Application Report
SPRA580B

Using the TMS320 DSP Algorithm Standard in a Dynamic DSP System

Carl Bergman *Digital Signal Processing Solutions*

Abstract

This application note illustrates some techniques used to manage data memory when using the TMS320 DSP Algorithm Standard. As system complexity grows and more functions and features are integrated into one system, data memory becomes increasingly precious. We can make the most of this precious resource by 'dynamically' sharing it between algorithms.

Contents

Figures

Definition of a Dynamic System

A dynamic system is one in which system memory may be reclaimed and reused while the application is executing. The details become very complex when the requirements of real-time algorithms and multi-tasking operating systems are considered. To keep things manageable, we will consider the case of a single-tasking system where only the data memory is reused.

Example

We will illustrate how memory is used in a dynamic system with a hypothetical secure-voice telephone. This phone has a built-in DSP with the ability to digitize and encrypt voice. When not used as a phone, the DSP has a voice-recognition algorithm that communicates with a desktop computer. These two demanding applications, voice recognition and secure voice, require the system to reconfigure data memory depending on which application is currently running. Before we go any further, let's review how the TMS320 DSP Algorithm Standard, which is part of TI's eXpressDSP (XDAIS) technology initiative, manages algorithms.

A Review of TMS320 DSP Algorithm Standard Object and Memory Management

The Create Function

The create function allocates memory for an instance of the algorithm and calls the algorithm's initialization function. Let's step through the create process to understand what takes place.

The Function Table

The starting point for any operation involving a standard algorithm is the `IALG_Fxns` structure. This function table is the means through which services are made available. The function table for the 'example' (`XMP`) algorithm looks like this:

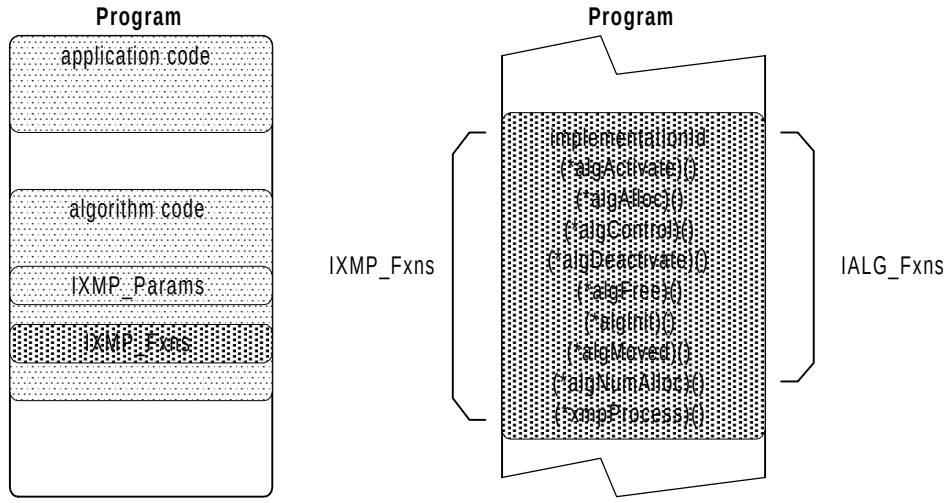
```
IXMP_Fxns XMP_IXMP ={
        &XMP_IXMP,      /* implementationId */
        xmpActivate,    /* algActivate   */
        xmpAlloc,       /* algAlloc      */
        xmpControl,     /* algControl    */
        xmpDeactivate,  /* algDeactivate */
        xmpFree,        /* algFree       */
        xmpInit,        /* algInit       */
        xmpMoved,       /* algMoved      */
        xmpNumAlloc,    /* algNumAlloc   */
        xmpProcess      /* algorithm specific processing */

  };
```

The first field of the above table is the unique implementation ID. The next 8 fields are the function pointers for the standard IALG functions. The last field is the function pointer for the algorithm-specific data processing function.

Figure 1 illustrates the function table and its location in memory. Note that the `IXMP_Fxns` structure differs from the `IALG_Fxns` structure in the last field.

Figure 1. The Function Table



The Algorithm-Specific Create Function

The `XMP_create()` function is the high-level, algorithm-specific function that creates an algorithm instance and returns the address of the instance object.

```
XMP_Handle handle;

handle = XMP_create(&XMP_IXMP, NULL);
```

`XMP_create()` accepts a pointer to the function table and a pointer to the creation parameters as its parameters (see Figure 2). The creation parameter pointer may be `NULL` if we are willing to accept the default creation parameters.

