

Practical solutions for DSP system developers

Embedded Edge

March 2001

Software Tools Unleash
Powerful DSP Systems



SPRN132

Making Algorithms
eXpress-DSP
Compliant

Component-based
Programming for
DSP Algorithms

Embedded Edge

A Texas Instruments Publication

Volume 2 March 2001 Number 1

Stan Runyon
Editor-in-Chief
 testman2@ear.thlink.net

Mike Robinson
Managing Editor
 mrobinso@cmp.com

Tim Moran
Creative Director

Donna Moran
Art Director

Genevieve Joerger
Director, Custom Solutions
 gjoerger@cmp.com

Gregory Montgomery
Director of Sales
 gmontgom@cmp.com

Grace Adamo
Project Manager

Robert Steigleider
Ad Coordinator
 rsteigle@cmp.com

Susan Harper
Circulation Director
 sharper@cmp.com

Embedded Edge is published by Texas Instruments, Inc. and produced in cooperation with CMP Media Inc. Entire contents Copyright © 2001. The publication of information regarding any other company's products or services does not constitute Texas Instruments' approval, warranty or endorsement thereof. To subscribe on-line, visit: www.edn.com/customsolutions/edge/subscribe.ftml

Code Composer Studio, TMS320, TMS320C6000, C6000, TMS320C5000, C5000, TMS320C2000, C2000, DSP/BIOS and eXpressDSP are trademarks of Texas Instruments, Inc.

Inside This Issue

Insighter: The Power behind the Power 4
Blazing chip speeds might impress, but software is right there on center stage as well.

Breakpoints 6
News from the providers of embedded systems development products and services.

Cover: Software Tools Spark DSP Systems 8
Part 2 of our report on how software tools are answering the call to speed the development of DSP-based embedded systems.

Making Algorithms eXpressDSP-compliant 14
Existing software components can be quickly converted into eXpressDSP-compliant algorithms.

Component-based Programming for DSP 22
With TI's TMS320 DSP Algorithm Standard and the proper framework, you can use DSP algorithms without modifying the original source.

Backplane Links DSPs to Workstations, Hosts 28
A communication strategy replaces the emulator with an open-architecture backplane.

Launchings 32
New products and services for embedded systems developers.

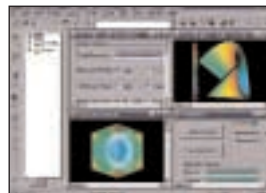
On the Edge 34
What do developers need from Web sites?



22



32



8

The Power behind the Power

But soft! what light through yonder window breaks?” OK, Romeo didn’t exactly have the latest technology breakthroughs in software development tools or DSPs in mind when he issued his famous tribute to Juliet back in the late 16th century. Nor did he answer his own question with, “It is the DSP chip, and object orientation is the sun!” as he may have if he were living today.

But perhaps the poets among us can appreciate that metaphor applied to the dazzling software advances lighting up the DSP stage. For that matter, who is to say that the star-crossed lovers wouldn’t appreciate them if they were around today?

Blazing chip speeds may impress, but software is right there on center stage as well, giving system developers the tools and standards needed to make sure that the chips deliver what they promise in performance and functionality. The latest tools don’t stop there, however. As part 2 of our report on advances in software development tools for DSP-based embedded systems shows, today’s tools are equipped to help users design multi-functional, multitasking, multiprocessing systems; integrate algorithms and real-time kernels; and compile listings for optimum speed or size.

For example, more and more developers are writing algorithms that comply with Texas Instruments’ TMS320 DSP Algorithm Standard, part of its eXpressDSP real-time software technology. The standard’s rules impose a level of excellence on third parties and assure the end user of consistency and compatibility, among other things.

Signals+Software is one company that was quick to recognize the benefits of complying with the standard. It’s churned out over 35 eXpressDSP-compliant algorithms—tested by TI for adherence to the standard—and found that the rules are simple to follow and don’t result in unwanted overhead. Read the contributed article inside for tips that help reduce conversion time and recommendations on how to prepare code for shipment to users.

The algorithm standard delivers another major benefit: it makes component-based programming for DSP algorithms a reality, as Spectrum Signal Processing describes in our next article. Spectrum Signal then shows how component-based programming is a signifi-

cant answer to the burgeoning complexity of application algorithms and the big jumps in system software/hardware ratios.

A system-level framework is required to encapsulate the components into applications, however. A rich framework allows developers to move entire algorithms—or parts of an algorithm—from one task or processor to another and reuse algorithmic building blocks in different algorithms. Importantly, there’s no need to modify—or even possess—the source code of the original algorithms or algorithmic building blocks, as there is with traditional programming techniques.

In the hardware arena, a backplane communication strategy to connect DSPs to workstations and hosts aids software development, debugging, and data transfer, in particular, for systems with multiple DSP boards—without using the ubiquitous emulator.

With an emulator, if all the DSPs are on a single board, the JTAG path slows down very slightly with each added processor, as the article from Pentek notes. For systems with multiple boards, all the boards must be daisy-chained externally, leading to extremely cumbersome cabling and possibly slowing performance further.

The backplane strategy, detailed in the article, exploits the open architectures of embedded systems, such as VMEbus or CompactPCI. Unlike the emulator, standard backplanes offer an ideal communication path between multiple DSP boards and between DSP boards and a workstation or a host.

—Stan Runyon
testman2@earthlink.net



Speech, Related Software Move to C54x DSPs

Speech Technology Center (St. Petersburg, Russia; www.speechpro.com) is porting high-quality, resource-conserving algorithms for speech recognition, speech enhancement, speech stretching, and noise and echo cancellation to the TMS320C54x DSP platform. The software will be eXpressDSP-compliant. Future versions will be developed for the C6000 processor.



DSP Packet Manager Prototypes Are Now Available

Tundra Semiconductor Corporation (Kanata, Ont.; www.tundra.com) is now shipping the Tundra Tsi920 DSP Packet Manager for prototyping. A fully functioning evaluation card is also available. The Tsi920 manages up to 32 DSPs, including Texas Instruments' C5000 platform, in a single board system. It increases system throughput and channel density by offloading the management of DSP traffic from a host processor.



Software Selected for In-flight Internet Access

Formation Inc. (Moorestown, N.J.; www.formation.com) has tapped Surf Communications Solutions Ltd. (Maynard, Mass.; www.surf-com.com) to supply multimodem software for Formation's in-flight digital communications and entertainment system. Among the software selected is Surf Multi-access Pool running V.90 high-speed modems to provide for Internet and e-mail services. The resulting modem cards are being built from reference designs using dual TMS320C6202 DSPs and Windows NT drivers.

BlueWave Board Picked for Network Security System

The Comstruct PCI/C6400 communications processing board from BlueWave Systems Inc. (Carrollton, Texas; www.bluews.com) has been selected by eNetSecure Inc. for its Model 2600 Telecommunications Intrusion Detection System (TIDS). The system detects unauthorized network access through computer-to-computer modems behind the network firewall by monitoring dial-up traffic. Optimized for high-density processing, the PCI/C6400 board carries up to four 200-MHz TMS320C6201 processors, with peak processing power pegged at 6,400 MIPS.



eXpressDSP Initiative Gains Support

Imagine Technology LLC (Lincoln, Neb.; www.imagnetechology.net) has committed to developing telephony, encryption, audio, and modem algorithms based on the TMS320 DSP Algorithm Standard, an eXpressDSP initiative. Six algorithms developed by Imagine have already passed compliance tests and more are planned. The algorithms are optimized C and assembly functions for the C5000 and C6000 DSP platforms.

System OK'ed to Keep Network Healthy

PolicyPoint, a QoS traffic and multiservice access platform from Natural Microsystems Corp. (Framingham, Mass.; www.nmss.com), has been certified by Concord Communications for compatibility with Concord's Network Health system. Network Health lets service providers and enterprises view network performance across applications, network services, and the network infrastructure. Information gathered by PolicyPoint is reported to Network Health and used by Concord's network managers to achieve predictable, consistent service-level management of IP-based services and converged IP applications.

SOFTWARE TOOLS UNLEASH POWERFUL DSP SYSTEMS

This is the second part of a two-part report on trends in DSP software development tools. Part 1 appeared in the October 2000 issue.

With DSPs jumping in performance and spreading throughout the embedded world, software tools are heeding the call to speed development.

By Stan Runyon



The latest crop of DSP chips may be dazzling system developers with blazing speeds, but the real heat is being generated by advances in development systems and support software. Not only are the latest tools letting users wring out the best performance from the chips, but the tools are comfortable with the latest trends in system design: multifunctionality, multitasking, multiprocessing, dual DSP and general-purpose processor chips, and more.

For instance, using both a DSP chip and a general-purpose processor (GPP) makes development even more interesting, but it sparks engineers to look for software capabilities aimed at that

dual arrangement. Consequently, DSP tools are becoming easier to use with GPPs, and some development environments offer the ability to mesh DSPs and GPPs more efficiently. Even so, both the DSP chips and development and embedded software are growing so powerful that in some applications no GPP is needed at all; the DSP alone takes care of the system functions.

However, noting the muscle of the latest crop of chips, engineers at Delphi Communications Systems in Maynard, Mass., are “trying to remove the microcontroller from our soft radio designs for picocellular reference applications, leaving just a powerful DSP chip,” says Rick Kane, vice president of business development. One way to tap that power is to write applications that comply with the TMS320 DSP Algorithm Standard. Kane notes that Delphi has one of the largest such algorithm inventories.

