Application Report
SPRA790 - October 2002

Techniques for Implementing Shared Relocatable Buffers Using the TMS320 DSP Algorithm Standard

Murat Karaorman Texas Instruments

ABSTRACT

This application note presents several mechanisms supported by the TMS320 DSP Algorithm Standard (referred to as XDAIS) which can be employed by algorithm writers to assist application developers in optimizing their system's performance and sharing of memory resources.

The first mechanism is based on using statically allocated global data structures. The second mechanism is built on the concepts of scratch memory and memory overlaying techniques based on algorithm activation and deactivation. The third mechanism is referred to as the write-once buffers technique and support sharing of relocatable read-only data across instances of a common algorithm. The last mechanism is referred to as the parent instance technique, whereby the algorithm developer defines and associates an anonymous parent instance with each algorithm instance that gets created. Parent instance memory can be shared by children algorithm instances.

Contents

1 Introduction . . . 2
2 Sharing Statically Allocated Global Data . . . 3
3 Sharing Memory Using Scratch Buffers . . . 3
4 Shared Relocatable Write-Once Buffers . . . 5
4.1 Write-Once Buffers Definition and Recent Spec Enhancements . . . 5
4.2 Creation and Sharing of Write-Once Buffers . . . 6
4.3 Relocation of Write-Once Buffers . . . 7
5 Shared Relocatable Buffers Using Parent Instance Technique . . . 7
5.1 Creation of Parent Instance . . . 8
5.2 Creation of Child Instances . . . 9
5.3 Relocation of Parent Instance . . . 10
5.4 Activation of Parent Instance . . . 10
6 Summary and Conclusion . . . 11
7 References . . . 11

List of Figures

Figure 1 An Optimal Instance Memory Sharing Layout . . . 2
Figure 2 Two Algorithm Instances Sharing Write-Once Memory . . . 5
Figure 3 Buffer Assignments to Two Algorithm Instances During algInit() . . . 6
Figure 4 A Parent Instance Memory Sharing Layout . . . 8
Figure 5 Creation of Parent Instance . . . 9
Figure 6 Creation of Child Instances . . . 10

1 Introduction

In typical multichannel, multirate applications several algorithm instances may need to share common structures, buffers, and lookup tables. Additionally, in order to ensure real-time deadlines, those shared buffers may need to be moved to external memory while the algorithm instances are not active and moved back to internal memory when they become active. XDAIS provides several mechanisms which can be employed by the algorithm writers to assist application developers in optimizing their system’s performance and memory usage. One or more of the following general techniques can be employed:

1. Algorithm instances share statically allocated global data structures/look-up tables. The algorithm requests no additional per-instance heap memory. Sharing is transparent to client applications, but the shared global data memory cannot be relocated at run time.
2. The algorithm requests some of its buffers that need to be in fast memory as scratch. Applications can arrange sharing of a common scratch area by overlaying the scratch buffers of any group of unrelated algorithm instances.
3. The algorithm requests (or provides) any sharable, read-only buffer as write-once persistent. Applications can grant multiple instances of the same algorithm the same write-one buffers (i.e., sharing).
4. The algorithm defines a parent instance which represents a sharable state. Applications must always create the parent instances separately. They may arrange sharing of the parent instance’s memory by initializing multiple instances of the same algorithm with the same parent.

Figure 1 illustrates the memory layout of a system which maximizes the sharing of instance memories of four algorithm instances. In the example, instances of algorithm A and B are created using the same parent, P, implicitly sharing its persistent memory (P). Instance 0 and 1 of algorithm C share the write-once buffer C.

Through the rest of this application note we discuss each mechanism and its applicability to various usage scenarios in separate sections.

