using proven technology, and therefore must be planned carefully.

Because of the relative youth of software, few people with software knowledge have also had the opportunity to develop management skills. Consequently, firms that wouldn't think of having a non-EE manage electronics development seem to slip easily into the mode of letting an individual without significant training or experience in software development manage that function. We have seen major companies commit this fault repeatedly. The results are predictable: managers without specific knowledge make improper decisions, and the project takes much longer to complete than should be the case.

Topic 3: Root Your Decisions in Project Economics

Many development projects are late to market simply because those involved often fail to factor the financial value of time into their decision-making adequately. As a result, they often undervalue time 10-fold when compared with development expense or unit manufacturing cost. Development teams can use straightforward techniques to calculate the financial value of a 1-month schedule slip [10, p. 17–41]. Such calculations must be done on a product-specific basis, because trade-offs are likely to balance at vastly different points for, say, a Sony Walkman than for a ship navigation system.

The result of the calculations is a set of trade-off rules for that product that enable everyone associated with the development project to make sound, consistent, and fast decisions when comparing time, money,

and product features. The trade-off rules are powerful tools for developing products quickly, simply because decisions with time—money—quality trade-off implications occur every day in all facets of a project, and without such rules the decision-making process bogs down. In particular, product developers can apply these rules in making trade-offs on software versus hardware implementation (Topic 10) and in many other areas that ultimately affect software development.

A simple example of such trade-off analysis lies in the decision of the type of memory to use in a product. For many software-driven products (automobiles, appliances, office equipment, etc.), the user of the product does not have to change the software in the product to use it. Indeed, the user may not even know that the product is software driven. For such products, the software may reside in read-only memory (ROM), a memory technology in which the software is "burned in" to the memory, never to be changed again. Such memory is inexpensive, yet it is inflexible; it cannot be changed. At the other end of the memory spectrum is dynamic random access memory (DRAM), which can be read from or written into at speeds comparable to the execution speeds of microprocessor instructions. Although this memory is more expensive than ROM, it offers flexibility in making changes. Thus software being developed in DRAM can be more easily changed in development, test, and field environments, which can accelerate product development and facilitate incremental improvement. When development groups select the memory technologies for use in a product, they usually consider trade-offs between the cost and flexibility of various memory types. An important part of this trade-off analysis should be the time to market benefit that a flexible memory type provides. Note that although this is a hardware selection issue on the surface, it has many software scheduling ramifications.

Software researchers have found that there are many other factors that substantially influence the amount of effort and the length of time required to develop software for a product. In his seminal work on the subject [2], Barry Boehm developed a constructive cost model (COCOMO) for estimating the effort and time to develop software, based on fifteen factors shown statistically to affect software development. Table 1 groups these fifteen factors into four categories.

A product development group can shorten the time required to deliver a software-driven product by addressing the following factors:

Table 1. Factors That Influence Software Development Time and Effort

Product attributes
Required software reliability
Database size
Product complexity
Computer attributes
Execution time constraint
Memory size constraint
Virtual machine volatility
Computer turnaround time
Personnel attributes
Analyst capability
Applications experience
Programmer capability
Virtual machine experience
Programming language experience
Project attributes
Modern programming practices
Use of software tools
Required development schedule

- Execution time constraint.
  Selecting a faster microprocessor for a product reduces the time software developers devote to optimizing their software to avoid real-time problems.
- Memory size constraint.
  Similarly, providing a large amount of memory reduces the time software developers devote to optimizing their software to fit into a constrained memory size.
- Computer turnaround time.
  Providing a powerful computer system that quickly runs software development tools reduces the time a software developer often spends waiting for results.
- Use of modern programming practices.
  Routine use of a suite of modern programming practices (structured or object-oriented analysis and design, top-down incremental development, design and code walkthroughs or inspections, etc.) simplifies the process for developing software, thus reducing the time to deliver a product to the market place, largely by reducing rework.
- Use of software tools.
  An advanced set of software tools (compiler, operating system, interactive debugger, design language, configuration management system, documentation system, project control system, etc.) reduces the effort software developers devote to

