Uncoupling Software Architectures from Platforms

Tone Relay in Voice-over-Packet Data Networks
DSP Algorithms that are ready before you are.

Don't start from scratch. With ready-to-go eXpressDSP™-compliant algorithms, you have more cost-effective options for your application design decisions and will get to market faster.

- More than 400 algorithms available today from more than 50 TI third parties
- Easy integration into virtually any application
- DSP industry's only program for Algorithm Compliance Testing
- Interoperable – mix and match algorithms from different sources
- Portable – algorithms move easily between TMS320™ DSP platforms
- Measurable – "apples to apples" comparison of algorithm performance data
- Flexible – use the same algorithms across a variety of simple to complex systems
- Consistent – all standardized algorithms have the same look and feel in delivery, documentation, and naming
- More dedicated support to help you move from concept to production faster

Third-party algorithms ready to integrate
Voice Coders/Decoders
G.711, G.722, G.723, G.728, G.729, G.729, GSM, and more
Voiceband Fax/Modems
V.37, V.21, V.22, V.23, V.27, V.29, V.32, V.34, V.90, and more
Telephony
VAD, CNG, DTMF, CPTD, CID, LEC, and more
Audio
MP3, AAC, AAC+, MP3, MP4, WAV, VIB, and more
Video/Imaging
JPEG, JPEG2000, MJPEG, MJPEG2000, H.263, MPEG4
New Markets
Voice Recognition Encryption, Fingerprint, GPS, DAB, DVB

For a complete list of algorithms for your application, for on-line training, or to download the TMS320 DSP algorithm standard, visit www.ti.com/sc/algos

The World Leader in DSP and Analog
Inside This Issue

Insighter: Startling Tales of Nonfiction 4
DSP applications are startlingly complex and more powerful, but they can be written and debugged in reasonable amounts of time.

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

Cover: Efficient Multirate Signal Processing 8
With its dual MAC, abundance of buses, and copious internal RAM, the TMS320C55x allows a high degree of parallelism—just the ticket for applications using decimating filter banks.

New Methodology Tames Distributed Apps 16
Coordination-centric design uncouples software architectures from platforms, taking the pain out of developing distributed applications, heterogeneous as well as homogeneous.

Passing Tones in Voice-over-Packet Systems 24
A tone relay can eliminate troublesome signaling tone distortion and other problems in packet data networks that employ speech compression.

Launchings 29
New products and services for embedded systems developers.

On the Edge 30
DSP RTOSs come of age, but developers need more.
If you’re amazed by the ongoing advances in DSP silicon and development tools, you ought to be nothing less than astounded by the commensurate reach of DSP applications. Not only are the applications startlingly complex and more powerful, but they can be written and debugged in reasonable amounts of time—without a mainframe computer and swarms of computer scientists.

Take the cover story, “Efficient Multirate Signal Processing with the C55x.” Here, Michael Tsiroulnikov, of MIKET DSP Solutions, shows how to take advantage of the copious on-chip features of the TMS320C55x generation of DSPs—a dual multiplier-accumulator, multiple buses, and lots of RAM—for multirate operations in the complex domain. In particular, he shows how to shape decimating filter banks, a key component of multirate signal processing systems.

Working in the complex domain helps sidestep the limitations of real-domain bandpass sampling, increases performance, and simplifies later signal processing. At the same time, the large store of RAM satisfies the often unquenchable thirst of multirate signal processing systems for fast RAM in which to store filter bank coefficients. In addition, the C55x's speed, boosted in large part by its dual MAC, enables you to compensate for burgeoning program code by running more instances of an algorithm in a given amount of time. All told, you should realize an order-of-magnitude cumulative improvement in performance over previous generations of DSPs when you implement signal detectors, decimating filter banks, and other functions that filter input data through an array of same-length complex filters.

Increasingly, such powerhouse signal processors are finding themselves working alongside general-purpose processors—on one chip, no less. This teamwork is great for distributed heterogeneous applications, but it poses a challenge if you’re seeking to wring the best efficiency out of the on-chip resources.

For instance, transformational algorithms are best performed by digital signal processors, whereas control-intensive and graphical user interface functions are best served by a general-purpose microprocessor. Accordingly, applications that are both control-intensive and algorithmically complex typically call for multiple-processor architectures.

Take heart. A new methodology, termed coordination-centric design, uncouples the architecture from the platform and takes the pain out of developing distributed heterogeneous applications. As David McCooey, Ken Hines, and Ross Ortega of Consystant Design Technologies relate in our second article, in developing distributed software applications software architects must consider multiple operating systems and think about how applications will map to specific processors. Specialists could be required to deal with obscure programming paradigms or tight performance requirements. Such implementation details can obscure the high-level software architecture.

Coordination-centric design lets designers cleanly separate the two and automatically produces efficient source code from high-level software models. Thus it supports heterogeneous distributed architectures and allows you to specify low-level implementation details without reference to the software design.

Eliminating trouble is the theme of our third article, this time the troublesome signaling tone distortion and other problems that can pop up in speech-compressed packet data networks. The solution: Use a tone relay to pass tones in voice-over-packet systems.

Adaptive Digital Technologies’ Scott Kurtz delineates the problems caused by low-rate speech compression algorithms and delves into the design and nuances of tone relays and their accompanying algorithms. For instance, although a tone passer seems straightforward at first glance, its implementation details can make or break performance. Kurtz’s thorough discussion should help clear up any distortion.

—Stan Runyon
testman2@earthlink.net
DSP Design Tools

The Leader in Portable Emulators for TI DSP's

We have your high performance tools for your DSP projects.

- Emulators:
  XDS510, XDS515, SPlx10, SPlx15, SPlx20, SPlx35, SPlx50, SPlx50

- Debuggers:
  TI Code Composer Studio

- Evaluation Modules:
  C54x, C5003/7206, F240/F240/F240/F2407,
  C54x/VCS49/VCS402/VCS406/VCS410/VCS416/VCS472/VCS509/VCS510

- eZdspTM and DSKs:
  F240/F241/F2407, eZdps for
  L2407/VCS407/VCS10/VCS33

- Software Tools:
  'C' compilers, assemblers, linkers

- Operating Systems:
  RTXC, Real Time Executive in 'C'

- Miscellaneous:
  Digital Motor Controllers, Surround
  Sound Modules, Prototype Modules,
  Power Supplies, JTAG Expander

Spectrum Digital = Tools for
DSP Development

12502 Exchange Drive, Suite 440
Stafford, Texas 77477

1: 281.494.4505
F: 281.494.5310
www.spectrumdigital.com
18/03/01
Spectrum Signal Wins Defense R&D Funds

Spectrum Signal Processing Inc. (Burnaby, B.C., www.spectrumsignal.com) is slated to receive over $500,000 (Canadian) from the Defence Industrial Research Program (DIRP) to create additional capability for its flexComm line of software-defined radios. The DIRP, an initiative of Defence R&D Canada, funds defense-related industry research and development. In this case, the money will go toward developing hardware and software that adds ultrawideband capability to Spectrum’s SDRs. The ultrawideband SDRs, in turn, will serve as subsystems in a modern processing system for a software-configurable, military satellite communications terminal.