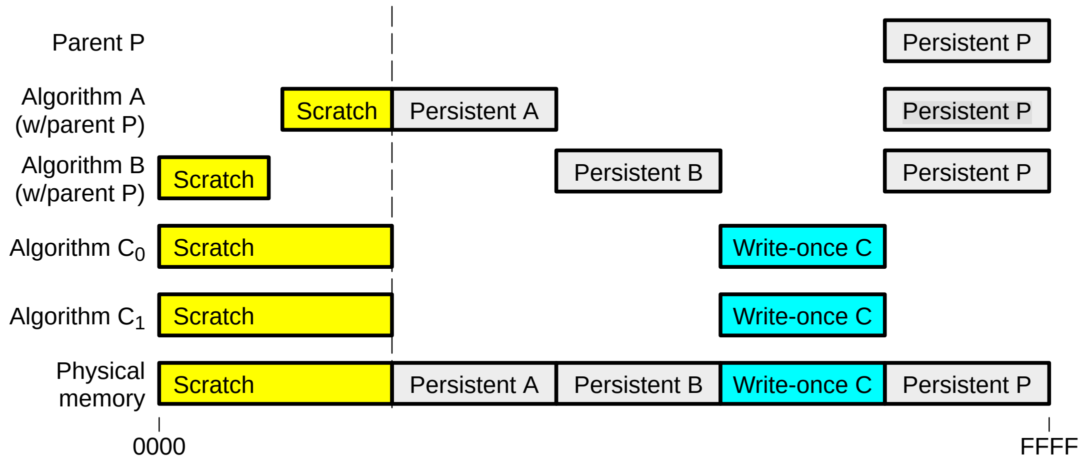

Figure 1. An Optimal Instance Memory Sharing Layout

2 Sharing Statically Allocated Global Data

The simplest method for sharing memory space among several algorithm instances is for the algorithm library to define data sections with global data structures, buffers, or look-up tables which get statically linked with the client's application.

Shared memory space gets allocated by the client application only once at link time without involving the IALG interface and each algorithm instance can readily refer to shared global memory at run time. The sharing is implicit and completely transparent to the client application. However, despite its implementation simplicity and the ease of integration offered to the client, this mechanism may not always be appropriate. Since the shared global memory is allocated statically at link time, it is not relocatable at run time. If performance constraints require placement of the buffers in fast internal memory, for example, this form of sharing may not be appropriate for sharing large buffers or look-up tables. Additionally, sharing read/write global memory is error prone which may easily lead to non-reentrant code and should generally be avoided.

This approach is most suited for sharing global read-only tables and structures. When the size of shared memory that needs to be allocated in internal memory is large, this approach can be combined with scratch buffering or write-once buffering techniques described in subsequent sections.

3 Sharing Memory Using Scratch Buffers

XDAIS defines scratch memory as memory that is freely used by an algorithm without regard to its prior contents, i.e., no assumptions about the content can be made by the algorithm and the algorithm is free to leave it in any state. The algorithm instance initializes its scratch buffers when the application activates the instance. It does this by granting it exclusive access to the scratch region and calling its IALG activation function, `algActivate()`. During initialization of its scratch buffers in `algActivate()`, the algorithm can only access its static memory and what is saved in its persistent instance memory. The application calls `algDeactivate()` when it wants to use/free up the scratch area granted to the instance. The algorithm saves to its persistent memory any information in its scratch buffers that it will need later during reactivation to re-initialize its scratch buffers.

After the standard algorithm initialization call to `algInit()`, all compliant algorithm instances with scratch buffers will be in either one of these two states:

1. Activated: no algDeactivate calls since the last algActivate call
2. Deactivated: no algActivate calls since algInit or since the last algDeactivate

The basic rule of operation is that an algorithm instance must be in an activated state when any of its processing functions are called.

The basic rule of sharing a system overlay scratch area is that at any given time, at most, one algorithm instance sharing the overlay area can be activated - all other instances must be de-activated. For example, based on the scratch memory assignments of Figure 1, if A or B is activated, C0 and C1 must be de-activated. If C0 is activated, A, B and C1 must be deactivated. Similarly, if C1 is activated A, B and C1 must be deactivated. However, if both C0 and C1 are deactivated then A and B can be simultaneously activated, since their scratch memories do not overlap.

It is entirely up to the application framework where to allocate scratch memory and decide which groups of algorithm instances (if any) will be sharing a common “scratch” overlay region. The application is also in full control of when a particular instance needs to be activated or deactivated. Framework overlay management activities are supported by algorithms which request scratch type memory and implement the standard XDAIS IALG interface functions: `algActivate()` and `algDeactivate`. This is a general mechanism that allows applications to overlay the scratch memories of any collection of algorithm instances.

The activation/deactivation can be expensive if large amounts of data need to be initialized or saved and restored at each context switch. A special case is when several instances of the same algorithm request (scratch) buffer space for referencing constants or read-only data, such as lookup tables that must be placed in fast internal memory only when the algorithm is active. A near optimal solution in terms of MIPS and memory is to reserve a section of the internal memory as part of the algorithm’s static allocation and have all instances refer to the same global shared lookup tables and buffers which gets placed in this section. This approach, however, introduces system integration complexity and is not applicable when relocation of buffers is needed or when data needs to be initialized dynamically based on instance creation parameters, etc. The remainder of this application note discusses two additional mechanisms that are supported by XDAIS and are simpler, yet only applicable to algorithm instances from the same vendor.