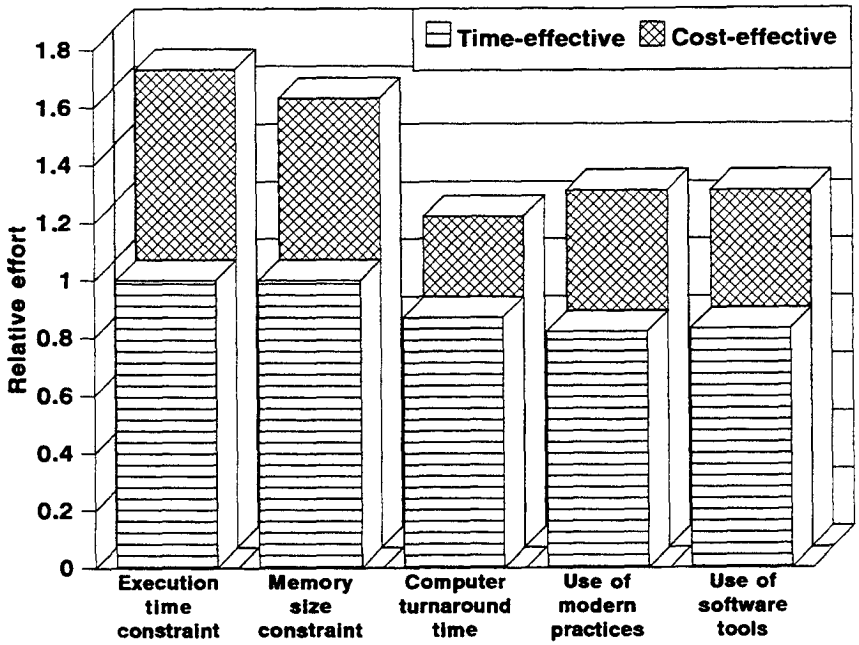


Figure 2. Boehm found that 15 factors had a significant effect on software development time and effort [2]. The graph shows five of these factors that are manageable. For each factor, the bars contrast relative effort for the time-effective versus cost-effective values of the parameters. The value 1.0 represents the value for a nominal project.

clerical tasks or assists software developers in intellectual tasks, thus improving time to market.

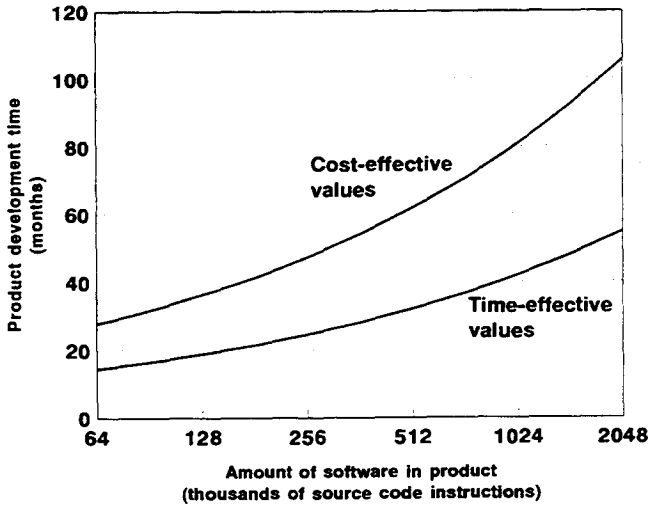
Boehm found that the effort required to develop software for a product varied significantly depending on the trade-offs made in a product development project. He was able to quantify the relative effect of trade-offs for these factors. In Figure 2, for example, we see that if the memory size constraint is extra high (95% of available memory is used, the cost-effective solution), then the effort required to deliver the product will be 56% more than the nominal case (less than 50% of the memory is used). Of the factors in Table 1 not shown in Figure 2, the combined set of personnel attributes has a huge effect on required effort and development time. Thus, in order to shorten development cycles continually over the course of several products, management should invest in increasing the capabilities and experience of the staff.