If we use a `NULL` parameter pointer, the following operation will take place.

```
if (prms == NULL) {
    prms = &XMP_PARAMS;
}
```

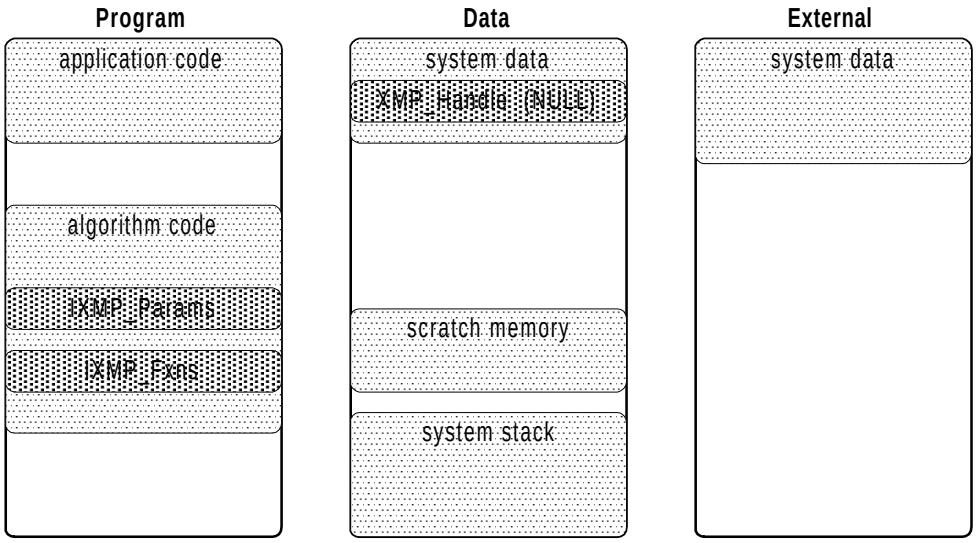
After ensuring that the creation parameters are defined, the generic `ALG_create()` function is called:

```
handle = (XMP_Handle)ALG_create (
    (IALG_Fxns *)fxns,  /* function table */
    (IALG_Handle) NULL, /* parent object or NULL */
    (IALG_Params *)prms /* creation parameters */
```

```
);
```

The generic `create` function does the remainder of the work: determines memory requirements, allocates memory, and initializes the new object.

Figure 2. Key Data Elements Used for Create



Memory Requirements

The next step in creating an instance of an algorithm is to allocate the required memory. The algorithm makes its memory requirements known by filling in a memory descriptor table.

Here is how we create this table:

The function `algNumAlloc()` tells us the maximum number of memory blocks the algorithm needs. If `algNumAlloc` is not implemented, then the default number of memory records is used. Currently, the default, `IALG_DEFMEMRECS`, is set to 4.

```
 /* find out the maximum number of records needed */
numRecs = fxns->algNumAlloc();
```

When the number of records is known, a table of memory descriptors (`IALG_MemRec`) is allocated.

```
/* allocate space for the memory descriptor table */
/* allow maximum number of records specified by algNumAlloc*/
IALG_MemRec *memTab;

memTab = malloc(numRecs * sizeof(IALG_MemRec));
```

This table is passed to `algAlloc()` to be filled in. `algAlloc()` fills in all the fields except the base address.

```
/*  fill in the memory descriptor table */
/*  returned value is actual number of records initialized */
/* it might be different than the maximum records required */
numRecs = fxns->algAlloc(prms, NULL, memTab);
```

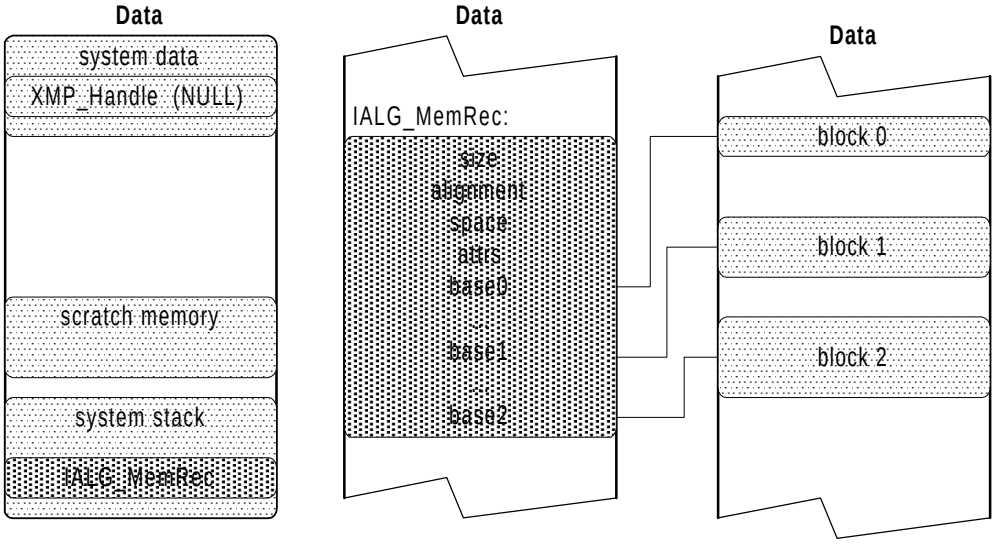
Allocate Memory.