We refer to one of the techniques as the parent instance mechanism. The main idea is for the algorithm developer to define and associate an anonymous parent instance with each algorithm instance that is created. The application uses the standard IALG interface and first creates the algorithm’s parent instance. This parent instance owns all buffers that can be accessed and shared by its children. All algorithm instances that are initialized with the same parent instance effectively share the memory buffers of the parent instance.

The other technique, discussed in the next section, is referred to as the write-once buffers technique. This is also the simplest technique, as it requires no extra parent instance creation or algorithm activation/deactivation.

4 Shared Relocatable Write-Once Buffers

Figure 2 illustrates an optimal instance memory layout where the application framework arranges algorithm instances C0 and C1 to share a single write-once buffer, and has overlaid the instance scratch buffers.

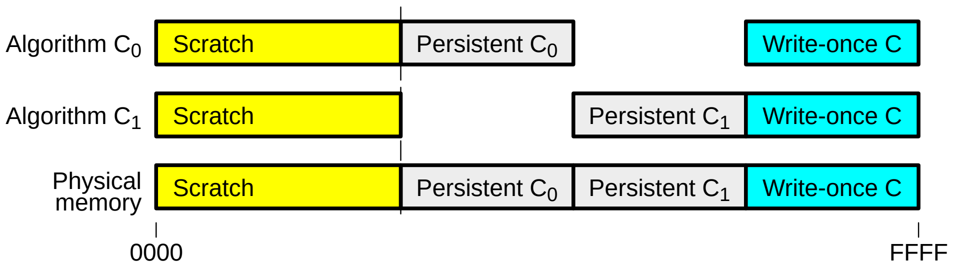

Figure 2. Two Algorithm Instances Sharing Write-Once Memory

4.1 Write-Once Buffers Definition and Recent Spec Enhancements

Recent revision of XDAIS includes specification enhancements to the IALG interface. This section summarizes the enhancements that lead to more effective allocation and sharing of write-once persistent memory.

Write-once type memory is specified using the IALG_WRITEONCE memory attribute of the IALG interface type, IALG_MemAttrs:

```
typedef enum IALG_MemAttrs {
    IALG_SCRATCH,        /* scratch memory */
    IALG_PERSIST,        /* persistent memory */
    IALG_WRITEONCE       /* write-once persistent memory */
} IALG_MemAttrs;
```

The algorithm uses the IALG_WRITEONCE memory attribute to indicate that the requested buffer must be treated as persistent by the framework, and the algorithm will only access the buffer as read-only. The contents of the buffer will be initialized by the algorithm either statically at link time or during algInit().

The applications are still free to treat write-once buffers in identical fashion to persistent (IALG_PERSIST) memory buffers, as this provides backward compatibility for older frameworks. However, the recent enhancements, summarized below, allow the applications to further take advantage of the write-once type memory requests for allocation and sharing, and simplify the sharing of read-only lookup tables.

During the algAlloc() call, the algorithms may now provide the the base address of any statically initialized IALG_WRITEONCE persistent buffers to the client. If the algorithm provides a base address, the client may simply use it to initialize the instance object without allocating any additional memory; otherwise, the client allocates and grants the memory like a standard memory request. Additionally, the client may arrange sharing by granting the same write-once persistent buffers to multiple instances of the same algorithm created with identical parameters.

4.2 Creation and Sharing of Write-Once Buffers

Sharing of write-once buffers is arranged by the application. Figure 3 illustrates an example where the application calls `algInit` providing the memory assignments for the algorithm instances c0 and c1.