Using a few simple formulas, one can estimate the effort and time required to develop the software for a product using the relative values of the factors in Table 1 and an estimate of the size of the software to be developed. Figure 3 shows the variation in development time between projects that make decisions to minimize software development time and effort versus projects that make decisions to minimize cost. The graph shows that development time could double when developers make decisions in favor of cost rather than schedule.

Topic 4: Insist on Measurable Progress

It is difficult to manage activities that cannot be measured, and accurate measurement is especially impor-

Figure 3. Effect of manageable factors on development time. These curves were calculated from the COCOMO model using cost-effective versus time-effective values for the five factors described in the text, which a project can control. Other factors could affect product development time further.



tant when timely completion is a key project requirement. On the surface, it would appear to be straightforward to measure development project completion in terms of the number of drawings completed for hardware and the number of lines of code written for software.

However, innovation does not proceed so methodically. Honest engineers fall prey to the 95% syndrome, in which 95% of the work seems to be completed, assuming that no more unforeseen difficulties occur. By the next month, some additional unanticipated problems have been solved, certainly placing us ahead of last month, so progress must now stand at 98%.

This trap is even more severe for software than for hardware, due to the fact that software is that much harder to see or touch. The problem areas are correspondingly more difficult to notice or manage, so the schedule tends to be softer.

For hardware development we often advise against highly structured phased development processes, because although they appear to contribute a degree of certainty to the process, they also act is several ways to slow down development [11]. However, software development requires more structure and clearer completion points because of its intangible nature.

Fortunately, there are some tools available to measure the progress of software development activities. One is to allocate effort by development phase according to past experience as to how much work is required in each phase [9, page 8]. This does not mean that one cannot overlap phases, but it does highlight areas in which overly optimistic thinking is occurring. For example, if one does not allocate enough testing effort relative to the coding effort that is planned, there is likely to be schedule slippage in the testing phase.

Another tool that teams use to manage progress is to break the whole project down into "bite sized" tasks, each of whose completion is verifiable, and measure completion according to the number of tasks that have been completed to date. No credit is given for partial completion of a task. Normally, a task should represent two to four person-weeks of effort.

Part of the project planning is to determine the staffing required to complete the defined tasks. From this information, one determines how many tasks will be completed as a function of time, which provides a measurable target of planned completion at any time. Actual completion can be plotted against this curve to see how well the project is tracking the plan. Kmetovicz provides details on this project management tool [8].

We suggest managing progress in terms of the percentage of small tasks completed because the alternatives do not work well for embedded software. The traditional alternative is lines of code written, but it has several flaws. If effective software development practices are used, actual code writing is a small part of the whole development process. Moreover, the lines of code written can vary greatly depending on the language used and the skill and diligence of the programmer. Some programmers are rewarded by the number of lines they write each day, encouraging them to be verbose, whereas others are under tight restrictions on memory usage for the program, which requires them to spend extra time being concise. Finally, just because a line of code is written doesn't mean that it is correct, and if it was done sloppily, many times the coding effort can be spent in finding the mistakes and correcting them.

Development Process Opportunities

Topic 5: Manage the Scheduling Links Between Hardware and Software

Hardware and software development each have their own scheduling demands that would appear to preclude developing the hardware and software concurrently. Consequently, product development teams must devote special attention and creativity to overcoming the natural tendency toward sequential development. A common example of this conflict is the desire to have a hardware vehicle available for testing the software before the target hardware is normally working. In many cases this situation is further complicated, because the hardware engineers want some software to drive their hardware and demonstrate that it works. Basically, each discipline believes it must wait until the other discipline has completed its work. There are several solutions to this apparent dilemma:

- Encourage development groups to use modern software development practices, which involve investing heavily in upfront effort to establish software requirements and design parameters before starting the coding [9,13]. These practices naturally defer the need for a test vehicle and minimize the need for tail-end software test efforts to catch problems that could have been an-

ticipated. They also produce code that is more mature and consequently is better positioned to support initial hardware testing. Thus, enforcing good programming practice minimizes the problem.

