TMS320 DSP Algorithm Standard Developer's Guide

Literature Number: SPRU424
June 2000

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

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

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

Contents

Figures

Tables

TMS320 DSP Algorithm Standard Developer's Guide

This document, along with the latest set of TMS320 DSP Algorithm Standard (referred to as "Algo Standard" throughout the document) Rules and Guidelines (literature number SPRU352), will help assist the algorithm developer with implementing the Algo Standard interface, as well as creating a test application. When the Algo Standard conversion is complete, the algorithm should conform to all the Algo Standard rules. Also required is the Algo Standard Code Gen Tool plug-in for Code Composer Studio. This greatly simplifies the Algo Standard development effort. To download the latest Algo Standard developer's kit:

From the TI DSP Developer's Village (http://dspvillage.ti.com), follow the link to the TMS320 DSP Algorithm Standard, and look in the documentation section for a link to the latest version of the standard.

The following procedure consists of seven sections. The first one requires the developer to determine the present state of the algorithm, decide on module names, and answer some preliminary questions. The second, third, fourth, and fifth sections give the developer some detailed insight into the Algo Standard implementation details by leading him/her through the process of acting as consumer, interface definer, and algorithm vendor. The sixth section describes how to create the Algo Standard library deliverable. The seventh section describes how to characterize the Algo Standard algorithm. Most of the procedures contain examples based on a G.729 encoder algorithm developed by Texas Instruments.

1 Preliminary Algo Standard Questions (Rules 1–10)

This part of the process focuses the algorithm developer on the operation of the algorithm and how the Algo Standard interface 'fits' onto it. This section verifies that the algorithm is in compliance with Rules 1–10 of the Algo Standard Rules and Guidelines document.

1) Is the testing environment already set up? There should be an application program that calls the algorithm to process a test vector(s) and generates an output vector(s). The output vector(s) is then compared to a known 'good' vector(s) for correctness. Make sure this test environment is set up and the algorithm successfully passes the test process before proceeding.

2) The original algorithm must conform to the first 10 rules stated in the Algo Standard Rules and Guidelines document. It is a good idea to verify this step before proceeding, but it could be temporarily postponed for the sake of expediting the Algo Standard process. Rule #8 (External Identifiers) can be skipped for now—it will be addressed later in the build procedure.

3) What are the names for the module, version (optional), vendor, architecture, and model? If there is more than one version of the algorithm, append it to the module name (refer to Algo Standard Rules and Guidelines document, section 3–18). Use Appendix A to record this information and refer to Table 1 for an example. For the C54/55x architectures, it is recommended to supply both near and far mode memory models. For the C6x architectures, all algorithms must, as a minimum, be supplied in little endian format, but it is recommended to supply a big endian model, as well. The different variants/models should not produce different interfaces for the same algorithm; the only difference will be the way the object files are compiled within each library variant/model (and the file name of each library).

4) Construct the Memory Table (memTab[]). Go to Appendix C and create a table based on the Instance Memory Table.

Table 1. Example Table for Naming the Module, Vendor, Variant, Architecture, and Model

Module Name	Vendor Name	Variant	Architecture	Variant (optional)
G729ENC	TI	None	[54 | 55 | 62 | 64 | 67]	[far (f) | mixed (m) | big endian (e)]

2 Module-Specific Abstract Interface (IMODULE)

If there are sample APIs in the Algo Standard Developers Kit which exist for your algorithm, please use them (these will be the i<MODULE>.[ch] files in “c:\ti\xdas\src” directories). If not, you must act as the Interface Definer (Service Provider) for your algorithm and create these files which are used by the application. Determine what the extended methods of the IALG interface will be for this algorithm. These are typically the module-specific “process and/or control” functions. This interface must be extended by at least one process method to allow the algorithm to be called, but any number of them can be defined in this section. Typically, **apply** and **mycontrol** methods are defined here, in this case we use **encode** and **mycontrol**. Use the following example as a guideline for naming the extended methods for the Algo Standard algorithm. These module-specific methods, along with the traditional IALG methods, make up the SPI (Service Provider Interface) which is supplied to the consumer/system integrator of the application.

Table 2 shows an example of how to enter the newly named methods in Table A–4 of Appendix A. Also, note that the *IALG_Handle* parameter does not need to be entered in this table (the code generation tool will add it to all functions automatically as the first parameter). “XDAS” types are encouraged for the module-specific interface (most likely located in “c:\ti\xdas\include\xdas.h”).

Table 2. Example Table for Naming Extended IALG Methods

No.	Return Type	Method Name	Param1 Type and Name	Param2 Type and Name	Param3 Type and Name
1	XDAS_Bool	mycontrol	IG729_Cmd cmd	IG729ENC_Status *status	N/A
2	XDAS_Int8	encode	XDAS_Int16 *in	XDAS_Int8 *out	N/A

Determine the interaction of the algorithm with the application (or framework). In particular, what are the instance creation parameters and what are the real-time status/control parameters? The instance creation parameters are values passed to the algorithm when it is instantiated (i.e., only one time) and the status/control parameters are algorithm status information that is read and/or written while the algorithm is in operation. The complex types that deal with these parameters are automatically generated by the “Algo Standard Code Generation plug-in tool” using the names specified by the user. The source code listed below is an example output of the Algo Standard Code Generation Tool. The developer simply supplies the structure member names. Determine the appropriate parameter names and enter Instance Creation Parameters in Table A–2 and the Real-Time Status and Control Parameters in Table A–3 in Appendix A. In both cases, the *size* parameter does not need to be entered

in the table because it is automatically generated by the Algo Standard Code Gen Tool as the first field of all Params and Status structures.