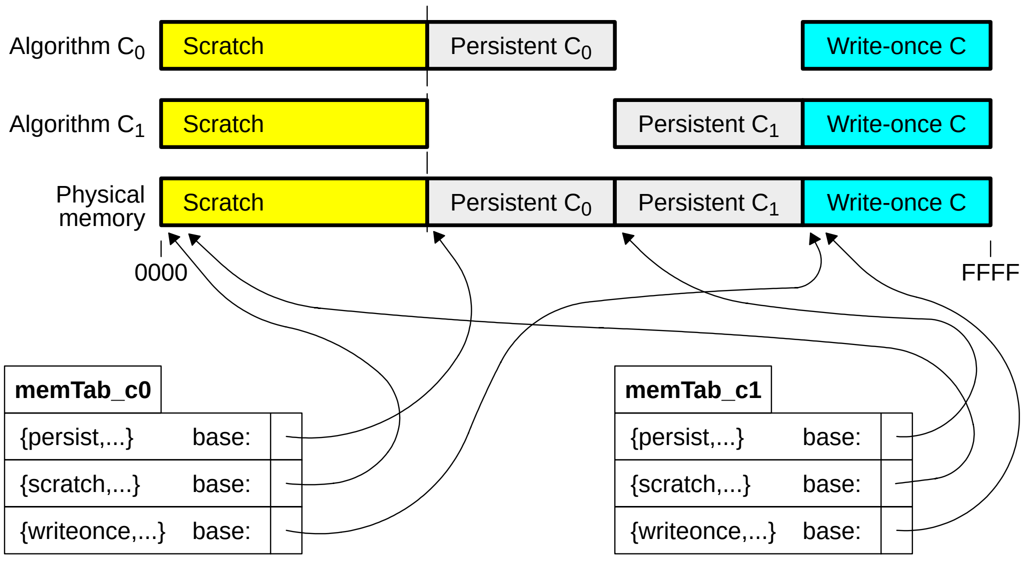

Figure 3. Buffer Assignments to Two Algorithm Instances During `algInit()`

The IALG_MemRec structure memTab_c0 is used to initialize c0 and memTab_c1 to initialize c1:

```
IALG_Handle c0 = memTab_c0[0].base;
IALG_Handle c1 = memTab_c1[0].base;

memTab_c0[2] = memTab_c0[2] = &<Write-Once Buffer>;

<MOD_VEND_IALG>.ialg.algInit(c0, memTab_c0, NULL, NULL);
<MOD_VEND_IALG>.ialg.algInit(c1, memTab_c1, NULL, NULL);
```

Allocation of the write-once buffers are enhanced to offer both the algorithm developer and the applications new options.

For the algorithms:

1. Algorithms can now provide statically allocated and initialized memory space for the write-once buffers. The addresses of such algorithm allocated write-once buffers are passed to the application during the call to `algAlloc`, by writing to the base field of the `IALG_MemRec` structure argument. This is the recommended approach when write-once buffers are statically allocated lookup tables or data structures.
2. Algorithms can still assign NULL to the base field and request the client application to allocate and assign the buffer space. This approach is used when the contents of the write-once memory is to be computed in `algInit` and depends on the creation parameters.

As for the applications:

1. Applications are free to utilize any algorithm-provided write-once buffers when the algAlloc call returns with a non-NULL base address. Applications can avoid allocating additional buffer memory this way, and they may instead pass back the same base address to the algorithm during the `algInit` call.
2. When the base address provided by the algorithm is NULL, or is not allocated in a desirable memory space (e.g., it is in external memory but must be placed in internal memory due to performance requirements, etc.), the application can allocate the buffer in a different location, utilizing the information provided by the algorithm in `algAlloc`. Application then calls `algInit` with the application assigned base address. The XDAIS specification of `algInit` ensures that the algorithm will properly initialize the write-once buffers at the provided base addresses. This behavior ensures backward compatibility in legacy application frameworks.

4.3 Relocation of Write-Once Buffers

The application can relocate the write-once memory buffers by treating them in the same way the persistent buffers (IALG_PERSIST) are treated in the XDAIS specification. The application can copy contents of any write-once buffer to another location and inform every affected algorithm instance by calling the algorithm’s `algMoved` function with the base field of the `IALG_MemRec` memory table structure initialized with the address of the new buffer location.

The application frameworks can do some further optimizations when moving write-once buffers. If the application context switch logic saves instance buffers to external memory and then later restores them back to internal memory, the copy overhead of saving the write-once buffers to external memory can be avoided by caching a copy of the write-once buffers once in a fixed external location, and using the cached read-only content to restore the buffer back. This optimization is possible both when:

1. the buffers gets restored back to the original internal memory base locations, which does not require an `algMoved` call, or when
2. the addresses are different, requiring an `algMoved` call.

Note that, as with all types of instant memory buffers, the IALG interface function, `algMoved`, should be implemented by the algorithm in order to support relocation of write-once buffers.