“The standard’s rules impose a standard of excellence on all third parties,” Kane says, “and our customers benefit from consistency and assurance that all will fit together. For example, rules ensure compatibility with C register conventions, reentrancy, relocatability, documentation of memory type and usage, and so on. That way, our users can allocate resources intelligently.”

“The rules impose an object-oriented design technique, and that’s good from our point of view in terms of helping our customers with integration,” Kane adds. (For more information on the algorithm standard, see page 14.)

Until recently, a typical DSP executed a single task or function, most likely running code in one giant loop. No longer. Today, many systems are designed to be

multifunctional or multitasking, calling for new kinds of development capabilities—perhaps an embedded real-time kernel or operating system.

KERNELS LINE UP FOR EMBEDDING

An embedded kernel can be thought of as one wavelet in a gathering of forces that threatens to become a tsunami in software design—that is, the construction of operating system software and applications from an off-the-shelf library of software components as well as in-house code.

Besides a kernel, new compiler and configuration technologies are bearing down on some common objections to generalized commercial software components. Those objections are the lack of application specificity, the hit to performance, and bloated code.

Imagine, however, techniques that allow a software library to adapt itself, remove unnecessary code, configure itself to the specific requirements of customers’ applications—even provide for debugging of a commercial black box. That’s not a dream but, increasingly, a reality.

Embedded Internet audio applications, such as MP3 players, the Sony Music Clip, and other appliances (actually, anything that is power-, space-, or cost-constrained)

are ripe for such capabilities.

Typically, a miniature audio device can contain a DSP chip that runs the main MP3 decoding algorithm but also has enough power left over to perform other system functions: the user interface, communications or other functions usually relegated to a microproces-

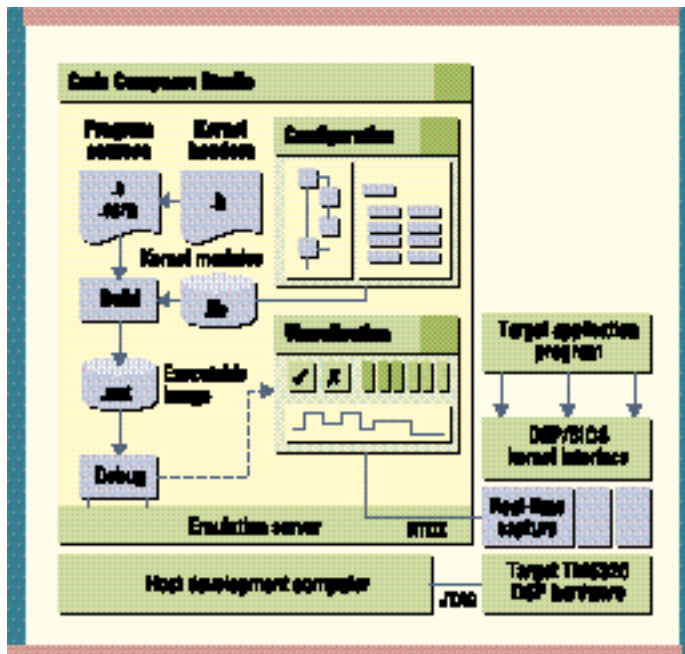


Figure 1. Real-time kernels have entered the DSP scene as a result of the growing number of tasks assigned to the target. For instance, application programs may engage simultaneously in digital signal processing and general-purpose system control functions; handle incoming events from multiple external sources; manage all kinds of ports, timers, DMAs, and host interfaces; move data; produce more than one data stream; execute algorithms at different frame rates—and even more. TI’s DSP/BIOS kernel essentially is a library of functions callable from C or assembly language. A palette of kernel modules can be included in a mix-and-match fashion within an application program.

sor or microcontroller. By eliminating the controller, designers save money and power; need only one set of tools instead of two; and can develop one code base, not two.

A standard kernel and advances in compiler technology as well as the natural ability to express functionality in a high-level language make such multitasking possible (Figure 1). Moreover, when such a kernel is highly integrated with the rest of the tooling, developers inherit some uncommon debugging and diagnostic capabilities—a welcome event considering that integration and debugging time have, in the words of one industry observer, “exploded to become by far the largest portion of any development.”

Take Blue Wave Systems, Inc. of Carrollton, Texas. A leading supplier of high-channel DSP subsystems used in telecom infrastructure equipment, such as VoIP gateways, digital wireless communications, and intelligent peripherals, it’s made Texas Instruments’ scalable, extensible kernel, DSP/BIOS II, the core infrastructure of its recently unveiled fax framework. Called ComStruct, the framework is actually a telephony development environment. The ComStruct voice- and fax-over-IP solution provides more than 120 channels of voice and fax relay or 120 channels of modem in an integrated deployment platform.

The escalating power of DSP hardware has led not only to the running of multiple algorithms, but also to a search for better compiler technology. One reason: the need to slash code size as applications continue to balloon in size and complexity.

“The embedded arena is one of the few places left where people care about the size of an executable program, because they pay for it to go into ROM,” says

The escalating power of DSP hardware has led to the running of multiple algorithms.

Keith Cooper, a professor and researcher at the Department of Computer Science at Rice University in Houston, Texas.

Compiler technology offers opportunities to reduce code size. For example, according to Cooper, many traditional compiler optimizations are designed to reduce the execution time of compiled code, but not necessarily the size of the compiled code.

Because much of the code for embedded systems is compiled once and then burned into ROM, the soft-

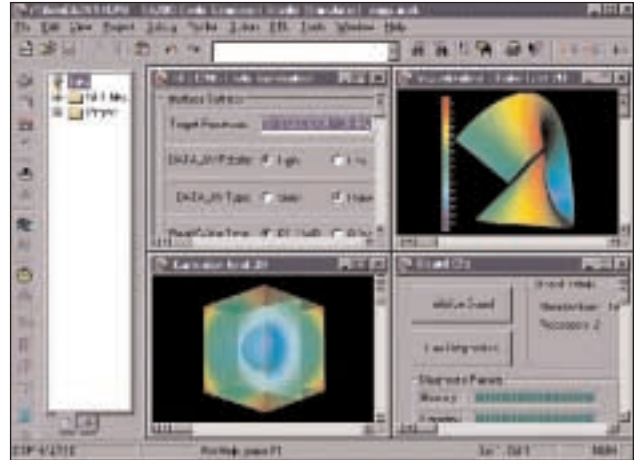


Figure 2. Users are asking for integrated development environments that provide an open, “pluggable” architecture. That is, third-party tool developers can produce plug-ins—software applications that use the environment’s application interface and are integrated with, customize, or extend the host and target development environment with additional, specialized functionality. Shown here are some typical plug-ins that can be integrated with and launched from Code Composer Studio: a TI data converter configuration and set-up tool (top left), a 3-D data visualization and analysis tool (top right and bottom left), and a configuration and diagnostics tool for a development board (bottom right).

ware designer will often tolerate much longer compilation times as a trade-off for smaller compiled code size.

Cooper’s approach is to take advantage of that by using a genetic algorithm, essentially a biased sampling search technique, to find optimization sequences that generate small blocks of object code, then compare the solutions to those found by using a fixed optimization sequence and by testing random optimization sequences.

Based on the results found by the genetic algorithm, the researchers develop a new fixed sequence to reduce code size. They also explore the idea of using different optimization sequences for different modules and functions of the same program.

SHORTENING THE WAIT

As that kind of work continues, commercial compilers have come down in some corners of the embedded world—that is, they are shaving lines of code from exe-

cutables. But perhaps just as important, they're cutting the lag time between new DSP architectures and compiler technology. Instead of the five-year wait typical in the mainstream commodity processor arena, C compilers for DSP development finally are beginning to get a jump on new architectural features, a boon for those who want to get to market fast with the latest hardware.

More accurately, the more enlightened semiconductor companies design advanced DSP functions with C coding in mind or design compilers and chips together. A case in point: the high-performance TI C6000 platform.

"We decided years ago to prepare for both a large leap in DSP performance and C efficiency," says Rich Scales, a manager in TI's Software Development Solutions Operation in Houston and an expert on compilers. That early insight may explain the reaction of Delphi's Kane when he says, "There's a lot more formalism and less seat-of-the-pants feel to DSP software over the last two years."

In fact, leading companies such as Cisco, Ericsson, Nokia, and 3Com—all of which want to reuse code in rather large diverse applications—are insisting on efficient C compilers. A profile-based compiler goes a long way toward satisfying that need.

Basically, such a compiler provides users with a

C compilers for DSP development are getting a jump on new architectural features.

graphical way of trading off code size for performance in their applications. The technique may be best applied in parallel processing, which can rocket in code size because of performance-boosting techniques such as aggressive unrolling and software pipelining.

In fact, in many applications, 80 percent of the con-

trol code falls into 20 percent of the nets. Instead of users' trying to optimize all control code, the profile-based compiler analyzes all of the code and then profiles, in a two-dimensional graph, the various options falling between the high-performance and small-code-size ranges. Somewhere along that graph is a knee representing a significant code size decrease with only a modest performance hit.