Each memory descriptor describes the following characteristics:

```
size (in bytes)
alignment (byte boundaries)
space (single access, dual access, external)
attributes (scratch, persistent, write-once)
base address
```

The memory allocation routine (typically a system function call) must reserve memory that matches these characteristics and fill in the base address. When the memory descriptor table is complete, it is passed to `algInit()`.

Figure 3. Memory Descriptor Table



Initialize the Object

By convention, the first memory descriptor is for the algorithm instance object (`IALG_OBJMEMREC`) is defined as 0. The handle to the object is set to the corresponding base address.

```
/*  note: IALG_OBJMEMREC is defined as 0 */

    handle = memTab[IALG_OBJMEMREC].base;
```

The object's function table pointer is initialized, making it possible to access algorithm functions.

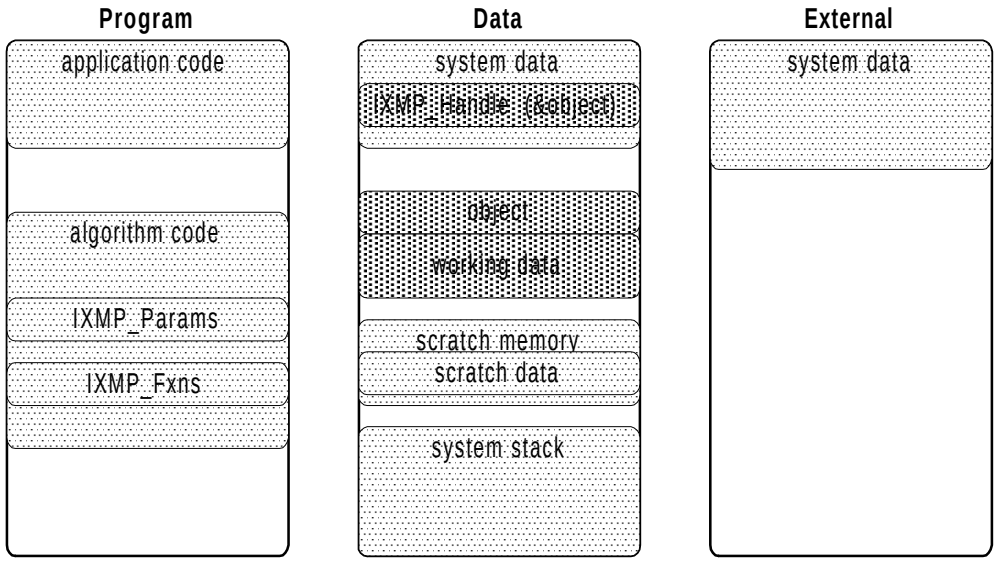
```
    handle->fxns = fxns;
```

Finally, the `algInit()` function is called to complete the initialization of the instance object and the algorithm.

```
    /* init the algorithm */
    if (fxns->algInit(handle, memTab, NULL, prms) == IALG_EOK) {
        return(handle);
    }
```

This completes the creation process. The returned handle is used for any future communication with the algorithm.

Figure 4. Algorithm Instance Created



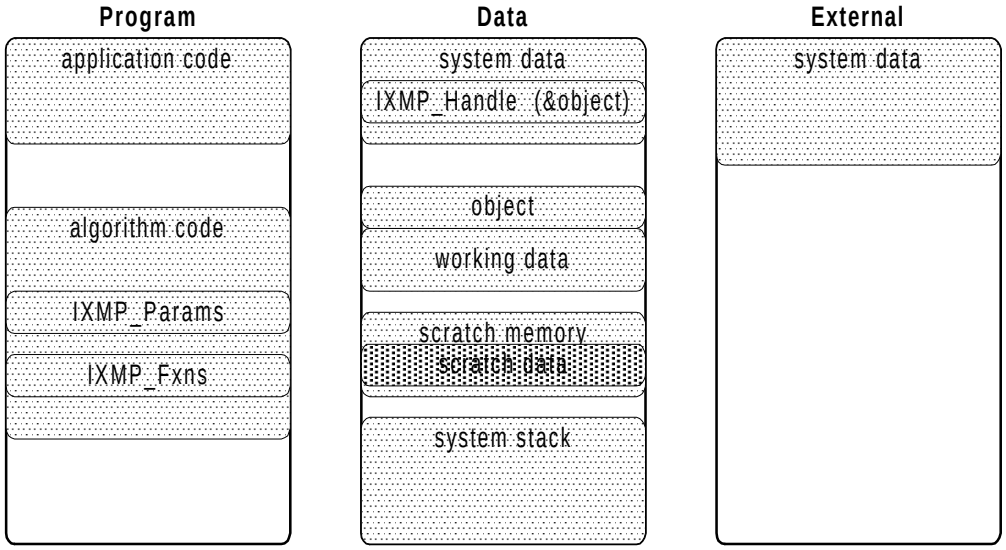
The Activate Function

The implementation of `algActivate()` and `algDeactivate()` is optional. If they are not implemented, a `NULL` is entered in the corresponding field of the function table.