5 Shared Relocatable Buffers Using Parent Instance Technique

The basic idea behind this technique is to encapsulate a shared structure within an anonymous instance object (called the parent instance) and arrange for the (children) algorithm instances to access the parent instance’s memory. Figure 4 depicts an example of such a sharing scenario where parent instance P exists as a standard standalone instance object, while the (child) algorithm instances A and B each have private non-overlapping scratch and persistent buffers. The dashed boxes labeled “Persistent P” in the child instances (A and B) are used to highlight the hidden and implicit shared state that belongs to the parent P.

In the subsequent sections we illustrate how this technique works by considering the following stages of processing: creation of the parent instance, creation of the child algorithm instance, relocation of the parent instance, and activation of the parent instance.

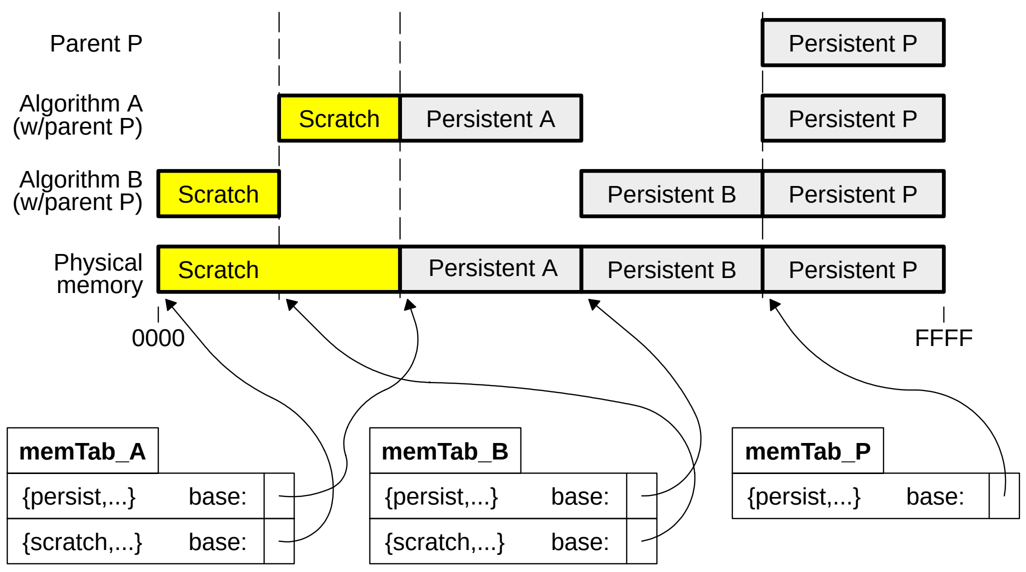

Figure 4. A Parent Instance Memory Sharing Layout

5.1 Creation of Parent Instance

The “discovery” of the presence of a parent instance that must be instantiated before the algorithm instance can be initialized is part of the specification of the `algAlloc` function. XDAIS does not require documentation or other static naming conventions to publish the IALG functions structure (vtable) or creation parameters of an algorithm’s parent module. So you cannot deduce if instances of an algorithm will have a parent or not, just by inspecting the header files provided by the vendor.

The application must provide the address of a pointer to the parent’s IALG_Fxns (`parentFxns`), to the `algAlloc` function. If the algorithm requires a parent instance, it assigns the address of the parent module’s IALG vtable to `parentFxns` . The application then uses it to call the parent module’s standard IALG functions and creates the parent instance like an ordinary instance object. Figure 5 shows an example code fragment for creation of a parent instance.

```
IALG_Handle parentFxns = NULL;
FIR_TI_IFIR.ialg.algAlloc((IALG_Params*)&firParams, &parentFxns, memTab);
[ . . . ]

if (parentFxns != NULL) {

    /* obtain parent instance memory requirements */
    n = parentFxns->algAlloc(NULL, NULL, memTab_P);

    /* allocate and assign memory requested by parent */
    if (_allocateMemory (memTab_P, n) ) {

        parentHandle = (IALG_Handle) memTab_P [0].base;
        parentHandle->fxns = parentFxns;
        parentHandle->fxns->algInit(parentHandle, memTab_P, NULL, NULL);
        /*
         * parentHandle points to successfully initialized instance.
         */
    } else
    {
       <report error: Can’t create parent instance!>
    }
}
```

