

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.

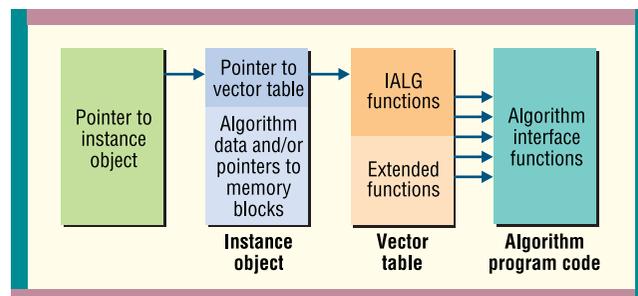
As an algorithm developer, you can quickly convert existing software components into eXpressDSP-

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 reentrant 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 ZAP_SIGSOFT_alloc(
    const IALG_Params *algParams,
    IALG_Fxns **fxns,
    IALG_MemRec memTab[]
)
{
    memTab[0].size = sizeof(ZAP_SIGSOFT_Obj);
    memTab[0].alignment = 8;
    memTab[0].space = IALG_DARAM0;
    memTab[0].attrs = IALG_PERSIST;

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

Int ZAP_SIGSOFT_free(
    register IALG_Handle handle,
    register IALG_MemRec memTab[]
)
{
    memTab[0].base = handle;

    return G729DEC_SIGSOFT_alloc(NULL, NULL, memTab);
}

Int ZAP_SIGSOFT_initObj(
    IALG_Handle handle,
    const IALG_MemRec memTab[],
    IALG_Handle p,
    const IALG_Params *algParams
)
{
    ZAP_SIGSOFT_Obj *inst = (ZAP_SIGSOFT_Obj *) handle;

    If (params==NULL) params = IZAP_PARAMS;

    ZAP_SIGSOFT_initialize(&inst->algMem, params->param1,
        params->param2);

    return IALG_EOK;
}

Void ZAP_SIGSOFT_moved(
    IALG_Handle handle,
    IALG_MemRec memTab[],
    IALG_Handle p,
    const IALG_Params *algParams
)
{
}

Int ZAP_SIGSOFT_numAlloc(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

```
XDAS_Bool ZAP_SIGSOFT_control(
    IZAP_Handle handle,
    IZAP_Cmd cmd,
    IZAP_Status *status
)
{
    ZAP_SIGSOFT_Obj *inst = (ZAP_SIGSOFT_Obj *)handle;
    XDAS_Bool retval = XDAS_TRUE;

    switch (cmd)
    {
        case IZAP_GETSTATUS:

            /* Copy status information from instance into status */

            status->param1 = inst->algMem.param1;
            status->param2 = inst->algMem.param2;

            break;

        case IZAP_SETSTATUS:

            /* Copy status information from status into instance */

            inst->algMem.param1 = status->param1;
            inst->algMem.param2 = status->param2;

            break;

        default:

            retval = XDAS_FALSE;

            break;
    }

    return retval;
}

Void ZAP_SIGSOFT_process(
    IZAP_Handle handle,
    XDAS_Int16 in[],
    XDAS_Int16 out[]
)
{
    ZAP_SIGSOFT_Obj *inst = (ZAP_SIGSOFT_Obj *) handle;

    ZAP_SIGSOFT_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

TCP/IP *for*

TI DSPs

Delivering precisely
what you need.

Since 1994, Precise has been a leading
supplier of embedded Internet protocols on
Texas Instruments' TMS320[™] DSP Family.

TMS320C3x[™]

1994

TMS320C4x[™]

1995

TMS320C62x[™]

1998

TMS320C67x[™]

1999

TMS320C54x[™]

1999



Precise Software Technologies Inc.
301 Moodie Drive, Suite 308
Nepean, Ontario
Canada K2H 9C4
Sales: 800-265-9833
Phone: 613-596-2251
Fax: 613-596-6713
E-mail: info@psti.com



www.psti.com

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 3: Normal test harness

```
void main()
{
    Int      InBuf[100];
    Int      OutBuf[100];
    Int      FrameCount;
    FILE     *f_in;
    FILE     *f_out;
    sXyx     ZapMem;                /* algorithm memory */

    InitializeZap(&ZapMem,10,20);    /* initialize algorithm */

    while( (fread(in, sizeof(Int), 100, f_in) == 100 )
    {
        ProcessZAP(&ZapMem, in, out); /* process data */

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

    fclose(f_in);
    fclose(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-

PLX PCI + TI DSP



"PLX definitely hit the mark with the 480 IO processor. By combining onchip PowerPC intelligence and a flexible bus architecture tuned to the C62 expansion port, PLX created a winner for DSP-on-PCI applications. Add a turn-key development system with reference drivers, firmware and schematics, and you have taken PCI-C62 design tools to the next level. PLX gets it."

Ted Raineault — Electric Sand



Find out more about PLX PCI + TI DSP at —

www.plxtech.com/tidsp

©2001 PLX Technology. The PLX logo is a registered logo of PLX Technology. All other company and product names are the property of their respective holders. All rights reserved.

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()
{
    XDAS_Int16  InBuf[100];
    XDAS_Int16  OutBuf[100];
    XDAS_Int16  FrameCount;
    FILE        *f_in;
    FILE        *f_out;
    ZAP_Params  ZapParams; /* parameter structure */
    ZAP_Handle  ZapHandle; /* algorithm handle */

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

    ZapParams.param1 = 10; /* place parameters in structure */
    ZapParams.param1 = 20;

    ZapHandle = ZAP_create(&ZAP_SIGSOFT_IZAP, &ZapParams);
    /* create and initialize algorithm */

    while( (fread(in, sizeof(XDAS_Int16), 100, f_in)) == 100 )
    {
        ZAP_apply(ZapHandle, in, out); /* process data */

        fwrite(out, sizeof(XDAS_Int16), 100, f_out);
    }

    ZAP_delete(ZapHandle);

    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.

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