Texas Instruments Incorporated (Dallas, www.ti.com) has joined the nonprofit, 50-plus-member RapidIO Trade Association (San Francisco; www.rapidio.org), an open-standards interconnect group that is pushing the use of “inside-the-box” high-performance switch fabric technology to multigabit-per-second transmission rates. Toward that end, the association’s RapidIO specification, released in March 2001, defines how chips and boards communicate within a system.

Dy 4 Systems (Kanata, Ont.; www.dy4.com) has been acquired by, and is now a business unit of, Force Computers (Fremont, Calif., www.forcecomputers.com). The acquisition is the result of a merger of Force’s parent company, Solectron, and C-MAC Industries. Dy 4 will keep its brand of COTS ruggedized embedded computing products for the defense and aerospace industry. At the same time, it will draw on Force’s hardware and software expertise in the commercial and telecom markets. For its part, Force aims to work Dy 4’s DSP expertise into its OEM business.

Questra Corporation (Rochester, N.Y., www.questra.com) has completed a $19 million second round of financing with Menlo Ventures and Trident Capital. Its enterprise software integrates mission-critical, intelligent devices with a company’s service and support infrastructure via the Internet.

Duo Joins PCTEL Executive Team

PCTEL (Milpitas, Calif.; www.pctel.com) has named John Schoen and Jeff Miller to its executive team. Schoen will serve as both chief operating officer and chief financial officer, positions he held at SAFCO Technologies. Before that, he filled various financial spots during 19 years at Motorola. Miller, who will take over as vice president of development, led SAFCO’s Test and Measurement Group for 3 years before it was acquired. Earlier, he led the implementation of the Cellnet cellular network and managed large software projects for Motorola’s Cellular Infrastructure Group.

TI Joins RapidIO Open-Standards Trade Group

Texas Instruments Incorporated (Dallas, www.ti.com) has joined the nonprofit, 50-plus-member RapidIO Trade Association (San Francisco; www.rapidio.org), an open-standards interconnect group that is pushing the use of “inside-the-box” high-performance switch fabric technology to multigabit-per-second transmission rates. Toward that end, the association’s RapidIO specification, released in March 2001, defines how chips and boards communicate within a system.

TI Embraces Linux for Its OMAP Platform

Texas Instruments, Inc. (Dallas; www.ti.com) has extended its OMAP wireless architecture to include the Linux operating system. The extension will give developers of Linux applications for 2.5G and 3G mobile devices easy access to the company’s DSPs. In addition, TI has selected RidgeRun, Inc. (Boise, Idaho www.ridgerun.com), developer of the DSPLinux operating system, to directly assist its customers. It has also worked with RidgeRun to enhance DSPLinux to take advantage of the real-time multimedia capabilities of the OMAP platform.

Questra Caps Second Financing Round

Questra Corporation (Rochester, N.Y., www.questra.com) has completed a $19 million second round of financing with Menlo Ventures and Trident Capital. Its enterprise software integrates mission-critical, intelligent devices with a company’s service and support infrastructure via the Internet.
Hand-In-Hand Remote Access + Power of DSP + Flexibility of FPGA

SMT363 @ a Glance

Features:
- Netsilicon 50 ethernet processor
- TI's 150MHz C6711 DSP with 16 Mbytes SDRAM
- Xilinx Spartan II or Virtex FPGA
- Support for PCI, ePCI, PXI and VME

Benefits:
- Remote development and access
- Simple implementation of multi-channel solutions
- Scalable system configuration

Applications:
- IF Radar Sampling
- Base-band Radar Sampling
- Spectrum Analysers
- Digital Radio Receivers
- Multi-channel Receivers
- Cellular/PCS base stations

Sundance Digital Signal Processing Inc.
Tel: (775) 827-3103 USA
Sundance Multiprocessor Technology Ltd.
Tel: +44 (0)1494 793667 UK
Email: sales@sundance.com  http://www.sundance.com
With its dual MAC, abundance of buses, and copious internal RAM, the TMS320C55x allows a high degree of parallelism—just the ticket for applications using decimating filter banks.

By Michael Tsiroulnikov

Compared with previous generations of digital signal processors, Texas Instruments’ TMS320C55x generation of DSPs is particularly well suited to multirate operations in the complex domain. Its single-cycle dual multiplier/accumulator (MAC), multiplicity of buses, and generous endowment of internal RAM give it about an order-of-magnitude cumulative improvement in performance over previous generations of DSPs when implementing signal detectors, decimating filter banks,
and other functions that filter input data through an array of same-length complex filters.

Complex-domain operations make it possible to overcome the limitations of real-domain bandpass sampling, thereby helping to maximize a filter’s decimation ratio without any loss in precision. A decimating filter bank can be viewed as an application-specific transform or a signal compression technique in which blocks of real signal data are converted into shorter sets of downsampled values.

**PROCESSOR HIGHLIGHTS**

Compared with its predecessors, the TMS320C54x generation, the C55x is highly advanced. Most notably, just its single-cycle dual MAC, supported by multiple buses that let it retrieve three 16-bit data words in a single cycle, gives the processor a twofold performance advantage. In addition, the new

---

**Figure 1.** Organizing the data into blocks of symmetric and asymmetric coefficients and then taking them in sequential order for the even and odd indices, respectively, keeps auxiliary pointer manipulations to a minimum. The coefficient data pointer (CDP) points to the prefolded input data; the auxiliary registers (ARx) point to the coefficient arrays.
processors have significantly more internal RAM. These resources can be used very effectively to realize various filter banks and functional transforms.

Relative to code written for the C54x, code for the C55x has less loop initialization and finalization overhead—an important advantage in staged multirate signal processing, when filters are often short and the overhead can therefore add significantly to the execution time.

Other valuable new features of the C55x are its coefficient data pointer (CDP) register and its auxiliary registers (ARx), which now allow long immediate offset values to be included in the op code without adding any extra cycles when the program is executing. This capability makes it easier to develop assembly code in a C-like fashion, thereby shortening the R&D cycle.

The key to the successful design of a DSP application is appropriate data organization for both the filter bank coefficients and the input data. Properly organized data makes for a smooth, natural flow of operations with minimal pointer manipulations and no unnecessary discontinuities in the program flow.

To exploit the advantages of complex-domain filtering, the input data vectors should be decomposed into symmetric and asymmetric components; the decomposition helps to boost performance as much as twofold. The new CDP register is exploited in this connection, being used to point to the decomposed data, while the auxiliary registers point to the filter banks.

An obvious drawback of multirate signal processing is its need for a large amount of fast RAM in which to store filter bank coefficients. That need usually won’t be a problem for the C55x generation because most of the processors have more than enough internal RAM. Also, their speed can compensate for the increase in program code by allowing more instances of an algorithm to be run in a given time.