- More specifically, plan the hardware and software testing early—simultaneously with system design—to anticipate scheduling conflicts and potential delays in the testing phase. In addition, this encourages test developers to focus on system functions, not design details.

- Recognize that the interaction of new hardware with new software will naturally complicate the testing process, thus stretching the testing schedule. Solutions include using the techniques of incremental innovation, reuse, and modularity, to minimize the amount of new design that requires testing [10, pp. 6979, 99—06], and the techniques of risk management to resolve different risks simultaneously but independently [10, pp. 214–221].

- Build a simulator, using the target microprocessor, for use as a software testbed. Then the software can be debugged before the actual hardware is functioning. Such simulators are also useful for debugging and testing the software while prototype hardware evolves through its design cycle, especially during periods when the hardware is temporarily decommissioned to incorporate design upgrades. Furthermore, simulators provide a consistent baseline that isolates software debugging, which makes it easier to determine which problems relate solely to software and which involve software–hardware interaction. Note that building simulators is an application of our principle of "building a tall junk pile" [10, p. 191].

Underlying these solutions is a common thread of close collaboration between hardware and software engineers. Because there is a high degree of interdependence between software and hardware activities, and opportunities abound to blame the other side for scheduling problems, the basic approach to accelerating the combined process is to change "they" situations into "we" situations. Possible solutions include:

- Organizing so that software and hardware personnel report to a common project leader.

- Co-locating this combined team.

- Jointly creating product specifications and project plans to promote joint ownership of objectives.

- Ensuring that evaluation, recognition, and reward systems support mutual accountability.

Although the value of this collaboration may seem obvious, it often fails to happen in practice. Software engineers are often viewed as scarce, specialized resources (Topic 1); the nature of programming often calls for lengthy, uninterrupted concentration, which seems to preclude a shared work space; and because management does not understand software development (Topic 2), they allow software engineering to become an isolated technical resource instead of an equal partner in the business of product development.

Once this joint team is in place, development groups should emphasize operating proactively rather than reactively to identify and exploit opportunities for getting activities off the critical path. This is why we have suggested previously that test plans be created early, so that potential schedule or technical risk difficulties can be identified while maneuvering room exists.

Topic 6: Let Users "Test Drive" the System Early

One factor is clearly more important than time to market: getting a product to market that is a commercial success. And obtaining a commercial success is 3.3 times as likely when the team has specified requirements, that is, there is a sharp problem definition [4, p. 59]. The difficulty, though, is that customers often don't know what they want until they see it. Consequently, there is great value in putting a likeness of the system into users' hands as early as possible in order to see if it satisfies them. In addition to the obvious customer satisfaction issues in getting their feedback, however, there are direct timesaving advantages: development teams can focus further development where it will create the greatest customer value and can drop features that evoke little customer enthusiasm, thus saving development time.

Hardware developers use prototypes to get user feedback, and one of the techniques of rapid hardware development is to get prototypes into users' hands early on. The same approach, known as software prototyping, applies to software development.

Consider how Siemens Medical Systems, Inc.,

Electromedical Group, used software prototyping to home in on user requirements quickly. They were developing a cardiac work station, and it wasn't certain what would be the best approach for designing the user interface. So the software engineers mocked up an interface using high-level, easily modifiable software tools. Then they brought in some cardiologists to "test drive" the prototype. Based on these user reactions, they modified the prototype in a few hours and had the same users try the modified version to see if they had it right yet. Once they had a firm grasp of what would be easiest for the doctors to use, they could plan the real software development efficiently.

