TEXAS INSTRUMENTS

Application Report
SPRA147C – April 2003

# Reference Frameworks for eXpressDSP Software: API Reference

*Alan Campbell*
*Davor Magdic*
*Todd Mullanix*
*Vincent Wan*

*Texas Instruments, Santa Barbara*

**ABSTRACT**

Reference Frameworks for eXpressDSP Software are provided as starterware for developing applications that use DSP/BIOS and the TMS320 DSP Algorithm Standard (also known as XDAIS). Several API modules have been created for use by these frameworks. These modules may be used with the Reference Frameworks or in other applications.

Some of the modules pertain to specific RF levels. Others (for example, the UTL module) are used in several of the frameworks.

This application note provides details about the functions in these API modules. Complete source code for these modules is provided with the Reference Frameworks.

Code Composer Studio, DSP/BIOS, eXpressDSP, and TMS320 are among the trademarks of Texas Instruments. See www.ti.com for a list of trademarks and registered trademarks belonging to Texas Instruments.

**Contents**

**1 Overview ... 4**
Folder Structure Overview ... 5
Rebuilding Libraries ... 5
**2 About IALG Interface Implementations ... 6**
IALG Functions ... 7
The ALGMIN Module ... 8
The ALGRF Module ... 9
**3 About the UTL Module ... 10**
**4 ALGMIN Module ... 12**
ALGMIN_activate ... 12
ALGMIN_deactivate ... 13
ALGMIN_exit ... 13
ALGMIN_init ... 13
ALGMIN_new ... 14
**5 ALGRF Module ... 15**
ALGRF_activate ... 16
ALGRF_control ... 16
ALGRF_create ... 17
ALGRF_createScratchSupport ... 17
ALGRF_deactivate ... 18

ALGRF_delete ........ 19
ALGRF_deleteScratchSupport ........ 19
ALGRF_exit ........ 20
ALGRF_init ........ 20
ALGRF_setup ........ 20
**6 CHAN Module ........ 21**
CHAN_close ........ 22
CHAN_create ........ 23
CHAN_delete ........ 23
CHAN_execute ........ 24
CHAN_exit ........ 25
CHAN_getAttrs ........ 25
CHAN_init ........ 25
CHAN_open ........ 26
CHAN_regCell ........ 27
CHAN_setAttrs ........ 28
CHAN_setup ........ 28
CHAN_unregCell ........ 29
**7 ICC Module ........ 30**
ICC_exit ........ 32
ICC_getBuf ........ 32
ICC_init ........ 33
ICC_linearCreate ........ 33
ICC_linearDelete ........ 34
ICC_setBuf ........ 34
**8 ICELL Interface ........ 35**
**9 SCOM Module ........ 38**
SCOM_create ........ 39
SCOM_delete ........ 40
SCOM_exit ........ 40
SCOM_getMsg ........ 41
SCOM_init ........ 41
SCOM_open ........ 41
SCOM_putMsg ........ 42
**10 SSCR Module ........ 43**
SSCR_createBuf ........ 44
SSCR_deleteBuf ........ 45
SSCR_exit ........ 46
SSCR_getBuf ........ 46
SSCR_init ........ 46
SSCR_prime ........ 47
SSCR_setup ........ 48
**11 UTL Module ........ 49**
UTL_assert ........ 51
UTL_logDebug ........ 52
UTL_logError ........ 53
UTL_logMessage ........ 53
UTL_logWarning ........ 54
UTL_setLogs ........ 54

UTL_showAlgMem ..... 55
UTL_showAlgMemName ..... 55
UTL_showHeapUsage ..... 55
UTL_stsDefine ..... 56
UTL_stsPeriod ..... 56
UTL_stsPhase ..... 57
UTL_stsReset ..... 57
UTL_stsStart ..... 58
UTL_stsStop ..... 58
**12 References ..... 59**

## Figures

Figure 1. Elements of a Reference Framework ..... 4
Figure 2. IALG Interface Function Call Order ..... 7
Figure 3. UTL Debugging Levels ..... 10
Figure 4. UTL Message Example ..... 11
Figure 5. ALGMIN Function Calling Sequence ..... 12
Figure 6. ALGRF Function Calling Sequence ..... 15
Figure 7. CHAN Function Calling Sequence ..... 22
Figure 8. ICC Data Flow Example ..... 31
Figure 9. ICC Function Calling Sequence ..... 32
Figure 10. SCOM Function Calling Sequence ..... 39
Figure 11. Scratch vs. Persistent Memory Allocation ..... 43
Figure 12. SSCR Function Calling Sequence ..... 44
Figure 13. Setting UTL_DBGLEVEL ..... 50

## Tables

Table 1. Modules Described in this Application Note ..... 4
Table 2. IALG Implementation Characteristics ..... 6
Table 3. ICELL_Obj Elements ..... 36
Table 4. ICELL_Fxns Functions ..... 37
Table 5. UTL Module Classes ..... 50

# 1 Overview

This application note provides reference information for modules created for use by the Reference Frameworks for eXpressDSP Software. These modules may be used with the Reference Frameworks or in other applications.

Figure 1 shows the elements that make up a Reference Framework on the target DSP. The modules described in this application note are part of the "Framework Components" element in this figure.

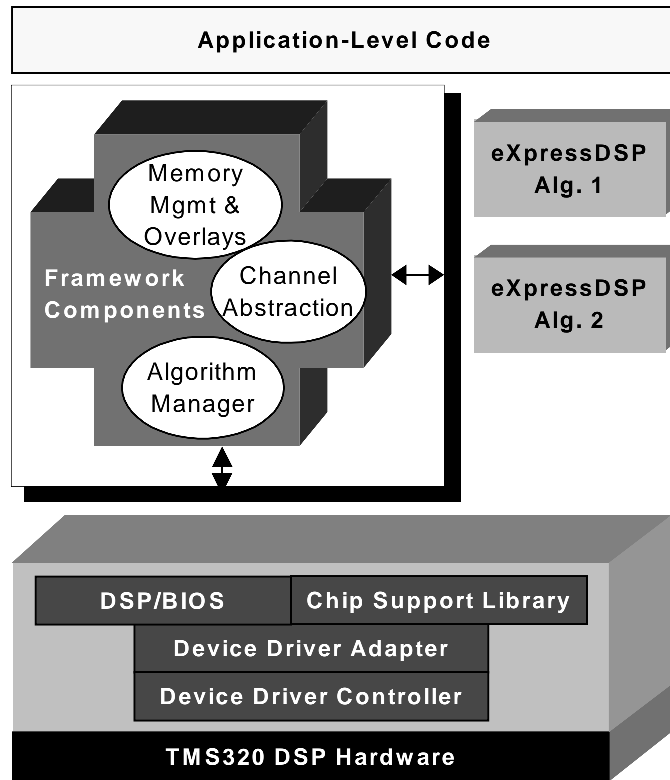


**Figure 1. Elements of a Reference Framework**

Table 1 lists the modules used in the current suite of Reference Frameworks.

**Table 1. Modules Described in this Application Note**

| Component | Description | Used In |
|---|---|---|
| ALGMIN | Minimum-footprint IALG interface implementation. Applies static buffer techniques instead of using dynamic (heap-based) memory allocation. | RF1 |
| ALGRF | IALG interface implementation. Provides functions for creating and deleting instances of XDAIS algorithms. The functions are similar to the functions in CCStudio's ALG module, but more efficient and controllable. | RF3, RF5 |
| CHAN | Manage execution of algorithms in cells. | RF5 |
| ICC | Manage inter-cell communication assignment. | RF5 |
| SCOM | Synchronized communication module | RF5 |
| SSCR | Shared scratch memory module. | RF5 |
| UTL | Supports debugging and diagnostics. | RF1, RF3, RF5 |

Complete source code for these API modules is provided with the Reference Frameworks. Typically, they need little or no modification.