Some compilers go much further. Possessing the knowledge of the expert hand-coded assembly writer, they provide interactive tuning and feedback, using an understanding of an entire application to optimize key components.

Just around the corner are a host of other compiler assets that will allow users to measure, tune, and build applications based on a myriad of important parameters, such as cycle

counts, cache hits and misses, and memory locations.

SOFTWARE DEFINES THE RELATIONSHIP

Compilers, kernels, algorithm standards: How do these relate to one another? Each has become a crucial part of a contemporary integrated development environment—a central nervous system, if you will, not only for differentiating products. In other words, not all Internet audio devices, for example, are alike; software differentiates one from the other, and the development environment that launched the winner takes much of the credit.

Such an environment, that can produce winning "X generation" products and reuse diverse software components across multiple sites, multiple developers, and even multiple companies, goes beyond the usual lineup of individual tools: It calls for heavy integration into a cohesive continuum. It also calls for the ability to accept plug-ins, compliant tools from diverse sources that use the IDE's application interface and are integrated with, customize, or extend the host and target development environment with additional specialized functionality (Figure 2).



Figure 3. Besides plug-ins, the latest development environments help boost productivity with advanced features, such as real-time analysis and the ability to visualize activity without stopping a processor, as shown here.

Compilers, kernels, BIOS, algorithm standards, debuggers, emulators, linkers, editors—all, then, must dovetail into a cooperative framework that users can put to work in optimizing their code. For example, in TI's vision a compiler would be aware of, and ready to support, TMS320 DSP Algorithm Standard components or stand ready to let users juggle code density and MIPS.

Like the compiler, other traditional components of the framework are undergoing plastic surgery. For example, facelifts for linkers and other tools are moving decidedly toward the visual.

The move away from cumbersome text-based languages should be appreciated by those who use extensive memory system overlays, different run-time and load-time addressing or code location trials, and the like. With a visual linker, all of those can be accomplished graphically merely by clicking and dragging various memory components—programs or data—and putting them into a graphical view of the memory

space. Not only can users drag and drop components into multiple memory types, but they also obtain immediate visual feedback and can choose from a library of standard devices.

What of the future? Can DSP development software get any better? Of course it can. Users are asking for many things, including even tighter integration with the chip of choice, larger file sizes, better ways of visualizing data (Figure 3), improved editors and project managers, integration of favorite tools—and the list goes on.

Look for gains in all of those areas, and more. Think, for instance, about real-time analyses, especially enabled by the BIOS; configurable editors; expanding libraries to support chips, boards and peripherals.

Users of TI's eXpressDSP technology or Code Composer Studio also can expect a deluge of third-party plug-ins that will make their development life a lot easier in a variety of application areas, from specific communications functions to general-purpose data converters. ◆

Existing software components can be quickly converted into eXpressDSP-compliant algorithms.

Making Algorithms eXpressDSP-Compliant

By David Miller

Texas Instruments' TMS320 DSP Algorithm Standard, part of its eXpressDSP real-time software technology, is a set of rules and guidelines for algorithm writers that greatly reduces the repetitive and time-consuming system integration tasks associated with algorithms. It standardizes many aspects of algorithm development, ranging from packaging and naming conventions to interfaces, memory management, and documentation. In addition, TI runs an extensive compliance test program that verifies that algorithms comply with the standard.

Algorithm developers—whether OEMs or third-party software companies—benefit from having a single accepted standard to write to and support, enabling them to focus on developing new algorithms rather than customizing existing algorithms for individual customers. In turn, customers benefit from compliant algorithms that show increased application consistency and reduce the necessity for system reengineering. The modular, off-the-shelf environment simplifies integration, provides algorithm compatibility, and promotes algorithm reuse, as well as reducing software risk and development time,

with little code or programming penalties. All these benefits lead to faster integration, a big gain for any algorithm consumer.

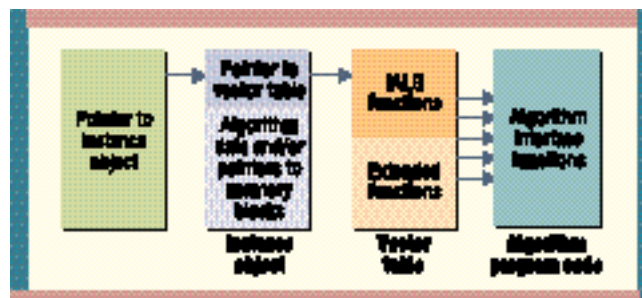
As an algorithm developer, you can quickly convert existing software components into eXpressDSP-

compliant algorithms. At Signals+Software, we've already produced over 35 eXpressDSP-compliant algorithms (tested by TI for compliance with the standard) on TI's C62x and C54x DSP generations. Our experience has shown that the rules are relatively simple to follow and don't result in unnecessary overhead at various points within the system. We've come up with tips and recommendations to reduce conversion time and ideas on how to prepare code for shipment to users. They are not intended to replace TI's existing documentation, nor are they exhaustive, but they do cover all the main points.

The algorithm standard specifies that algorithms be distributed in the

form of archives or libraries containing all the object files necessary for the particular algorithm. Algorithms must be re-entrant and multichannel, and each channel or instance can have any number of data memory blocks. The blocks are allocated at the system level (see the figure) and can be either static or dynamic.

The first of these blocks



Algorithms that comply with the TMS320 DSP Algorithm Standard use a memory structure with multiple levels of abstraction to provide standardization and flexibility.

Listing 1: Example IALG interface code

```

int IALG_ALLOW_allow(
    const IALG_Params *algParams,
    IALG_Params **pParam,
    IALG_Handle handle[])
{
    handle[0].size = sizeof(IALG_ALLOW_obj);
    handle[0].alignment = 0;
    handle[0].space = IALG_DEFAULT;
    handle[0].attr = IALG_FINALIZED;

    return 1; /* number of memory blocks */
}

int IALG_ALLOW_free(
    register IALG_Handle handle,
    register IALG_Handle handle[])
{
    handle[0].base = handle;

    return 0;
}

int IALG_ALLOW_initObj(
    IALG_Handle handle,
    const IALG_Handle handle[],
    IALG_Handle h,
    const IALG_Params *algParams)
{
    IALG_ALLOW_obj *inst = (IALG_ALLOW_obj *) handle;

    if (param==NULL) param = IALG_PARAMS;

    IALG_ALLOW_initialize(inst->algData, param->param1,
        param->param2);

    return IALG_OK;
}

void IALG_ALLOW_freeObj(
    IALG_Handle handle,
    IALG_Handle handle[],
    IALG_Handle h,
    const IALG_Params *algParams)
{
}

int IALG_ALLOW_createAllow(void)
{
    return 1;
}

```

is known as the instance object. Each instance of an algorithm is identified by a handle, which is a pointer to the instance object.

Interfaces to an algorithm are implemented using vector tables containing pointers to functions. Each interface is identified by the address of its vector table. Thus only a single label is needed to specify a particular algorithm and interface. The first entry in the instance object is always a pointer to the vector table, and once the instance has been created, the handle is all that's needed to uniquely identify that instance.

All algorithms must implement an interface called IALG, which contains basic functions relating to the creation and deletion of instances of the algorithm. In addition, an extended, algorithm-specific interface is needed, which must be derived from IALG.

We'll use an imaginary algorithm called "ZAP" as an example. ZAP has a straightforward interface with a persistent memory structure and initialization and processing functions:

```

zap.h
typedef struct
{
    /* ZAP persistent data */
    int param1;
    int param2;
    /* etc */

    /* note: it is assumed that no pointers to
    data within */
    /* the structure are contained in the structure */
} sZAP;
void InitializeZAP(sZAP *mem, int param1,
int param2);
void ProcessZAP(sZAP *mem, int *in, int
*out);

```

The process of making any algorithm eXpressDSP-compliant can be broken down into three main stages. A typical first step is to make the

Listing 2: Example extended interface code

```
KIRK_Decl MAP_WINDOW_extended(  
    IKIRK_Handle handle,  
    IKIRK_Col col,  
    IKIRK_Status *status  
)  
{  
    MAP_WINDOW_obj *inst = (MAP_WINDOW_obj *)handle;  
    KIRK_Decl retval = KIRK_FALSE;  
  
    switch (col)  
    {  
        case IKIRK_STATUS:  
  
            /* Copy status information from instance into status */  
  
            status->sgrownd = inst->algMem.pgrownd;  
            status->sgrownd = inst->algMem.pgrownd;  
  
            break;  
  
        case IKIRK_HANDLE:  
  
            /* Copy status information from status into instance */  
  
            inst->algMem.pgrownd = status->sgrownd;  
            inst->algMem.pgrownd = status->sgrownd;  
  
            break;  
  
        default:  
  
            retval = KIRK_FALSE;  
  
            break;  
    }  
  
    return retval;  
}  
  
void MAP_WINDOW_process(  
    IKIRK_Handle handle,  
    KIRK_Int16 in[],  
    KIRK_Int16 out[]  
)  
{  
    MAP_WINDOW_obj *inst = (MAP_WINDOW_obj *) handle;  
  
    MAP_WINDOW_process(&inst->algMem, in, out);  
}
```

existing algorithm source code comply with the algorithm standard's rules and conventions. When you've done that, you can add the necessary abstract interfaces. Although at this stage the algorithm is complete and you can produce the necessary documentation, it's normally necessary to also produce application interfaces and a test harness.