The *size* field **must** be the first field of all Params and Status structures. This variable is typically used to determine whether a Params or Status structure has been extended by the Interface Definer. When Params and Status structures are extended (i.e., more fields are added), the original set of members must still work, so any IALG function will be able to tell which structure (original or extended) to use based on the value stored in the *size* field.

```
/* ig729enc.h */
#ifndef IG729ENC_
#define IG729ENC_
#include <ialg.h>
#include <ig729.h>
/*
 *  ======== IG729ENC_Obj ========
 *  This structure must be the first field of all G729ENC instance objects.
 */
typedef struct IG729ENC_Obj {
    struct IG729ENC_Fxns *fxns;
} IG729ENC_Obj;
/*
 *  ======== IG729ENC_Handle ========
 *  This handle is used to reference a G729ENC instance object.
 */
typedef struct IG729ENC_Obj *IG729ENC_Handle;
/*
 *  ======== IG729ENC_Params ========
 *  This structure defines the creation parameters for all G729ENC instance ob-
jects.
 */
typedef struct IG729ENC_Params {
Int size;    /* MUST be the first field */
Int frameLen;
Int pfo;
Int vad;
} IG729ENC_Params;
/*
```

```
 *  ======== IG729ENC_Status ========
 *  This structure defines the status parameters or values that can be
 *  read and/or written while the algorithm is ‘live’.
 */
typedef struct IG729ENC_Status {
Int size;    /* MUST be the first field */
Int maxChannels; /* Can be read/write */
Int frameLen;    /* Can be read/write */
Int signalStatus;   /* Read only */
} IG729ENC_Status;
/*
 *  ======== IG729ENC_Fxns ========
 *  This structure defines all of the operations on G729ENC objects.
 */
typedef struct IG729ENC_Fxns {
    IALG_Fxns   ialg;
    XDAS_Bool (*mycontrol)(IG729ENC_Handle handle, IG729_Cmd cmd, IG729ENC_Status
*status);
    XDAS_Int8 (*encode)(IG729ENC_Handle handle, XDAS_Int16 *in, XDAS_Int8 *out);
} IG729ENC_Fxns;
#endif  /* IG729ENC_ */
```

3 Vendor-Specific Algorithm Interface (IALG)

When properly followed, the steps outlined in this section automatically enforce Rules 11–17 of the Algo Standard Rules and Guidelines document. When finished with this section, review Rules 11–17 to make sure that they are all satisfied.

1) Create the build directory and copy the existing test application source files and algorithm object files into this directory. This directory name will also be used as the Project Location entry in the Algo Standard Code Gen Tool.

2) Now we have all the necessary information to generate the Algo Standard template code using the Algo Standard Code Gen Tool. A representation of the Algo Standard Gen Tool Dialog box is shown in Figure 1 on page 7 with some sample entries. If not already launched, start Code Composer Studio. Launch the Algo Standard Code Gen Tool plug-in by following this link on the CCS toolbar: **Tools –> XDAIS –> Template Code Generator**. Use the following seven guidelines for entering the information for your particular algorithm. The Create Template button invokes the plug-in to generate the template code and create a Code Composer Studio project based on the information in the Dialog box.

   a) Algorithm Name – comes from Appendix A, Table A–1, Module name entry

   b) Vendor Name – comes from Appendix A, Table A–1, Vendor name entry

   c) Project Location – comes from step 2 above

   d) Instance Creation Parameters – comes from Appendix A, Table A–2 entries

Note:

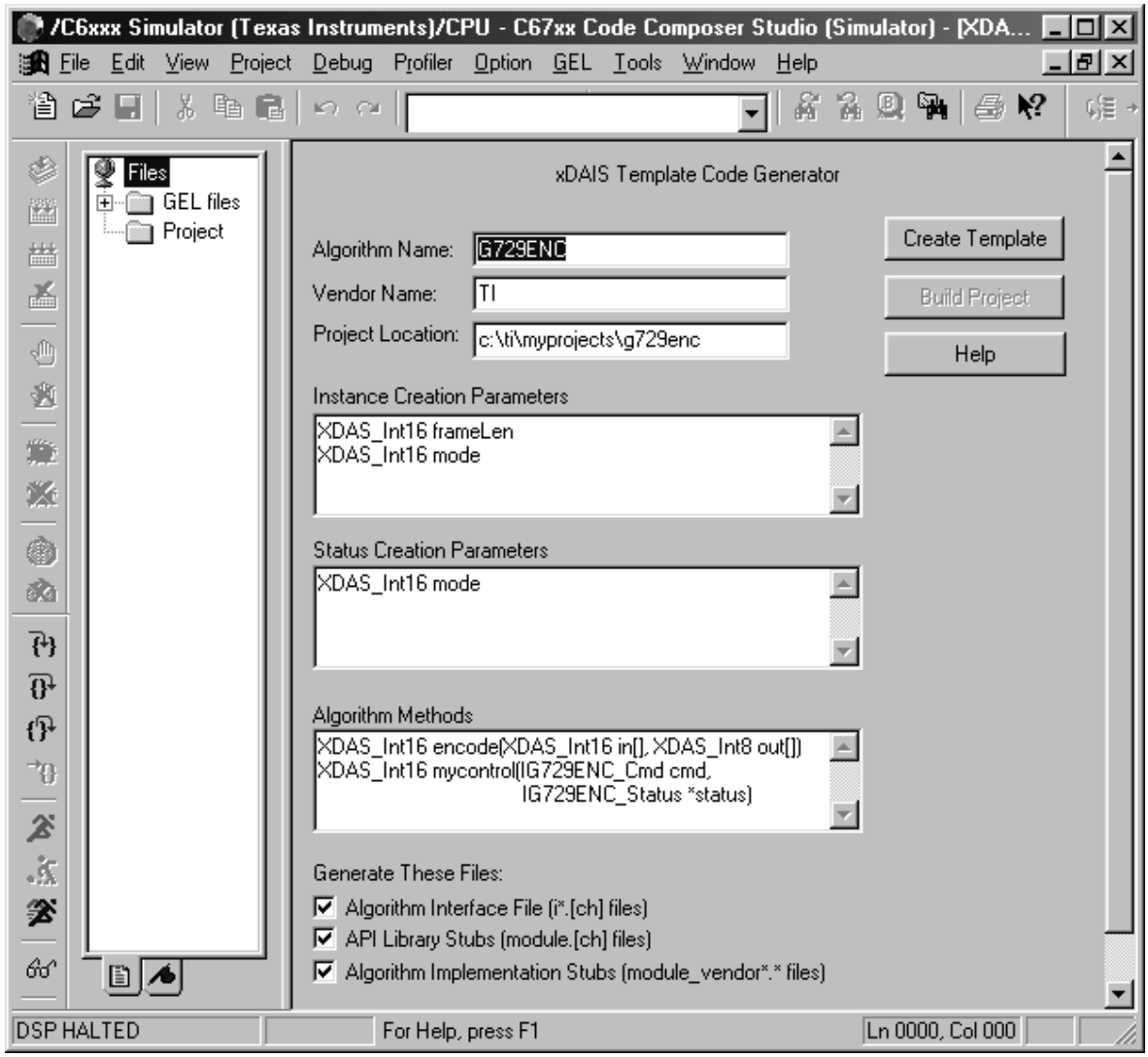
When entering this information, do not use semicolons because the tool adds them automatically.