## Folder Structure Overview

For each module described in this application note, files are located in the following folders (where *RF_DIR* is the top-level "referenceframeworks" folder):

- ***RF_DIR*\include.** Contains public header files. Public header files are typically referenced by both the module itself and the framework. In contrast, private header files are stored with the source code that includes them. Each module has one header file. Example: algmin.h.
- ***RF_DIR*\lib.** Contains pre-built library files for this module. Where possible, a library file is provided for each supported compilation model. For example, algmin.l54 (near model), algmin.l54f (far model), algmin.l55, algmin.l55l, algmin.l62, and algmin.l64.
- ***RF_DIR*\src.** Root folder for the module source code and project files (.pjt). A readme.txt file is provided in each subfolder. These files tell which frameworks use the module and answer questions about each module.

In addition to these modules, Reference Frameworks provide "dummy" algorithms, device adapters, and mini-drivers used by the pre-built frameworks. For information about the algorithms, see the application notes for each Reference Framework. For information about device adapters and mini-drivers, see the *DSP/BIOS Driver Developer's Guide* (SPRU616).

Source code is provided with Reference Framework modules for two reasons. First, you can modify and recompile libraries if you wish. Second, source code is provided for debugging purposes. If you halt execution while in a library module, CCStudio asks if you want to find and open the source file for the module. This allows you to step into module functions and inspect internal and external variables, even if you do not intend to modify the code. (Source code for driver-related modules is not included in the Reference Frameworks distribution, but is available as part of the DSP/BIOS Driver Developer's Kit.)

**Hint:** In CCStudio, using Options→Customize→Directories menu option, you can specify which folders CCStudio should search to locate the source file. If you specify source code folders for the modules used in RF3, CCStudio opens windows with their source code automatically as you step into a library module procedure.

## Rebuilding Libraries

Libraries are built with debugging enabled (-g) and no optimization. For performance reasons you may wish to rebuild the libraries using optimization switches for post-development versions of your applications.

By default, RF3 and RF5 modules are built with UTL_DBGLEVEL set to 70. By default, RF1 is built with UTL_DBGLEVEL set to 60.

If you rebuild a library and then rebuild the Reference Framework application, either delete the executable file (app.out) or use Project→Rebuild All in order to build with the new library. CCStudio does not currently check for dependencies on rebuilt libraries. This problem will be corrected in the near future.

## 2 About IALG Interface Implementations

A fundamental requirement of the TMS320 DSP Algorithm Standard (also known as XDAIS) is that all algorithms implement the IALG interface to define their memory requirements. This enables efficient use of on-chip data memories in client applications. The uniform memory management scheme enables multiple algorithms to co-exist without contention in a single application.

Two implementations of the IALG interface are available in the Reference Frameworks. Each implementation has different strengths and characteristics. Table 2 compares these implementations.

**Table 2. IALG Implementation Characteristics**

| | ALGMIN | ALGRF |
|---|---|---|
| **Provided with** | Reference Frameworks | Reference Frameworks |
| **Best used with** | RF1<br>(Compact, static systems) | RF3 and higher |
| **Memory allocation** | Static | Dynamic; uses DSP/BIOS MEM module |
| **Key points** | • Super-compact.<br>• ALGMIN_new is key instantiation function.<br>• ALGMIN_new calls algInit but not algAlloc since buffers are pre-generated. | • Uses DSP/BIOS MEM module for dynamic allocation.<br>• Smaller footprint than the CCStudio-supplied ALG implementation.<br>• Supports sharing of scratch memory. |

RF1 provides an implementation of the IALG interface called ALGMIN. In keeping with the aim of RF1, it is a "small but effective" module. Totaling no more than 100 lines of source code, it serves to statically instantiate any XDAIS algorithm. ALGMIN is the smallest IALG implementation.

Higher-level Reference Frameworks typically use the ALGRF module to create, configure, and delete instances of XDAIS algorithms. Yet another implementation exists in the form of the ALG module supplied with CCStudio. ALG is for general-purpose use, while ALGRF is tuned to effectively use the DSP/BIOS MEM module for memory allocation. ALGRF is also smaller and more controllable than ALG.

Naturally these modules are mutually exclusive. Only one should be used in a particular end-application.

## IALG Functions

The IALG memory interface defines various types and constants with the key element being a global structure of type IALG_Fxns. It contains a set of function pointers, which is commonly called the v-table. Some of the functions are optional, while algAlloc(), algInit() and algFree() must always be implemented.

```
typedef struct IALG_Fxns {
  Void *implementationId;
  Void (*algActivate)(IALG_Handle);
  Int  (*algAlloc)(const IALG_Params *, struct IALG_Fxns **, IALG_MemRec *);
  Int  (*algControl)(IALG_Handle, IALG_Cmd, IALG_Status *);
  Void (*algDeactivate)(IALG_Handle);
  Int  (*algFree)(IALG_Handle, IALG_MemRec *);
  Int  (*algInit)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *);
  Void (*algMoved)(IALG_Handle, const IALG_MemRec *, IALG_Handle, const IALG_Params *);
  Int  (*algNumAlloc)(Void);
} IALG_Fxns;
```

It is the application's responsibility to initialize a pointer, usually of type IALG_Handle, to point to the v-table structure when creating an instance of the algorithm. This provides access to each of the algorithm methods through the function table, without necessarily exposing the vendor's specific function names. The algorithm enables this operation by specifying an instance object containing all of the code's state or context information. Its first field is always of type IALG_Obj, which, in turn, makes the IALG functions accessible to the client. The reserved field memTab[0] allows the application to point to the instance object thus implying reentrancy since all read/write algorithm data memory is encapsulated in a per-channel structure.

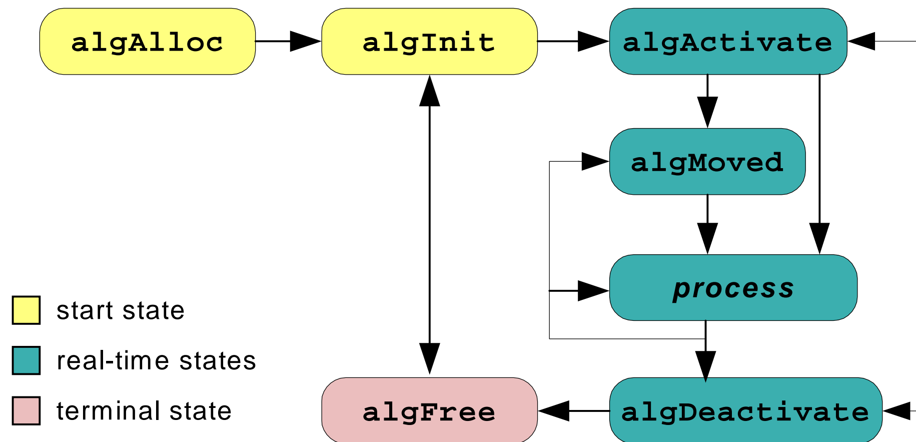


**Figure 2. IALG Interface Function Call Order**

- **algAlloc().** Returns a table of memory records that describe the size, alignment, type, and memory space of all buffers required by an algorithm.
- **algInit().** Based on the information retrieved from the memTab[ ] descriptor structure, the application allocates the requested memory before calling the algInit() initialization function. algInit() performs all initialization necessary to complete the run-time creation of an algorithm instance. After a successful return from algInit(), the object is ready to be used to process data.