REALIZING A GENERIC COMPLEX FILTER BANK

Listing 2: Assembly Code for the Filter Bank Function

```
.mmregs
 .cpl_on
 .arms_off
 .text
 ; ---------------------------------------------
 .global APP_filter_bank
 APP_filter_bank
 ; ---------------------------------------------
 ; ---- function arguments:
 ; ar0 = APP_tSc *pSc
 ; ar1 = ptr to _aaEvenFilterBlock[0][0]
 ; ar2 = ptr to _aaOddFilterBlock[0][0]
 ;
 ; pushboth(xar5);
 ; pushboth(xar6);
 ; bit(ST1, &ST1_ARMs) = 0;
 ; bit(ST1, &ST1_FRCt) = 1;
 ; sum / sub --------------
 xar3 = mar[ar0(&APP_tSc.zpDatat + APP_FB_PSZ-1)];
 xar4 = mar[ar0(&APP_tSc.zpDataPlus)];
 xar5 = mar[ar0(&APP_tSc.zpDataMinus)];
 xar6 = mar[ar0(&APP_tSc.zpDataData)];
 brc0 = #(APP_FB_PSZ/2-1);
 localrepeat {
   t0 = *ar3;
   HI(ac0) = *ar6 + t0, LO(ac0) = *ar6 - t0;
   *ar5 = LO(ac0), *ar4 = HI(ac0);
 }
 ; filter -------------------------
 xcdp = mar[ar0(&APP_tSc.zpDataPlus)];
 xar3 = mar[ar0(&APP_tSc.zcOut + APP_tOut.sG)];
 t0 = #(APP_FB_PSZ/2+1);
 t1 = #(APP_tOut.sG);
 brc0 = #(2-1);
 brc1 = #(APP_FB_PLS/2-1);
 localrepeat {
   localrepeat {
     ac0 = *ar1 + *coef(*cdp+);
     acl = *ar2 + *coef(*cdp+);
     || repeat (#(APP_FB_PSZ/2-3))
     ac0 = ac0 + (*ar1 + *coef(*cdp+));
     acl = acl + (*ar2 + *coef(*cdp+));
     ac0 = rnd(ac0 + (*ar1 + *coef(*cdp+));
     acl = rnd(acl + (*ar2 + *coef(*cdp+));
   }
   xcdp = mar[ar0(&APP_tSc.zpDataMinus)];
   xar3 = mar[ar0(&APP_tSc.zcOut + APP_tOut.sA)];
 }
 ; exit ------------------------
 bit(ST1, &ST1_FRCt) = 0;
 bit(ST1, &ST1_ARMs) = 1;
 xar6+popboth();
 xar5+popboth();
 return;
```
given sampling frequency, \( f_s \), a signal with frequency \( f_s / 2 + d \) is aliased by a “mirrored” signal, \( f_s / 2 - d \), for any deviation, \( d \). In effect, the whole frequency plan is folded multiple times within the interval \( 0 \ldots f_s / 2 \).

The operations on analytic signals in the complex domain are much simpler, because the frequencies \( (k * f_s + d) \), for any \( k \), are aliased into themselves—the frequency plan can be viewed as going around in a circle. Consequently, the central frequency of the signal band can be in any relationship to the sampling frequency, and the choice of the decimation ratio is much relaxed, although it’s still constrained by the filter properties, the signal/noise spectrum, and the application requirements.

So how, exactly, do you convert a real-domain signal, \( X(t) \), into complex-domain signal \( q(t) \)? For:

\[
X(t) = \{x(t), x(t - 1), x(t - 2), \ldots, x(t - L + 1)\}^T
\]

\[
q(t) = os_i(t) + j*oa_i(t)
\]

where \( L = \) the length of the filter, what’s needed is a complex filter bank made up of \( N \) subfilters, \( F_z \), each of which has the form:

\[
F_z = F_{s_i} + j * F_{a_i}
\]

where \( F_s \) indicates real components of \( F_z \), and \( F_a \) indicates imaginary ones:

\[
F_{s_i} = \{fs_{i,0}, fs_{i,1}, \ldots, fs_{i,L-1}\}^T
\]

\[
F_{a_i} = \{fa_{i,0}, fa_{i,1}, \ldots, fa_{i,L-1}\}^T
\]

\( i = (0 \ldots N - 1) \)

(all of the subfilters are assumed to be of the same length) and \( q(t) \) can be given as:

\[
q(t) = F_z^T * X(t)
\]

For a very wide class of filter banks, \( F_z \) can be represented as the sum of two components, a strictly symmetrical real one, \( F_{s_i} \), and an asymmetrical imaginary one, \( F_{a_i} \), and both components of each subfilter will still comply with the rules for the Hilbert transform.

The straightforward approach to performing these calculations involves using the \texttt{fir()} and \texttt{firn()} instructions of the C55x, but there’s a superior approach. If the bank has many subfilters of equal length, it’s more efficient to prefold the input data, \( X(t) \), around its center point, \( (t_c) \), into two sets, one a sum of the samples \( x(t_c - t) \) and \( x(t_c + t) \) and the other the difference between those same samples:

\[
s(d) = x(t_c + d) + x(t_c - d)
\]

\[
a(d) = x(t_c + d) - x(t_c - d)
\]

With the input data in complex form, our task now is to optimize the processing operations so as to exploit the C55x’s dual MAC as fully as possible. Two things that can be done in pursuit of that goal are:

- Use CDP (which is, by default, a pointer to coefficients) as a pointer into the prefolded data arrays, while using auxiliary pointers to operate with coefficients.
- Regroup filter banks. When CDP points to the summed data, \( s(t) \), we need both auxiliary pointers to operate with arrays from the \( F_{s_i} \) set. Therefore we can set up two “parallel” blocks of filter coefficients for either \( F_{s_i} \) or \( F_{a_i} \) and use even and odd subfilters sets, \( \{F_{s_0}, F_{s_2}, F_{s_4}, \ldots\} \) and \( \{F_{s_1}, F_{s_3}, F_{s_5}, \ldots\} \), linkable into different single-access RAM segments. Then the filtering with \( F_{s_2} \) will be performed simultaneously with the filtering with \( F_{a_2} \). The modifications of operations with \( F_{a_i} \) are similar. Of course, the code

Embedded Edge
March 2002 11
Multirate Signal Processing

will definitely be simpler if the number of filters in the filter bank is even. This approach becomes more efficient as the number of subfilters grows; indeed, it makes sense for four or more subfilters.

You can use a very similar approach for the more general case of a complex input and a complex filter bank; the processing flow and the code are essentially the same. Be sure to use the Hermitian (conjugate transposed) operator while doing complex domain filtering; otherwise the frequency plan will be inverted.

Determining the best trade-off among decimation filter, block size, and decimation ratio must be decided case by case. Sometimes it makes perfect sense to use longer filters with better shape factors (narrower transient bands), which allow a higher decimation ratio. But since it’s impossible to provide general advice appropriate for all circumstances, it’s up to you to decide on the trade-offs when you develop your algorithm.

In general, because the filter length is typically longer than the block size, the processing flow involves concatenating incoming data with previously saved block(s) of data and then updating the “saved signal” array. The filter bank function is C-callable:

```
void APP_filter_bank(APP_tSc *pSc, Int *psEven, Int *psOdd);
```

The filter function accepts as parameters a pointer to a scratchpad structure and pointers to the even and odd filter blocks (Listing 1; the code for concatenation and updating described above has been omitted, since it’s straightforward), each grouped into an array of the form:

```
Int_aaAppEvenFilterBlock[2*N/2][L/2];
```

where the first N/2 rows of half-sized filters (L/2) are taken in sequential order from a symmetric set, $F_{si}$ and the rest are taken from asymmetric set $F_{ai}$ for the even and odd indices, respectively. Organizing the data in that manner (Figure 1) helps keep auxiliary pointer manipulations to a minimum.

