A Technical Overview of eXpressDSP-Compliant Algorithms for DSP Software Producers

Stig Torud *Organization*

ABSTRACT

Advances in digital signal processor (DSP) technology in the application areas of telephony, imaging, video, and voice are often results of years of intensive research and development. For example, algorithm standards for telephony have taken years to develop. The implementation of these DSP algorithms is often very different from one application system to another because systems have, for example, different memory management policies and I/O handling mechanisms. Because of the lack of consistent system integration or programming standards, it is not possible for a DSP implementation of an algorithm to be used in more than one system or application without significant reengineering, integration, and testing.

This document targets these algorithms and assists the reader with making the algorithms compliant with the TMS320 DSP Algorithm Standard which is part of TI's eXpressDSP™ technology initiative. Algorithms that comply with the standard are tested and awarded an eXpressDSP compliant mark upon successful completion of the test.

It is assumed that the reader of this document has read or is familiar with the specifications of the standard.

Contents

List of Figures

1 Introduction

The eXpressDSP Algorithm Standard (XDAIS) activities can be divided into three categories: clients (users) of algorithms, producers of algorithms, and creators of algorithm-specific interfaces. This application note focuses on the producers of algorithms and assumes that the interface is defined.

In order for an algorithm to be compliant with the standard, the algorithm must implement the IALG interface specified by the standard. The IALG interface is an *abstract* interface or a *Service Provider Interface* (SPI) and is defined in the `ialg.h` header file. In addition to implementing the IALG interface, the algorithm must obey a number of rules. For example, external identifiers need to follow the naming conventions, algorithms must never directly access any peripheral device, and the algorithm code must be fully relocateable. See the TMS320 Algorithm Standard specification for the complete set of rules.

The algorithm must also implement an algorithm-specific interface, defined by an algorithm interface creator, who extends the IALG interface in order to run the algorithm. This application note uses the IG723ENC interface, defined by TI, as an example, and describes the process of making a TI version of the ITU G.723.1 annex A encoder algorithm eXpressDSP compliant.

The client of the algorithm can manage an instance of the algorithm by calling into a *table of function pointers* (v-table). Every algorithm must create the v-table to be compliant with the standard. Through the v-table, the application can manage the algorithm instance, for example, create and delete an instance object of the algorithm as well as run the algorithm. See Figure 1 for an illustration.

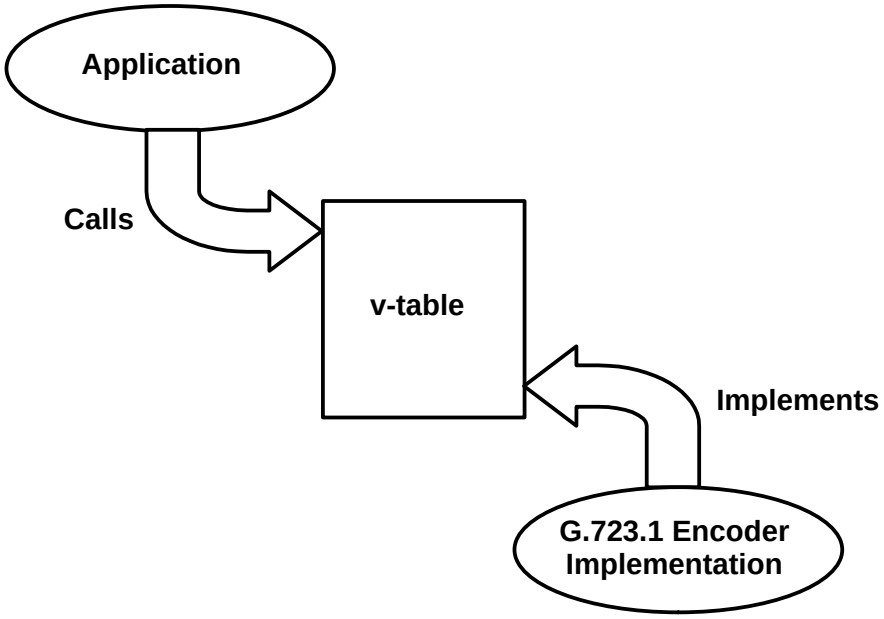

Figure 1. Application, Implementation and the v-table

2 The IALG Interface

The IALG interface is the core interface defined by the standard that the algorithm is required to implement. It defines types and constants as well as the v-table, `IALG_Fxns`. Every algorithm needs to define and initialize a variable of type, `IALG_Fxns`, to declare the v-table. Some of the functions in the v-table are optional, while `algAlloc()`, `algInit()`, and `algFree()` are required.