Hardware prototypes are broadly applicable tools that can be used to test user interaction issues or determine whether the interior portions of the machinery will operate properly. For example, some hardware prototypes are used to check out ergonomic issues, whereas others run for millions of cycles under adverse conditions just to see if the design will endure. In contrast, software prototypes have a more focused area of application. They can be even more powerful than their hardware counterparts in specifying user interfaces with the product, because interface design requirements can be more elusive in the intangible software medium than in hardware. For interface design, the time taken to understand user needs through prototyping can save much redesign time later. On the other hand, the use of a prototyping, cut-and-try approach for the interior kernel, or "engine," portion of the software is an ineffective, time-consuming means for writing code when sound engineering methods are available for building these clearly specified portions of the system. Thus, the use of hardware prototypes can be embraced universally, but the software counterparts are quite advantageous in some areas and detrimental in others.

Similarly, embracing software prototypes leads to an important methodological dilemma. Software used to be written in an informal cut-and-try style, but as software projects became larger and more complex, this style became ineffective. This has led to the modular software methodologies described in Topics 8 and 9 and also to making sure that programmers switch to these methodologies. In a way, software prototyping is a throwback to the "obsolete" informal methods from which modern software leaders are just weaning their workers. Thus, it seems that software prototyping is headed in exactly the wrong direction in terms of the development of a science of programming.

This brings us to an important principle of rapid product development, which is to be flexible about the process used. In many cases a formal engineering approach will be fastest, because it will allow effective subdivision of the work and thus a more concentrated effort, and it will reduce dead ends and rework. Yet, there is also a place for the powerful but informal tool of prototyping. The fast product developers are those who recognize the power and pitfalls of each tool and use a combination of approaches that works most effectively for the case at hand. To arbitrarily rule out an unstructured approach like prototyping is to slow down—or worse, overlook—the crucial task of understanding what the customer wants.

Topic 7: Institutionalize a Fluid Design Review Process

Hardware developers often ascribe considerable importance to design reviews. But these same people seldom conduct design reviews on a regular basis, nor are they often happy with their review process. For hardware, reviews are a useful design assurance technique, but other techniques, like prototyping, can also assure the design quality.

Software design requires a design assurance process to a greater degree than hardware, as design flaws are more difficult to spot in the less visible medium of software. Moreover, software does not have tools such as prototypes available as alternatives to reviews. (Software prototypes, discussed in Topic 6, are valuable for certain purposes but not for verifying design quality.)

Thus, whereas hardware reviews might be put into the nice-to-have category, software reviews—sometimes called walkthroughs or inspections—are an essential part of the software development process. Software engineers who are working effectively spend 20%–25% of their time on reviews: preparation, the review itself, and follow-up activities. Projects without this level of review are likely to bog down in their later stages as previously undiscovered problems arise. When such mistakes are discovered late, they are likely to be time-consuming to correct, in part because any successful testing done to date will have to be repeated to assure that the fix hasn't introduced new bugs.

The temptation might be to go light on reviews, especially for those coming from a hardware develop-

ment background, because reviews are seen as a non-value-adding activity that does not directly result in lines of code written. Yet, reviews really fit into the ounce-of-prevention category.

Because reviews are such a large part of the software development process, the real schedule issue isn't so much the amount of review but the responsiveness of the review. In the fastest projects, development groups use software reviews as an informal but essential means of doing business. They are handled at a relatively low level and on a frequent basis, by using competent peers. The low level tends to keep them informal, and doing them frequently keeps the task from becoming overwhelming and catches difficulties when they can be fixed easily and without bruising egos. When peers are used routinely for review, they soon learn to be thorough but fair in their assessment, because they are likely to be the reviewee rather than the reviewer next time. Management's main responsibility in this informal system is simply to make sure that reviews occur on a frequent, regular basis and to monitor the overall effectiveness of the system to assure that it makes the best use of people's time.

Several conditions are necessary for such a system to take root in an organization, the sorts of things we have all heard before: top management support, supported by adequate and ongoing training, in alignment with corporate goals, and consistent with the reward and compensation systems. Along these lines, one issue stands out when development urgency is paramount: an adequate source of accessible reviewers is needed so that reviews can be scheduled routinely on reasonably short notice. Otherwise, projects experience ongoing delays waiting for review, and review is done on work that has solidified to the point that changes become costly. There are two implications here, both of them fundamental and obvious but nonetheless difficult to achieve. First, the compensation and recognition system must encourage reviewing, which can be as simple a matter as having a charge number for reviewing, so that it does not have to be charged to the reviewer's or the reviewee's project. Second, managers must not so overload their most valuable reviewers with various assignments that their backlog precludes timely reviews.

