My favorite stage of design is proof of concept. Given unlimited resources and time, of course, you can implement every design. However, I get immense satisfaction from figuring out how to implement a complex concept within the constraints of real-world engineering. For example, can you do it fast enough with a single processor and in the bit of memory left over from the last redesign?

In an EDN hands-on project last year, Technical Editor Bill Schweber test drove several communication-simulation tools ("Communication-simulation software smooths system design," Aug 3, 1998, pg 87). For this hands-on project, I decided to take a closer look at one of those tools aimed at turning digital-signal-processing concepts into reality. The DSP domain imposes its own set of unique constraints, usually involving the processing of massive amounts of data within tightly defined time slices. I chose the Matlab environment, from The Mathworks (www.mathworks.com). The Mathworks created Matlab for designing control systems but has since added many communication and signal-processing toolboxes as well as other kinds of modules.

For my exploration, I used Matlab as the basic development platform. I added the optional DSP Blockset, which provides libraries of DSP functions for use in Simulink, a graphical block-diagram tool. I also chose Stateflow, which supports state-machine models, and Real-Time Workshop, which brings all of your work together into a PC runtime executable. All of these tools integrate with each other through Matlab, allowing you to make on-the-fly changes to your system and immediately evaluate the results. I spent a good span of time getting to know the various components and testing the tools using assorted demonstration designs. My goal was to better understand the process for developing in a structured environment and to explore the advantages and limitations of such a process.

The Mathworks also offers many modules and libraries that I didn't use. Do be careful, though: Although Matlab's base price is lower than other algorithm-development tools, adding the options you want can quickly bring its price right up to the same level.
Different media

I uncovered five ways of developing in Matlab. I'll call each one a medium, bringing to mind the way artists use different media (paint, ink, and tempera, for example) to capture images. The five media are Matlab's matrix-based M-language, block diagrams (Simulink), state machines (Stateflow), s-functions (system functions you write yourself in C), and libraries. Functions in libraries take the form of one or more of the other media. Note that, although Matlab forms the platform for the other tools, you do not need to code with M-language. You could go directly to Simulink and design in blocks without writing any code. To get the most out of Matlab, however, you'll want to learn to use all of the tools together, which requires a time investment.

I found that one of Matlab's most powerful features is the integration between these media. In some cases, such as when writing C-code, you must follow compatibility guidelines to ensure that data successfully passes between your code and the rest of the system model. But, by and large, the pieces integrate smoothly. This flexibility is both one of Matlab's greatest strengths and one of its greatest frustrations.

The strength is that you can work in the medium that best suits your application or preference. I remember struggling during the shift from command-line editors to graphical-user-interface (GUI) word processors. If you are someone who rarely uses a tool, GUIs can be a blessing. However, expert users find complex GUIs much less efficient than command-line interfaces. I liken Matlab to a combination of both. If you choose, you can work at the block-diagram level, defining signals as lines, or at the state-machine level using states and transitions. Or, you can work at the code level. Some design work is easier as blocks; some, as straight code. You get to decide which method is best for each aspect of your design.

This flexibility can give rise to frustration, however, because the overall tool becomes more complex. I had to learn several distinct interfaces. For example, Matlab is a matrix-based engine, meaning that notation and data take a different form from that of other programming languages. Matrices offer a more organized format for representing and manipulating data than do arrays, and Matlab offers a selection of commands to manage the matrices. If you've struggled to make arrays work for you, it may take awhile to get used to the new format, but it's well worth it. Both Simulink and Stateflow have fairly straightforward interfaces, but if you've never used block-diagram tools or state machines, you've got a learning curve ahead. Connecting blocks is easy enough, but understanding the exact nature and flow of the data that the lines represent sometimes requires some head scratching. Taking the time to learn the tools gives you access to a lot of power. With all tools, the value you gain is directly proportional to how proficiently you learn how to use them. (see sidebar "Reporter's notebook: Matlab").

Ignore the details

The beauty of algorithm-development tools is that they temporarily remove certain constraints to facilitate proving a concept. For example, I specified double-precision data types to verify that the math worked out. Of course, a final system may be unable to support that many significant bits, but I postponed dealing with the question. The development environment let me focus on the meat of the problem—proving the algorithm—and let me leave how many significant bits I would need until later.