If they are implemented, these functions manage the scratch and shared memory. The activate function prepares the scratch data to enable the algorithm to run. The deactivate function saves what is necessary from the scratch data so that another algorithm can run.

It is important to remember that `algActivate()` must be called before calling `algMoved()`. This ensures that the algorithm's data is ready to run and that any pointer calculation will reflect the 'ready to run' state of the system.

Figure 5. Algorithm Activate



The Move Function

When it is necessary to relocate the algorithm's data, the application notifies the algorithm with a call to `algMoved()`. The same parameters that are passed to `algMoved()` are also passed to `algInit()`. The only difference is in the pointer values. The algorithm must update its internal data references based on the new pointer values. This function is optional, and if not implemented, a `NULL` is entered in the corresponding field of the function table. Remember that the algorithm must be 'active' before calling `algMoved()`.

Figure 6 illustrates memory that is 'defragmented' by moving the algorithm's data so that it is contiguous with other existing blocks of memory.

Figure 6a. Relocating Data with algMoved()

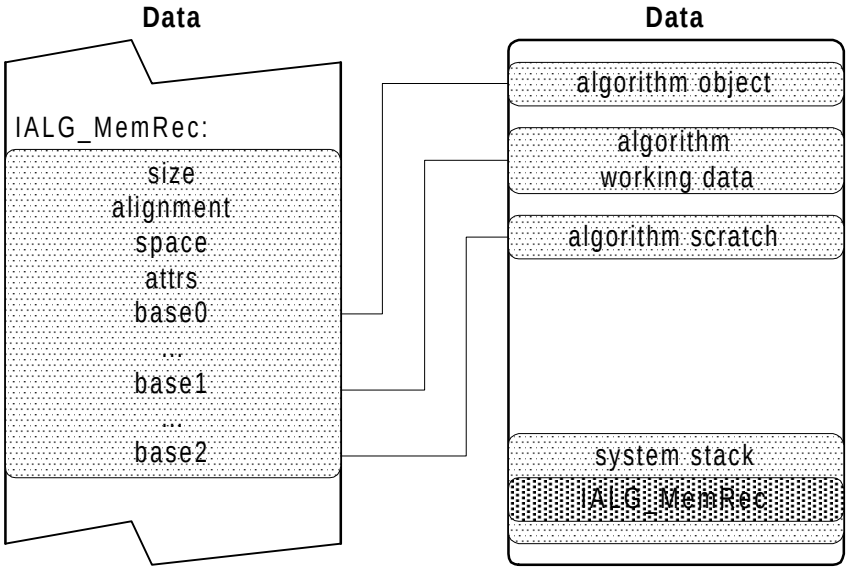
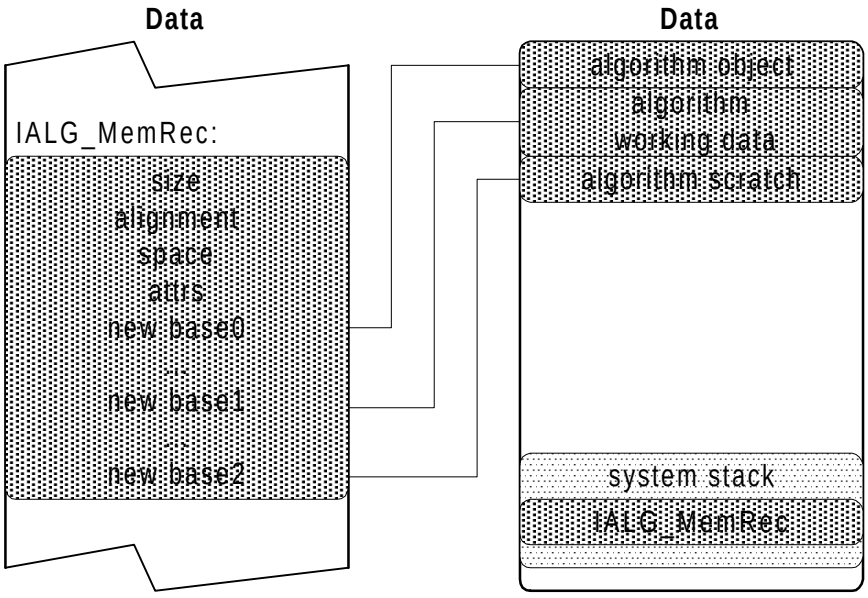


Figure 6b. After Move, Memory Descriptor Points to New Location



The Delete Function

The object management function, `ALG_delete()`, calls the IALG function `algFree()`. The `algFree()` function operates in a way similar to `algAlloc()` in that it returns a completed memory descriptor table. The algorithm's memory is freed, based on the information returned to the application in the table. There is no change to the program memory - the algorithm code still exists in the system code image.