- **algActivate() and algDeactivate().** These methods give the algorithm a chance to initialize and save scratch memory outside the main processing loop. Partitioning memory into two groups, scratch and persistent, allows data placement flexibility and footprint optimization. An algorithm can freely use scratch memory without regard to its prior contents. Persistent memory is used to lock down state information that must be preserved between successive invocations by the application. This distinction enables allocation optimization through scratch buffer overlays.
- **algFree().** The algorithm must make the client aware of the current base addresses and size of each memory block previously requested in algAlloc(), so that the application can delete the instance object and all its buffers without creating memory leaks.

Complete technical details on the IALG interface can be found in *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352) and the *TMS320 DSP Algorithm Standard API Reference* (SPRU360). See Section 12, References, page 59 for a list of related documentation and application notes.

## The ALGMIN Module

RF1 provides an implementation of the IALG interface called ALGMIN. In keeping with the aim of RF1, it is a "small but effective" module. Totaling no more than 100 lines of source code, it can statically instantiate any XDAIS algorithm.

ALGMIN requires use of the RF1 genbufs application or some other method of generating the data buffers required the algorithm. No dynamic memory is allocated from a heap when instantiating an algorithm. Instead, purely static techniques are used—thus reducing footprint size.

Why bother with a separate module for the small amount of code in ALGMIN? The answer is that the ALGMIN module makes RF1 easier to adapt and reuse in other applications. In fact ALGMIN can be used independently of RF1 if required.

The application requirements are simply to encapsulate the individual pre-generated buffers as an array of buffer pointers. This is done as follows for the FIR XDAIS algorithm:

```
Char *firChanBufs[] = { firChanBufId00, firChanBufId01, firChanBufId02Scr };
```

In this case, firChanBufs is then passed as a parameter to ALGMIN_new. The function loops through the buffers assigning XDAIS memTab[].base entries to each. The resulting memory descriptor table is passed to the algorithm’s algInit function thus enabling it to use these data buffers for processing. The cycle is repeated for each new algorithm or channel.

ALGMIN is applicable to all 'C5000 and 'C6000 designs. It can also be used independent of Reference Frameworks. Syntax and descriptions for all the functions are provided in Section 4, *ALGMIN Module*, page 12.

## The ALGRF Module

The ALGRF module exists to create and delete XDAIS algorithms by using the DSP/BIOS MEM memory manager. ALGRF fits the needs of RF3 and RF5. It is likely to be use in other planned RF levels. Any XDAIS-compliant algorithm can be used with ALGRF.

In comparison to the ALG implementation provided with CCStudio, the ALGRF implementation has the following advantages:

- **Smaller footprint.** As a generic module ALG supports both malloc / free Run-Time Support Library and DSP/BIOS MEM_alloc / MEM_free dynamic memory allocation. ALGRF supports only DSP/BIOS allocation, which saves code-space for the designer. Additionally, ALGRF ensures that no "dead code" exists; only functions that are called are linked in to the executable.

- **Scratch Memory Support.** The following API has been introduced in ALGRF:

```
ALGRF_Handle ALGRF_createScratchSupport(IALG_Fxns *fxns, IALG_Handle parent,
    IALG_Params *params, Void *scratchBuf, Uns scratchSize)
```

  This function allocates memory requested by algorithms, except in the case where IALG_SCRATCH, internal data buffers are requested. Instead, the scratchBuf and scratchSize parameters indicate that a buffer already exists in the application, which can be reused by the current algorithm. Such controlled sharing saves precious internal data memory.

  The CHAN module used in RF5 makes use of this API.

  RF3 does not use this API by default. It merely calls ALGRF_create() without sharing scratch data memory. This design choice was made since many unknown application factors exist which may influence the scratch re-use policy. For example, applications using a single priority have few scratch considerations compared to applications with many priority levels.

- **Abstraction from DSP/BIOS heap labels.** ALGRF uses the DSP/BIOS MEM module dynamic memory allocation. A heap identifier label or memory segment name can be passed to MEM_alloc() indicating which heap to allocate from. Instead of hard-coding these labels, they are passed in via:

```
/*  Configure the ALGRF module to use:
 *  1st argument - memory for internal heap
 *  2nd argument - memory for external heap
 */
 ALGRF_setup( INTERNALHEAP, EXTERNALHEAP );
```

The ALG module will remain in CCStudio to support legacy content and enable non-DSP/BIOS, or "generic" applications. ALGRF sits side-by-side as an alternative XDAIS instantiation module.

ALGRF is applicable to all 'C5000 and 'C6000 designs. It can also be used independent of Reference Frameworks. Syntax and descriptions for all the functions are provided in Section 5, *ALGRF Module*, page 15.

# 3 About the UTL Module

The UTL module contains a set of debugging macros, which you can elect to expand to actual code or to nothing by setting certain flags at compile time.

You define the desired amount of debugging features by setting the UTL_DBGLEVEL flag to a desired level in your application's Build Options. A debugging level includes the class for its level and all the classes from previous levels. You can also individually include or exclude classes regardless of the specified level. Figure 3 shows the debug levels and the features they enable:

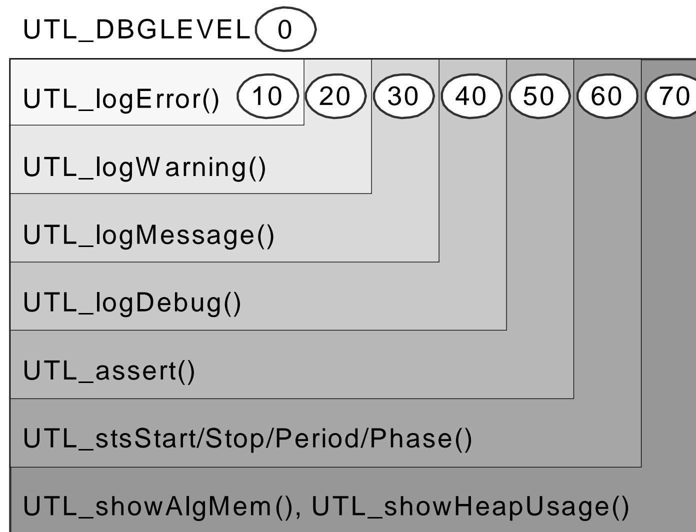


**Figure 3. UTL Debugging Levels**

For example, you may want some diagnostic messages printed to a LOG object while you develop your application, but not in the deployment phase. Rather than using LOG_printf() and physically removing it from the source code when building a release version (and then possibly having to put it back to debug subsequent releases), you can use the following UTL_logDebug() macro in your code:

```
UTL_logDebug( "Entered local procedure 'encrypt()'" );
```

If you set the variable UTL_DBGLEVEL to 40 or higher, the previous code expands to:

```
LOG_printf( UTL_logDebugHandle, "Entered local procedure 'encrypt()'" );
```

The UTL_logDebugHandle variable is a LOG_Handle that points to your default LOG object, which is initialized at startup via UTL_setLogs(). If the UTL_DBGLEVEL macro variable is set to 39 or less, the previous UTL_logDebug() macro expands to nothing.