2.1 algAlloc()

The algorithm must implement the `algAlloc()` function to declare its memory requirements. In the following example, we request one block of persistent memory. The size of this block must be large enough to hold the object definition of the algorithm as well as any working buffers required during the execution of an instance of the algorithm.

```
Int G723ENC_TI_algAlloc(const IALG_Params *algParams, IALG_Fxns **pf, IALG_MemRec
memTab[])
{
    /* Request memory for G723ENC instance object */
    memTab[0].size = sizeof(G723ENC_TI_Obj);
    memTab[0].alignment = 0;
    memTab[0].space = IALG_DARAM0;
    memTab[0].attrs = IALG_PERSIST;

    return(1)    /* return number of memory blocks requested */

}
```

The `algAlloc()` implementation informs the application of its memory requirements by filling the `memTab` structure. Based on this information, the application allocates the requested memory before it calls the function to initialize the object.

If your code is not organized as an object containing the state or context information, you must do so. This is good coding practice and motivates reentrant code since all references to the object is now through a pointer to the object. Consider the following example of the `G723ENC_TI_Obj`:

```
typedef struct G723ENC_TI_Obj {
   IALG_Obj   ialg;            /* Points to the v-table */
   IG723_Rate workingRate;     /* 5.3 or 6.3 kbps */
   XDAS_Bool  hPFilter;        /* High Pass filter on/off */
   XDAS_Bool  VAD;             /* Voice activity detection on/off */

    ...................        /* specifics to the implementation */
} G723ENC_TI_Obj;
```

Notice that the first field in the object is `IALG_Obj`, which is a pointer to the v-table we will create later. This parameter *must* be the first field in any object definition. It is the application's responsibility to initialize this pointer to point to the v-table when creating an instance of the algorithm. Also, `G723ENC_TI_Obj` must occupy the first block of memory, `memTab[0]`.

2.2 algInit()

The algorithm implements `algInit()` to initialize the instance persistent memory requested in `algAlloc()`. After the application has called `algInit()`, the instance of the algorithm pointed to by `handle` is ready to be used. See the example below:

```
Int G723ENC_TI_algInit(IALG_Handle handle, const IALG_MemRec memTab[], IALG_Han-
dle p, const IALG_Params *algParams)
{
    G723ENC_TI_Obj *enc = (Void *)handle;
    const IG723ENC_Params *params = (Void *)algParams;

    if (params == NULL) {
        params = &IG723ENC_PARAMS;  /* set default parameters */
    }

    /* Copy creation params into the object */
    enc->workingRate = params->rate;
    enc->hPFilter = params->hpfEnable;
    enc->VAD = params->vadEnable;

    g723EncInit(enc);  /* Initialize all other instance variables */

    return(IALG_EOK);
}
```

If the application passes in a `NULL` pointer for the creation parameters, the algorithm should use a default set of creation parameters. The default set of parameters are defined by the interface definer.

2.3 algFree()

The last of the required functions the algorithm needs to implement is `algFree()`. It is the algorithm's responsibility to set the addresses and the size of each memory block requested in `algAlloc()` such that the application can delete the instance object without creating memory leaks. See the example below:

```
Int G723ENC_TI_algFree(IALG_Handle handle, IALG_MemRec memTab[])
{
    G723ENC_TI_Obj *enc = (Void *)handle;

    algAlloc(NULL, NULL, memTab);      /* Fill the memTab struct */
    memTab[0].base = (Void *)&enc;

    return(1);
}
```

The call to `algAlloc()` will set all parameters in the `memTab` structure except the `base` field which holds the address of the block of memory.