A PC or workstation makes a powerful platform on which to develop and test. You can simulate perfect signals (handling imperfect signals is a detail, albeit an important one) and access all processing stages and data states without having to probe hardware with an expensive high-speed emulator. Additionally, the PC takes care of all I/O for you. You can generate incoming signals or
simulate them from file. Likewise, you can save outgoing signals to file for later analysis.

You also postpone the details of assembly and C programming. Working in assembly often amounts to the time-consuming art of register management, which factors more into algorithm implementation than design. I used to spec out assembly using Basic because Basic supported complex math functions and I had access to a strong debugging environment. Algorithm-development tools offer the same advantages, but they also provide debugging capabilities. These capabilities give you the ability to look at an algorithm from a perspective that isn’t obvious or common. With Simulink, I could animate execution of signal processing by attaching a nonintrusive scope anywhere in a block diagram. I could also capture data that I wanted to analyze and then run it through an independent set of processing blocks, such as an FFT, for statistical analysis. Much in the same way that you can process data for the algorithm, you can process the data for test and analysis. Many common functions, such as FFTs or filters, are straightforward to place between the signal you want to test and the output scope.

Designing a DSP algorithm is challenging because testing and analyzing a particular implementation can be as complicated as the algorithm itself. If you try to develop in a high-level language, such as Basic, you have to design your own suite of graphing tools and filters. One of the benefits of working in the Matlab environment is that these functions already exist and are quite powerful. Additionally, you have access to the source code of most functions, allowing you to customize analysis functions to handle special quirks or boundary cases specific to your algorithm. You can also quickly add complex debugging triggers within your system, a key feature for tracking difficult and “transitory” bugs. Much the same as you would add lines of debugging code to a program, you can anticipate special cases and set a signal or state when the event occurs.

The electronic napkin

You can learn several lessons from my exploration of the Matlab family (see sidebar “Hard-earned lessons”). One of the most important is the influence of a tool on the final design. The chief purpose of a development tool is to help move you from concept to implementation. To accomplish this goal, a good tool imposes structure.

Structure lays down a framework that helps simplify a problem. For example, the rules of a block diagram let us make many accurate assumptions about how a system operates. We can follow the flow of an algorithm much more easily than we can with other formats, such as code or equations. Structure also confines design, primarily to ensure that you do not violate any of the rules that you are voluntarily following. After all, if you violate a rule, then all of the assumptions based on that rule become invalid. Structure cuts design time by distilling complexity into simpler constructs and assists in testing the robustness of a design. Structure also organizes data so that it is easier to work with, changing its format to illuminate hitherto hidden patterns or enhance the meaning of the data. As I got used to the rules of a structure, such as a state machine, I would begin to think of a design in these terms, making natural the transition from ephemeral idea to captured design. Thus, the proof of concept becomes the template for implementation.

Before today’s GUI design tools became widely available, a good portion of design took place on napkins. An idea struck you, and you would sketch it. Then, you might flesh out the idea through a mix of flow charts and pseudocode. Next, you would determine which sections of the concept would be the most problematic and have the most influence on the overall system. Implementing these problem areas in a higher level language would then expose the feasibility of the overall concept. For example, I worked on one project in which I wanted to implement a more accurate algorithm than the one that was currently in use. However, I could not get the required accuracy with the new algorithm because the system’s numeric library offered too few significant digits. I could have
created new math functions to double the size of the numbers, but I decided that the effort was not worth the gain.

Because the proof of concept becomes the template for implementation, the structure that the development tool imposes greatly affects your final design. For example, flow diagrams restrict how you can lay down an idea. Complex structures create a maze of lines that become unreadable after a point, making the flow tool more of a hindrance than a help. Therefore, you must break down the concepts in the flow chart into simple components. In general, simple components lead to good design. You also reduce the number of variables you have to consider at one time by spreading them across many conceptual layers so that at each layer the problem becomes manageable and solvable. This methodology is commonly called “top-down” design and is one that Matlab silently imposes on the design process. Having layers of abstraction captures complexity. For example, I worked with one demo in which the top layer represented two modems communicating with each other. Moving down into the layers resulted in the further breakdown of each modem into more complex components, which were really simple components for that level. Standard process is to sketch out the entire system from a high level and drill down, defining and partitioning the system as you go.

Structure, however, does not necessarily lead to better design. When Pascal first appeared, its proponents touted how the lack of a GOTO statement created structure and, thus, better programs. As a result, these programs substituted one form of spaghetti code for another. For example, if you wanted to be able to drop out of a loop based on a certain error condition, you had to make that condition part of the loop. Thus, every iteration of the loop required an extra conditional instead of only checking for the conditional when it mattered and then GOTOing out of the loop.