**DATA FORMAT**

In the same manner, the output of the algorithm should represent the data in the format that’s most suitable for the further processing. A C-like structure `APP_tOut` is declared, which may contain other fields if required by an application. The function `APP_filter_bank()` only stores $O(t)$ (decimated complex signal pairs per subfilter) into predefined locations. Note that the filter length doesn’t have to be even.
greatly helps with writing, debugging, and testing assembly code by running bit-exact comparisons with a C version of the same algorithm. Despite the relative complexity of the C55x instruction set, I found that I could write C55x assembly code faster than I could write code for previous DSPs, like the C54x or earlier generations.

As you can see from the listing, filtering with Fs and Fa is done using an external loop over the block repeat counter register (BRC0). The pointers to the input array (pSc->asPlus or pSc->asMinus) and the output array (pOut->sC or pOut->sS) are updated when the first iteration is over, but the pointers to the coefficients remain free-running.

Note that the internal loop initiation and finalizing overhead take only three processor cycles. In certain cases it may be possible to decrease the number of cycles even further.

THE FILTER BANK

Let’s demonstrate the use of this procedure to detect multifrequency R1 signals as specified in the ITU-T Q.320, Q.322, and Q.323 recommendations. Although a very simple example, it demonstrates the advantages of this approach very nicely.

Valid multifrequency signals are represented by two, and only two, tones from a set of six frequencies \( f_i = 700 + i \times 200 \text{ Hz}, i = (0 \ldots 5) \), with 1.5% tolerance and a duration of at least 60 ms. In this kind of application, noise isn’t usually a problem because tone detection is done before a call is established; hence there’s no voice on the line to corrupt the tone signal.

Our filter bank consists of six subfilters of length 100 (input signal sampling frequency \( f_s = 8 \text{ kHz} \)), derived from a Hamming prototype and modulated by \( \cos(2\pi f_i [t - t_c]) \) and \( \sin(2\pi f_i [t - t_c]) \). (Figure 2 shows the filter shape for the second frequency bin, 900 Hz.) The passband (tinted area) is \( \pm \Delta f \) (about 1 dB), \( \Delta f = 40 \text{ Hz} \), and the stopband is \( 200 - \Delta f = 160 \text{ Hz} \) (42 dB). These values provide some headroom and allow signal processing with a maximum frequency deviation of 2.35% even for the highest-frequency bin, 1,700 Hz.

The choice of the decimation ratio depends greatly on the operations to be performed on the decimated signal. Although signal detection applications allow for deep subsampling, other applications, such as subband adaptive filtering, may impose another set of (obviously, much harder) constraints on how low a signal may be decimated. In our application, the transition bands (-160:-40) and (40:160), relative to the central frequencies, may be overlapped (Figure 3).

If we consider the shape of the decimation filter only, the Nyquist barrier can be seen as deliberately broken. A signal with the spectrum falling into transition bands beyond the tinted area will be aliased. Fortunately, no fatal consequences result, because the signal there is only noise and there’s no need to reconstruct the original signal. Note that the area of interest (~40:40 off the central frequency, 900 Hz) is “clean.” Nevertheless, it’s overlapped with the signals in adjacent frequency bins and the accompanying noise, all in the stopband area of the filter. Therefore, to utilize such a high decimation ratio as 40, the prototype filter will be “sufficiently good” and have an adequate shape factor and stopband attenuation.

THE INPUT

The input into the function is a block of 40 samples (appended to the saved preceding data), which is transformed into six complex pairs. This number of samples corresponds to 5 ms, an interval that fits well with other signal processing algorithms, like ITU-T...
G.72x vocoders. The further processing of the decimated signal, to estimate the instantaneous frequency and energy in this low-noise case is obviously simple. If you’re interested in more advanced techniques, Modern Spectrum Analysis [2] provides an excellent starting point.

For this decimation ratio \((dr = 40)\), \(N = 6\), the filter length \(L = 100\), and the sampling frequency \(f_s = 8,000\) Hz; the theoretical, straightforward MAC count is \(2 * L * N * (f_s / dr) = 0.24M\) MACs. The processing figure for \(APP\_filter\_bank\) is only 0.11 MIPS (550 clock cycles per function call), including the overhead of the function call. That low number clearly demonstrates the superiority of this approach over the straightforward one and, of course, over traditional approaches, like Goertzel filters, which are usually slower by an order of magnitude.

You can lower both the MIPS and memory requirements even further by using a staged decimation process. For example, the MFR.1 input signal allows simple decimation-by-2 filtering, since the highest frequency is sufficiently offset from 2 kHz for the filters to be organized in a way that makes half of the coefficients equal to zero. In addition, the filters in the second stage become half as long, also lowering program RAM requirements.

**MANY APPLICATIONS**

You can use very similar techniques for different applications: almost any kind of tone detector, signal classifier, or spectrum analyzer and a wide variety of filter banks, can achieve better performance with this approach.

But how much that performance gain benefits the end user depends on many factors. Although you can improve the performance of an entire signal detection and classification subsystem significantly with these techniques, the subsystem usually represents only a small part of the whole picture. If an entire voice processing system is built using a so-called “universal port”
approach—one in which a single DSP performs all the functions required for one or few channels—the benefits will be modest because the savings in required processor power are minor relative to a typical vocoder’s requirements. If, on the other hand, the voice processing is distributed—that is, if various DSPs are assigned to different functions—and each DSP executes only a few smoothly coexisting algorithms, the difference can be dramatic. There are three main reasons.

For one, such a system may be composed of the best available interoperable algorithms from different vendors. For another, fewer DSPs will likely be needed to achieve the same channel density, since every algorithm may trade off between program memory and per-channel performance. Finally, there'll likely be no need to attach any external RAM to the DSPs because the DSPs won’t have to keep the entire set of algorithms in their program space. As an added bonus, power consumption will be reduced to just the very low C55x core power plus serial peripherals.

Taken together, these factors will reduce the overall system size and price of future telecommunication infrastructure equipment while increasing its performance, quality, and reliability.

NOTES


Michael Tsiroulnikov (mikets@telus.net) is the principal DSP engineer at MIKET DSP Solutions in Richmond, B.C. He enjoys developing high-quality, high-performance customer-specific applications on the newest DSPs, especially ones presenting significant R&D challenges. His expertise includes system design, adaptive algorithms, and practical implementations of functional transforms.
A new methodology takes the pain out of developing distributed heterogeneous applications.

Design Methodology Uncouples Software Architectures from Platforms

By David McCooey, Ken Hines, and Ross Ortega
ponents and special communication “coordinators”—which are created or selected from libraries of preexisting components. Then, the components and coordinators are graphically composed to create a software design.

Once it’s been created, the design is simulated to validate its functional behavior. After that, an architectural mapping phase assigns the pieces to the computing resources, that is, components to processors and coordinators to communication mediums. Commented C code is generated that’s tuned for each operating system running on a specified processor and implements the behavior captured by the coordinators.

Let’s visit the mapping and code generation stages in more detail.