3 Module-Specific Interface

So far, we have implemented three functions required to create, initialize, and delete the instance object. We also need to implement the IG723ENC interface, which is the SPI for the TI version of the ITU G.723.1 annex A encoder interface. The IG723ENC interface is defined in the `ig723enc.h` header file. `IG723ENC_Fxns` extends the `IALG_Fxns`, as seen in the excerpt from the header file:

```
typedef struct IG723ENC_Fxns {
    IALG_Fxns  ialg;    /* IG723ENC extends IALG */
    XDAS_Bool  (*control)(IG723ENC_Handle handle, IG723_Cmd cmd,
                          IG723ENC_Status *status);
    XDAS_Bool  (*encode) (IG723ENC_Handle handle, XDAS_UInt16 *in,
                          XDAS_UInt16 *out);
} IG723ENC_Fxns;
```

The standard introduces algorithm standard data types, e.g., `XDAS_UInt16`, to ensure type consistency. See `xdas.h` for the complete definitions of these data types.

You need to create both the `control()` and `encode()` functions in order to have a complete algorithm that implements the IG723ENC interface. Most often, these functions are wrappers to existing implementations, as seen in the example below for the `encode`() function:

```
XDAS_Bool G723ENC_TI_encode(IG723ENC_Handle handle, XDAS_UInt16 *in, XDAS_UInt16
*out)
{
    G723ENC_TI_Obj   *enc = (Void *)handle;

    if(encoder(enc, in, out)      /* do the processing */
        return(XDAS_TRUE);

    return(XDAS_FALSE);
}
```

The IG723ENC interface defines the default creation parameters in the `ig723enc.c` source file. If the application wants to use creation parameters other than those provided in the interface, the application will make a local copy of the default algorithm parameters and modify the desired fields before creating an instance. The application then needs to pass this modified parameter structure to `algAlloc()` and `algInit()` to create an instance with the modified parameters.

3.1 V-table

We now want to define and initialize the IG723ENC v-table. Since `IG723ENC_Fxns` extends `IALG_Fxns`, the IG723ENC v-table must also include the functions in the IALG v-table:

```
#define IALGFXNS \
   &G723ENC_TI_IALG,            /* module ID */     \
   NULL,                        /* activate */      \
   G723ENC_TI_algAlloc,         /* alloc */         \
   NULL,                        /* control */       \
   NULL,                        /* deactivate */    \
   G723ENC_TI_algFree,          /* free */          \
   G723ENC_TI_algInit,          /* init */          \
   NULL,                        /* moved */         \
   NULL                         /* numAlloc */      \
IG723ENC_Fxns G723ENC_TI_IG723ENC ={
   IALGFXNS,                    /* IALG functions */
   G723ENC_TI_control,
   g723ENC_TI_encode
} G723ENC_TI_IG723ENC;
asm("_G723ENC_TI_IALG .set _G723ENC_TI_IG723ENC");
```

The first field in the v-table is the address of the table. This field is used as a unique identifier of the implementation. Notice that only the three required IALG functions and the module-specific functions are defined in the v-table. All the other function pointers are set to `NULL`.

The `asm()` statement defines the symbol `G723ENC_TI_IALG` to be equal to `G723ENC_TI_IG723ENC`, which means that the IALG and the IG723ENC v-tables are shared.

Notice the naming conventions used as the symbol for the v-tables.

- G723ENC_TI_IALG Reads: "TI's implementation of the IALG interface for the G723ENC module"
- G723ENC_TI_IG723ENC Reads: "TI's implementation of the IG723ENC interface for the G723ENC module"

If you make a call into the v-table with `IALG_Handle`, you will only be able to access `IALG_Fxns`. You will need to use `IG723ENC_Handle` to access the `encode()` and `control()` functions.

4 Algorithm Structure

Figure 2 illustrates how the object, the v-table, and the actual implementation are laid out in memory. The handle to the instance object is located on the stack for illustration.