With conditional expansion of macros to code you can reduce code size and remove unnecessary functionality in the deployment phase without having to remove development debugging/diagnostics aids.

Figure 4 shows some example calls to UTL functions and some sample UTL output messages.

■ Display memory usage

```
UTL_showAlgMem(thrAudioproc[i].algFIR);
UTL_showHeapUsage( INTERNALHEAP );
```

■ Avoid success/error codes at initialization

```
thrAudioproc[i].algFIR =
  FIR_create( &FIR_IFIR, &firParams );
UTL_assert(thrAudioproc[i].algFIR != NULL);
```

■ Check “must be true” conditions at runtime

```
UTL_assert( bufPtr != NULL );
```

■ Measure intervals between function calls

```
Void thrProcessRun() {
  /* process new frame of data ... */
  UTL_stsPeriod( stsProcess );
}
```

■ Display diagnostic messages

```
UTL_logDebug( "Application started." );
```

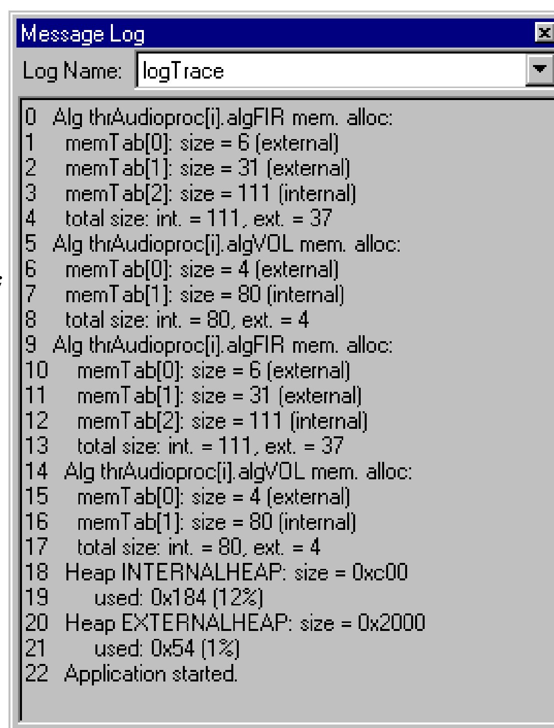


**Figure 4. UTL Message Example**

The UTL module implements the following classes of debugging features:

- **Message severity macros.** Errors, warnings, diagnostic messages, and debugging messages can be sent through macros that expand to LOG_printf()-style calls. The macro names are UTL_logError(), UTL_logWarning(), UTL_logMessage(), and UTL_logDebug().

- **Assertion macro.** Stops execution for a failed assertion and brings the offending source code line to focus in the debugging window. The macro syntax is UTL_assert(condition). This function is particularly useful in a system where all initialization operations (such as algorithm instantiation) *must* succeed. Of course, some may fail as you debug applications. So, rather than returning various success/failure codes from initialization functions, we halt the target if a crucial operation fails.

- **Time measurement macros.** These macros can report execution time, time between two periodic executions of a point in the program, and phase between two periodic points. The statistics are shown in STS objects in CCStudio. The macros are UTL_stsStart() and UTL_stsStop() for execution times, UTL_stsPeriod() for measuring time between periodic executions of a point in the program, and UTL_stsPhase() for measuring phase between two periodic points.

- **XDAIS algorithm diagnostics macros.** These macros report XDAIS algorithm memory requirements and heap usage. The macro names are UTL_showAlgMem() and UTL_showHeapUsage().

Details on the UTL macros are provided in Section 11, *UTL Module*, page 49.

# 4 ALGMIN Module

**Name** ALGMIN – XDAIS-Algorithm Instance Manager

**Synopsis** `#include <algmin.h>`

**Functions**

```
ALGMIN_activate()          /* Init. scratch buffers before processing */
ALGMIN_deactivate()        /* Save instance's persistent state */
ALGMIN_exit()              /* Shut down ALGMIN */
ALGMIN_init()              /* Initialize the ALGMIN module */
ALGMIN_new()               /* Instantiate a static XDAIS algorithm */
```

**Description** The ALGMIN module provides a set of XDAIS algorithm instantiation functions for use in compact, static applications. ALGMIN can be used on both 'C5000 and 'C6000 platforms. For background on and a comparison of IALG implementations, including ALGMIN, see Section 2, *About IALG Interface Implementations*, page 6.

Figure 5 summarizes the sequence in which ALGMIN module functions may be called.

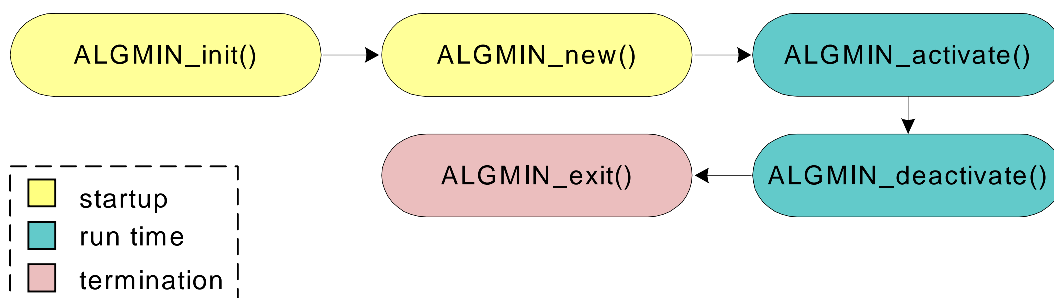


**Figure 5. ALGMIN Function Calling Sequence**

## ALGMIN_activate

Initialize scratch buffers before processing

**Syntax** `Void ALGMIN_activate(alg);`

**Parameters** `IALG_Handle alg` /* Algorithm Instance handle */

**Return Value** `Void`

**Description** ALGMIN_activate() initializes any scratch buffers and shared persistent memory using the persistent memory that is part of the algorithm's instance object.

The first (and only) argument to ALGMIN_activate() is an algorithm instance handle. The algorithm uses this handle to identify the various buffers, which must be initialized prior to any processing methods being called.

**Constraints**

- Should only be called after ALGMIN_init() and ALGMIN_new() have been called.

## ALGMIN_deactivate

Save instance's persistent state

**Syntax** `Void ALGMIN_deactivate(alg);`

**Parameters** `IALG_Handle alg` /* Algorithm Instance handle */

**Return Value** `Void`

**Description** ALGMIN_deactivate() saves an instance's persistent state information back from scratch buffers to persistent memory.

The first (and only) argument to ALGMIN_deactivate() is an algorithm instance handle. The algorithm uses this handle to identify the various buffers, which must be saved prior to the next ALGMIN_activate() and processing calls.

**Constraints**

- Should only be called after ALGMIN_init() and ALGMIN_new() have been called.
- Should only be called after ALGMIN_activate() and processing methods have been called one or more times.

## ALGMIN_exit

Shut down the ALGMIN module

**Syntax** `Void ALGMIN_exit(Void);`

**Parameters** `Void`

**Return Value** `Void`

**Description** ALGMIN_exit() finalizes use of the ALGMIN module. It is included for module completeness, and does not perform any actions.

**Constraints**

- Should only be called if the ALGMIN module is no longer needed.

## ALGMIN_init

Initialize the ALGMIN module

**Syntax** `Void ALGMIN_init(Void);`

**Parameters** `Void`

**Return Value** `Void`

**Description** ALGMIN_init() initializes use of the ALGMIN module. It is included for module completeness, and does not perform any actions.