The existing algorithm will undoubtedly breach several of the algorithm standard's rules, and those breaches must be rectified. The amount of work involved here varies—a well-designed, re-entrant algorithm with no hardware accesses will generally need only minor changes, whereas other algorithms could require something approaching a complete redesign.

NAMING CONVENTIONS

The most common breach is that of rule 8, naming conventions. The rule states that all external definitions must be either API identifiers or have an API and vendor prefix. It ensures that no name space clashes can take place between different components and vendors. Although you can use the linker to "hide" noncompliant symbols, this is regarded as bad practice, and adding the appropriate prefix to all symbols is a straightforward exercise.

Another change you might need to make is to the way an algorithm handles its memory, which for a compliant algorithm is allocated outside of the algorithm. Therefore you must remove any memory allocation functions that are contained inside the algorithm.

A related issue is that of re-entrancy. Rule 2 states that all algorithms must be re-entrant within a preemptive environment (including time-sliced preemption). This requirement generally means that the use of global data memory (other

than for read-only tables) is to be avoided. If all persistent memory is contained in structures passed to the algorithm and all temporary memory is stack-based, the requirement will be met. Alternatively, for temporary memory, an algorithm can have scratch memory blocks that are handled in a similar manner to persistent ones.

The issue of hardware accesses has already been mentioned. Rule 6 states, “Algorithms must never directly access any peripheral device. This includes but is not limited to on-chip DMA controllers, timers, I/O devices, and cache control registers.” Indeed, it’s generally considered good practice not to access peripherals directly, but some algorithms may do so, in which case you’ll need to redesign them.

Finally, rule 1 requires algorithms to be callable from C. Most algorithms will comply with this requirement, but software that was designed to be called from assembly

These techniques originate from the object-oriented nature of the algorithm standard, and in particular the principle of encapsulation (the hiding of internal data from the user). There are two common areas where you may need to make changes.

The first is interface buffers. The inputs and outputs to eXpressDSP algorithms are normally done via user-defined data buffers, pointers to which are passed to the processing functions. If an existing algorithm has internal interface buffers, you’ll need to modify it.

USE THE CONTROL FUNCTION

Secondly, algorithm control and status functionality may need to be changed. It’s good practice not to permit the user to access internal data directly, but instead to use a control function that enables status information to be read or written.

```
IALG_Fxns fxns;  
sZAP algMem  
} ZAP_SIGSOFT_Obj; /* ZAP  
instance object */
```

As a result, the algorithm will require only a single memory block.

IMPLEMENTING THE IALG INTERFACE

Now, you can add the mandatory IALG interface. For a simple algorithm, only three compulsory functions need to be implemented, but it’s also good practice to implement two of the optional ones:

- `algAlloc`—this function returns a table of memory blocks required by the algorithm. ZAP has one block whose size is that of the `ZAP_SIGSOFT_Obj` structure given above.
- `algInitObj`—this function initializes the algorithm. Note that it simply “wraps” the existing initialization function. In eXpressDSP, initialization parameters are contained in

It’s good practice not to permit the user to access internal data directly, but instead to use a control function that enables status information to be read or written

code (typically older products) will need C interface functions added. The rule doesn’t mean that, for example, internal assembly language functions with their own calling convention can’t be used or that algorithms can’t change status registers to different values than those used by the C compiler, but it does require that the C environment be properly restored before the algorithm exits.

In addition to breaches of the rules, several standard techniques are used in the algorithm standard that may require changes to be made to an existing algorithm.

This technique is very flexible and can be adapted to almost any application. You may, however, need to make minor changes to an existing algorithm so that eXpressDSP control functions can be written.

After making all the necessary changes to the algorithm, you must consider data memory issues before you can add the abstract interfaces. For an algorithm with simple memory requirements, like ZAP, you can incorporate the algorithm’s memory into the instance object:

```
typedef struct  
{
```

a structure, a pointer to which is passed to this function. If the pointer is null, default parameters are used.

- `algFree`—this function is called when an algorithm is no longer required. It returns a table of memory spaces that can be deleted.
- `algMoved`—this optional function is called if the system has to move an algorithm’s data memory. In our case, no action is required and the function can just be empty (omitting the function indicates that the memory can’t be moved). If an algorithm uses more than one memory block or a memory block contains

pointers to internal locations (or both), the `algMoved` function must change or “fix up” the pointers to take the moved memory into account.

- `algNumAlloc`—this optional function returns the maximum number of memory blocks that the algorithm requires. In our case, it simply needs to return a value of one. If it's omitted, the system assumes that the algorithm needs a maximum of four memory blocks.

Listing 1 gives example code for the IALG interface.

In addition to IALG, it's necessary to implement an extended, or module-specific, interface. In the case of the ZAP algorithm, we define two additional functions for control and processing: `control` and `process`.

The two initialization parameters may need to be changed at run time, and that's made possible by defining a status structure. The user can then read the parameters into the status structure or write them from the status structure by passing appropriate commands to a control function. The actual processing of data is done by a process function, which merely wraps the existing algorithm's process function.

Example code for the extended interface is shown in Listing 2.

DOCUMENTING THE CODE

The next important step in developing an eXpressDSP-compliant component is documentation. Rules 19 through 24 require you to characterize and document various algorithm performance information. This information is extremely important to users of algorithms, but it's not always readily available without considerable effort. Note that the characterization requirements apply to the algorithm itself, as called through the vector table, and should not include any over-

head from any optional application interface (as discussed below).

The first documentation requirement is for heap memory. The information required is that passed to the system by the `algAlloc` function. It is easy to obtain and is mostly available immediately. The most common exception is when the `sizeof(xxx)` function is used (as in the example code in Listing 1), but as this function is evaluated during compilation, the quickest way to obtain the result is to look in the assembler file produced by the C compiler. For more complex algorithms, memory sizes may be related to initialization parameters, and the appropriate formulas should be supplied.

The next requirement is for algorithms to characterize their worst-case stack memory requirements. There are two basic methods of doing that. You can calculate the size by hand by adding up the stack usage of the worst-case path through

the algorithm. In the case of a complex algorithm with many levels of subroutines, the calculation can be extremely difficult and time-consuming. An alternative method is to fill the memory area used by the stack with a known pattern. You can then determine the amount of stack usage. This method can also be automated so that no manual intervention is required.

Thirdly, you must document static memory. For the algorithm standard, static memory means memory that is allocated at link time and is typically used for data tables. The use of static memory for other uses is discouraged. Rule 21 requires algorithms to characterize their static memory requirements. You can obtain the necessary information by linking the algorithm into a test harness and looking at the map file.

A similar requirement is for the program memory sizes. Again, you can obtain that information easily

Listing 2: Normal test harness

```

void main()
{
    int    InBuf[100];
    int    OutBuf[100];
    int    FrameCount;
    FILE   *f_in;
    FILE   *f_out;
    algm   AlgMem; /* algorithm memory */

    InitializeAlg(&AlgMem, 10, 30); /* initialize algorithm */

    while( !feof(f_in) || !feof(f_out) )
    {
        ProcessData(&AlgMem, in, out); /* process data */

        WriteOut(OutBuf, sizeof(InBuf), 100, f_out);
    }

    fflush(f_in);
    fflush(f_out);
}

```

Compliant Algorithms

by linking the algorithm into a test harness and looking at the map file. It is often easiest to create a specific program memory output section just for the algorithm; the linker will then add up all the necessary object modules and produce a total figure.

INTERRUPT LATENCY

The final two requirements relate to the real-time operation of the algorithm. Rule 23 requires algorithms to characterize their interrupt latency. The techniques used here vary according to the processor. On the C5000, there is normally no need for an algorithm to disable interrupts, and consequently the latency is simply the longest time that inter-

rupts are blocked by the noninterruptible RPT instruction. A manual investigation is usually needed to find the longest RPT loop. In cases where that's data-dependent, you should calculate the theoretical maximum.

On the C6000, interrupts will cause software-pipelined code to malfunction, and therefore they must be disabled around such code. If the software is written in C, you can have the compiler automatically do it by using the `-mi<n>` option, and `<n>` can be used as the documented interrupt latency. For assembly code, interrupts should be disabled and restored by means of calls to the appropriate DSP/BIOS routines. It's usually very time-consuming to calculate the latency manually in the

latter case; an easier method is to use the on-chip timer to calculate the number of cycles between the interrupt disabling and restoring function calls.

The last characterization requirement is rule 24, which requires algorithms to characterize their typical period and worst-case execution time for each operation (that is, the extended interface function). The period is frequently either a fixed property of the algorithm being implemented or dependent on initialization parameters. For control functions, typical periods generally can't be given. The worst-case execution time is similar to the stack size and can be difficult, if not impossible, to calculate theoretically for complex algo-

rithms. In those cases, you should measure or profile it using appropriate input data.