In the mapping phase, components must be assigned to processing resources—an operating system process, for example. Coordinators can be assigned to processing resources or communication mediums. Figure 1 shows three possible mappings of a simple software design to different target architectures.

The final stage of coordination-centric design, automatic code synthesis, translates the engineer’s high-level model into C code. The synthesized code provides the same functionality as the high-level design model, but important optimizations have been made. During code synthesis, component and coordinator distinctions are dissolved; components and coordinators are an artifact of logical design only. This is a significant advantage of coordination-centric code synthesis over other, more traditional object-oriented code generation approaches: by exploiting the semantic information provided by the coordination-centric model, synthesis generates code that’s much more efficient in terms of memory footprint and run-time performance.

In traditional software design, a developer must select an appropriate communication mechanism, depending on the execution location of the interacting component. If the components execute in the same operating system process, shared memory is an efficient mechanism. If they’re in different processes, an operating system interprocess communication call is appropriate. If they reside on different processors, a remote communication mechanism must be selected. All of these choices are low-level implementation decisions that complicate software architectures. In coordination-centric design, these details are dealt with before code generation and aren’t embedded in the software design.

In traditional component-based

**The synthesized code provides the same functionality as the high-level design model.**
Coordination-centric Design

March 2002

Embedded Edge

design methodologies, each component must contain not only its core functional behavior, but also its behavior interacting with other components. The entanglement of functional and interaction behavior forces software developers to create tightly coupled software components. If the interaction behavior among components changes, all those components must be updated to reflect the changes. These intrusive modifications are required even when the core functionality of the component isn’t altered. What’s more, tightly coupled software components are difficult to understand, debug, and maintain and even more difficult to reuse in different designs.

BEHIND THE BENEFITS

With a coordination-centric framework, however, software components are loosely coupled through the use of coordinators, which contain the interaction code between components. A design rule requires a coordinator between any interacting components. Thus a coordination-centric framework encourages reuse by allowing system components to be interchanged in a “plug and play” fashion; indeed, entire subsystems can be designed independently from other system components. Also, the debugging phase is greatly reduced because the correctness of the software functionality is separated from its interaction behavior. Essentially, both debugging and maintaining software are simplified because there are no “back door” interaction paths with the components. All communication and interaction is explicit and clearly defined.

A distributed MP3 player application is a good example of how coordination-centric design works. Let’s look at an MP3 player we designed using Strata, a commercial implementation of coordination-centric design.

MP3 PLAYER

The design steps were incorporating existing code into components, using models to validate the behavior of the architecture, mapping the design to a single processor and synthesizing the code. The very last steps were to map the same design to a distributed-processor architecture and synthesize the code for each processor.

We used a PC running Linux as the development host for Strata, for design entry and behavioral simulation of the system prior to mapping. The same PC served as the target and to create a Linux executable that plays MP3 songs from a file system on the PC. For the distributed-processor mapping, we used an

Figure 2. With Strata, you compose a software system by dragging and dropping components and coordinators and making connections through bindings. The Controller manipulates the MP3FrameReader, which in turn reads an MP3 file and provides it to the Decoder. The rounded blue blocks are reusable software components, the square green blocks the coordinators.

Figure 3. An action triple model allows software contained in action trapezoids to be executed upon the occurrence of certain trigger events, provided that the related mode of operation is true. For instance, within the Controller, when the Play mode is active and the Check event occurs, the code snippet inside playing_check is executed.
Ethernet crossover cable that connected the PC to a Texas Instruments TMS320C6711 DSP board. The distributed mapping of the player placed the control and file reader portion on the PC and the decoder and playback on the DSP board. We loaded the synthesized code for the DSP board using TI's Code Composer Studio IDE.

The first task was to partition the player functionality into coordination-centric components and coordinators. Because of the streaming nature of the application, the coordinators required were simple data pipes. (Figure 2 shows a Strata tool view of the software design for the MP3 player.) To demonstrate the main decoding algorithm, we wrote a simple C-based control GUI to start, pause, and stop the song.

Pushing into the Controller component, shown in Figure 3, highlights some interesting features of the coordination-centric design framework. The graphical syntax captures the event-driven programming model, which uses modes, events, and actions. Modes are internal state variables that guard the action. Actions can turn modes on or off, generate events, modify variables, and make foreign subroutine calls. Events are generated either automatically by the system or explicitly by an action. The combination of a mode, an event, and an action constitutes an "action triple." For example, when the Play mode is true and the Check event arrives, the action playing_check is triggered.

**USING LEGACY CODE**

A foreign subroutine is one written in C or Java. A major advantage of coordination-centric design is that such legacy code can be intermingled with new software components. The playing_check action code, below, communicates with the GUI controller via the MP3_Control foreign call, which returns the identity of the button pressed. After activating the corresponding mode, the action generates the Check event.

```c
foreign function int Mp3_Control ();
int result;
result = Mp3_Control ();
if ((result == 1)) { +Play;
} else {
    if ((result == 2)) { +Pause;
    } else {
        if ((result == 3)) { +Stop;
        }
    }
}->Check ();
```

In Figure 3 is a user-defined constraint block. It enforces a mutual exclusion constraint between the modes Play, Pause, and Stop, which are connected to the inputs m1, m2, and m3, respectively. Constraints clearly declare and enforce the intended use of a component. In standard programs, in contrast, constraints are usually embedded into the structure of the code or contained in a comment to remind the developer and others about the intended use of a body of code. Also, note that like components and coordinators, constraints are reusable elements of a software design.

Coordination-centric design provides a very powerful abstraction in the ability to share states among components without committing to a communication mechanism. Com-
ponent and coordinator interactions occur via interfaces. Figure 3 shows the Controller's graphical connection to its Control_Out interface. Graphically the modes Play, Pause, and Stop are bound to this interface and therefore shared with the MP3Frame_Reader via the coordinator ctrl_pipe, as shown in Figure 2.

INDEPENDENT SIMULATION
To validate the functional behavior of the design, we first simulated it independent of operating system and hardware issues. By debugging the design at this level of abstraction, we gained confidence in the algorithm's correctness. In a traditional debugging environment, an error could be in the algorithm; it could be an improper use of the operating system or a communications protocol; it could be a hardware bug or a timing problem; and so on.

The developer must consider all the possibilities simultaneously. To ease the task, the coordination-centric framework provides a visualization called “evolution diagrams” that graphically illustrate interactions among the components and coordinators, as shown in Figure 4. (Note that this simulation isn’t a real-time execution of the system.) Each horizontal trace is a component or coordinator. Within a component or coordinator, colored horizontal bars represent the component’s or coordinator's mode. Vertical bars are events. Colored arrows among the traces show control communications, message sends, and variable propagation. Of particular interest are the control and variable arrows, which show interaction behavior that wasn't hand-coded by the developer but instead generated automatically.

The sound quality demonstrated by the model simulation proved that the architecture was functioning properly.

TARGET MAPPING
Confident in the correctness of the architecture, we created a Linux MP3 player by mapping the design to a single Linux process. Since coordinators are logical constructs, the ones in Figure 2 were optimized

Four of a kind!
Being first to market is a sure bet when your hand includes cards like the Quatro6x...