Some companies are using outside contractors as reviewers. This is an example of trading development expense for schedule time, which the project's economic analysis (Topic 3) may show to be an excellent buy. Outside reviewers have no axe to grind, and they can be called in when desired.

Product Design Opportunities

Topic 8: Specify Requirements First

As described in Topic 1, today's embedded software systems can be very large, often reaching hundreds of thousands of statements written in a high-level computer language. Such complexity makes it difficult to comprehend the task of developing a feature-rich embedded software system or even to determine a starting point in understanding the design of a large system based on software developed for a previous product. Effective software development groups use two key techniques to guide the design of and enable comprehension of large software systems: requirements-driven design (covered here) and modular design (covered next).

Although requirements-driven design may seem natural to most product developers, it isn't natural to software engineers. Most software developers learn their craft by studying some basic computer language constructs, then creating small programs. For each program they develop, they spend some time coding and then debugging the program until it works the way they want. Although this incremental approach is a wonderful way to learn programming, it is ineffective in developing large embedded software systems, because it emphasizes implementation before the understanding of the structure of the entire system to be developed. Many companies still use this approach because they believe they can measure progress in terms of lines of code written. This is usually not an effective measure, because it does not indicate the correctness of the software developed or indicate the amount of effort that remains.

In contrast, software development teams should spend much of their time on the most important phase of software development, namely, the requirements specification phase. The purpose of this initial phase of the software development process is to produce a requirements specification document that defines what the software system must do: the functions to be performed, the performance to be achieved, and the design and development constraints. If the team fails to develop a specification properly, the software developers involved in subsequent design and test activities

must make assumptions about the exact nature of the requirements. The assumptions are seldom entirely correct, and this causes errors in later design and coding phases of embedded software development. Note for software that is to be reused in other products, the specification is critically important.

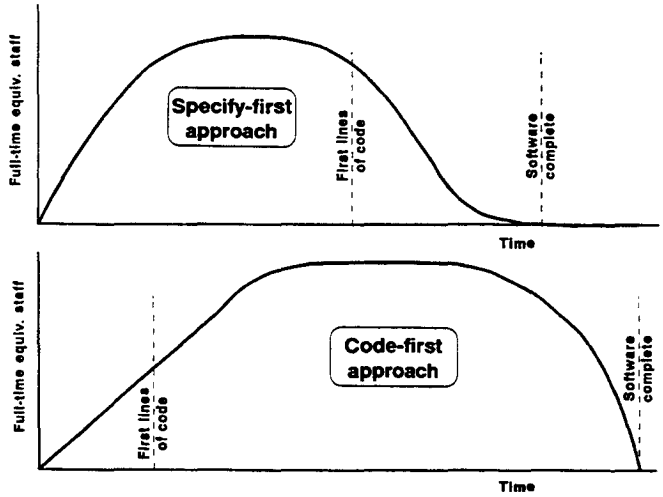
The next phase of software development, design, transforms the requirements specification document into a detailed physical structure of modules, which are later coded into software components. Like the architectural blueprints of a construction project, a good design provides speed advantages in delivering a product to market [3, p. 143]:

- From the detailed design, the developers can determine more accurately the time required to complete software development.

- The partitioning of modules avoids system bugs, which otherwise would not be found until late in the development process when they would be time-consuming to correct.

Although specifying requirements first requires an investment of time early in the project, it saves much more time later on. The problem with the code-and-debug approach to design is that it masks fundamental interface and integration issues that are time-consuming to fix late in the development process. Figure 4 illustrates the time-to-market differences between these two approaches to software development.