APPLICATION INTERFACE

To simplify the calling of compliant algorithms, an application, or “concrete,” interface is usually implemented that gives the algorithm a similar interface to a non-eXpressDSP application. This interface is entirely optional and doesn’t form part of the algorithm itself. The application interface consists of functions to create and delete algorithm instances and to process data and read/write control information. TI provides some generic functions that assist the writing of application interfaces, but they may need customizing to suit particular systems:

- create—this function, which can directly map to the TI `ALG_create` function, calls the various IALG and memory allocation functions in order to create an algorithm instance.
- delete—this function, which can directly map to the TI `ALG_delete` function, calls the `algFree` IALG function and frees all the relevant memory.
- apply—this function applies the algorithm to a frame of data. In simple algorithms, like ZAP, it simply needs to map to or call the process function, but for more complex algorithms it may also need to call the `algActivate` and `algDeactivate` functions.
- control—this function can directly map to the control function described above.

The algorithm is now complete, but it’s usually necessary to write a test harness, first so that the operation of the algorithm can be verified and second so that customers can run the code and gain a degree of confidence that it works as intended. A typical test harness will process data files in non-real time. The use of the standard C file I/O

Listing 4: eXpressDSP test harness

```

void main()
{
    XDMG_Int16 InBuf[100],
    XDMG_Int16 OutBuf[100],
    XDMG_Int16 FrameCount;
    FILE *f_in;
    FILE *f_out;
    XEP_Params XepParams; /* parameter structure */
    XEP_Handle XepHandle; /* algorithm handle */

    f_in = fopen(INP_FILE, "rb");
    f_out = fopen(OUT_FILE, "wb");

    XepParams.param1 = 10; /* place parameters in structure */
    XepParams.param2 = 10;

    XepHandle = XEP_create(XEP_ALGORITHM_NAME, &XepParams);
    /* create and initialize algorithm */

    while( !feof(f_in) && fread(XDMG_Int16, 100, f_in) == 100 )
    {
        XEP_apply(XepHandle, in, out); /* process data */

        fwrite(out, sizeof(XDMG_Int16), 100, f_out);
    }

    XEP_delete(XepHandle);

    fclose(f_in);
    fclose(f_out);
}

```

functions will enable files on the host PC to be passed through the DSP code via the JTAG interface when the software is run under Code Composer Studio.

As well as the test harness, all necessary files needed to integrate and test the algorithm should be supplied. It’s logical to include a Code Composer Studio project file to simplify the build process.

A test harness for the non-eXpressDSP-compliant algorithm, in Listing 3, needs very little change for the eXpressDSP-compliant version, in Listing 4. The main differences

are the memory allocation and the initialization code. The application interface conceals the internal workings of the eXpressDSP-compliant algorithm from the user. Thus all the benefits are gained without requiring major changes to the high-level software. ◆

David Miller (David.Miller@signalsandsoftware.com) is the TI Product Manager at Signals+Software Ltd. in Harrow, Middlesex, U.K. He has overseen the development of a wide range of eXpressDSP-compliant telecommunications software algorithms.

With TI's algorithm standard and the proper framework, you can use DSP algorithms without modifying the original source.

Component-based Programming Comes to DSP Algorithms

By Arda Erol

The DSP industry is similar to the general-purpose processor industry. Every year new DSPs significantly outdo older models in speed, power consumption, and price. As a result, these processors are used in areas once considered out of reach. Although faster clock speeds relieve the signal-processing software developer from hand-optimizing code, the use of more complex algorithms and the higher ratio of software versus hardware in many applications result in more complex software. However, complex software is nothing new for the mainstream software industry, which has embraced component-based programming techniques such as COM (Component Object Model), DCOM (Distributed Component Object Model), and CORBA (Common Object Request Broker Architecture).

Although these models don't target real-time applications, component-based programming techniques can be used at many levels of complexity, allowing you to easily bal-

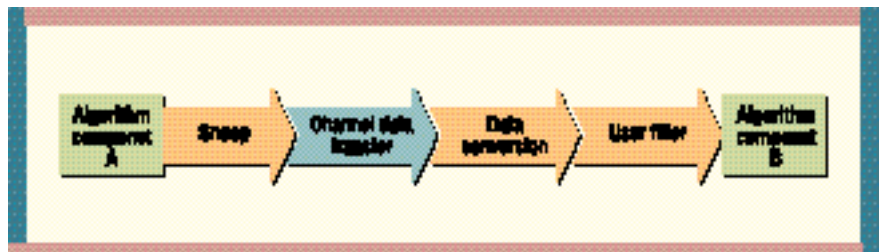


Figure 1. Component-based communication channels enable additional operations to be inserted in the data path, allowing advanced features such as real-time monitoring of data and data format conversions in heterogeneous systems.

ance modularity (which comes with a whole set of advantages) and performance. Despite widespread belief in the DSP industry to the contrary, component-based methods can be easily applied in C and assembly language without an object-oriented language, such as C++.

A RICH FRAMEWORK

With the advent of Texas Instruments' TMS320 DSP Algorithm Standard, component-based programming is feasible at the algorithm level. The standard lets you easily

use DSP algorithms from other companies in your own applications. You also need a system-level framework to encapsulate those components in applications. A powerful framework becomes more important when an application requires multiple DSPs or a mix of DSPs and general-purpose processors.

A rich framework enables you to move entire algorithms or even parts of an algorithm from one task or processor to another and reuse algorithmic building blocks in different algorithms (see "Bringing It All Together," page 24). More impor-

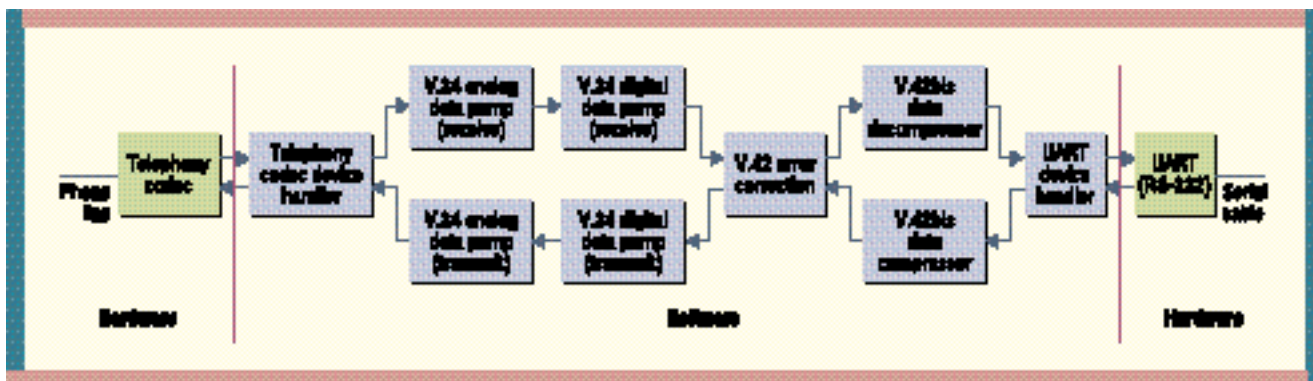


Figure 2. Like most DSP applications, a V.34 modem can be implemented as a collection of independent software components, even hardware device handlers. You can do that even with hardware device handlers, which usually represent I/O points of an application, like the telephony codec device handler and the UART device handler shown here.

tantly, you can do that without modifying—or even possessing—the source code of the original algorithms or algorithmic building blocks.

If you use traditional programming techniques, modifying the source code is unavoidable: Functions must be organized into separate builds to run on different processors, and the necessary communications must be added between the separated modules. Even then, every time you move part of

decreases, the interchangeability of components decreases and the system ceases to be scalable.

HEAVYWEIGHT AND LIGHTWEIGHT

Algorithms can be characterized as either heavyweight or lightweight. A heavyweight algorithm is a complete DSP algorithm such as a G.723 vocoder, an MPEG-2 video encoder, or a V.34 data pump. Lightweight algorithms, or algorithmic

source code, and easier system-level debugging.

With this approach, the DSP industry can define standard interfaces for different categories of algorithms. TI's TMS320 DSP Algorithm Standard, part of eXpressDSP, is an excellent example. Establishing such standards is key to enabling developers to interchange different implementations of an algorithm from different vendors without substantial effort.

At a lower level, algorithms can be

Component granularity plays an important role in finding the correct balance between meeting run-time requirements and maintaining modular programming.

an algorithm from one processor to another, you have to repeat the same tedious process.

Component granularity plays an important role in finding the correct balance between meeting run-time requirements and maintaining modular programming. As the granularity of components increases, so does the chance to incur unnecessary overhead in an application. On the other hand, as the granularity

mic building blocks, are smaller DSP functions, such as a finite impulse response filter or a discrete cosine transform.

At the highest level of component granularity, each heavyweight algorithm is bundled into a distinct component. The advantages include distribution of algorithms in binary form, deployment of complete algorithms to different tasks and processors without modifying algorithm

split into smaller building blocks, such as filters, echo cancellers, and fast Fourier transforms. The finer granularity has additional benefits: deployment of a single heavyweight algorithm over multiple processors without modifying (or, again, even possessing) the algorithm source code, block-diagram-based design of DSP algorithms, reuse of algorithmic building blocks, and easier algorithm-level debugging.

BRINGING IT ALL TOGETHER

Spectrum Signal Processing is currently developing a component-based application framework called Accelera. Accelera allows you to distribute components from various sources to multiple processors. On each processor, you can group components into one or more threads or tasks. At run time, you can create snoop connections to monitor intercomponent communications without stopping the application code. Overall, Accelera supports binary algorithm code distribution; code reusability;

with two target processors and one development (host) processor. The solid circles show algorithm components (lightweight or heavyweight), and the dashed circles show threads and tasks. Components can be implemented as collections of other components, as in the case of c5.