Features
- Four 160 MHz TMS320C6701 DSPs (fastest-point)
- Plug-n-Play PCIe Interface
- FIFO link interconnects between processors
- Three FIFOPort interfaces
- Comprehensive C/C++ cross development tools & algorithm templates
- Windows 9x/NT/2000 drivers

Applications
- Numeric coprocessing
- Video coprocessing
- Large-scale data reduction
- Huge, multiprocessor arrays

Get your data sheets now! Visit the Quatro6x site at www.innovative-dsp.com/q6x

Innovative Integration... real time solutions!
Ported from architecture to architecture, typical DSP operating systems never reach their full potential on any particular processor. But now, the engineers at Blackhawk™ have painstakingly crafted an individually perfect DSP RTOS for each modern TI DSP architecture.

Each Blackhawk™ TAILWIND™ DSP reference board has its own TALON™ RTOS written specifically for it. Eliminating the compromises of ported OS’s; enabling the highest possible performance and the quickest out-of-the-box bring-up.

Blackhawk™ provides a full spectrum of DSP products, tools, and services that bring your DSP designs to market faster, at less cost. Before you start your next TI DSP project, wing your way over to www.blackhawk-dsp.com/handcrafted.

Blackhawk™ is the perfect match for your DSP development project.
away during code generation, in this case because all interaction occurs
in the shared address space of a Linux process. Code generation cre-
ated commented C code and a corresponding make file. The code was
then compiled with gcc to create the binary image. Because the simula-
tion foreign subroutines were written for Linux, we reused them for
the target. The resulting MP3 player executable played in real time.

As described earlier, we then distributed the design across a Linux
PC and a C6711 DSP evaluation board connected by an Ethernet
crossover cable. The distributed MP3 player plays MP3 songs by
retrieving them from the Linux side, transmitting them to the DSP board,
decoding them, and playing them through an external speaker.

Figure 5 shows the mapping table for the distributed player. We
mapped the decoder component to the DSP board, the connecting coor-
dinator to the TCP link, and the remaining control and file aspects to
the Linux computer. Code generation created two sets of C files, one
for the Linux side and one for the DSP/BIOS side, as well as TCP calls
for Linux and for DSP/BIOS. Using Code Composer Studio, we com-
piled the C files for the DSP board and downloaded the binary image
onto the board. We compiled the Linux side with gcc, then ran both
executables and listened to an MP3-encoded song.

RESULTS
The memory requirements for the distributed MP3 player synthesized
software are as follows: for the Linux MP3 player, the Linux footprint was
233 kilobytes; for the distributed MP3 player, it was 119 KB, and the
DSP footprint was 805 KB.

The large DSP footprint needs further explanation. Because we had
essentially a “raw” board, the entire
software image had to be down-
loaded. The TCP/IP stack provided
with the development kit—down-
loaded to drive the Ethernet con-
nection—is 301 KB (37% of the DSP
image). Other DSP/BIOS libraries
and objects required an additional
68 KB (8%). The DSP architecture
requires the alignment of various
data structures; it enforces the align-
ments by introducing “holes,” or
gaps, in memory, which took up 285
KB (35%). We had to write initializa-
tion code to bring up the TCP/IP
stack, which required 20 KB (2%).
The legacy code, including the MP3
decoder algorithm itself, required 96
KB (12%), and the generated synthe-
thesized code 35 KB (4%).

David McCooey (david.mccooey@consyst-
tant.com) is a senior software engineer at
Consystant Design Technologies, Inc.,
focusing on platform differences and how
to capture them in APIs. For the previous
12 years, he worked on the internals of
the LynxOS RTOS at Lynx (later to
become LynuxWorks). Before that, he
was a member of the technical staff at
AT&T Bell Labs, working on the Access
Network System (ANS). Ken Hines
(ken.hines@consystant.com) is Consyst-
ant’s chief scientist, a vice president, and
co-chairman of the board; Ross Ortega
(ross.ortega@consystant.com) is chief
technology officer, a vice president, and
co-chairman of the board. Hines and
Ortega founded Consystant based on
their doctoral research.
SOFTWARE SOLUTIONS FROM SUNDANCE
for TI’s C6000 DSP Platform

RTOS From 3L:
- Diamond Toolset Supporting
- Multi DSP with Multi Tasking
- Multi Threading
- Pre-Emptive Scheduling
- Virtual Channel Routing
- Host I/O
- DMA Engines and Comports

GDD Hand-Coded, Optimized:
- DSP/Vector Library
- Maths Library
- CBLAS Library
- LINPACK Library
- EISPACK Library

SUNDANCE DIGITAL SIGNAL PROCESSING INC. Tel: (775) 827-3103 USA
SUNDANCE MULTIPROCESSOR TECHNOLOGY LTD. Tel: +44 (0)1494 793167 UK
Email: sales@sundance.com  http://www.sundance.com

PCMCIA JTAG Emulator

IDEAL LAPTOP/NOTEBOOK EMULATOR

- FAST:
  As good as most PCI Emulators.

- CONVENIENT:
  Hot pluggable, compact, robust, no external power supplies.

- FLEXIBLE:

- VERSATILE:
  Converter available for desktop use.

"The Predator works great on every Texas Instruments C5000 and C6000 target that I Program." - Bob Davenport, TI.

"I am now the proud user of a Predator JTAG. The Emulator came in today and in a few minutes I was able to use it." - Carlo Treves, TI.

www.kanecomputing.com - Tel: +44(0)1606 351006
A tone relay can eliminate troublesome signaling tone distortion and other problems in packet data networks that employ speech compression.

**Passing Tones in Voice-over-Packet Data Systems**

By Scott Kurtz

Low-rate speech compression algorithms for voice-over-packet data networks can often distort signaling tones excessively. Tone relays and tone relay algorithms especially developed for voice-over-packet and other systems that employ speech compression enable you to avoid the problem. These algorithms address a number of subtle issues that affect robustness and performance. (Even when using one, however, you need to do a significant amount of testing under a variety of conditions to ensure robustness.) A good one should be flexible enough to handle the right set of signaling tones yet simple enough to integrate into a host application.

The idea of carrying speech over packet data networks is gaining acceptance in the world of telephony. Many standards bodies have scrambled to determine the best way to do that over the various types of packet data networks, like IP, ATM, and frame relay. All packet networks have limited bandwidth; for that reason, speech compression is an essential ingredient of voice-over-packet standards, in order to make the best use of the channels.

Speech compression algorithms remove redundancy from speech data by extracting key information from speech signals. This information often includes parameters that model the human vocal tract. A
A good model requires only a few bits to specify the parameters while still providing good speech quality when the speech signal is regenerated. In general, a higher degree of compression (lower bit rate) results in lower speech quality.

Unlike Morse and Huffman coding, speech compression algorithms result in the loss of information. Although good algorithms hold down the loss in perceived speech quality, the lower-rate algorithms aren't adept at passing many non-speech signals, including DTMF, MF R1, MF R2 Forward and Reverse, and Call Progress tones. In fact, many lower-rate speech compression algorithms distort signaling tones beyond the point of reliable detection.

**SYSTEM OVERVIEW**

A tone relay implementation consists of two layers of functionality: the relay encoding and decoding and the underlying tone detection and generation, as shown in Figure 1. The tone relay has a component that operates at the encoding side of the link, the tone relay encoder, and one that operates at the decoding side, the tone relay decoder. In the figure, the input is a sequence of a sample representing the voiceband signal. Normally, the sampling rate is 8,000 samples per second.