Figure 4. The "specify-first" approach delivers the first lines of code later but delivers the entire software package earlier with less effort. These graphs are based on experience regarding the relative probability of making errors and the cost of fixing errors using the two different approaches.



Topic 9: Break up the Software Monolith

The concept of modular design evolved from the observation that people cannot simultaneously comprehend the large number of details in a monolithic structure. Modular design organizes this monolith into comprehensible components, called modules. Developing the design of a software system, or subsystem, then becomes a task of deciding on the capabilities of each module and the interfaces among the modules. The modules in a good design exhibit minimal coupling, so that the development and understanding of one module does not depend on or affect the design of another module. Inside the modules of a good design, the data and functions are highly cohesive, so that the development and understanding of the module does not depend on too many details. Although they are beyond the scope of this article, the design principles of coupling and cohesion are well established [9, pp. 104–107]. Thus, the value of a modular design is its presentation of the necessary information in pieces that can be studied independently yet can be easily understood.

Beyond the improvements that accrue from improving the understanding of a complex software system, modular design offers additional opportunities for accelerating the development of embedded software systems:

- Relatively independent teams can be put to work on different modules, thus multiplying staffing without paying the communication burden normally associated with large teams.

- Testing becomes more flexible to schedule, because modules can be tested fairly independently whenever they are ready, rather than having to wait for completion of other modules.

- Modular design facilitates reusing software developed and deployed in previous products. This is a powerful technique for reducing the time required to develop a product by simply reducing the amount of software that has to be written and tested. Indeed, many companies have invested in developing libraries of reusable modules, from which product development teams can check out software that has already been developed and tested. Their task can then be characterized as one of integrating existing software with new software. For example, Xerox frequently bases the software of new copiers on software modules de-

veloped previously, thus significantly reducing time to market.

Topic 10: Factor Time into Hardware/Software Trade-off Decisions

The traditional battleground between software and hardware engineers is whether a specific feature should be implemented in hardware or software. Many factors enter into such decisions, including manufacturing cost, production volume, development effort, system performance, and ease of accommodating design changes. Our experience, however, is that product developers do not take into account the appropriate factors in making such decisions. Typically, they base their decisions on manufacturing cost but do not consider the dollar cost of development time.

Although such hardware-software implementation decisions appear to be technical trade-offs, their basis is in fundamental business and product strategy principles. To obtain trade-offs that properly balance development time with other project objectives, software engineers will have to acquaint themselves with end users, competitive offerings, field service procedures, and the like. This demands that they be integrated into the project at a level far deeper than just "designing to spec." On the other hand, those from marketing, purchasing, and similar functions will have to learn more about software development (Topic 2) to see through the smoke screen that software engineers sometimes try to erect when it seems simpler to make decisions unilaterally.

Consider a trade-off decision made by one of our clients—Siemens Medical Systems, Inc., Electromedical Group—when they developed a special input-output board for one of their instruments. The board had two quite different and reasonably independent functions, but technical considerations suggested that both functions would fit on one circuit board, sharing a microprocessor. However, when they looked at the design as a time to market issue, they decided on quite a different solution. Both functions were placed on one board, but each had its own portion of the board and its own microprocessor. Thus, they had to buy an extra microprocessor on the hardware side, but they gained key schedule advantages on the software side:

- They could divide the project into two relatively independent subprojects, which meant clearer objectives, smaller teams, better communication, and easier testing.
- They were working well down on the microprocessors' processing capability curve, which meant that the software engineers didn't have to spend extra time planning and writing unusually efficient code to satisfy the machine's cycle time requirements (see Topic 3).

Although Siemens made this decision to facilitate software development at the expense of hardware cost, many such decisions go in the other direction, especially late in the development cycle, when the hardware seems "locked in," and software seems so flexible. When features tend to migrate toward software implementation, the software portion of the project becomes overburdened, then slips. Often product development groups fail to detect this effect because the changes in requirements occur progressively—what aptly has been called "creeping elegance." At each step the software change seems relatively simple but the effort required to make the hardware change is clearer.