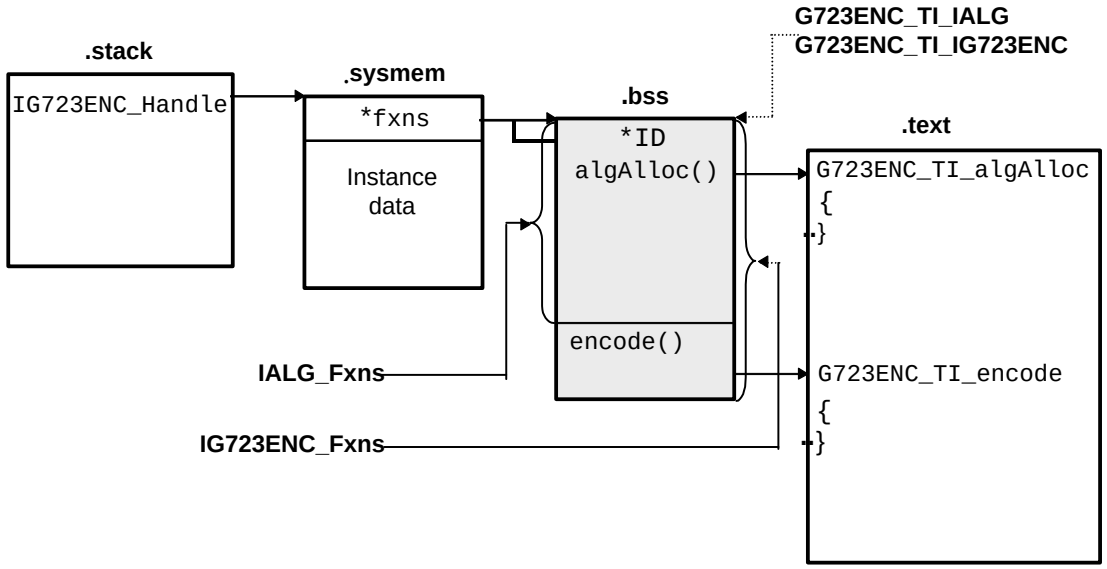

Figure 2. TMS320 DSP Algorithm Standard Structure

Notice that `G723ENC_TI_IG723ENC` and `G723ENC_TI_IALG` share the same v-table.

5 Summary of Some Important Rules

5.1 TI C Language Run-Time Conventions

This rule ensures that all algorithms can be called from the C language. The entire algorithm code can still be in assembly. However, every function in the v-table is required to follow the processor-specific conventions for interfacing C and assembly. The functions internal to the algorithm are not required to follow these conventions.

5.2 Core Run-Time APIs

The core run-time support for the algorithm is the TI C-language run-time APIs with the exception of heap management, I/O, and miscellaneous non-reentrant functions. The algorithm is also free to use the DSP/BIOS run-time support library except the scheduling APIs. This does not mean that an application using an algorithm with calls to DSP/BIOS APIs needs to run DSP/BIOS. The application can stub these APIs or replace them with its own implementation of the APIs. See the TMS320 DSP Algorithm Standard Specifications, Appendix B, *Core Run-Time APIs*, for a complete overview of the core run-time support APIs.

5.3 Near/Far Models

5.3.1 Program Model

All the entry points for the module, i.e., the functions in the v-table, and the core runtime support functions must be *far* calls/returns on c54x chips with extended addressing. Functions internal to the algorithm can be either *far* or *near* calls/returns..

5.3.2 Data Model

All ’C6x algorithms must access all static and global data as *far* data. However, this does not imply that all look-ups in a static table will be *far* accesses. Only the reference to the look-up table will be a *far* access; the actual look-up will be an offset to the look-up table address.

5.3.3 Cache

The algorithm should not assume that it will or will not be used in cache mode. The algorithm should work in either case.

5.3.4 Naming Conventions

Every external identifier and type definition must follow the naming conventions with the prefix <*module*>_<*vendor*>_. The library and the header file following the library must adhere to these naming conventions as well.

A simple way to check whether or not all the external identifiers follow the naming conventions is to extract the library and run the *nmti* tool found in the /bin directory of the TMS320 DSP Algorithm Standard Developer’s Kit on every object file. See the example code below, where *nmti* is run on the example object file `scale.obj`:

```
[C:/test] nmti scale.obj
00000000 S .text
00000000 S .data
00000000 S .bss
0000000f A FP
0000002e A DP
0000002f A SP
00000000 T _Scale     /* should be G723ENC_TI_Scale */
00000030 t RL0
000000f8 t L1
00000130 t L2
00000180 t RL1
0000018c t L3
00000000 t .text
00000000 d .data
00000000 b .bss
       U _Sqrt_LBC    /* should be G723ENC_TI_Sqrt_LBC */
       U _Comp_En     /* should be G723ENC_TI_Comp_EN */
```

We can identify an external symbol by looking at the letter in the second column. The letter is in uppercase if the symbol is external. In other words, all the symbols with an uppercase letter in the second column need to follow the naming conventions.