## ALGMIN_new

Statically instantiate an XDAIS algorithm

**Syntax**

```
IALG_Handle ALGMIN_new( IALG_Fxns *fxns, IALG_Params *params,
                        Char *algChanBufs[], SmUns numAlgChanBufs);
```

**Parameters**

| | |
|---|---|
| `IALG_Fxns *fxns` | /* pointer to algorithm functions */ |
| `IALG_Params *params` | /* pointer to parameters for algorithm */ |
| `Char *algChanBufs[]` | /* pointer to buffer array for channel */ |
| `SmUns numAlgChanBufs` | /* number of buffers used per channel */ |

**Return Value**

| | |
|---|---|
| `IALG_Handle alg` | /* Algorithm Instance handle */ |

**Description** ALGMIN_new() provides a very low overhead method of bringing up an algorithm. No memory heaps are used, and no buffers are allocated. This function initializes an algorithm to use buffers whose alignment and size have been preconfigured to match algorithm needs.

ALGMIN_new runs the algorithm's algInit function. If this function is successful, ALGMIN_new() returns an algorithm instance handle.

**Constraints**

- Should only be called after ALGMIN_init() has been called.

**Example**

```
firHandle0 = (FIR_Handle)ALGMIN_new( (IALG_Fxns *)fxns,
                                     (IALG_Params *)params,
                                     firChanBufs,
                                     firNumChanBufs) );
```

# 5 ALGRF Module

**Name** ALGRF – XDAIS-Algorithm Instance Manager

**Synopsis** `#include <algrf.h>`

**Types and Constants**

```
typedef IALG_Handle ALGRF_Handle;
```

**Functions**

```
ALGRF_activate()                  /* Init. scratch buffers before processing */
ALGRF_control()                   /* Algorithm specific control and status */
ALGRF_create()                    /* Create an algorithm instance object */
ALGRF_createScratchSupport()      /* Create an instance, supporting scratch */
ALGRF_deactivate()                /* Save instance's persistent state */
ALGRF_delete()                    /* delete an algorithm instance object */
ALGRF_deleteScratchSupport()      /* Delete an instance, supporting scratch */
ALGRF_exit()                      /* Shut down ALGRF */
ALGRF_init()                      /* Initialize the ALGRF module */
ALGRF_setup()                     /* Configure system to use selected heaps */
```

**Description** The ALGRF module provides an interface to create, delete and invoke XDAIS algorithms on data. It institutes memory management policies suited to medium- to high-level Reference Frameworks. ALGRF can be used on both 'C5000 and 'C6000 platforms. For a comparison of IALG implementations, see Section 2, *About IALG Interface Implementations*, page 6.

Figure 6 summarizes the sequence in which ALGRF module functions may be called.

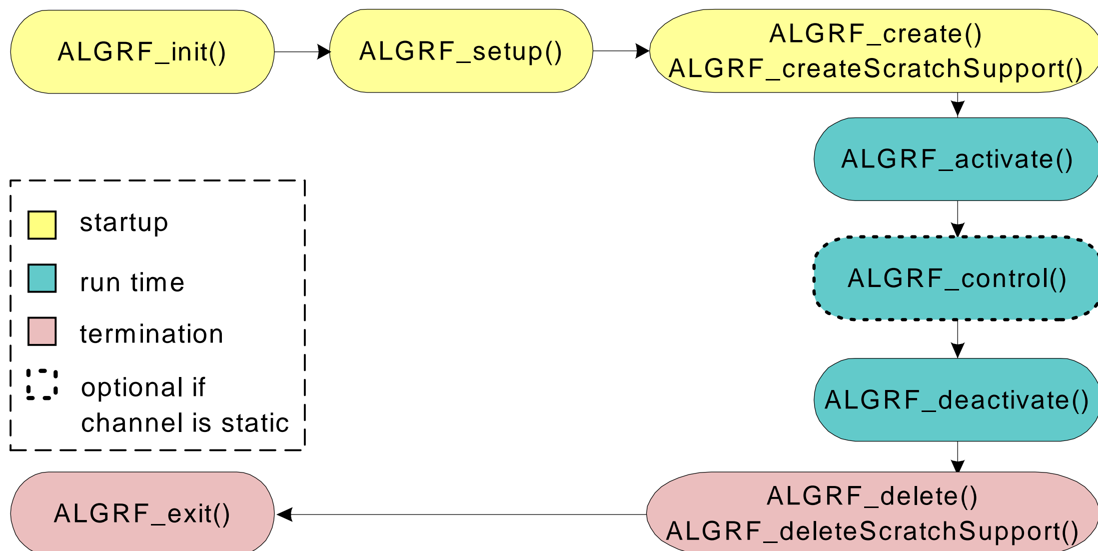


**Figure 6. ALGRF Function Calling Sequence**

## ALGRF_activate

Initialize scratch buffers before processing

**Syntax** `Void ALGRF_activate(alg);`

**Parameters** `ALGRF_Handle alg;` /* Algorithm Instance handle */

**Return Value** `Void`

**Description** ALGRF_activate() initializes any scratch buffers and shared persistent memory using the persistent memory that is part of the algorithm's instance object.

The first (and only) argument to ALGRF_activate() is an algorithm instance handle. The algorithm uses this handle to identify the various buffers, which must be initialized prior to any processing methods being called.

**Constraints**

- Should only be called after ALGRF setup and create functions have been called.

## ALGRF_control

Send control command to algorithm

**Syntax** `Int ALGRF_control(alg, cmd, *statusPtr);`

**Parameters**

| | |
|---|---|
| `ALGRF_Handle alg;` | /* Algorithm Instance handle */ |
| `IALG_Cmd cmd;` | /* Algorithm-specific command */ |
| `IALG_Status statusPtr;` | /* Algorithm specific in/out buffer */ |

**Return Value** `Int` /* Return status (IALG_EOK) */

**Description** ALGRF_control() sends an algorithm-specific command, cmd, and a pointer to an input/output status buffer pointer to an algorithm's instance object.

The first argument to ALGRF_control() is an algorithm instance handle. The second two parameters are interpreted in an algorithm-specific manner by the implementation.

The return value of ALGRF_control() indicates whether the control operation completed successfully. A return value of IALG_EOK indicates the operation completed successfully; all other return values indicate failure.

**Constraints**

- Should only be called after ALGRF setup and create functions have been called.

## ALGRF_create

Create an algorithm instance object

**Syntax** `ALGRF_Handle ALGRF_create(fxns, parent, params);`

**Parameters**

| | |
|---|---|
| `IALG_Fxns *fxns;` | /* pointer to algorithm functions */ |
| `IALG_Handle parent;` | /* parent instance object */ |
| `IALG_Params *params;` | /* pointer to algorithm parameters */ |

**Return Value** `ALGRF_Handle` /* Algorithm Instance handle */

**Description** ALGRF_create() instantiates any XDAIS algorithm. Independent of the type of memory requested, physical memory is always allocated. That is, algorithms requesting scratch buffers receive a unique buffer nonetheless.