At one point in a design, I faced the challenge of deciding how to implement error checking. Once an error condition arises, you must have a mechanism for addressing it. If signal integrity fails, for example, there are several ways to manage the error (Figure 1):

- The function that uncovered the integrity error can break out to an error-handling routine, resulting in termination of processing.
- The function can call an error-handling routine, resulting in recovery possible upon return.
- The function can try to handle the error itself, which results in high-knowledge handling of the error. However, error-handling as a whole becomes more complex because it is fragmented throughout the design.
- The function can set a flag and continue processing. This flag can cause a branch later in the processing chain, resulting in delayed handling of error and potential difficulty tracking error flow throughout the design.
- The flag can trigger error handling in a parallel state flow; that is, error occurs in process flow but is handled in parallel and distinct control flow. In this case, error handling would be delayed and become part of another flow.

Part of the challenge in handling an error flag in a state diagram is deciding whether to immediately address the error. In that case, each state must have transitions accounting for every error possible in that state or in a parallel state. This approach resembles a control loop running on the system, which does something only when the program sets a control flag. If a state addresses each possible error state, you may have to create substates to handle the varied number of transitions. Additionally, connecting similar errors from different states makes it more complex to resume process or control flow from where the error occurred. In this case, the trade-off is between efficient error control and recovery (handling the error where it occurs) and centralized error handling.

These examples demonstrate one lesson: Don’t immediately jump into a structured development environment. Imposing structure too soon on a budding idea can alter the course of design, driving
you in a particular direction before you consciously decide that you want to move in that direction. A paper napkin allows you complete freedom from restraint and structure. You can sketch an idea using whatever conceptual constructs make sense at the time. The pieces don't yet have to match up. Your design is flexible and can flow. In contrast, starting directly with an "electronic napkin," invites structure into the design, perhaps too soon. You limit yourself to the variety of structures and schemes the environment supports. If you're just trying to get an idea down, fighting your tool can distract your thinking.

One of the most devilish temptations of design work is to immediately jump into the hard—read fun—stuff: the core algorithm. After all, if the core of the idea proves unworkable, then why bother defining the rest of the system? I discovered why: The rest of the system can profoundly impact how to properly implement the algorithm.

For example, consider a simplified network with several nodes and a main processor (Figure 2). In this example, you must route a block of data coming to Node A to Node B. One obvious place to start design is on the routing algorithm, perhaps based on some protocol. But we haven't yet asked some important questions: Who will control the data, and where will that control reside? For example, each node could have its own intelligence and decide how to handle data passing through it (Figure 2a), or each node could pass all data to the main processor and follow its commands (Figure 2b). Even the data itself could contain the control information (Figure 2c). Depending upon how you implement the initial algorithm, you may find that you have locked out certain options (who can be slave, who can be master) in your rush to prove the concept.

Thus, structure can save you time by simplifying design. However, if you use structure rashly, it can actually burden or hamstring design when it comes time for the details. Environmental concerns notwithstanding, paper is cheap and easy to scrap. As soon as you begin to build block diagrams on your screen, you begin to make a possibly permanent investment in design. After I'd laid down 50 or so blocks, I felt that changing the course of my design would be more costly. Instead of rethinking the entire design—that is, crumpling up the latest napkin—I considered kluging around the initial design so that I wouldn't have to scrap all that work. In such a case, the algorithm constrains the system instead of the system constraining the algorithm. Either way is fine. You're just better off if you consciously make the choice rather than being surprised later. This is not to say that structure is bad; just be aware of the limitations involved with each medium of modeling. Each medium has its strengths (the patterns and ideas it exposes) and weaknesses (the patterns and ideas it inhibits).

**Getting down to details**

In the end, design is all in the details. A good algorithm-development tool allows you to focus on what you consider the key details. However, at some point, you must strip away all of the assumptions and structure to transform concept into implementation—to cross the implementation gap.

The Mathworks does not bill Matlab as an implementation tool. Rather, Matlab allows you to create models that closely simulate real-world systems. You have to build the bridge to cross the implementation gap. Matlab can help by creating C code from blocks or states; however, this stage is when you begin to see the effect of design on implementation.