Another convenient approach to satisfying the naming convention requirements is to use the symbol hiding and renaming tool, *rename*, in the /bin directory of the TMS320 DSP Algorithm Standard Developer’s Kit.

6 Library and Header File

After implementing the required `IALG_Fxns` and `IG723ENC_Fxns`, and making sure we are following all the rules in the specification, we are ready to create the object library, `g723enc_ti.l62`. This library and a header file are delivered to an application that uses the eXpressDSP-compliant algorithm that implements the TI version of the G.723.1 encoder interface.

The header file `g723enc_ti.h` needs to declare the symbols to the v-table as in the following line of code:

```
extern IG723ENC_Fxns G723ENC_TI_IG723ENC;
extern IALG_Fxns G723ENC_TI_IALG;
```

7 Examples

The following examples illustrate how an application can use your algorithm. Notice how the v-table appears as the indirection between the implementation of the algorithm and the application using the algorithm.

7.1 Dynamic Object Creation

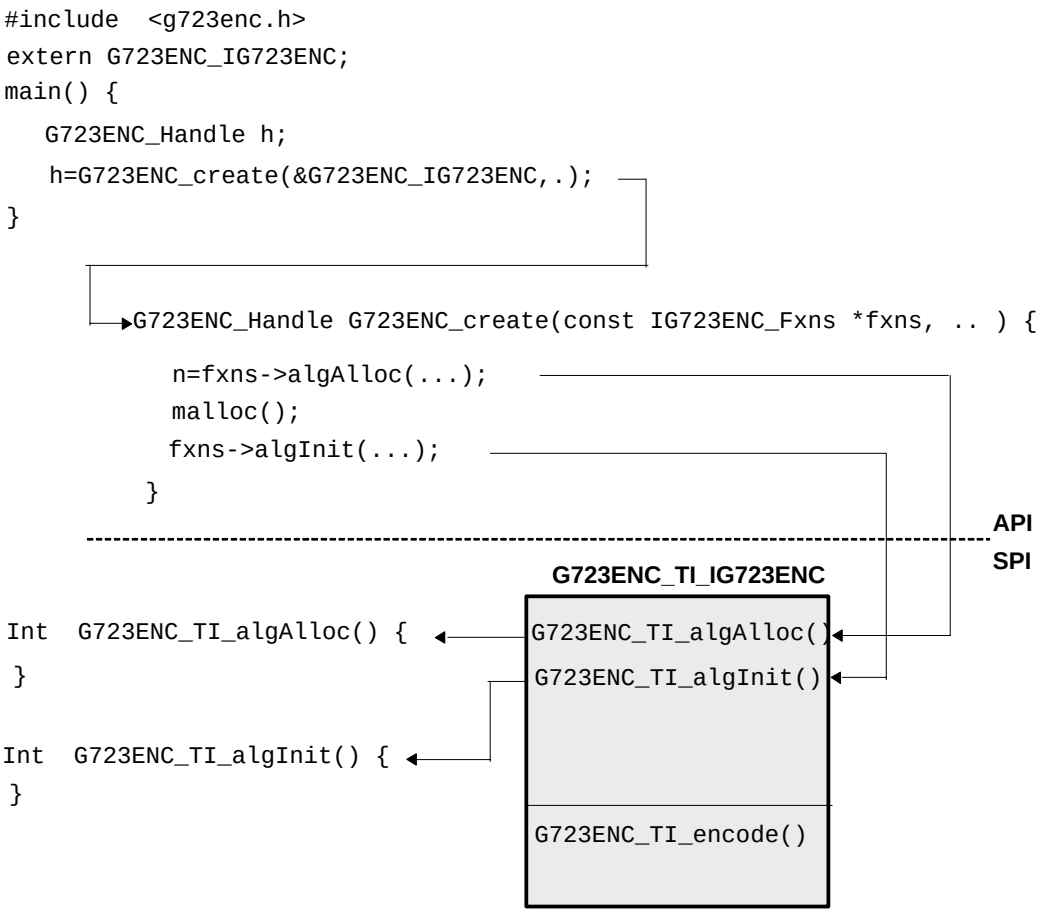

Figure 3. Example of Dynamic Creation of an Instance of the TI G.723.1 Encoder Module