   e) Status Creation Parameters – comes from Appendix A, Table A–3 entries

   f) Algorithm Methods – comes from Appendix A, Table A–4 entries

   g) If you are using the TI-supplied recommended module-specific interface for your algorithm (IMODULE), check ONLY the “Algorithm Implementation Stubs” checkbox. Otherwise, check all 3 boxes at the bottom of the dialog window.

Figure 1. Algo Standard Template Code Generator Tool Interface



3) The Algo Standard files will be generated and put into the newly created project specified by the Project Location field. The default location is “c:\ti\myprojects\<MODULE>_<VENDOR>” directory if nothing is typed in this field. Refer to Appendix B for a description of the files generated by the Template Generator. Place a copy of the following files in the project directory (the alg_<xxx>.c files are usually found in “c:\ti\xdas\src\api”): alg_create.c, alg_malloc.c, and the file(s) that contain the main() application code to the project directory. If implementing the sample APIs in the Algo Standard Developer’s Kit, copy the corresponding i<MODULE>.[ch]

& <MODULE>.[ch] files (e.g. ig729enc.[ch] and g729enc.[ch]) into the project directory (since they were not generated by the tool).

4) Add all the following files to the newly created project:

   - alg_create.c and alg_malloc.c
   - The object files containing the original vendor algorithm
   - rts[dsp].lib (the DSP-specific run time system library)
   - The framework or application containing the main() function

5) Change the build options so the 'Include Search Path' contains the following paths; order is important: <path for project directory>, <path for any header files required for original algorithm>, and "c:\ti\xdas\include". Follow **Project –> Options** and add these project build changes.

6) Edit the i<MODULE>.c file. This contains the Params structure default values. Fill in each default value with a reasonable value as the tool puts in zero by default for each parameter.

7) Rename the <MODULE>_<VENDOR>.c file to <MODULE>_<VENDOR>_ialg.c, and edit the file:

   - Add the following #pragma statements to the existing group of #pragma statements at the top if the algorithm implements these interfaces:

```
#pragma CODE_SECTION(<MODULE>_<VENDOR>_activate,
".text:algActivate")
#pragma CODE_SECTION(<MODULE>_<VENDOR>_deactivate,
".text:algDeactivate")
#pragma CODE_SECTION(<MODULE>_<VENDOR>_moved,
".text:algMoved")
```

   - Go to the <MODULE>_<VENDOR>_Obj structure and add the object data types required by the algorithm. This is usually the existing algorithm handle, but can include other items.

   - Complete the <MODULE>_<VENDOR>_alloc() function by entering the algorithm memory requirements. A code sample gives an example of how to do this. If memory alignment is needed, set memTab[].align = (byte_boundary * 8) / (BITS_PER_CHAR); for c54/55x, BITS_PER_CHAR = 16, for c6x, BITS_PER_CHAR = 8. If memory alignment for a particular block is not needed, set the *align* field to 0. The units for the memTab[].align field are in Minimum Addressable

Units (MAUs). Each memTab[] record should map to an entry in the Instance Memory Table created from Appendix C.

```
/* Request memory for a G729ENC object */
memTab[0].size = sizeof(G729ENC_TI_Obj);
memTab[0].alignment = 0;
memTab[0].space = IALG_EXTERNAL;
memTab[0].attrs = IALG_PERSIST;
/* Request aligned scratch memory */
memTab[1].size = 256;    /* size in bytes */
memTab[1].alignment = 32;/* [(byte_boundary * 8) /
BITS_PER_CHAR] */
memTab[1].space = IALG_DARAM0;
memTab[1].attrs = IALG_SCRATCH;
<additional memTab[] entries>

return(MEMTAB_NRECS); /* The return value must match
the memTab array size */
```

This function is used by the framework to get the memory requirements from the algorithm. Each memTab[] record contains the memory requirements for a block of memory which is needed by the algorithm. The first entry in the memTab[] array (index 0) is *always* reserved for the memory requirements for a single algorithm instance object. The remaining entries can be used to communicate memory requirements for aligned working/scratch/history buffers, etc. that the algorithm instance needs in addition to itself in order to execute correctly. This function returns the number of records in the memTab[] array so that the framework knows how many blocks of memory it needs to allocate. Once the framework has successfully allocated the required block(s) of memory, it needs to update the *base* fields in each of the records and then pass this updated memTab[] into the <MODULE>_<VENDOR>_initObj() function.