Why a Dynamic System?

It is common in DSP systems to have both on-chip and external memory. To get the most out of a DSP, the algorithm must keep its data structures in on-chip memory with its faster access times. Inevitably though, the number of features desired for the system exceeds the ability of on-chip memory to support them. When that happens, the dynamic use of on-chip memory becomes desirable. Furthermore, if we can quickly switch between functions, and quickly move an algorithm's working data between on-chip and off-chip memory, we can achieve the effect of parallel processing.

How Does a Dynamic System Work?

Our illustration of a dynamic system will be a telephone with DSP capability. This specialized (and hypothetical) telephone operates in the normal way; additionally, it has the ability to do voice recognition and also to establish a secure voice connection. The system in our illustration has the following characteristics:

The system is non-preemptive:

- Task or algorithm switching happens under application control.

The program memory usage is static:

- There is adequate program memory and all of the application code fits in program memory.

The data memory usage is dynamic:

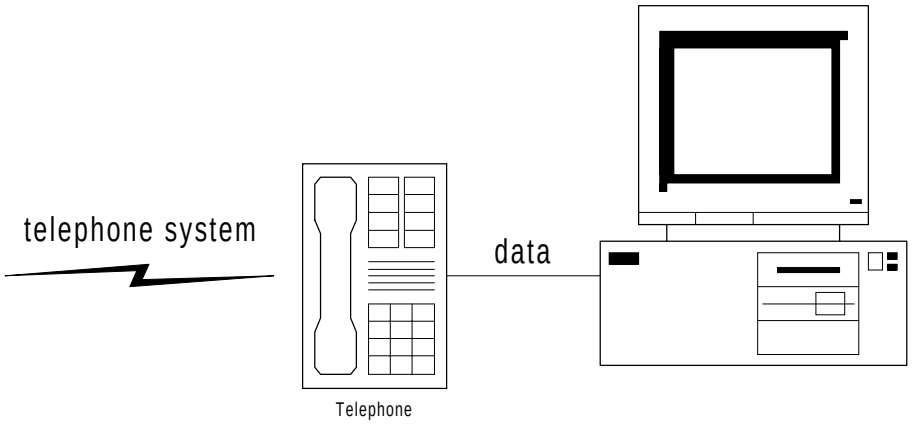
- On-chip memory is limited and must be shared. There is adequate off-chip memory.

We will run through the following scenario to show how data memory can be managed. Through this sequence we will use the algorithm standard interface to manage algorithm instances.

1. Power on. The voice recognition software (via the speaker phone) provides voice input and output for the computer.
2. Initiate a call. The DTMF tone generation and echo cancellation software are used to make a normal phone call.
3. Initiate secure voice. The secure voice algorithm provides us with an encrypted voice connection to the compatible telephone at the other end.
4. The call is discontinued. Back to the voice recognition mode for communication with the computer.

Figure 7 illustrates the system.

Figure 7. The Telephone in Our Example

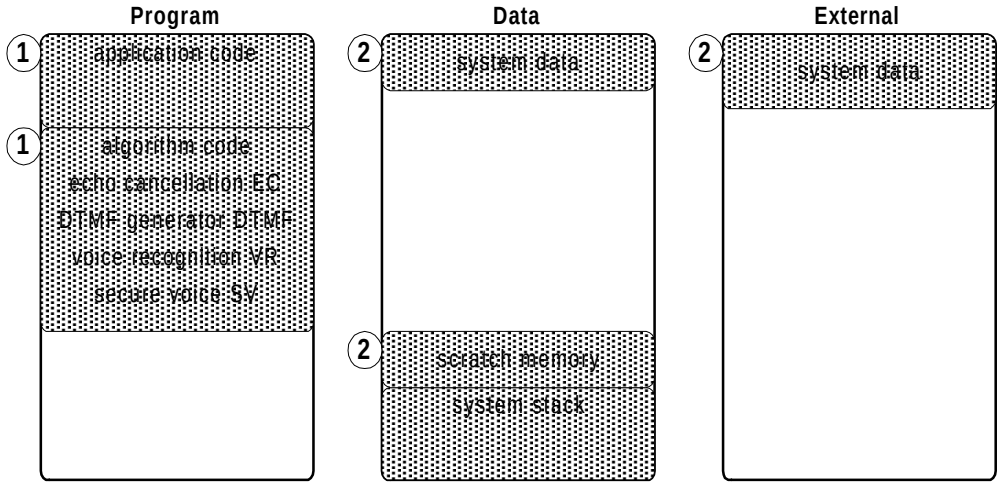


The Power-On State

At power-on, the code for the telephone application is stored in an inexpensive low-speed ROM. The on-chip program memory, on-chip data memory, and external memory are all RAM and are undefined.