**Constraints**

- Should only be called after the ALGRF setup function has been called.
- If the return value of ALGRF_create() is NULL then the algorithm cannot be instantiated.
- A stack size of at least ( ALGRF_MAXMEMRECS * sizeof(IALG_MemRec) ) must be reserved. By default the maximum number of memory records is set to 16. On 'C54x for example, the user must therefore allocate a stack size greater than 16 * 5 = 80 words. If an algorithm requests more than 16 descriptors, NULL is returned. In this case the user is free to modify ALGRF_MAXMEMRECS in algrf.h, and rebuild the library to handle these extreme cases. Note that memory descriptor table allocation was not done dynamically in order to reduce the risk of memory fragmentation.

## ALGRF_createScratchSupport

Create an instance, supporting scratch

**Syntax**

```
ALGRF_Handle ALGRF_createScratchSupport(fxns, parent, params,
scratchBuf, scratchSize);
```

**Parameters**

| | |
|---|---|
| `IALG_Fxns *fxns;` | /* pointer to algorithm functions */ |
| `IALG_Handle parent;` | /* parent instance object */ |
| `IALG_Params *params;` | /* pointer to algorithm parameters */ |
| `Void *scratchBuf` | /* pre-allocated scratch buffer for sharing */ |
| `Uns scratchSize` | /* size of shared scratch buffer */ |

**Return Value** `ALGRF_Handle` /* Algorithm Instance handle */

**Description** ALGRF_createScratchSupport() instantiates any XDAIS algorithm.

Physical memory is allocated from the heap(s) in the case where:

- IALG_PERSIST memory attribute is requested
- IALG_SCRATCH, external memory is requested

No physical memory is allocated when the algorithm requests:

- IALG_SCRATCH, internal data memory

In the latter case, the parameters scratchBuf and scratchSize inform the algorithm of the base address and size of the buffer to be reused.

**Constraints**

- Should only be called after the ALGRF setup function has been called.
- If the return value of ALGRF_createScratchSupport() is NULL then the algorithm cannot be instantiated.
- scratchBuf must be non-NULL if the algorithm requests internal scratch memory.
- It is the application's responsibility to ensure the size of the scratch buffer passed in is greater than or equal to the worst-case algorithm scratch request. In fact, an algorithm may request more than 1 scratch buffer. If such an algorithm is the worst-case scratch, the scratchSize must account for the sum of its scratch sizes plus their alignments. If any algorithm's total scratch requests exceed scratchSize, NULL is returned.
- A stack size of at least ( ALGRF_MAXMEMRECS * sizeof(IALG_MemRec) ) must be reserved. By default the maximum number of memory records is set to 16. On 'C54x for example, the user must therefore allocate a stack size greater than 16 * 5 = 80 words. If an algorithm requests more than 16 descriptors, NULL is returned. In this case the user is free to modify ALGRF_MAXMEMRECS in algrf.h, and rebuild the library to handle these extreme cases. Note that memory descriptor table allocation was not done dynamically in order to reduce the risk of memory fragmentation.

## ALGRF_deactivate

Save instance's persistent state

**Syntax** `Void ALGRF_deactivate(alg);`

**Parameters** `ALGRF_Handle alg;` /* Algorithm Instance handle */

**Return Value** `Void`

**Description** ALGRF_deactivate() saves an instance's persistent state information back from scratch buffers to persistent memory.

The first (and only) argument to ALGRF_deactivate() is an algorithm instance handle. The algorithm uses this handle to identify the various buffers, which must be saved prior to the next ALGRF_activate() and processing calls.

**Constraints**

- Should only be called after the ALGRF_setup() and ALGRF_create() functions have been called.
- Should only be called after the ALGRF_activate() and processing methods have been called one or more times.

## ALGRF_delete

Delete an algorithm instance object

**Syntax** `Bool ALGRF_delete(alg);`

**Parameters** `ALGRF_Handle alg;` /* Algorithm Instance handle */

**Return Value** `Bool` /* Success or Failure. TRUE (1) = success, else failure. */

**Description** ALGRF_delete() deletes the dynamically created instance object and all of its previously requested data buffers.

**Constraints**

- Should only be called after the ALGRF_setup() and ALGRF_create() functions have been called.
- If alg is NULL then ALGRF_delete() simply returns.
- A stack size of at least ( ALGRF_MAXMEMRECS * sizeof(IALG_MemRec) ) must be reserved. By default the maximum number of memory records is set to 16. On 'C54x for example, the user must therefore allocate a stack size greater than 16 * 5 = 80 words. If an algorithm requests more than 16 descriptors, failure (FALSE) is returned. In this case the user is free to modify ALGRF_MAXMEMRECS in algrf.h, and rebuild the library to handle these extreme cases. Note that memory descriptor table allocation was not done dynamically in order to reduce the risk of memory fragmentation.

## ALGRF_deleteScratchSupport

Delete an instance, supporting scratch

**Syntax** `Bool ALGRF_deleteScratchSupport(alg);`

**Parameters** `ALGRF_Handle alg;` /* Algorithm Instance handle */

**Return Value** `Bool` /* Success or Failure. TRUE (1) = success, else failure. */

**Description** ALGRF_deleteScratchSupport() deletes the dynamically created instance object. It also deletes all of its previously requested data buffers, except for those with both IALG_SCRATCH and internal attributes. This function is used as a counterpart to the ALGRF_createScratchSupport() function.

**Constraints**

- Should only be called after ALGRF setup and create functions have been called.
- If alg is NULL then ALGRF_deleteScratchSupport() simply returns.
- It is the responsibility of the client to delete any ‘left over’ scratch buffers after all algorithms using that buffer have been deleted.
- A stack size of at least ( ALGRF_MAXMEMRECS * sizeof(IALG_MemRec) ) must be reserved. By default the maximum number of memory records is set to 16. On 'C54x for example, the user must therefore allocate a stack size greater than 16 * 5 = 80 words. If an

algorithm requests more than 16 descriptors, failure (FALSE) is returned. In this case the user is free to modify ALGRF_MAXMEMRECS in algrf.h, and rebuild the library to handle these extreme cases. Note that memory descriptor table allocation was not done dynamically in order to reduce the risk of memory fragmentation.

## ALGRF_exit

Shut down the ALGRF module

**Syntax** `Void ALGRF_exit(Void);`

**Parameters** `Void`

**Return Value** `Void`

**Description** ALGRF_exit() finalizes use of the ALGRF module. It is included for module completeness, and does not perform any actions.

**Constraints**

- Should only be called if the ALGRF module is no longer needed.

## ALGRF_init

Initialize the ALGRF module

**Syntax** `Void ALGRF_init(Void);`

**Parameters** `Void`

**Return Value** `Void`

**Description** ALGRF_init() initializes use of the ALGRF module. It is included for module completeness, and does not perform any actions.

## ALGRF_setup

Configure ALGRF to use selected heaps

**Syntax** `Void ALGRF_setup(internalHeap, externalHeap);`

**Parameters**

| | |
|---|---|
| `Int internalHeap;` | /* Internal heap label */ |
| `Int externalHeap;` | /* External heap label */ |

**Return Value** `Void`

**Description** ALGRF_setup() takes internal and external heap arguments to match DSP/BIOS MEM module heap identifier labels with algorithm IALG_MemSpace requests. This keeps the ALGRF library independent of the user’s heap labeling.

**Constraints**

- Must be the first ALGRF function called after ALGRF_init().
- Valid DSP/BIOS MEM heap labels must be passed in to this function. Memory section names themselves are also valid labels.