- Complete the <MODULE>_<VENDOR>_initObj() function by placing calls to the vendor algorithm initialization routines. Also, initialize the algorithm state with data from the incoming Params structure and memTab[].base fields:

  ```
  g729enc–>frameLen = params–>frameLen;
  g729enc–>workBuf = memTab[1].base;
  ```

- The <MODULE>_<VENDOR>_free() function has the minimum code required. Add any other clean up code for the algorithm after the ‘n = alloc(…)’ statement. It is the application’s responsibility to perform any memory allocation/deallocation, and this function is used by the applica-

tion *only* to ”query” the memory resources previously allocated and assigned to the algorithm instance object, which *may* subsequently be reclaimed. This IALG function *cannot* actually free any algorithm instance memory. These statements will most likely look like the reverse of the statements in initObj():

```
memTab[1].base = g729enc–>workBuf;
```

This is how the algorithm communicates which blocks of memory can be freed by the framework. This function updates the *base* fields of the memTab[] array records and passes them back to the framework via the pointer to memTab[].

- Implementing the <MODULE>_<VENDOR>_moved() function is not required by the Standard, however, if implemented it can be used by the system integrator to more flexibly manage application memory resources by moving algorithm instance data. This basically contains the same memTab[].base statements as the ones found in initObj(), for example:

  ```
  g729enc–>workBuf = memTab[1].base;
  ```

  This function is called by the framework to inform the algorithm instance that a particular block (or blocks) or memory has (have) been moved. The framework communicates these updated base addresses by passing a memTab[] array with updated *base* fields into the moved() function.

- Implementing the <MODULE>_<VENDOR>_activate() function is not required by the Standard, however, it may be needed to set up scratch memory from persistent memory which was saved during the last process function call. For example, a vocoder processing function may need to do its calculations based on the results of the last process call. It is the framework’s responsibility to make sure that the scratch/persistent memories do not get corrupted during the lifetime of the algorithm instance. The algorithm must assume that memory is uncorrupted at all times.

  Void G729ENC_TI_activate(IALG_Handle handle)

  ```
  /* Copy encoder history from external slow memory
  into working buffer */
  {
      G729ENC_TI_Obj *g729enc = (Void *)handle;
      /* copy saved history to working buffer */
      memcpy((Void *)g729enc–>workBuf, (Void *)g729enc–>histo-
  ry, g729enc–>frameLen * sizeof(Int));
  }
  Void G729ENC_TI_deactivate(IALG_Handle handle)
  ```

```
/* Copy encoder history from working buffer to ex-
ternal slow memory */
{
    G729ENC_TI_Obj *g729enc = (Void *)handle;
    /* copy history to external history buffer */
    memcpy((Void *)g729enc->history, (Void *)g729enc->work-
Buf, g729enc->frameLen * sizeof(Int));
}
```

The same can be said for the <MODULE>_<VENDOR>_deactivate() function. Its only purpose is to save the current contents of any scratch memory to persistent memory needed for the next processing function call. Think of algActivate() and algDeactivate() as the "context switching functions" at the algorithm level. The algActivate() will prepare the scratch memory before the actual processing function is called, and the algDeactivate() will save the scratch memory after the processing call exits for the next iteration. In other words, the framework needs to call algActivate() before calling the process function and then call algDeactivate() after the process function exits:

```
XDAS_Int16 G729ENC_apply(G729ENC_Handle handle,
XDAS_Int8 *in, XDAS_Int16 *out)
{
ALG_activate((IALG_Handle)handle);
handle->fxns->encode(handle, in, out);
ALG_deactivate((IALG_Handle)handle);
}
```

- Complete the extended IALG methods at the end of this file. The **apply** method (or in this case, **encode**) is where the vendor's original algorithm code is called from. Usually, the **mycontrol** method also needs to be completed and an example of how to do this follows:

```
XDAS_Bool G729ENC_TI_mycontrol(IG729ENC_Handle han-
dle, IG729ENC_Cmd cmd, IG729ENC_Status *status)
{
    G729ENC_TI_Obj *g729enc = (Void *)handle;

    if ( cmd == IG729ENC_GETSTATUS )
    {
      status->maxChannels = g729enc->maxChannels;
   status->signalStatus = g729enc->signalStatus;
    }
    else if ( cmd == IG729ENC_SETSTATUS )
```

```
        {
          g729enc->maxChannels = status->maxChannels;
          g729enc->frameLen = status->frameLen;
        }
        else /* Invalid command */
        {
            return (XDAS_FALSE);
        }
        return (XDAS_TRUE);
    }
```

8) Rename the <MODULE>_<VENDOR>_vtab.c file to <MODULE>_<VENDOR>_ialgvt.c.

9) Add the following function definitions to the bottom of the <MODULE>_<VENDOR>.h file, before the final #endif statement:

```
/*
 *  ======== <MODULE>_<VENDOR>_init ========
 *  Initialize the <MODULE>_<VENDOR> module as a whole.
 */
Void <MODULE>_<VENDOR>_init(Void);

/*
 *  ======== <MODULE>_<VENDOR>_exit ========
 *  Exit the <MODULE>_<VENDOR> module as a whole.
 */
Void <MODULE>_<VENDOR>_exit(Void);
```

4 Consumer/System Integrator (CONCRETE API)

This section describes how to create sample consumer (framework) files to build a client application to test our Algo Standard interface. These files make up the Concrete API and are *not* required for Compliance Testing submission, but are definitely helpful. Algo Standard Rules 25–30 will be addressed.