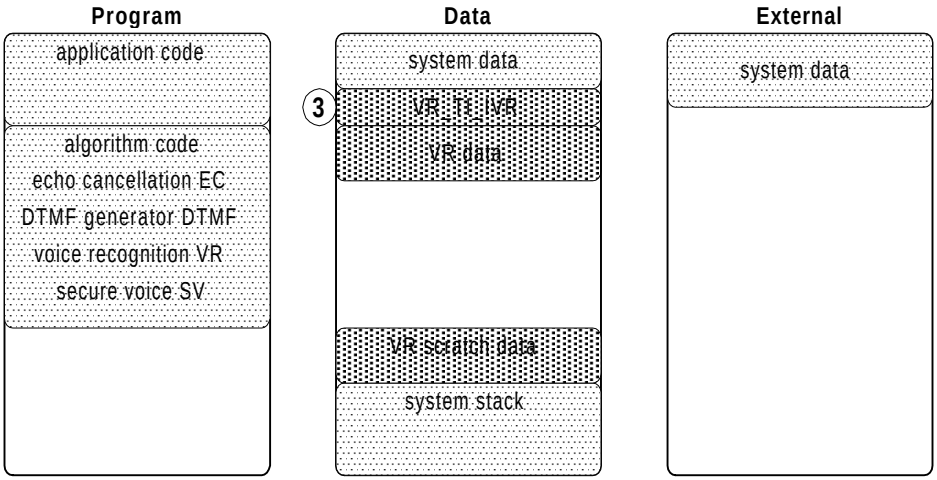
The system boot program loads the application into on-chip program memory from the ROM (Figure 8 (1)). The initialization code runs (Figure 8 (2)) and establishes the system stack in data RAM and establishes whatever other data structures the system needs in data RAM or external RAM.

Figure 8. Memory Diagram at Power On



At this point, the application is running with no algorithms active. Since the default state of the phone is with voice-recognition active, the application creates an instance of the voice recognition algorithm and activates it (Figure 9 (3)). This enables voice commands via the speakerphone to the computer. Note the addition of the algorithm object, working data, and scratch data to the memory usage diagram in Figure 9.

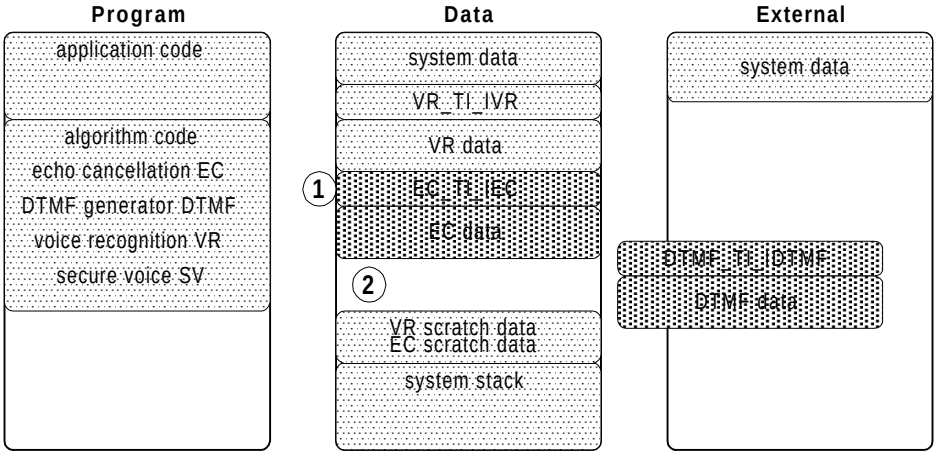
Figure 9. Algorithm Instance Data



Initiating A Call

To initiate a call, the echo cancellation and DTMF tone generation algorithms must be created and activated. Referring to the memory usage diagram, the echo cancellation can be instantiated (Figure 10 (1)) but there is not enough data memory for the DTMF tone generator (Figure 10 (2)).