Perceived cost is another factor that drives product developers too strongly toward software solutions. A software change often seems free, but changing the hardware has clearer manufacturing cost, development expense, and scheduling implications. The solution here is not only to estimate realistically all of the costs associated with the software change but also be aware of the trade-off factors between cost and time. A slip in the schedule is usually much more costly than just the extra salary of the engineers involved. To make rapid, accurate decisions between software and hardware implementation, everyone influencing a project must know just how much one month of schedule slippage will affect the company's bottom line, which can be calculated as indicated in Topic 3.

Conclusions

Two themes underlie the advice given in this article. One is that because software is a new and dramatically growing element of many manufactured products, the software development process is subject to many types of growing pains. These include a scarcity of software engineers, the need for a strong program to recruit and train such engineers (even if the company is not recruiting in general), the need to procure new software development hardware and software and replace such development tools that have become obsolete quickly, and the need for everyone in the company involved with new products to become as comfortable with soft-

ware development as they are with hardware development. Because of the explosive growth of software-based functionality in the "afflicted" products, the implementation of these changes is likely to lag the need for the changes, even in the best-run companies. And until implementation catches up with the need, software development problems are likely to be the cause of many product launch delays.

The other theme is that software development issues must be integrated into the daily decision-making process of a product's development. This journal has carried many articles on the integration of marketing with R&D, and the Product Development and Management Association has even sponsored annual conferences on this subject. On a related front, the popular concurrent engineering movement has helped companies to integrate manufacturing issues into a product's daily design decision processes so that the resulting product design is manufacturable at target cost. A logical extension of this trend is that if software is a crucial part of a new product, software engineers will have to be woven into the ongoing decision-making that makes for a successful, rapidly developed product.

Yet, the obstacles to making the software engineers regular members of a development team may seem more difficult to surmount than for other disciplines. Software is a highly technical specialty—which some software engineers may prefer to keep to themselves. Software development requires long uninterrupted periods of concentration, which tends to conflict with team co-location. And software engineers are usually the last to come into contact with the customer. For anyone who has not experienced software development directly, *The Soul of a New Machine* [7] and *Showstopper!* [14] illuminate this distinctive world of the software engineer effectively.

Although the path may be difficult, we have tried to provide a good map and signposts here, and we know from experience that the destination—faster development of embedded software—makes the journey worth taking.

References

1. Boddie, John. *Crunch Mode: Building Effective Systems on a Tight Schedule*. Englewood Cliffs, NJ: Yourdon Press, 1987.
2. Boehm, Barry W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
3. Brooks, Jr., Frederick P. *The Mythical Man-Month—Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975.
4. Cooper, Robert C. *Winning at New Products*. Reading, MA: Addison-Wesley, 1993.
5. Halfhill, Tom R. 80X86 wars. *Byte* 19(6):74–88 (June 1994).
6. Humphrey, Watts S. *Managing the Software Process*. Reading, MA: Addison-Wesley, 1989.
7. Kidder, Tracy. *The Soul of a New Machine*. New York: Avon Books, 1981.
8. Kmetovicz, Ronald Eugene. *New Product Development: Design and Analysis*. New York: John Wiley & Sons, 1992.
9. Rauscher, Tomlinson G. and Ott, Linda M. *Software Development and Management for Microprocessor-Based Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1987.
10. Smith, Preston G. and Reinertsen, Donald G. *Developing Products in Half the Time*. New York: Van Nostrand Reinhold, 1991.
11. Smith, Preston G. and Reinertsen, Donald G. Shortening the product development cycle. *Research-Technology Management* 35(3):44–49 (May–June 1992).
12. Whiting, Rick. Hewlett-Packard's software initiative from the top. *Electronic Business* 18(9):53 (June 1992).
13. Whitten, Neal. *Managing Software Development Projects*. New York: John Wiley & Sons, 1990.
14. Zachary, G. Pascal. *Showstopper!: The Breakneck Race to Create Windows NT and the Next Generation at Microsoft*. New York: The Free Press, 1994.