Once you've implemented the lowest-level components (in C or assembly) or imported them (for example, eXpressDSP-compliant algorithms or Simulink-generated code), you can bring

them into the Accelera block-diagram design tool and combine them to form more complex components or a complete application to be deployed on one or more target processors. The set of processors can be a combination of DSPs, microcontrollers, and general-purpose processors. When an application is executed on the target processors, the block-diagram design tool turns into a real-time debugging tool that lets you to snoop on communications between components.

If you notice load-balancing problems, you can move components from processor to processor without touching the components' source code. For embedded solutions, you can detach the host system from the target processors at the end of the development cycle by encoding the default configuration of all components into the DSP software.

ty; integration with DSP algorithm design tools, such as Simulink; support for algorithm standards, such as eXpressDSP; partitioning of algorithms to heterogeneous multiprocessor systems, including hosts; and advanced real-time debugging support with the ability to monitor and modify intercomponent communications.

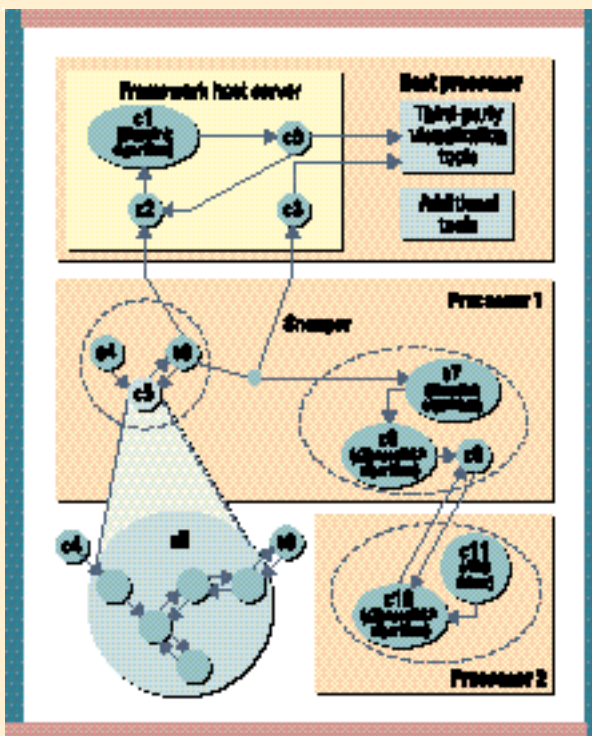
The figure shows Accelera in a system

Several approaches to component-based programming at the algorithmic building-block level have been tried. One approach—defining each component as a thread or task—is taken by a number of commercially available development environments. The one shortcoming is insufficient component granularity. Those systems don't prevent you from implementing even small algorithmic building blocks as separate components; because of the context switch overhead between components, however, you must minimize the number of algorithm components to maximize performance.

The system integrator can introduce a custom implementation.

Hence you must consider the capabilities and number of processors to efficiently split the algorithm into components. This approach greatly increases the dependency on a specific target platform.

A more advanced approach to component-based software development involves defining algorithm components to be independent of operating system entities, such as threads and tasks. With this approach, you group multiple components inside a single thread or task and schedule the components in a static order or a dynamic order programmed by the user. That eliminates context switching when it's not a requirement of the application. In addition, since components within a thread are inherently synchronized, the need to synchronize communications between components is also eliminated. Con-



sequently, you have very efficient data-passing techniques, such as passing pointers to data, rather than transferring actual data.

Component-based programming can be extended to the implementation of communication libraries that provide the means of exchanging data between algorithm components. With such libraries, the system integrator can choose the optimal communications implementation based on the communication needs of the algorithms used and their physical locations. Moreover, when faced with unsupported communications hardware, the system integrator can introduce a custom implementation, which can then be

Furthermore, interaction between components can even be tampered with, which can be turned into a very powerful debugging tool (Figure 1).

One important advantage of component-based design at the algorithmic building-block level is its ability to provide a natural integration path for DSP development tools, like MathWorks' MATLAB and Simulink. Following the thread-independent component approach, you can convert even individual Simulink blocks into components without incurring much overhead. That way, once the application is deployed on a target platform, only frequently called

ing the processor is fast enough to handle a multichannel modem implementation.

Moreover, the connections between the components are easy to modify for different tasks. For example, you can add an additional component before the data compressor and decompressor blocks to multiplex data to multiple instances of the error correction and data pump components to send data over multiple phone lines. Best of all, that can be done without access to the existing components' source code.

EASE OF DEVELOPMENT

If the component model in use is also supported on a host processor, it becomes easier to develop each component independently. For example, the programmer of the data compressor can connect the input and output of the component to a host processor that reads from a file and another one that writes to a file. The programmer can then test the data compressor on actual hardware before the other components are completed by other programmers.

Similarly, the programmer of the error correction component can create two instances of it and connect them to each other for debugging and then stream test data from the host PC via host-to-DSP channels. During system integration, the data pump blocks and the compressor and decompressor blocks can be grouped into two threads, thereby avoiding task switching and synchronization overhead among components within each thread. ♦

Arda Erol (arda_erol@spectrumsignal.com) leads a software group at Spectrum Signal Processing, Inc. in Burnaby, B.C., that develops future software products. His expertise includes DSP software development, real-time operating systems, and video/image processing.

The very same component-based approach can be extended to the entire set of operating system entities.

used by all existing algorithm components.

Similarly, there's no reason to stop at the communications layer. The very same component-based approach can be extended to the entire set of operating system entities, including heaps and threads, to provide the same level of flexibility throughout the system. For example, by letting the user create multiple heaps in different memory regions, where each memory region may have distinct physical properties, a component-based heap implementation solves the thorny problem of having more than one physical type of directly accessible memory.

A well-designed component model can also provide a means of monitoring interactions between components. The user gains insight into any part of the system at run time without placing debug statements inside the application code.

algorithm components need to be ported to assembly language (or reimplemented by hand in C), and you don't have to worry about other components of the application. Even if all components are eventually reimplemented by hand, using imported blocks from DSP algorithm design tools makes development a lot easier, as it provides a functional system much earlier in the development cycle.

A REAL-LIFE EXAMPLE

V.34 modem software, for example, can be implemented as a collection of individual components (Figure 2), and each component can consist of yet another set of subcomponents. The components can be easily distributed to multiple processors, or multiple instances of the same collection of components can be executed on a single processor, assum-

A communication strategy replaces the emulator with an open-architecture backplane.

Backplane Links DSPs to Workstations, Hosts

By Rodger H. Hosking

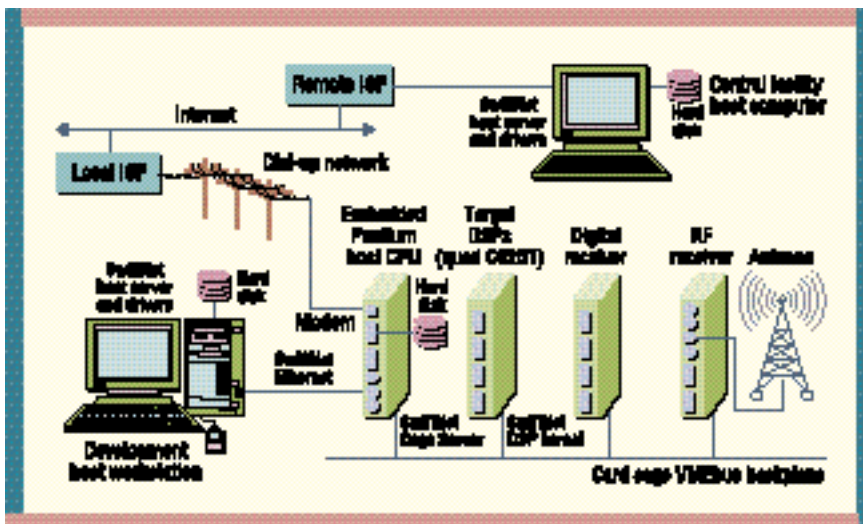


Figure 1. This remote software radio data acquisition system analyzes and stores data locally and relays it once a day across the Internet to a central computer.

Engineers developing DSP-based embedded systems are challenged to bridge the gap between the DSP devices and more conventional peripherals, such as workstations, to support the critical tasks of software development, debugging, and data transfers.

A common technique for connecting the embedded DSP to the workstation is to use an emulator. Virtually all of Texas Instruments' DSPs are equipped with a serial

interface that complies with the IEEE JTAG specification, which allows access to the processor for debugging purposes. A JTAG emulator board is installed in the host computer and connected to the DSP over a serial cable with a standard multipin connector that matches the JTAG signal pins on the target DSP board.

Although the emulator approach may provide a reasonable solution for a single-processor system, it's

not satisfactory for multiprocessor systems, as the emulator lines from all the DSPs must be connected in a daisy-chain fashion. If all the processors are on a single board, the lines can be connected as part of the board design, although the JTAG path slows down very slightly with each added DSP. For systems with multiple boards, all the boards must be daisy-chained externally. That leads to extremely cumbersome cabling and can additionally affect performance.