Figure 10. Algorithm Can Not Be Created

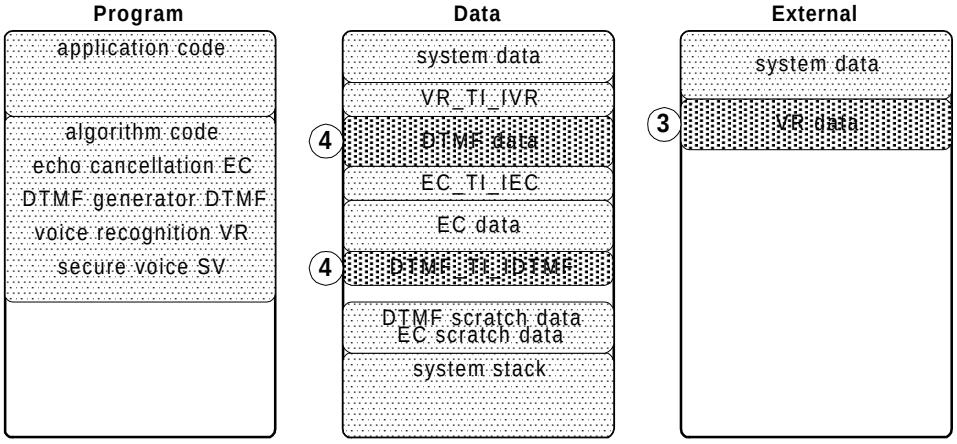


Our solution is to swap out the private data memory used by voice recognition (Figure 11 (3)). The steps are:

1. Copy the voice recognition's private data memory to a location in external memory.
2. Make a call to `algMoved()` to inform the algorithm where its data is.
3. Call `VR_deactivate()` to preserve the current state of the algorithm.

With this done there is room to instantiate the DTMF tone generator (Figure 11 (4)).

Figure 11. Enough Space Available for Another Algorithm



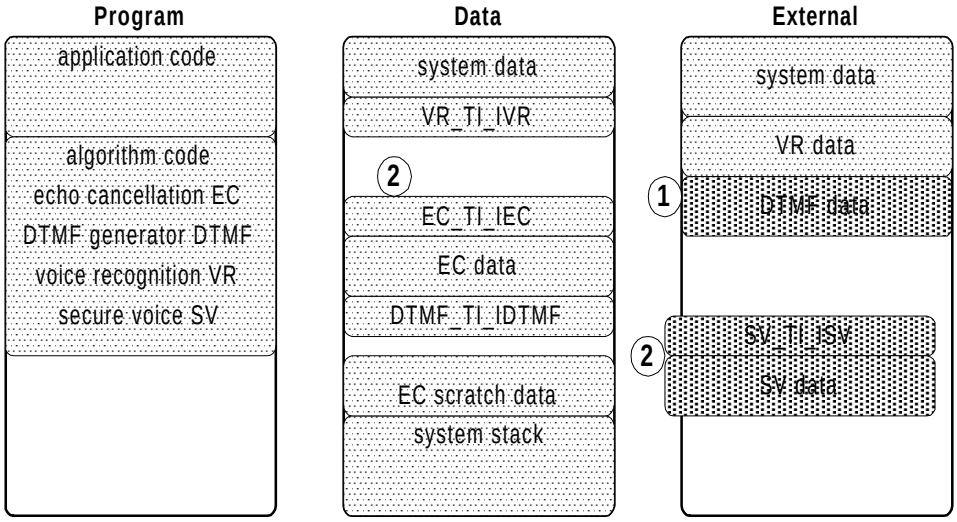
Note that if we want to move an algorithm's data and deactivate it, we must do them in this sequence: move data, call `algMoved()`, call `algDeactivate()`. This ensures the algorithms' data pointers are current the next time it is activated.

Switching to Secure Voice

The combination of voice compression, data encryption, and modem functionality allows us to have a secure communications channel to a compatible telephone at the distant end.

Looking again at the data memory usage map from the previous step, we see there is very little free memory. To start the secure voice algorithm we first have to make some room. Since the tone generator will not be used while in secure-voice mode, the first step is to move its private data memory and deactivate it (Figure 12 (1)).

Figure 12. Again, More Memory Needed



Still looking for more memory (Figure 12 (2)) we delete the voice-recognition algorithm (Figure 13 (3)). Now we have enough free memory, but not in a big enough block. We will solve this problem by relocating the object and private data memory for the echo cancellation. Let's look again at the move sequence.