# 6 CHAN Module

**Name** CHAN – Manage execution of algorithms in cells

**Synopsis** `#include <chan.h>`

### Types and Constants

```
typedef enum CHAN_State {           /* channel states */
    CHAN_ACTIVE,
    CHAN_INACTIVE
} CHAN_State;

typedef struct CHAN_Obj *CHAN_Handle;

typedef struct CHAN_Obj {
    ICELL_Obj  *cellSet;            /* set of cells in the channel */
    Uns         cellCnt;            /* number of cells in cellSet. Must be >= 1 */
    CHAN_State  state;              /* state of the channel */
    Bool        (*chanControlCB)(CHAN_Handle chanHandle);
                                    /* control callback function */
} CHAN_Obj;

typedef struct CHAN_Attrs {         /* attributes of a channel for creation */
    CHAN_State  state;              /* state of the channel, default is CHAN_ACTIVE */
    Bool       (*chanControlCB)(CHAN_Handle chanHandle);
                                    /* control function, default is NULL */
} CHAN_Attrs;

CHAN_Attrs CHAN_ATTRS = {           /* Default channel attributes */
    CHAN_ACTIVE,                    /* state */
    NULL                            /* chanControlCB */
};
```

In addition, the CHAN module makes use of the structures defined to match the ICELL interface. See Section 8, *ICELL Interface*, page 35 for details on those structures.

### Functions and Macros

```
CHAN_close()                        /* Closes all cells in a channel. */
CHAN_create()                       /* Creates a channel. */
CHAN_delete()                       /* Deletes a channel. */
CHAN_execute()                      /* Executes algorithms for all cells in channel. */
CHAN_exit()                         /* Exits from the CHAN module. */
CHAN_getAttrs()                     /* Gets the attribute structure for a channel. */
CHAN_init()                         /* Initializes the CHAN module. */
CHAN_open()                         /* Creates algorithm instances for channel cells. */
CHAN_regCell()                      /* Registers a cell within a channel. */
CHAN_setAttrs()                     /* Sets the attributes of a channel. */
CHAN_setup()                        /* Sets up the CHAN module. */
CHAN_unregCell()                    /* Releases memory allocated for ICC arrays. */
```

**Description** The CHAN module manages the serial execution of XDAIS algorithms contained in cells. This module manages one or more channel objects (CHAN_Obj), which are a collection of cell objects (see Section 8, ICELL Interface, page 35).

Figure 7 summarizes the sequence in which CHAN module functions may be called. CHAN functions not shown here must be called after CHAN_setup() and before CHAN_exit() as appropriate.

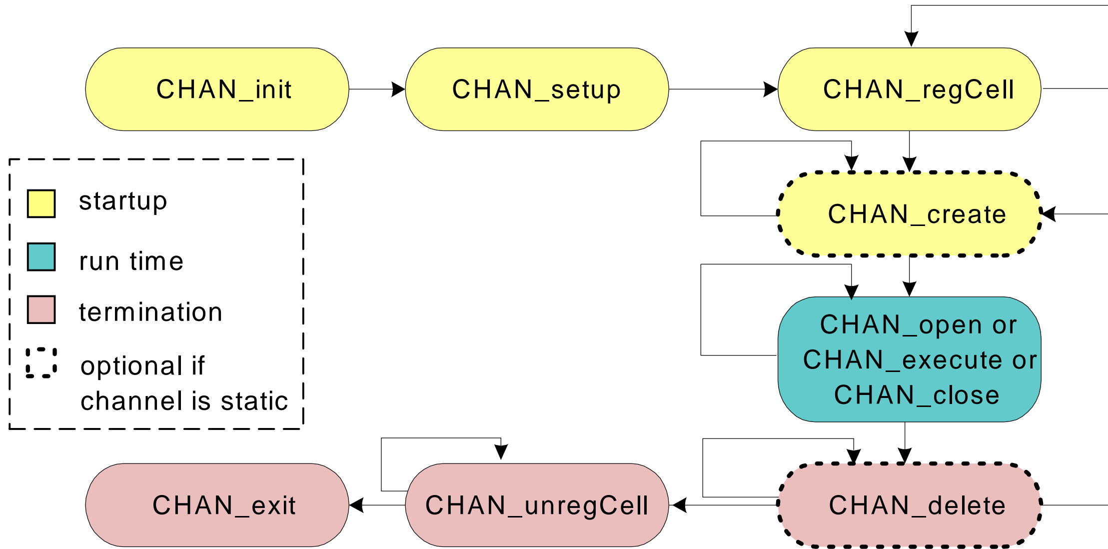


**Figure 7. CHAN Function Calling Sequence**

| **CHAN_close** | Closes all cells in a channel |
|---|---|

**Syntax** `status = CHAN_close(chanHandle);`

**Parameters** `CHAN_Handle chanHandle;` /* channel handle */

**Return Value** `Bool` /* Success or failure. TRUE (1) = success, else failure. */

**Description** CHAN_close() closes the cells in the specified channel and frees the algorithms instantiated by the channel in CHAN_open().

For each cell in the channel, this function calls the cell's cellClose() function, ALGRF_deleteScratchSupport(), and SSCR_deleteBuf(). If a failure occurs during any of these calls, the function returns FALSE immediately.

This function uses the UTL_assert() macro to confirm that the channel handle specified is not NULL.

**Constraints**

- Subsequent calls to CHAN functions may not be made for the channel specified in a call to CHAN_close() until the channel is re-opened via a call to CHAN_open().

## CHAN_create — Creates a channel

**Syntax** `chanHandle = CHAN_create( );`

**Parameters** `Void`

**Return Value** `CHAN_Handle` /* channel handle */

**Description** CHAN_create() uses MEM_alloc() to dynamically allocate a CHAN_Obj from the segment for DSP/BIOS objects. It returns a pointer to this object. Programs can also declare a CHAN_Obj statically, and need not call CHAN_create().

This function returns NULL if the allocation fails.

The attributes for a dynamically created channel are initially set to the default attributes.

**Constraints**

- May be called to dynamically create channel objects at run-time. Channel objects may also be declared statically.

## CHAN_delete — Deletes a channel

**Syntax** `status = CHAN_delete(chanHandle);`

**Parameters** `CHAN_Handle chanHandle;` /* channel handle */

**Return Value** `Bool` /* Success or failure. TRUE (1) = success, else failure. */

**Description** CHAN_delete() uses MEM_free() to dynamically free the CHAN_Obj pointed to by chanHandle. This function can only be called for dynamically-created channels.

This function returns the result of MEM_free().

This function uses the UTL_assert() macro to confirm that the channel handle specified is not NULL.

**Constraints**

- May be called only if CHAN_create() was called earlier for this channel.

## CHAN_execute — Executes algorithms for all cells in channel

**Syntax** `status = CHAN_execute(chanHandle, arg);`

**Parameters**

| | | |
|---|---|---|
| `CHAN_Handle` | `chanHandle;` | /* channel handle */ |
| `Arg` | `arg;` | /* user-supplied pointer */ |

**Return Value** `Bool` /* Success or failure. TRUE (1) = success, else failure. */

**Description** CHAN_execute() is used to serially execute the algorithms contained by all the cells in the channel.

If the specified channel is active, this function calls the chanControlCB() function for the channel if one is specified in the channel attributes. Then, it calls the cellExecute() function for all cells contained by the channel. The cellExecute() function typically activates the algorithm.

The arg parameter is passed as an argument to the cellExecute() function of each cell. You may choose to use this argument to control the cells in some manner.

If the channel is CHAN_INACTIVE, this function returns TRUE and performs no actions.