1) Modify the test application main() function to call into the Algo Standard interface functions rather than the algorithm directly. We can use the type definitions and functions defined in our sample <MODULE>.[ch] files to help us manage Algo Standard objects. The following is a common example:

```
#include <std.h>
#include “g729enc.h”
#include “g729enc_ti.h”
G729ENC_Params params;
G729ENC_Handle alg;
G729ENC_Status status;
G729ENC_Cmd cmd;
Int* sample;
Short* output;

/* Create an instance of an Algo Standard object */
if ((alg = G729ENC_create(&G729ENC_TI_IG729ENC, &pa-
rams)) != NULL)
   G729ENC_encode(alg, sample, output);
/* To check the status, make the following call */
cmd = G729ENC_GETSTATUS;
G729ENC_mycontrol(alg, cmd, &status);
/* To set a control parameter, make the following call
*/
cmd = G729ENC_SETSTATUS;
status->maxChannels = 32;
G729ENC_mycontrol(alg, cmd, &status);
/* To delete this instance, make the following call */
G729ENC_delete(alg);
```

2) Before building the project, there is a set of processor-specific rules that must be followed. The goal is to verify that Rules 25 – 27 are followed for C6x; Rules 28–30 are followed for C54/55x. The following step(s) address how to configure your project to satisfy these DSP-specific rules:

   - For C6x, Rule 26: “All C6x algorithms must access all static and global data as far data”. The ‘–ml3’ cl6x compiler option defaults all data and functions to far.

- For C54/55x, do the following extra steps if a ‘far’ model is desired: In the Compiler build options, check the “Use Far Calls” box and type “548” in the Processor Version field under the Code Gen (II) Category. In the Assembler build options, check the “Define __far_mode Symbol” box and type “548” for Processor Version.

3) Replace all instances of CALL with [.if __far_mode FCALL .else CALL .endif] in all assembly source files.

4) Replace all instances of RET with [.if __far_mode FRET .else RET .endif] in all assembly source files.

5) For far mode run-time support, include the library “rts_ext.lib” into the project.

   NOTE for far mode compilation: The FCALL instruction pushes **two** words onto the stack, as opposed to one word for the near CALL instruction. If you have any C run-time convention code in assembly that pops items off the stack during the return of an assembly function that has been called by a C function, make sure to modify all your assembly modules to account for this.

   - Finally, for all C54/55x object files, make sure the size of each object file is **less than or equal to 32K words**.

6) Build the project. Run the program with the same test vectors you normally use to test your algorithm. The newly “Algo Standardized” algorithm should execute with the same results as before.

5 OPTIONAL: Real-Time Trace Control Interface (IRTC)

Now that we have implemented the IALG interface and created a test client to check it, we can now implement an additional interface that will allow the algorithm to take advantage of the Real Time Analysis capabilities of DSP/BIOS. IRTC defines an interface, that when implemented, allows a module's various RTA modes to be enabled, disabled, and controlled in real time.

1) Make a copy of the file "rtc.c" from the Algo Standard Developers Kit directory (normally c:\ti\xdas\src\api). Add this file to the project.

2) Copy over "fir_ti_irtcvt.c" and "fir_ti_irtc.c" from c:\ti\xdas\src\filter, and then copy over "fir_ti_priv.h" from c:\ti\xdas\include. Rename each file to reflect <MODULE>_<VENDOR> and replace all internal references in all 3 files (make sure to match the case in each operation):

   - "fir_ti" to lower case <module>_<vendor>" (do *not* match whole word)
   - "FIR_TI" to upper case <MODULE>_<VENDOR>
   - "fir" to lower case <module>
   - "FIR" to upper case <MODULE>
   - "FIR filter" & "Filter" to upper case <MODULE>
   - "ti" to lower case <vendor> (match whole word)
   - "TI" to upper case <VENDOR> (do *not* match whole word)
   - "mask" to "biosMask"

Add the 2 C source files that were just modified to the project. Also, in "irtc.h", make sure IRTC_Handle points to IALG_Obj type, NOT IRTC_Obj type.

3) In the current <MODULE>_<VENDOR>.h file, add the following:

```
#include <irtc.h>
#include <log.h>
/*
 *  ======== <MODULE>_<VENDOR>_IRTC ========
 *  <VENDOR>'s implementation of the IRTC interface for
<MODULE>
 */
extern IRTC_Fxns <MODULE>_<VENDOR>_IRTC;
```

4) In the current <MODULE>_<VENDOR>_ialg.c file, add the following lines to the top of the file and to the <MODULE>_<VENDOR>_Obj structure, respectively:

```
#include <<MODULE>_<VENDOR>_priv.h>
IRTC_Mask biosMask; /* Current DSP/BIOS RTA mask setting
*/
```

Move the entire structure definition to the top of the “<MODULE>_<VENDOR>_priv.h” file (in place of the existing <MODULE>_<VENDOR>_Obj), and then add “#include” statement for the original algorithm’s header file. Delete the external function declarations for “filter” and “tst”.

1) Create a DSP/BIOS LOG Object. If no DSP/BIOS CDB file exists for the project, under the CCS toolbar, go to the link: **File –> New –> DSP/BIOS Configuration**. Create a LOG Object by right-clicking the LOG Event Manager and select “Insert LOG”, then rename the object to “trace”.

2) If a linker command file already exists for your project, comment out the MEMORY sections in the original linker command file. In the DSP/BIOS configuration tool, select the Memory Section Manager (MEM Module). Edit the IPRAM and IDRAM sections to match those defined in the original linker command file by right-clicking on each of the MEM Objects and selecting “Properties” (if needed). Define additional MEMORY sections by right-clicking on the MEM Module and selecting “Insert MEM”.

3) Save this DSP/BIOS configuration file as “<MODULE>_<VENDOR>.cdb” (or whatever name is desired), and then close the file. Add this DSP/BIOS CDB file to the project, and then add “<MODULE>_<VENDOR>cfg.cmd” to the project; if a linker command file already exists, make the following changes to it:

   - Add “–l <MODULE>_<VENDOR>cfg.cmd” at the top of the original linker command file.
   - Comment out the MEMORY section (<MODULE>_<VENDOR>cfg.cmd already specifies its own memory map based on DSP/BIOS config tool). If there are memory section conflicts, they need to be resolved between the two linker command files.
   - Comment out the SECTIONS in the original linker command file.
   - Close the file and save the changes.
   - Remove the “rts[dsp].lib” file from the project, since the DSP/BIOS CMD file automatically links in the C-run time support library.