The input samples feed both the speech encoder and the tone relay encoder. The speech encoder compresses the speech data, and the tone relay encoder detects the presence or absence of a signaling tone. If a tone is present, the tone relay encoder encodes information about the tone that enables it to be reproduced at the other end of the communications link. The tone relay encoder also issues an Active flag that, when set, indicates that valid tone data is present.

The speech data, tone data, and Active flag form the inputs to the encoded-packet processor, which creates a speech or tone packet in accordance with the appropriate voice-over-packet specification. The packet then goes to the opposite end of the link via the packet network.

The packet-decoding processor receives packets from the packet network and sends compressed speech data to the speech decoder. In some systems, speech data isn't sent when tone data is sent. If that's the case, the processor informs the speech decoder that the frame is missing and the speech decoder acts accordingly. The processor also sends the tone data and the Active flag to the tone relay decoder. If tone activity is present, the tone relay decoder regenerates the original tone based on the parameters included in the packet.

A switch determines whether the decoded speech or the regenerated tone data serves as the output. The tone relay decoder controls the switch, which is set to the speech decoder unless a tone is being regenerated.

**TONE PASSER CHARACTERISTICS**

On the surface, the design of a tone passer might appear straightforward. There are, however, some implementation details, such as frame size, that can have a significant effect on its performance.

Speech compression algorithms operate on a frame-by-frame basis. The input speech is therefore divided into frames. Each frame contains a given number of samples, as defined by the algorithm. Typical frame lengths and their corresponding sample counts are 2.5 ms (20 samples), 10 ms (80 samples), and 30 ms (240 samples).

To reconstruct signaling-tone...
bursts more precisely, a tone relay should use finely specified durations. Consequently, the tone relay encoder must be able to analyze the input signal using small frame sizes so that the burst length is quantized in sufficiently small intervals.

Although a smaller interval allows a more accurately reconstructed tone, more processing power is needed to detect the smaller intervals. A trade-off is therefore necessary.

A good tone relay should generate tone bursts with an accuracy of ±3 ms. The tone burst interval may not coincide with the frame size of the vocoder or with the analysis frame size of the tone detector in the tone relay encoder. Accordingly, the tone relay algorithm must work out those differences.

Amplitude quantization is another important detail. Note that the tone relay quantizes both the time interval and the signal amplitude. Tone detectors in the network must be able to quantize the signal amplitude with quantization intervals of 3 dB or less.

The dynamic range of the amplitude quantizer should span the range called for by the associated signal detector specification. For example, if the detector specification indicates that a detector must detect signals between 0 and -25 dBm, the tone relay amplitude quantizer should span that range.

Leading-edge suppression is another parameter to worry about. To detect a signaling tone reliably without excessive probability of a false alarm, many samples of input data must be analyzed before deciding whether a signaling tone is present. The tone relay encoder output is therefore delayed with respect to the onset of the signaling tone. In a system that employs tone relay, the leading edge of the signaling tone could possibly pass through the vocoder before the tone relay detects the tone.

If a long enough segment of tone passes through the vocoder, two adverse effects may result, as shown in Figure 2. Here, a sequence of two 50-ms tone pulses are the input to the encoding side of a communication link (Figure 2a). The vocoder introduces both delay and distortion. (Figure 2b).

The first effect is that the tone burst that emerges from the decoding side is extended in duration, and distortion remains at the start of the tone burst (Figure 2c). The beginning of the output signal is the portion passed by the vocoder, and the tone relay decoder generates the

---

**Figure 3.** The tone relay encoder function includes tone detectors, which supply status information the encoder uses to produce coded tone data. Similarly, the tone relay decoder function includes a tone generator. The tone generator uses control information from the relay decoder to re-create the tone.
remainder. As a result, the interdigit time is shortened when signaling tones occur in rapid succession. If the interdigit time isn’t met, the detector at the far end can make errors. Leading-edge suppression eliminates the effect, as well as the distortion (Figure 2d).

The second effect is similar to the first. Since it’s presumed that the vocoder distorts the signaling tone, the portion that it passes is distorted, but the portion that the tone relay passes is not. However, a phase discontinuity occurs between the two portions of the output signal. The discontinuity could cause the detector at the far end to detect a split digit or two separate digits for a single input digit.

Consequently, a tone relay should be capable of detecting the leading edge of a signaling tone as soon as possible. When the leading edge is detected, a tone suppressor should be used to remove the signaling tone’s frequency components from the signal going to the vocoder.

Tone suppression could be done more crudely by muting the input to the speech encoder. However, although muting is a simpler solution, it isn’t desirable. The leading-edge detector isn’t as robust as the detector itself because it must make a decision on a shorter-duration signal. The leading-edge detector is therefore more prone to false alarms. Muting the input to the speech encoder when the leading-edge detector goes off makes it more likely that speech will be muted, rather than the signaling tone. So it’s better to suppress only the frequency components that are part of the detected signaling tone. That way, if speech is present, the speech signal isn’t changed too much.

**FROM THEORY TO PRACTICE**

Now let’s take a closer look at a tone relay implementation. Figure 3 breaks down the architecture shown in Figure 1.

The input PCM signal is fed into the tone relay encoder, which in turn feeds it to the various types of tone detectors. The tone detectors return status information including
Tone Relay

A carrier-class DSP-based tone relay software package can demonstrate how tone relay functionality can be integrated into a host application. For the discussion of such software, go to http://edtn.com/cs/EE/tonerelaysw.fhtml

Scott Kurtz (skurtz@adt-inc.com) is the vice president of engineering at Adaptive Digital Technologies, Inc. in Conshohocken, Pa., where he has focused on the development of the company’s DSP based products and technology. He has specialized in digital signal processing and digital communications over the course of his 18-year career, beginning at RCA Corporation’s Government Communications Systems Division. At InterDigital Communications Corporation, he was instrumental in developing the first digital wireless local loop telephone system, which was the precursor to today’s digital cellular telephone systems. He holds six patents, with two more pending.
Analog-I/O DSP Board Stars 16 Independent Channels
Toro, a PCI-based analog-I/O DSP board powered by a 150-MHz TMS320C6711 DSP, features 16 simultaneous, independent channels, plus 32 MB of one-wait-state SDRAM, and a wide choice of triggering modes. Each channel wields dedicated 16-bit, 250-kHz A/D and D/A converters, allowing for simultaneous, nonmultiplexed sampling. A built-in real-time event log precisely records trigger times and user-defined events. Multiple Toro boards can share triggers for additional channel capacity. Toro sells for $2,850. An 8-channel version costs $1,850.


Telecom Software Suites Add eXpressDSP-Compliant Packages
A line echo canceller for the TMS320C54x generation and an adaptive multirate (AMR) encoder-decoder for the TMS320C55x generation are the first in a series of eXpressDSP-compliant additions to the HelloVoice, HelloWireless, and HelloWLAN software suites. For release later this year are both line and acoustic echo canceller, G.729, A-/B encoder-decoder, GPRS PHY layer, and DTMF packages for the C55x DSP, as well as an 802.11x PHY layer for the C64x DSP. License fees range from $75,000 to $125,000 for the encoder-decoder and from $50,000 to $150,000 for the echo canceller. Royalties are negotiable.