The linker command file for the application has the following statements:

```
-u _G723ENC_TI_IG723ENC
-l g723enc_ti.l62
_G723ENC_IG723ENC = _G723ENC_TI_IG723ENC;
```

Notice that the binding of G723ENC_IG723ENC to the TI v-table G723ENC_TI_IG723ENC happens at link time. If the application wants to change to another implementation which implements the same interface, it simply changes the binding of G723ENC_IG723ENC to the v-table of the other implementation. In other words, the application code stays exactly the same and no recompilation is required. Only a relink is required to switch to another implementation.

7.2 Multichannel Encoding

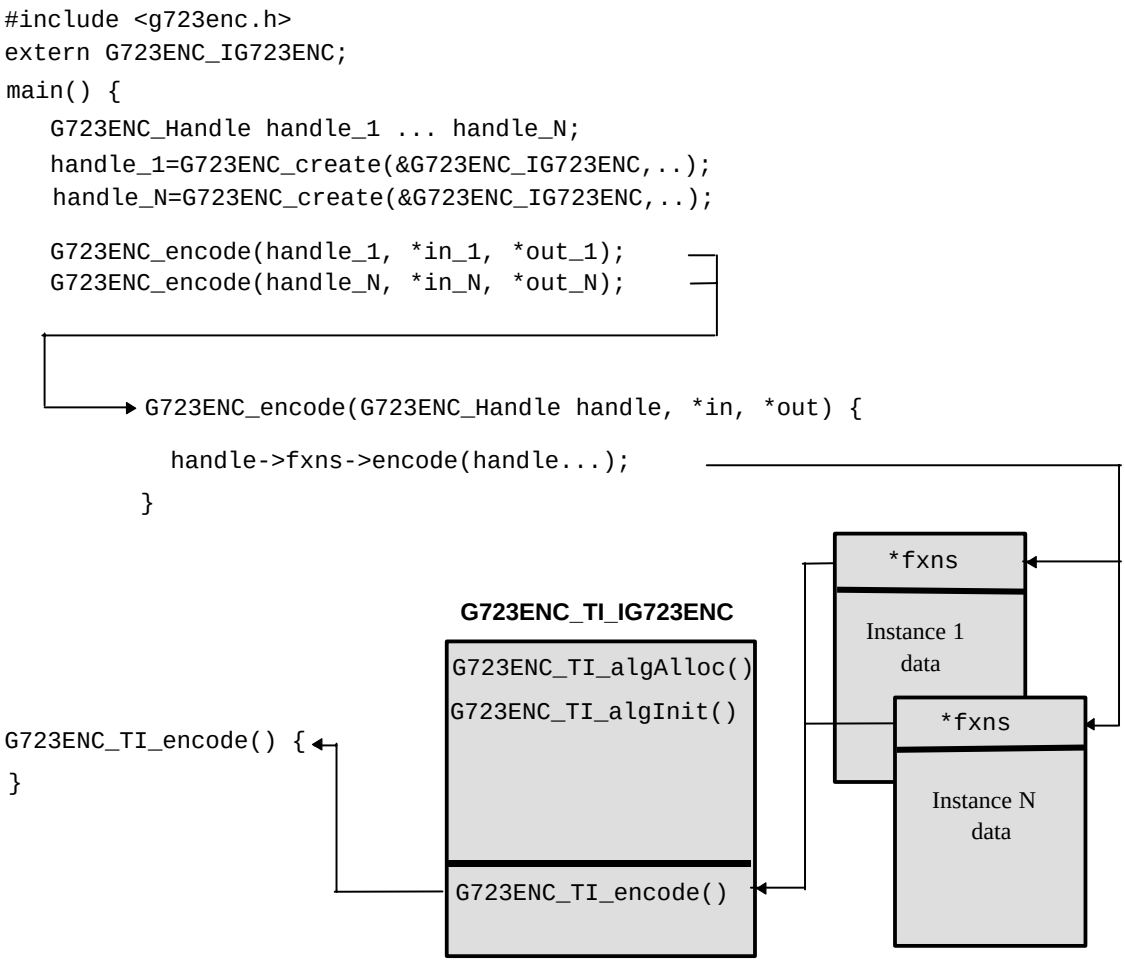

Figure 4. Example of a Multichannel System With TI G.723.1 Encoder Modules

Notice the first field in the instances is a pointer to the v-table, i.e., they are running the same program code. The other parameters in the instances contain the interframe state information.

The binding of G723ENC_IG723ENC to the TI v-table G723ENC_TI_IG723ENC happens at link time as explained previously.

8 Improvements

We now have an algorithm that has implemented the TI-defined IG723ENC interface and is eXpressDSP-compliant. However, the algorithm is not optimized to utilize the mechanism provided in the standard for optimal memory utilization.

8.1 Scratch Versus Persistent Memory

THE TMS320 DSP Algorithm Standard labels the types of memory into two categories: scratch and persistent. Persistent memory is used to store state information while an algorithm instance is not executing. On the other hand, the algorithm uses scratch memory as a working area during execution. After the execution of the algorithm, the application is free to assign the scratch memory to other algorithms in the system. In other words, an algorithm instance cannot make any assumptions about the contents of the scratch memory when it is being assigned to the instance of the algorithm by the application.

In the `G723ENC_TI_algAlloc()` function, we requested one block of memory to hold the state information and potential working buffers. We recommend that an algorithm divide its memory requirements into several blocks for two reasons. First, it makes the memory management more flexible for the application with regards to fragmentation issues. Second, the application can make efficient use of the scratch memory since it can be overlaid among instances of algorithms.