4) Add the <MODULE>_<VENDOR>_trace() macro anywhere in the original algorithm source code where it makes sense to output diagnostic messages to the LOG Object. Customize the diagnostic messages at each point in the algorithm so that the diagnostic messages that appear on the Host are useful to the application for real-time analysis purposes. You may want to use the *biosMask* field of the instance object to set different levels of debugging in the trace() macro. Refer to the “<MODULE>_<VENDOR>_irtc.c” file for an example of how the trace() macro is called.

5) We can use the type definitions and functions defined in our sample rtc.[ch] files to help us manage real-time trace descriptors. Each **active module** can only bind to **one** instance type of DSP/BIOS object (e.g. 10 active instances of an algorithm all must reference a single DSP/BIOS object).

   Make a backup copy of "main.c". Now, all of the processing functionality must be taken *out* of main() and put into a separate function (main() is only called once in DSP/BIOS). Use the Config Tool to set up a PRD, CLK, SWI, or IDL Object to call your function via the DSP/BIOS scheduler. Modify the main() function to initialize the RTC trace descriptor:

```
#include <rtc.h>
extern LOG_Obj trace; /* Created by the DSP/BIOS Con-
figuration Tool */
RTC_Desc rtc;
/* Bind output log to G729ENC_TI module */
RTC_bind(&G729ENC_TI_IRTC, &trace);
/* if the instance creation succeeded, create a trace
descriptor */
if (alg != NULL && RTC_create(&rtc, alg,
&G729ENC_TI_IRTC) != NULL)
   RTC_set(&rtc, RTC_ENTER);
```

6) Build and load the program to the target. Go to **Tools –> DSP/BIOS –> Message Log** to open a Message Log window and configure it to display the LOG Object messages by right-clicking on the window, selecting "Properties", and selecting "trace" from the pull-down menu. Go to **Tools –> DSP/BIOS –> RTA Control Panel**, right-click the panel and select "Enable All". Run the program. The program should still run with the same results as before Algo Standard conversion, and the LOG_printf() messages from the <MODULE>_<VENDOR>_trace() calls in the algorithm should show up in the DSP/BIOS Message Log window. This verifies that the ITRC interface and DSP/BIOS calls are working properly. The Algo Standard algorithm is now officially "BIOSized". If an application does not elect to use DSP/BIOS, the <MODULE>_<VENDOR>_trace() macro has the logic to detect if a LOG Object exists. If no LOG Object exists, then the DSP/BIOS calls are never initiated during run-time. Therefore, the algorithm works in any environment (DSP/BIOS & non-DSP/BIOS).

7) The other type of DSP/BIOS object that can be used in Algo Standard instances is the Statistics (STS) Object. The IRTC interface can be modified to work with STS Objects, as well as LOG Objects. The BIOS mask can be divided into two regions (e.g., upper half and lower half) to correspond to either STS or LOG diagnostics levels. For example, the <MODULE>_<VENDOR>_bind() function can be altered to accept two DSP/

BIOS Object handles, one for LOG and one for STS Objects. Refer to the latest DSP/BIOS Users Guide for additional information on how to use STS Objects and API's.

6 Algo Standard Library Creation (Rules 8, 11, 13, 15)

Now that the algorithm runs correctly with the Algo Standard interfaces implemented, the actual Algo Standard library deliverable can be created. To be in compliance with Rule 15 of the Algo Standard Rules and Guidelines document, follow the naming convention described in section 3–17. The files listed below need to be partially linked into a single, relocatable, non-executable object, and then archived as a file that is named using the typical Algo Standard naming convention:

```
<module>v<vers>_<vendor>_<variant>.a<arch><model>
(e.g. g729enc_ti.a54f).
```

This name comes directly from Table 1, where module, vers, vendor, processor, and variant are entered (<vers> is required if there are different versions of the Algo Standard algorithm available).

Since multiple algorithms and system control code are often integrated into a single executable, the only external identifiers defined by an algorithm should be those specified by the algorithm API definition (Rule 8). The way to achieve this is to hide all symbols of the object code, and then expose only the symbols the application needs to see (i.e. the v-tables and init/exit functions). Use the –h & –g _<symbol> options. Do not forget the underscore '_' before the symbols (assembler symbols are prefixed with it).

To give the single relocatable object file a unique name, use the –o <object_filename> option.

To specify partial linking (build a single, relocatable, non-executable object file), use the –r option.

To generate a MAP file for documenting the memory characteristics in the next section, use –m option.

Combining all of our desired linker options together, we can create a separate linker command file, e.g. "enclib.cmd" (file **must** have the .cmd suffix to work with the linker), that consists of the following:

```
–r

–m g729enc_ti.map
–o g729enc_ti.o54f
–h
–g _G729ENC_TI_IG729ENC

–g _G729ENC_TI_IALG

–g _G729ENC_TI_init

–g _G729ENC_TI_exit

g729enc_ti_ialg.obj
g729enc_ti_ialgvt.obj
```

```
<original algorithm object file(s)>

SECTIONS
{
     .text
     {
          g729enc_ti_ialgvt.obj (.text)
          <original algorithm object files(s)> (.text)
     }

     .text:algAlloc {}
     .text:algInit {}
     .text:algFree {}
     .text:algActivate {}
     .text:algMoved {}
     .text:algDeactivate {}
     .text:init {}
     .text:exit {}
}
```

From the DOS command prompt, create the relocatable, non-executable object file:

```
> lnk500 enclib.cmd

> lnk6x enclib.cmd
```

Then create the final library file in an archived format:

```
> ar500 –r g729enc_ti.a54f g729enc_ti.o54f

> ar6x –r g729enc_ti.a62 g729enc_ti.o62
```

To test the newly created Algo Standard library, remove all the source files from the project which are already included as part of the Algo Standard library. To link the library into the project, go to the CCS Toolbar and follow the link: **Project –> Options**. Select the **Linker** tab. Locate the “Include Libraries” field, type in the full name of the Algo Standard library file, and hit the **OK** button. Build the project and run the program. The program should still execute with the same results as expected.