For example, using state machines is an efficient way to conceptualize an idea. Graphically representing the states can make them easy to follow. However, looking at the generated code may boggle you, because the code must also include the mechanism overhead for maintaining and transitioning between states. If a process is fairly linear, using states may create complexity. For example, you can use flags to more easily maintain power-off, power-on, and sleep states. In this
case, the conceptual framework of states helps to clearly define the problem and process, but you may want to discard it when you actually implement the system.

Matlab has great strength in proving algorithms; however, I found that it does not provide as strong a connection to the real world as you might like. For example, after I prove an algorithm, I wonder how to determine what kind of processor I need to implement the design. Often, you can add more memory (to a point) to reduce processing power, or vice versa. Finding the "sweet spot" between these two important resources is a critical phase in reducing the cost of a signal-processing algorithm. A report generator shows how much memory Matlab required to perform the algorithm but does not tell you how many MIPS you're going to need. Matlab generates Pentium-executable C code. Although this code is ANSI-compatible, you have some work ahead of you to port the algorithm to your target processor and system. You can get some timing information by defining certain parameters during simulation, but, again, algorithm execution time is a function of processing power. In contrast, other vendors' development tools support specific DSP boards and can thus accurately profile timing and performance metrics.

Crossing the implementation gap can be a significant challenge. Now you have to write all of those I/O drivers you postponed during development and port Matlab's C code if you want to use it. Additionally, an efficient model for proving an algorithm may be costly in the added difficulty it takes to implement. Matlab lets you think about the parts of a problem in several ways,—like building a model out of paper, cardboard, wood, and plastic. Using different media (matrices, state machines, blocks, and code) to solve a problem is useful, but integrating them adds complexity to your final solution. Remember, Matlab is a simulation tool. It has to deal with more issues than you are concerned with in a final implementation. For example, it manages the interface between event-(Stateflow) and time- (Simulink) driven models, as well as between discrete (DSP Blockset) and continuous (Simulink) blocks. Matlab may quite effectively take care of these issues during simulation, but you may later find yourself having to understand these complex constructs yourself.

Finally, once you port an implementation to a target processor, you lose your connection to your models. If you want to make changes, you have to decide whether to change the models and then port the algorithm again (the tool remains a part of the maintenance and upgrade process) or abandon the development tool and work with your code from here on out (the tool is used for proof of concept only). In either case, you have to intimately understand the code that Matlab generates. Also understand that the more conceptual layers you employ, the more model overhead you create.

**Final analysis**

The implementation gap is still wide; a tool such as Matlab will neither help you with resource allocation, nor optimally code your algorithm. Rushing into design and haphazardly using the modeling media may reduce the time it takes to prove a design, but you may pay for this during implementation.

By all counts, though, a development tool like Matlab is invaluable in understanding a complex problem. By providing several media with which to paint your problem, you can look at your problem from several perspectives, which can lead to insights on how to solve it.

To give you an idea of the kind of design possible using Matlab as an algorithm and system-level development tool, I've included some program examples. Remember: Matlab is an open-ended framework that aims to help you reduce design risk. Make sure you drive the tool, not the other way around.
Program examples
The following links lead to two program examples that show how Matlab represents and organizes designs.
The first is V34stateflow, which simulates two v.34 modems communicating with each other. This example uses both Simulink and Stateflow models. Click here to download a zip file containing the files necessary to run the demo in Matlab. Or click here to view the HTML output of the Simulink Report Generator, which compiles system information about Simulink and Stateflow models, such as what the subsystems are, who calls each blocks, memory requirements, the calling string, the variables used in the block and their values, simulation parameters (step size, tolerances), and graphics of the models. The report is a "long version" of what you can access in the program files using the GUI interface.
The second example is the demo program "dsplpc," which you can load from the Matlab command line by typing "dsplpc". Click here to view the Report Generator output (HTML). Click here to download a zip file containing the C code created by the Real Time Workshop (Simulink models) and Stateflow Coder (Stateflow models).