Because the algorithm uses the persistent memory to hold state information between executions, this information should be assigned to slow external memory. At the time the algorithm has a frame of data for processing, the state information needed for processing the frame needs to be copied into a scratch buffer in fast memory. When the algorithm is finished processing the buffer, the application may decide to assign the scratch buffer to another instance. The information in the scratch buffer required for processing the next frame of data then needs to be copied back to the persistent memory. The IALG interface provides this mechanism through the functions `algActivate()` and `algDeactivate()`. The application makes the decision if it wants to call `algActivate()` and `algDeactivate()` for every frame of data, or if it wants to call e.g., `algActivate()`and never share the scratch memory with other instances until it calls `algDeactivate()`.

The IALG interface also provides the function `algMoved()`which enables the application to move an instance's memory block around at runtime. This is a powerful capability for some applications since they get even more flexibility in the management of the instance objects. An algorithm implementing this function is only responsible for updating the internal references to its memory blocks.

8.2 Stack Issues

One way to achieve reentrant code is to declare all scratch data on the stack. There are at least two problems with this approach. First, an algorithm has no control over the placement of the system stack. Second, it is very difficult for the application to share scratch stack memory between instances. We strongly recommend that algorithms avoid using the system stack more than necessary and encourage algorithms to request scratch memory blocks in `algAlloc()`.

9 Conclusion

TMS320 DSP Algorithm Standard activities can be divided into three categories: clients (users) of algorithms, producers of algorithms and creators of algorithm specific interfaces. This application note focuses on the producers of algorithms. It provides the new XDAIS software producer with a technical overview of the basic components needed to be eXpressDSP-compliant, and also highlights some important rules that are often misunderstood and provides some tips for future improvements.

Please refer to the TMS320 DSP Algorithm Standard Developers Guide (SPRU424) for a complete procedure to make any algorithm eXpressDSP-compliant:

1. Implement `algAlloc(), algInit()` and `algFree()`.
2. Implement the module-specific interface.

3. Create the v-table.
4. Make sure you follow all the rules in the standard.
5. Build the library and create the header file.

At this point, the algorithm should be eXpressDSP-compliant. Other optional steps you might consider are:

1. Divide memory requests into several block.
2. Implement other functions in the IALG interface (`algActivate(), algDeactivate()`, etc)