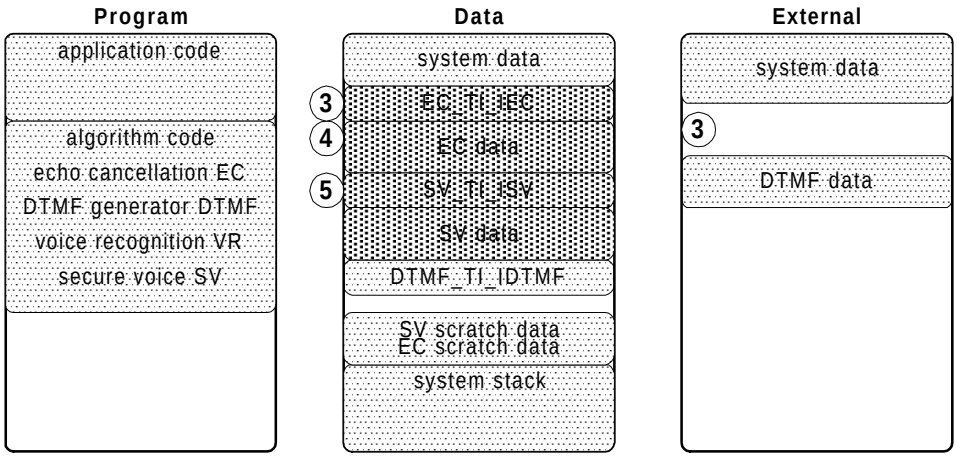
We must build a new memory descriptor table representing the new locations of the algorithm's memory and pass the table to the algorithm via a call to `algMoved()`.

This is the sequence:

1. Allocate a memory descriptor table.
2. Pass it to the `algAlloc()` routine to get it initialized to the same values that were used when the algorithm was created.
3. Fill in the new addresses for the algorithm's data memory.
4. Copy (or DMA) the contents of the memory to the new location.
5. Call the `algMoved()` function with the updated memory descriptor table.

With the echo cancellation data moved (Figure 13 (4)) there is enough room to create an instance of the secure-voice algorithm (Figure 13 (5)).

Figure 13. Secure Voice Algorithm Instantiated



Our telephone conversation continues on the newly created secure channel, over the same phone lines.

Hanging Up the Phone

Hanging up the phone returns us to the voice-recognition mode for our computer through the following sequence.

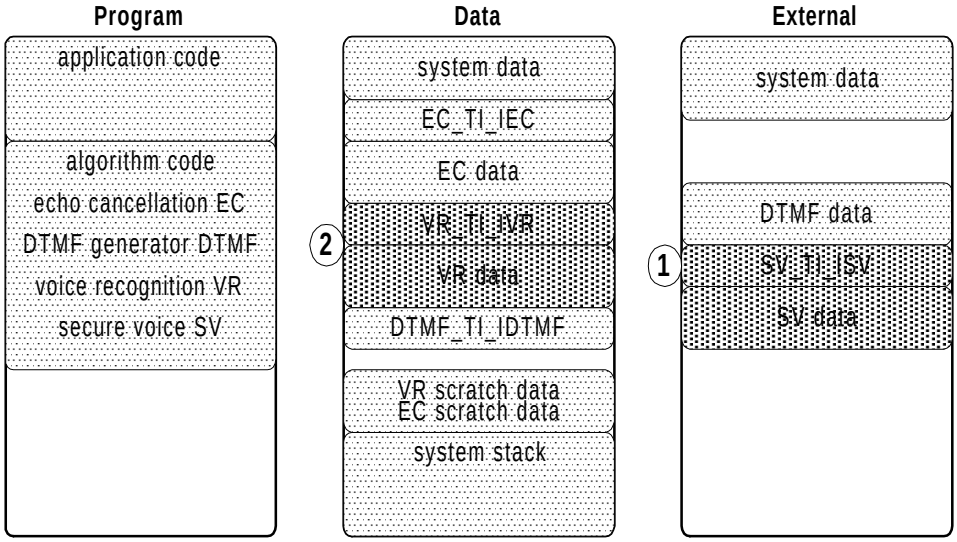
1. The secure voice object and private data memory are moved to external memory (Figure 14 (1)). Then the algorithm is deactivated, freeing up the scratch memory it was using.

2. The voice-recognition algorithm is re-created (Figure 14 (2)) and activated, establishing it's object and private data memory and scratch memory.

3. The DTMF tone generator is simply deleted and its memory reclaimed.

The computer is available via voice commands until the next phone call.

Figure 14. Voice Response Active Once Again



Conclusion

Although hypothetical, this limited example shows how the functions provided by the TMS320 DSP Algorithm Standard interface can be used to manage algorithms in a dynamic system.

The functions to create, activate, move, deactivate, and delete algorithm instances provide the ability to manage our memory usage as the system requirements change in a running system.

References

1. *TMS320 DSP Algorithm Standard Rules and Guidelines*, SPRU352.
2. *TMS320 DSP Algorithm Standard API Reference, SPRU360.*
3. *Making DSP Algorithms Compliant with the TMS320 DSP Algorithm Standard, SPRA579.*
4. *Using the TMS320 DSP Algorithm Standard in a Static DSP System, SPRA577.*

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.