For open-architecture, embedded systems like those based on the VMEbus or CompactPCI, the standard backplane offers an ideal communication path between boards. Further, these systems are often equipped with one or more system controller CPU boards with Ethernet interfaces to support communications to the workstation and between card cages.

SwiftNet was developed to solve the connectivity problems, not only for single-processor systems, but particularly for distributed multiprocessor systems spanning several boards or deployed in different locations. By taking advantage of the resources of open-architecture, embedded systems to solve the DSP

connection problem, SwiftNet eliminates the awkward JTAG cables and the need for daisy-chaining.

SwiftNet can save weeks of writing custom code and drivers to establish the types of connections that are typical of embedded systems. Furthermore, the DSP and the workstation can assume the different roles of client and server to meet system needs for both development and run-time applications.

SwiftNet fits extremely well within TI's new eXpressDSP software paradigms because it effectively replaces the low-level JTAG primitives. In this way, Code Composer Studio, RTDX (Real-Time Data Exchange), and the DSP/BIOS software components from TI all can ride on top of the SwiftNet backbone.

WHAT IS SWIFNET?

SwiftNet is actually a large collection of drivers, libraries, and utilities that support software development and run-time connectivity between workstations and any number of DSPs. Every DSP development system consists of a workstation, a DSP, and some type of electrical connection

The DSP and the workstation can assume the different roles of client and server to meet system needs

between them. To support dozens of possible combinations among the three elements, SwiftNet drivers are modular and based on the popular TCP/IP communication protocols.

To properly steer messages, SwiftNet tags each element in the system with a unique "handle" that specifies the workstation, the embedded CPU board, the DSP board name, and the processor number on the board. Each SwiftNet

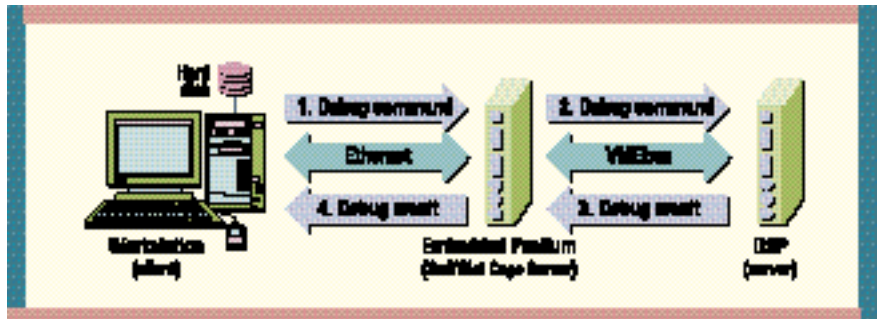


Figure 2. Debugging across the backplane is supported by SwiftNet client requests from the workstation and server responses from the DSP target.

signal is a TCP/IP packet that consists of the header with the SwiftNet handle followed by a variable-length data payload. The signal allows an unlimited number of DSPs to be accessed by either a distributed work group or a single user.

SwiftNet software includes both client and server functions that generate and respond to TCP/IP SwiftNet signals. Since SwiftNet signals are uniform across all hosts, any combination of workstations can be connected to the network and participate successfully in SwiftNet messaging to any DSP target, regardless of the operating system

Each DSP runs a small kernel that

view. This symmetry allows the same set of functions to be used across all SwiftNet platforms.

To see how you can use SwiftNet for development and run-time applications, let's look at an intelligent digital receiver data acquisition system (Figure 1). Deployed for detecting and archiving specific types of radio traffic in sparsely populated areas, several systems are connected over the Internet to a central monitoring facility.

Each system consists of an RF front end, a multichannel digital receiver module, a quad TMS320-C6701 DSP processor, and an embedded Pentium host processor equipped with a hard disk drive and modem. All the components are VMEbus boards housed in a small chassis. The DSP application issues coarse tuning commands to the RF front end and then sweeps the digital receiver section within each band, searching for radio signal energy much like a scanner.

When a signal of interest is detected, the narrowband digital receivers lock on to it, sending the down-converted receiver output to the DSPs. Here, the baseband signals are classified by modulation type by analyzing the frequency and time characteristics. Further DSP processing completes the demodulation and then determines if the information is of interest.

Linking DSPs to Workstations

If a signal needs to be stored, it is sent to the embedded Pentium host for archiving on the hard disk along with the time and date, ready for periodic uploading to the central facility.

PROGRAM DEVELOPMENT

For development, SwiftNet TCP/IP drivers provide an Ethernet link from the embedded host to an external Pentium workstation running Code Composer Studio (CCS) for debugging, editing, compiling, and profiling the entire application. The workstation launches the SwiftNet Debug Manager application for Windows, which allows each DSP on the target boards to be accessed by the software development tools.

The embedded Pentium computer board in the VME card cage uses the SwiftNet Cage Server application to link SwiftNet messages between CCS running on the workstation and each DSP within the card cage (Figure 2). In this case, as each CCS command is generated during debugging, the SwiftNet client function on the workstation generates a SwiftNet command that is sent out through the Ethernet link to the embedded Pentium CPU board in the card cage.

Here the SwiftNet Cage Server function on the embedded Pentium translates the message into reads and writes across the VMEbus backplane to the target DSP device. The commands typically require a value or status from the DSP that's passed across the VMEbus to the embedded Pentium. The return value is then sent to the workstation as a SwiftNet message over Ethernet to complete the debugging transaction.

STANDARD I/O FOR DSPs

Many programmers find it extremely helpful to embed temporary `printf()`

commands at strategic points in DSP programs to track program execution and status. To receive the messages, the workstation must first be set up to run the SNIOWorkstation Server function by invoking the command:

```
snio recvr_cage board_1 DSP_3
```

In this example, the embedded Pentium host computer name in the VME card cage is `recvr_cage`, the processor board name is `board_1`, and the target processor is `DSP_3`.

In the DSP program, you set up the SwiftNet Standard I/O library functions (`snstdio`) by including the header file at the beginning of the DSP program:

```
#include <snstdio.h>
```

The standard I/O connection to the workstation (in this example, named `ws1`) is then opened up by issuing:

```
snstdio_init(ws1);
```

The programmer can now send a message to the workstation from any point in the program by simply inserting a `printf()` instruction like this one:

```
printf("Now Entering Demod Function\n", demod_num);
```

When the DSP program is executed, the message conveniently appears in a text window on the workstation.

To more easily evaluate the content and format of the files that the DSP application will eventually send to the embedded host disk drive, send these files to the workstation file system during the development phase.

Since we've already set up the standard I/O Ethernet connection to

the workstation for the `printf()` functions, we can take advantage of the SNIOWorkstation file functions. First we define a file pointer, and then we open or create a file on the workstation for writing, as follows:

```
FILE *fp;  
fp = fopen("data_file", "w+");
```

Data from a memory buffer of a specified size on the DSP board can be easily written to the workstation file and then closed by executing:

```
fwrite(fp, buffer, size);  
fclose(fp);
```

Once the file is closed, it can be opened by any application on the workstation for either analysis or inspection.

The DSP program is successfully creating and writing files to the workstation file system, so we can begin to wean the workstation away from the system and replace it with the embedded host. The embedded host must accept and store data files from the DSP subsystem during the day in preparation for a nightly file transfer to the central facility computer.

The SwiftNet Cage Server function, previously loaded on the embedded host at start-up to support the workstation Ethernet connection, now also supports local file system access by the DSP. In the C program for the DSP, all we need to do is change the server name in the DSP `snstdio` initialization function from `ws1` (the workstation) to the name of the embedded host, `recvr_cag`:

```
snstdio_init(recvr_cage);
```

By changing just this one argument in the DSP program, we've redirected all SwiftNet messaging and file I/O to the embedded Pentium. The change allows the DSP

Linking DSPs to Workstations

board to create and write output files directly to the embedded host file system for archiving.

Just like the development workstation, the central facility computer needs to run the SNIOSwiftNet Server application to serve system resources to the DSP system:

```
snio recvr_cage board_1 DSP_3
```

Note that the remote host can be running under any operating system supported by SwiftNet, including Solaris, Digital Unix, and HP-UX, as well as Windows.

At the receiver system, to establish the Internet connection to the remote computer, the DSP program can invoke a SwiftNet system function to launch the dialer program on

the embedded host:

```
stat = system("dialer");
```

The function starts the dialer program and returns a success code to the DSP program using the variable `stat`. It first accesses a local ISP and then connects to the IP address of the central facility computer. With the Internet connection established, the DSP needs to add server access to the remote central facility host:

```
snstdio_init(remote);
```

With server access added, the DSP can send messages and files directly to the remote host. Once connected, the DSP program issues SwiftNet file commands to move all the archived

files stored during the day on the embedded Pentium local hard disk across the Internet to the central facility computer file system.

Note how easy it is to change the standard I/O services for the DSP board from the Ethernet link to the development workstation, to the embedded host across the backplane, and to the remote computer over the Internet. In each case, the only change to the DSP code was the `hostname` argument in the `snstdio()` function. ◆

Rodger H. Hosking (rodger@pentek.com) is Vice President of Pentek, Inc. in Upper Saddle River, N.J., and was a co-founder of the company in 1986. He was previously engineering manager at Rockland Systems and Wavetek Rockland.