To check the Algo Standard Library symbols, run the nmti.exe utility on the relocatable OBJECT file from DOS:

```
>  nmti g729enc_ti.o54f | more
```

Each symbol listed will have a ’type letter’ to the left of it. If it is in *upper* case, then it is exposed, if *lower* case, then it is hidden. Make sure that the v-table symbol(s) is/are exposed with the *upper* case letter. Then make sure the rest of the symbols have a *lower* case type letter. Ignore the *upper* case ’S’s because those are the section names and they need to be exposed (you have no control over them; they cannot be hidden).

7 Performance/Memory Characterization (Rules 19–24)

1) The last step produces the performance and memory characterization data required for the compliance process. The required information can be entered in the tables of Appendix C. For an explanation of the details, refer to Algo Standard Rules and Guidelines, Chapter 4. The best thing to do is to create an electronic copy of these tables using the name <MODULE>_<VENDOR>.doc.

2) A DSP/BIOS framework could be developed to gather the performance information.

8 Conclusion

This document helps the Algo Standard algorithm developer through the entire Algo Standard conversion process. When completed, the algorithm should be submitted immediately for compliance testing. Make sure all header files that are submitted (file names and contents) follow the Alfo Standard naming conventions and header file rules. A minimum of five items must be submitted:

- ❏ <Algo Standard library name>.doc (see Appendix C for an example)
- ❏ The Algo Standard Library file
- ❏ <MODULE>_<VENDOR>.h (and any other header files included in this file)
- ❏ i<MODULE>.h (and any other header files inclluded in this file)
- ❏ ialg.h (unmodified system header file usually found in c:\ti\xdas\include)

Appendix A

Algo Standard Name Tables

Table A–1. Name Selection

Module Name	Vendor Name	Variant	Architecture	Variant (optional)
			Cores: ‘54’, ‘55’, ‘62’, ’64’, ‘67’	‘f’ = far calls/returns ‘e’ = big endian ‘m’ = mixed calls

Table A–2. Instance Creation Parameters

```
typedef struct I<MODULE>_Params {
```

No.	Params Type	Params Name
1.		
2.		
3.		
4.		

```
} I<MODULE>_Params;
```

Table A–3. Real-Time Status and Control Parameters

```
typedef struct I<MODULE>_Status {
```

No.	Status/Control Type	Status/Control Name
1.		
2.		
3.		
4.		

```
} I<MODULE>_Status;
```

Table A–4. Extended IALG (IMODULE) Methods

No.	Return Type	Method Name	Param1 Type and Name†	Param2 Type and Name	Param3 Type and Name
1					
2					
3					
4					
5					

† The IALG_Handle parameter does not need to be specified as the Standard Gen tool will insert it by default

Appendix B

Template File Descriptions

The Algo Standard Code Gen Tool (with all GUI checkboxes checked) generates most of the following files that make up the Algo Standard layers of abstraction:

APPLICATION FRAMEWORK Concrete Interface (API)
(framework-specific set of sample
functions for framework to manage
algorithm instance objects)

<MODULE>.c (implementation of client API functions)

<MODULE>.h (client API interface definitions)

rtc.c (used as is from Algo Standard Developers Kit –– client API functions for real-time trace control)

rtc.h (used as is from Algo Standard Developers Kit –– client API definitions for real-time trace control)

MODULE –– PUBLIC Module-Specific Interface (IMODULE)
(abstract interface)

i<MODULE>.c (definition of default parameter structure settings)

i<MODULE>.h (abstract interface definition header –– PUBLIC data types & methods)

ALGORITHM –– PRIVATE Vendor Specific Interface (IALG / IRTC)
(algorithm-specific)

<MODULE>_<VENDOR>.h (vendor implementation header file; used by application)

<MODULE>_<VENDOR>_ialg.c (vendor-specific algorithm functions)

<MODULE>_<VENDOR>_ialgvt.c (function v-table definitions)

The following files, which are modified versions of sample source files in the Algo Standard Developer's Kit, make up the IRTC interface which can be viewed at the same level as the IALG interface:

```
<MODULE>_<VENDOR>_irtc.c
```

```
<MODULE>_<VENDOR>_irtcvt.c
```

```
<MODULE>_<VENDOR>_priv.h (PRIVATE data used by both IALG &
IRTC)
```

Appendix C

Algo Standard Performance Characterization

This section contains a set of tables that help characterize the Algo Standard algorithm. This information can be extremely useful to the system integrator who is trying to integrate algorithms into a system that has limited memory. Refer to the Algo Standard Rules and Guidelines document, for details on how to fill in the correct information for the following tables.

Table C–1. Module

Module	Vendor	Variant	Arch	Model	Version	Doc Date	Library Name
G729ENC	TI	none	54	far	none	05.05.2000	g729enc_ti.a54f

Table C–2. ROMable (Rule 5)

Yes	No
X	

Table C–3. Heap Data Memory (Rule 19)

memTab	Attribute	Size (bytes)	Align (MAUs)	Space
0	Persist	54	0	External
1	Scratch	256	32	DARAM0
2	Scratch	130	4	SARAM0
3	Persist	712	0	External
4	Scratch	656	64	DARAM1
5	Persist	256	16	SARAM1

Notes: 1) The unit for size is (8–bit) byte and the unit for align is Minimum Addressable Unit (MAUs).