Figure 5. Creation of Parent Instance

5.2 Creation of Child Instances

Only after creating the parent instance can the application finalize the creation of the child algorithm instance. After allocating the child algorithm instance memory requests, the application passes the parent handle to the algorithm’s `algInit` function. Since the parent and child algorithm modules are co-designed by the algorithm vendor, the child algorithm can initialize its pointers to access the parent’s shared buffer(s). It can either keep a pointer to the parent handle and access the shared buffer via the parent handle, or keep a pointer directly to the shared buffer. The application cannot make any assumptions about how the child instance accesses the parent’s shared memory. Figure 6 shows an example code fragment for creation of the child instance.

```
n = FIR_TI_IFIR.ialg.algAlloc((IALG_Params*)&firParamsA, &parentFxns, memTab_A);
n = FIR_TI_IFIR.ialg.algAlloc((IALG_Params*)&firParamsB, &parentFxns, memTab_B);
[. . .]
/*
 * create instance A as child of parentHandle */
 */
if (_allocateMemory (memTab_A, n) ) {

   handleA = (IALG_Handle)memTab_A[0].base;
   ((IFIR_Handle) handleA)->fxns = &FIR_TI_IFIR;

   /*
    * initialize instance A: assign memory & the parent instance
    */
   ((IFIR_Handle) handleA)->fxns->ialg.algInit(handleA, memTab_A, parentHandle,
                                               (IALG_Params*)&firParamsA);
}

/*
 * create instance B as child of parentHandle */
 */
if (_allocateMemory (memTab_B, n) ) {

   handleB = (IALG_Handle)memTab_B[0].base;
   ((IFIR_Handle) handleB)->fxns = &FIR_TI_IFIR;

   /* initialize instance B */
   ((IFIR_Handle) handleB)->fxns->ialg.algInit(handleB, memTab_B, parentHandle,
                                               (IALG_Params*)&firParamsB);
}
```

Figure 6. Creation of Child Instances

5.3 Relocation of Parent Instance

The buffers assigned to the parent instances can be relocated after creation only if both the child and parent algorithm modules implement the IALG `algMoved` function. Since child instances keep a reference to the shared buffers that are owned by the parent instance, any relocation related with the parent instance requires informing the child instances about the relocation, in addition to informing the parent instance about the relocation. So, first the parent must be informed about the relocation by calling its `algMoved` function, which results in the parent instance updating its references. Subsequently, the application must call the child instance's `algMoved` function, passing the parent's handle.

5.4 Activation of Parent Instance

If the parent has declared any scratch buffers, it must also implement the `algActivate` function. In that case, an additional requirement for the child instance is to activate the parent before any of its processing functions can be called. Otherwise, the child instances may access corrupted shared state.

Additionally, a parent instance cannot be deactivated if there is at least one child instance that is still active, since the child's de-activation function may need to save some state in a shared parent scratch buffer.

6 Summary and Conclusion

The static global memory buffer approach is the simplest technique for sharing global read-only tables and structures. When the size of shared memory that needs to be allocated in internal memory is large, this approach can be combined with scratch buffering or write-once buffering techniques described in subsequent sections.

The standard algorithm activation/deactivation functions help application developers perform efficient and general purpose overlay management of scratch buffers. For sharing non-scratch type memory the parent instance mechanism or write-once buffer techniques can be used if supported by the algorithm.

Write-once buffers offer a simple and efficient approach but are applicable only to persistent read-only type buffers, whereas the parent instance technique is general. When parent instances are used the algorithm developer decides whether the algorithm instances are to share some buffers or not. However, with write-once buffers, the algorithm informs the application about its read-only buffers and lets the application make the sharing decision. It is possible that some applications require more optimization than others and an algorithm should run correctly regardless of whether its instances share the read-only buffers or not.

Parent instances and write-once memory types provide a standard and powerful mechanism for sharing common global persistent structures, read-only tables, etc., across multiple algorithm instances from the same vendor. The primary challenge encountered by the application designers using these techniques is the need to keep track of all memory assignments as well as the instance sharing information and other implicit or explicit dependencies between algorithm instances and parents and their children.

Algorithm developers should implement the simplest mechanism that gives the application designers enough flexibility to be able to integrate the algorithm.

7 References

1. *TMS320 DSP Algorithm Standard Rules and Guidelines*, SPRU352.
2. *TMS320 DSP Algorithm Standard API Reference*, SPRU360.