Hard-earned lessons
Some of these lessons may seem obvious, but they're tempting to overlook in the rush to start producing a design.
Get intimate with your tool: Matlab, for example, is much, much more than an electronic cocktail napkin. If you just sit down and start using it, you may overlook some of its best features. For example, typing "Demo" at the command line brings up a long list of demonstration programs from which you can borrow.
Do it the old-fashioned way: Pencil and paper are still your friends. Spend some serious time with them before diving into a design environment. Trashing an initial idea is sometimes best. For some reason, it's much easier to discard an idea on a piece of paper than it is to scrap a block diagram you've been struggling with for a few hours.
Get down and dirty: Developing in a block-diagram environment can shield you from having to understand the intricacies of the blocks you use by having you work at a higher, less detailed abstraction layer (that is, it's a filter, not a complex polynomial with coefficients). However, if you don't understand the fundamentals of the blocks you use, you may be unable to discern which method of design and implementation is the best or most efficient at the chip and assembly level. For example, you might find the right filter by changing the number of taps until you get the results you need. However, if you understand filter theory, you can choose a different specialized filter, compensate for its peculiarities, and use fewer taps.
Don't prematurely simplify: Reducing the complexity of a problem sometimes merely postpones having to deal with it. Something that may help speed design may complicate implementation and end up increasing overall development time.
Use it again: Devote some thought to identifying and creating reusable components. Today, a single DSP may be able to handle multiple instantiations of your algorithm. Thus, the design unit is not a single signal algorithm running multiple times on a single DSP; a more efficient architecture supports a scalable number of instantiations. You probably want your algorithm to reflect this to optimize resource sharing and reduce cost.
I discovered many well-thought-out and useful features during my time using Matlab. But I also ran into some problems and discovered some idiosyncrasies, as with any new tool. To be fair, you can chalk up some of my difficulties to operator error, but others reveal that the Mathworks needs potentially to make some future enhancements to Matlab.

**Nice touches:**

- Well-organized libraries. You can easily add your own functions to the libraries instead of maintaining several distinct libraries.
- Clean, well-defined, and—for the most part—transparent integration between the tool boxes.
- The ability to link documentation files directly to blocks, which allows you to sparsely and cleanly document diagrams for day-to-day expert use. If you are unfamiliar with the design, you can easily consult the documentation.
- Background file saving, which keeps you working without delay and could save you from disaster.
- The ability to create limited graphical-user interfaces (switches, buttons, displays, and others) that ease simulating a system. You can also create a virtual prototype. This feature is useful for demonstrating proof of concept and garnering feedback from those who will want to make sure that you designed what they wanted you to design, which is not always the same as what they asked you to design.
- Support for creating document formats, such as Adobe Illustrator and Acrobat (PDF) files for printing out your design.
- A selection of demos, many of which can serve as building blocks.
- The Filter Realization Wizard, which eases the building of filters by simply asking you to plug in a few parameters.
- The report generator, which takes a snapshot of all diagrams, records the names of every block and subsystem, and charts memory usage. By taking daily reports, you can track changes, especially when many people are working on the same project.

**Stubbed toes, skinned knees, and sprained ankles:**

- Lack of a comprehensive tutorial. Although, the documentation for each tool box is fairly complete, a more comprehensive tutorial—complete with simple build-it-yourself projects to get you up to speed on some of the fundamental concepts—would be helpful. For example, I found it difficult to discern how the automatic transmission demo worked using parallel states until I had gained a strong understanding of Stateflow.
- The system can crash if you don't save before beginning a simulation. I occasionally forgot to save and paid the price when the system crashed.
- The lack of documentation for the demos (accessible by typing "Demo" at the Matlab command line). If you understand the problem in question, the diagrams make sense, but you often inherit projects that you do not initially understand.
- The lack of an "undo" option. This omission annoyed me, especially when I accidentally changed the characteristic of the wrong state. When your mind has already pressed onward, it's difficult to remember what it was you just changed.
- If you try to close or minimize a window and you accidentally close the main Matlab window, it doesn't ask whether you're sure you want to exit. Your entire session just ends.
- Error messages that lack clarity. For example, I failed to place the initial transition into a state diagram. When I tried to simulate the design, I met with the message, "Stateflow Runtime Error: failed assertion in sfc_hm.c at line 922: len >= objectProperties->size". An error message such as this one is sure to send you on a long goose chase the first time you encounter it.
- Matlab's assumption that you are an expert user. Although you can find the syntax rules in the extensive help files, access to just the formats or a "dummy mode" displaying the format would be helpful.
Author info

You can reach Technical Editor Nicholas Cravotta at 1-510-558-8906, fax 1-510-558-8914, e-mail cravotta@compuserve.com.

Acknowledgment

I would like to thank The Mathworks Inc (Natick, MA) for generously providing Matlab and extend special thanks to The Mathworks' Anne Mascarin for her valuable assistance with this project.