Table C–4. Stack Space Memory (Rule 20)

	Size (bytes)	Align (MAUs)
Worst Case	256	0

Table C–5. Static Data Memory (Rule 21)

.data					.bss				
Object File	Size (bytes)	Align (MAUs)	Read/ Write	Scratch	Object File	Size (bytes)	Align (MAUs)	Read/ Write	Scratch
g729encTI.obj	325	0	R	No	g729enc_ti_ialgvt.obj	22	0	R	No

Table C–6. Program Memory (Rule 22)

	Code	
Code Sections	Size (bytes)	Align (MAUs)
.text	7,782	0
.text:algAlloc	128	0
.text:algInit	209	0
.text:algFree	86	0
.text:algActivate	7	0
.text:algMoved	9	0
.text:algDeactivate	7	0
.text:init	3	0
.text:exit	3	0
.cinit	24	0

Table C–7. Interrupt Latency (Rule 23)

Operation	Typical Call Frequency (microsec)	Worst-Case (Instruction Cycles)
encode()	2,000	50
control()	2,000	0

Table C–8. Period / Execution Time (Rule 24)

Operation	Typical Call Frequency (microsec)	Worst-Case Cycles/Period	Worst-case Cycles/Period	Worst-case Cycles/Period
encode()	2,000	16,000	No periodic execution	No periodic execution
control()	2,000	200	No periodic execution	No periodic execution

Notes:

- ❑ This algorithm follows the run-time conventions imposed by TI's implementation of the C programming language.
- ❑ This algorithm is re-entrant within a preemptive environment (including time-sliced preemption).
- ❑ All algorithm data references are fully relocatable.
- ❑ All algorithm code is fully relocatable.
- ❑ This algorithm does not directly access any peripheral device.
- ❑ This algorithm does not include definitions specific to a debug variant.
- ❑ This algorithm accesses all static and global data as far data (c6x only).
- ❑ This algorithm operates properly with program memory operated in cache mode (c6x only).
- ❑ This algorithm was compiled in little endian mode (c6x only).

Appendix D

Algo Standard Compliance Report

This section contains the official checklist used during compliance testing. It is recommended to go through the entire checklist before submission.

TMS320 DSP Algorithm Interoperability Standard

Algo Standard Compliance Testing

Compliance Test Report (preliminary)

Note: All references to Rules and Guidelines are from the February 2000 revision of SPRU352.

Date:

Vendor:

Algorithm

Incoming Inspection ———————————————————————

documentation:

header file(s):

Rule 1: All algorithms must follow the run-time conventions imposed by TI's implementation of the C programming language.'

Report: Vendor should supply statement that this is correct.

Rule 2: All algorithms must be reentrant within a preemptive environment (including time-sliced preemption).

Report: Vendor must state that algorithm is reentrant according to the definition of the DSP Algorithm Standard

---

Rule 3: All algorithm data references must be fully relocatable (subject to alignment requirements). That is, there must be no "hard coded" data memory locations.

Report:

Rule 4: All algorithm code must be fully relocatable. That is, there can be no hard coded program memory locations.

Report:

Rule 5: Algorithms must characterize their ROM-ability; i.e., state whether they are ROM-able or not.

Report: Vendor must document that their code is ROM-able

Rule 6: Algorithms must never directly access any peripheral device. This includes but is not limited to on-chip DMAs, timers, I/O devices, and cache control registers.

Report:

Rule 7: All header files must support multiple inclusions within a single source file.

Report:

Rule 8: All external definitions must be either API identifiers or API and vendor prefixed.

Report:

Rule 9: All undefined references must refer either to the operations specified in Appendix B (a subset of C runtime support library functions and the DSP/BIOS) or other Algo Standard-compliant modules.

Report:

Rule 10: All modules must follow the naming conventions of the DSP/BIOS for those external declarations disclosed to the client.

Report:

Rule 11: All modules must supply an initialization and finalization method.

Report:

Rule 12: All algorithms must implement the IALG interface.

Report:

Rule 13: Each of the IALG methods implemented by an algorithm must be independently relocatable.

Report:

Rule 14: All abstract algorithm interfaces must derive from the IALG interface.

Report:

Fxns:

Params:

Status:

Rule 15: Each Algo Standard-compliant algorithm must be packaged in an archive which has a name that follows a uniform naming convention.

Note: Refer to section 3.4 of the eXpressDSP Algorithm Standard Rules and Guidelines (SPRU352) for further guidance on file naming conventions.

Report:

Rule 16: Each Algo Standard-compliant algorithm header must follow a uniform naming convention.

Header file name should be of the form:
<module><vers>_<vendor>_<variant>.h

Report:

Rule 17: Different versions of an Algo Standard-compliant algorithm from the same vendor must follow a uniform naming convention.

Report:

Rule 18: If a module's header includes definitions specific to a "debug" variant, it must use the symbol _DEBUG to select the appropriate definitions;

_DEBUG is defined for debug compilations and only for debug compilations.

Report:

Rule 19: All algorithms must characterize their worst-case heap data memory requirements (including alignment).

Report:

Rule 20: All algorithms must characterize their worst-case stack space memory requirements (including alignment).

Report:

Rule 21: Algorithms must characterize their static data memory requirements.

Report:

Rule 22: All algorithms must characterize their program memory requirements.

Report:

Rule 23: All algorithms must characterize their worst-case interrupt latency for every operation.

Report:

Rule 24: All algorithms must characterize the typical period and worst-case execution time for each operation.

Report:

DSP-specific Rules ————————————————————————————

Rule 25: All C6x algorithms must be supplied in little endian format.

Report:

Rule 26: All C6x algorithms must access all static and global data as far data.

Report:

Rule 27: C6x algorithms must never assume placement in on-chip program memory; i.e., they must properly operate with program memory operated in cache mode.

Report:

Rule 28: On processors that support large program model compilation, all core run-time support functions must be accessed as far functions; for example, on the C54x, the calling function must push both the XPC and the current PC.

Report:

Rule 29: On processors that support large program model compilation, all algorithm functions must be declared as far functions; for example, on the C54x, callers must push both the XPC and the current PC and the algorithm functions must perform a far return.

Report:

Rule 30: On processors that support an extended program address space (paged memory), the code size of any object file should never exceed the code space available on a page when overlays are enabled.

Report:

End Report ——————————————————————————————