CCS Plug-ins Aid Debugging

SwiftNet Debug Manager and a Register Configuration Tool are seamless eXpressDSP-compliant plug-ins for Code Composer Studio. The debug manager lets Code Composer Studio communicate with a virtually unlimited number of DSP targets in multiple locations, allowing live debugging over the Internet. The register configuration tool configures hardware registers for Pentek models 4290, 4291, and 4292, all TMS320C6000-based boards. Available now, the tools are supplied free when Code Composer Studio is purchased through Pentek (Model 4977) for \$2,995. **Pentek, Inc.**, Upper Saddle River, N.J.; 201-818-5900, www.pentek.com

Media Processor Harnesses Eight DSPs



The SPIRIT-6022 PCI-based high-density media processing platform delivers a dense, scalable architecture for applications like voice and fax over packet, CTI/IVR, speech recording and playback, and audio conferencing. Based on a high-performance architecture that builds on eight TMS320C6203 DSPs and a 200-MHz PowerQUICC II processor, the platform is complemented by a full suite of voice coders, telephony algorithms, and fax software, as well as speech recording and playback and other CTI/IVR APIs. The SPIRIT-6022 is available now. Prices start at \$5,421 each in single quantities. **RadiSys Corp.**, Portland, Ore.; (800) 950-0044, www.radisys.com

PCI Board Forms Voice Compression Channels

Using 12 TMS320C54x processors, the PCI-based Voice Shuttle delivers up to 72 scalable, high-quality voice

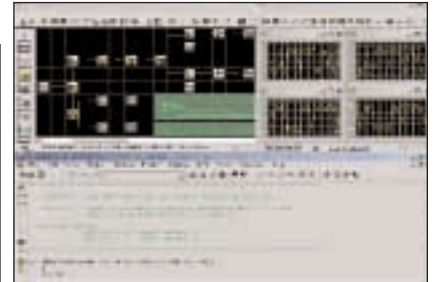
compression channels. The board generates Internet formats on telephony hardware, offering the option to select a preferred coding format, bit rate, and architecture. In this way, the Voice Shuttle optimizes voice quality at low bit rate while tailoring the output format to meet personal preferences. The Voice Shuttle sells for \$200 to \$300 per channel and is available now. **VoiceAge Corp.**, Montreal; (514) 737-4940; www.voiceage.com

CCS Plug-In Tests Software Components

The VectorCAST for Code Composer Studio plug-in lets designers test individual software components before testing at the system level. The software, which is eXpressDSP-compliant, allows designers to build complete test harnesses for each software component as well as construct test cases, execute code, view pass/fail test results, and get code coverage details for each test executed. Prices for VectorCAST, which is available now, start at \$6,850. **Vector Software, Inc.**, North Kingstown, R.I.; (401) 295-5855, www.vectors.com

eXpressDSP Telephony Algorithms Save Memory

A complete set of eXpressDSP-compliant telephony software for the TMS320C54x serves applications—like point-of-sale systems, ATM terminals, and set-top boxes—requiring low-bandwidth transfers of data, voice, or both. A unified, robust design reduces the system memory requirements of the dozen algorithms, which include code for V.32bis/V.32, V.22bis/V.22, DTMF, Caller ID type I and II, G3 fax, and 2.4-kb/s vocoders. The price for the V.32bis/V.32 algorithm starts at \$9,950. **Spirit Corp.**, Moscow; +7 095 912-7024, www.spiritcorp.com



Design Suites Achieve eXpressDSP Compliance

Two SystemView bundled design suites for the TMS320C5000 and C6000 platforms, the Real-Time DSP Architect and the Real-Time Communications Design Suite comply with the eXpressDSP specification for Code Composer Studio plug-ins. The SystemView portion of the suite promotes efficient design, simulation, and analysis. DSP Architect adds C code generation and a seamless interface to Code Composer Studio for rapid prototyping, real-time analysis, debugging, and hardware-in-the-loop simulation. The communications suite adds communications, RF, analog, and logic libraries. Both SystemView Real-Time DSP Architect and the SystemView Real-Time Communications Design Suite are available now and sell for \$8,995 and \$11,995, respectively. **Elanix, Inc.**, Westlake Village, Calif.; (800) 535-2649, www.elanix.com

Low-Power VC33 Module

A TMS320VC33-based, business-card-sized controller board, the D.Module.VC33 complements the DSP with microcontroller-like I/O peripherals optimized to minimize CPU load. The board provides up to 1 MB of additional SRAM, flash memory, a software-configurable external bus interface, high-speed UART, watchdog, in-system reconfigurable logic to accommodate application-specific interfaces, and BIOS. It's available now, starting at \$290. **D.SignT**, Kerken, Germany; ++49 2833 570977, www.dsignt.de

Development Software Spans Network Protocols

A complete embedded-software tool suite, the DelCORE Universal Framework can serve as a building block for developing DSP-based wired and wireless network applications that work across multiple, disparate protocols of the developer's choice, both commercial and proprietary. Touting a simplified connection manager and upper-layer functionality, the framework accommodates a range of DSPs, including the TMS320C5x, C54x, and C6000. Prices for the DelCORE Universal Framework, which is available now, start at \$25,000 plus royalties. **Delphi Communications Systems**, Maynard, Mass.; (978) 897-5650, www.delcomsys.com

eXpressDSP Components Focus on Communications

A large series of digital communications software components for the TMS320C6000 and C5000 platforms offers eXpressDSP compliance. Numbering 50 in all, the series includes G.723.1, FR-GSM, EFR-GSM, other telephony elements, and fax and modem components. Prices start at \$7,000 each. The components are available now. **Signals+Software Ltd.**, London; +44 (0) 20 8872 9000, www.signalsandsoftware.com



PCI-based DSP Board Targets Data Acquisition

The model AVR-32 is a PCI-based DSP and data acquisition board dri-

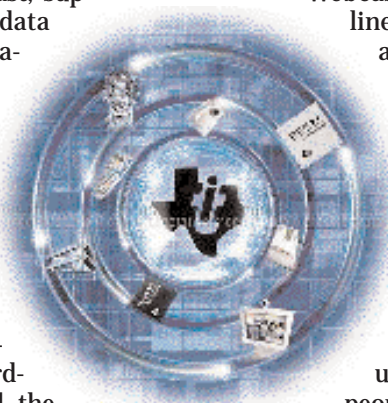
ven by a TMS320C32. The board's connector, which can tie directly to an IDE drive for fast data acquisition, consists of the inputs and outputs of 3-MHz 12-bit D/A and A/D converters. For flexibility, an onboard complex FPGA can be configured as a FIFO, dual-port RAM, digital filter, or other circuit. An assembler, debugger, Windows drivers, utility software, and sample programs are included. The model AVR-32 is available now and sells for \$995 each in single quantities. A software development kit for the board costs \$400. **Dalanco Spry**, Rochester, N.Y.; (713) 473-3610, www.dalanco.com



What Developers Need from Web Sites

By David Peterman

The Internet is changing the way designers develop their products, as well as the way vendors support designers and distribute the tools and the software they use. Designers now purchase development tools, software—even algorithms—on-line. Most vendors' Web sites, at the very least, supply manuals, data sheets, and application notes. Yet users need more, especially in the DSP embedded-systems world. Developers of such systems want vendors' sites to supply complete solutions, with information about hardware, software, and the links between them. Vendors face the challenge of creating the most useful, interactive, and information-rich on-line presence possible.



SEARCHING FOR ANSWERS

A series of focus groups conducted by Texas Instruments indicates that designers are most likely to use such sites in the early stages of product development, returning later for answers to specific questions. They would rather search a site than navigate through it to find those answers, and their chief complaint is the frustration they feel when trying to do so. Merely typing a question into a knowledge base and

receiving the response would be heaven, but most designers don't expect to see it in their own lifetimes.

A truly helpful Web site for designers of DSP-based embedded systems would have at least three important features: interactive

Webcasts, interactive on-line training, and the automatic notification of bug patches.

Each is needed if designers are to work as efficiently as possible.

A single interactive Webcast on a specific design problem permits a virtually unlimited number of people to receive more information about it than

they could by exchanging days' worth of e-mail: Watching a screen that shows code being written or a sensor moving on a diagram is a more immediate and comprehensible experience than reading application notes or listening to abstruse technical discussions. A tip or a solution from an interactive Webcast can reduce development time by weeks.

The second essential feature—interactive on-line training—makes it possible to train more designers in a single month than could be trained in 12 months of seminars in the physical world. Last year, more than 15,000 registrants participated in TI's on-line training courses, in

which designers, proceeding at their own pace, take 12-hour classes at their desktops. For both Webcasts and on-line training courses, questions submitted by e-mail or telephone should be answered quickly.

Last, automatically notifying designers about bug patches would help alleviate the common complaint that engineers sometimes spend days or even weeks working on a bug problem before discovering that a fix was available. Although some vendors do post bug patches on the Web, automatic notification is now rare in the embedded systems industry, not only for patches, but also for software updates and other relevant information.

Of course, the danger of reliance on the Web is the loss of the personal touch. Vendors can minimize this problem by creating vital on-line communities that cater to designers and answer their need for information and support. Interactive chat rooms provide additional opportunities for dialogue among users.

Designers are moving to the Web as a one-stop shop for all their professional requirements, notably purchasing. Vendors must now rise to the challenge.



David Peterman is the manager of Worldwide Software Development Systems at Texas Instruments Inc., based in Houston.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265