Modem and Telephony Development Kits
The Modem Developers Kit and Client-side Telephony Developers Kit for the TMS320C54V90 and TMS320C54CST DSPs allow processing throughputs of up to 2,400 MIPS and can run in a PC or as a stand-alone system. Suited for developing sophisticated telecom systems, it carries a 32-bit PCI interface, 16 MB of SDRAM, and 4 MB of flash memory and accepts H.100 bus connections and XBUS and EMIF daughtercards. It also includes a stereo audio and phone interface. Prices start at $3,350.

DSP Research, Inc., Sunnyvale, Calif.; (408) 773-1042, www dsp.com

Protocol Stack Boasts eXpressDSP Compliance
A serial communications protocol algorithm, bf3Scp helps to design Windows user interfaces for DSP applications. The architecture is based on an ActiveX host protocol engine and an eXpressDSP-compliant target-side protocol engine algorithm. It's available for the TMS320C54x generation and TMS320C6000 DSP platform. A second package, bf3Net, is a TCP/IP protocol stack for the C54x that connects DSP algorithms to the Internet. Its modular architecture allows for customization; Code Composer Studio plug-ins speed configuration, code generation, and debugging. License fees are $7,500 for bf3Scp and $20,000 for bf3Net.

In recent years, the way that DSP software applications are developed has undergone a major transformation. With the number of functions performed by a DSP exploding, writing the majority of code in hand-optimized assembly language has become impractical, and most DSP code is now written in C.

As applications continue to grow in complexity, developers are turning increasingly to real-time operating systems as well. Indeed, DSP RTOSs are coming of age, but more is needed for complete solutions, in particular, device drivers, frameworks, and algorithms.

Three years ago, few DSP system developers used commercial RTOSs. Since then, their use has grown so rapidly that, for example, over half of Texas Instruments’ customers now use a commercial RTOS.

Two trends are driving the adoption of DSP RTOSs. In some applications, DSPs perform all system control functions in addition to signal processing and so need an RTOS. Cost-sensitive consumer electronic applications, such as MP3 players, where eliminating the microcontroller reduces the overall system cost, are good examples.

Most DSP applications, however, use one or more DSPs communicating with a microprocessor. In those cases, the DSP executes multiple signal processing functions, like speech compression and dial tone detection. For maintainable multi-functional systems, a multithreading DSP RTOS is ideal.

Given the size and performance constraints of most DSP applications, most developers today prefer to use a real-time DSP kernel rather than more heavyweight RTOS kernels typical of microprocessor applications. Obviously, simply porting a standard microprocessor or microcontroller kernel to a DSP is inappropriate for most applications. Unlike their cousins, DSP kernels must scale to provide basic multithreading support in just a few kilobytes. In addition, interrupt latency and interrupt-to-task time are critical, since unlike microprocessors, DSPs almost always process streams of real-time data.

Now, the market is starting to demand more complete solutions. To meet those demands, DSP RTOSs need to evolve beyond simple kernels to address three key areas—device drivers, frameworks, and algorithms.

Device drivers are notoriously difficult to debug and test. Moreover, no DSP application can be seriously tested on the embedded target without working drivers. Working drivers for a DSP’s on-chip peripherals and commonly used external peripherals enable developers to bring up an application on real hardware much faster. They also eliminate the need for a significant effort to develop software that doesn’t differentiate the final product.

Frameworks are particularly critical for developing robust DSP applications quickly. They’re the glue that integrates algorithms, I/O, and control code and are ultimately responsible for managing system resources efficiently. By nature, frameworks are application-specific. Successful DSP RTOSs must therefore offer a range of frameworks optimized for different types of applications.

Frameworks must also provide standardized interfaces for integrating proprietary as well as third-party algorithms. With DSP applications now supporting multiple functions, more development teams are buying algorithms from third-party providers. By simplifying the process of integrating algorithms from multiple sources, frameworks can help reduce the system integration time, which already consumes much of the product development cycle.

Nick Lethaby is product manager for the DSP/BIOS real-time kernel at Texas Instruments, Inc’s Software Development System group in Santa Barbara, Calif.
Features of the DspIceLVx include:

- High-level language debugging support (as per XDS510)
- Code Composer Studio, RTDX, DspBios etc., Support
- Code Composer Studio PDM (Parallel Debug Manager) Support
- Heterogenous (mixed) and Homogenous processor support
- TI DSPs and ARM support (simultaneously)
- Rugged compact design
- PCI bus PnP support
- Variable TCLK from 20Mhz to 158Hz
- 1.0V - 5.0V processors (auto-detect)
- Supports 1 processor JTAG scan chain as per XDS510 with multiple processors in a “daisy chain”

AND/OR

- Supports up to 16 separate JTAG scan chains (not daisy chain)
  - With multiple processors in a “daisy chain”
  - All from one PCI slot

Processor Support

<table>
<thead>
<tr>
<th>JTAG</th>
<th>TMS470</th>
<th>TMS320 DSP Family</th>
</tr>
</thead>
<tbody>
<tr>
<td>TI_14Pin</td>
<td>OMAP</td>
<td>C20x</td>
</tr>
<tr>
<td></td>
<td>ARM7</td>
<td>C24x</td>
</tr>
<tr>
<td>MPSD_12PIN</td>
<td>ARM9</td>
<td>C27xx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C28xx</td>
</tr>
<tr>
<td>DspIceLVx</td>
<td>✔</td>
<td>✔/✔/✔/✔/✔/✔</td>
</tr>
<tr>
<td>DspIceLV</td>
<td>✔/✔/✔/✔/✔/✔/✔/✔</td>
<td>✔/✔/✔/✔/✔/✔/✔/✔</td>
</tr>
<tr>
<td>DspIce(mpsd)</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Software Support

- TI Code Composer Studio™ CCS1.2 CCS2.0
- Texas Instruments trademarks XDS510™, TMS320™, TMS470™, RTDX™, DSBios™, Code Composer Studio™
- ARM Limited trademarks ARM™, ARM9™

Softronics Emulators

- xICE* -> PCI Bus Multi-Target Emulator
- nICEx -> 100baseT Ethernet Emulator
- DspIce® -> PCI Bus Emulator
- Ice*Pack -> ISA (16 bit) Bus Emulator
- Mice*Pack -> Parallel Port Emulator
MATLAB and Code Composer Studio.
Together at last.

The leaders in DSP development now work together. With the new Developer's Kit for Texas Instruments DSP from The MathWorks, you can execute Code Composer Studio™ functions in MATLAB to interactively debug applications, analyze real-time data, and automate testing on TI DSPs. And you can streamline implementation by generating real-time prototypes and Code Composer Studio projects from Simulink.

Demos, tutorials, and trials are available online.
See the fastest way to develop, test, verify, and implement DSP software. Visit www.mathworks.com/eme.

The MathWorks
Visit www.mathworks.com/eme or call 508-647-7040
Visit our web site for information on training courses online, on-site, and at public locations around the world. All trademarks are property of their respective owners.
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

Copyright © 2002, Texas Instruments Incorporated