If the chanControlCB() function or the cellExecute() function returns FALSE, CHAN_execute() also returns FALSE.

This function uses the UTL_assert() macro to confirm that the channel handle specified is not NULL.

**Constraints**

- May be called only after calls to CHAN_init(), CHAN_setup(), CHAN_regCell(), and CHAN_open() have occurred.

**Example** This example is from thrProcess.c in RF5.

```
for( chanNum = 0; chanNum < NUMCHANNELS; chanNum++ ) {
   ...
   /* execute the channel */
   rc = CHAN_execute( chanHandle, NULL );
   UTL_assert( rc == TRUE );
   ...
}
```

## CHAN_exit

Exits from the CHAN module

**Syntax** `CHAN_exit( );`

**Parameters** `Void;`

**Return Value** `Void`

**Description** CHAN_exit() shuts down use of the CHAN module. Internally, it calls the ALGRF_exit(), and SSCR_exit() functions.

**Constraints**

- No CHAN module functions may be called after CHAN_exit() has been called.

## CHAN_getAttrs

Gets the attribute structure for a channel

**Syntax** `CHAN_getAttrs(chanHandle, *chanAttrs);`

**Parameters**

| | | |
|---|---|---|
| `CHAN_Handle` | `chanHandle;` | /* channel handle */ |
| `CHAN_Attrs` | `*chanAttrs;` | /* pointer to attribute structure */ |

**Return Value** `Void`

**Description** CHAN_getAttrs() returns the attributes for the channel specified. These attributes are the channel state and its control callback function. CHAN_Attrs has the following structure:

```
typedef struct CHAN_Attrs {        /* attributes of a channel for creation */
    CHAN_State  state;             /* state of the channel, default is CHAN_ACTIVE */
    Bool       (*chanControlCB)(CHAN_Handle chanHandle);
                                   /* control function, default is NULL */
} CHAN_Attrs;
```

This function uses the UTL_assert() macro to confirm that the chanHandle and chanAttrs specified are not NULL.

## CHAN_init

Initializes the CHAN module

**Syntax** `CHAN_init( );`

**Parameters** `Void;`

**Return Value** `Void`

**Description** CHAN_init() initializes the CHAN module. Internally, it calls the ALGRF_init() and SSCR_init() functions.

**Constraints**

- Must be called during the program initialization phase.

## CHAN_open — Creates algorithm instances for channel cells

**Syntax** `status = CHAN_open(chanHandle, cellSet[], cellCnt, *chanAttrs);`

**Parameters**

| | | |
|---|---|---|
| `CHAN_Handle` | `chanHandle;` | /* channel handle */ |
| `ICELL_Obj` | `cellSet[];` | /* list of cells in the channel */ |
| `Uns` | `cellCnt;` | /* number of cells in cellSet */ |
| `CHAN_Attrs` | `*chanAttrs;` | /* optional pointer to attribute structure */ |

**Return Value** `Bool` /* Success or failure. TRUE (1) = success, else failure. */

**Description** CHAN_open() creates algorithm instances for the cells in the channel.

This function sets values for elements in the CHAN_Obj for the specified channel using the parameters passed to this function. If the chanAttrs parameter is NULL, this function uses the default channel attributes.

Then, this function loops through all cells in the cellSet[] array. For each cell, this function calls the following:

- Calls SSCR_createBuf() to get the scratch buffer and size of the buffer.
- Calls ALGRF_createScratchSupport() to instantiate the algorithm using the scratch buffers returned by SSCR_createBuf().
- Calls UTL_showAlgMemName() for the algorithm.
- Calls the cell's cellOpen() function if one exists.

If any of these calls fail, CHAN_open() returns FALSE.

This function uses the UTL_assert() macro to confirm that the chanHandle and the cellSet[] array are not NULL.

**Constraints**

- May be called only after calls to CHAN_init(), CHAN_setup(), and CHAN_regCell() have occurred.
- Must be called before any calls to CHAN_execute().

**Example** This example is from thrProcess.c in RF5.

```
for (chanNum = 0; chanNum < NUMCHANNELS; chanNum++) {
   ...
   // open the channel: this causes the algorithms to be created
   rc = CHAN_open( &thrProcess.chanList[ chanNum ],
                   &thrProcess.cellList[ chanNum * NUMCELLS ],
                   NUMCELLS,
                   NULL );
   ...
}
```

## CHAN_regCell — Registers a cell

**Syntax**

```
status = CHAN_regCell( cellHandle, iccIn[], iccInCnt,
                       iccOut[], iccOutCnt);
```

**Parameters**

```
ICELL_Handle  cellHandle;       /* Cell to register */
ICC_Handle    iccIn[];          /* Input ICC objects */
Uns           iccInCnt;         /* Number of input ICC objects */
ICC_Handle    iccOut[];         /* Output ICC objects */
Uns           iccOutCnt;        /* Number of output ICC objects */
```

**Return Value**

```
Bool          /* Success or failure. TRUE (1) = success, else failure. */
```

**Description** CHAN_regCell() registers a cell.

This function first calls SSCR_prime() to determine the worst-case scratch buffer requirements for the algorithm specified in the cell. It then assigns “in” and “out” ICC objects to a cell. See Section 7, ICC Module, page 30 for information about the ICC module.

To have a cell that does not contain an algorithm, the algFxns field in the ICELL_Obj (that is, cellHandle->algFxns) must be NULL. When algFxns is NULL, CHAN_regCell() does not call SSCR_prime(), but still assigns the ICC objects to the cell.

This function is used when setting up cells.

The iccIn[] and iccOut[] arrays may contain zero or more handles to ICC objects. The number of objects in each array is specified by the iccInCnt and iccOutCnt parameters.

This function returns FALSE if SSCR_prime() returns FALSE.

This function uses the UTL_assert() macro to confirm that the cellHandle specified is not NULL.

**Constraints**

- Must be called after calls to ICC_linearCreate() or related calls that create ICC objects passed to this function.
- CHAN_regCell() must be called for each cell prior to any calls to CHAN_open() or CHAN_execute().
- If the cell does not use input ICC objects, the iccIn parameter must be set to NULL, and the iccInCnt must be set to 0. Similarly, iccOut must be NULL and iccOutCnt must be 0 if no output ICC objects are used.

**Example** This example is from thrProcess.c in RF5.

```
inputIcc  = (ICC_Handle)ICC_linearCreate( thrProcess.bufInput[ chanNum ],
                        FRAMELEN * sizeof( Sample ) );
outputIcc = (ICC_Handle)ICC_linearCreate( thrProcess.bufIntermediate,
                        FRAMELEN * sizeof( Sample ) );
rc = CHAN_regCell( cell, &inputIcc, 1, &outputIcc, 1 );
```

## CHAN_setAttrs — Sets the attributes of a channel

**Syntax** `CHAN_setAttrs(chanHandle, *chanAttrs);`

**Parameters**

| | | |
|---|---|---|
| `CHAN_Handle` | `chanHandle;` | /* channel handle */ |
| `CHAN_Attrs` | `*chanAttrs;` | /* pointer to attribute structure */ |

**Return Value** `Void`

**Description** CHAN_setAttrs() sets the attributes for the channel specified. These attributes are the channel state and its control callback function. CHAN_Attrs has the following structure:

```
typedef struct CHAN_Attrs {        /* attributes of a channel for creation */
    CHAN_State  state;             /* state of the channel, default is CHAN_ACTIVE */
    Bool        (*chanControlCB)(CHAN_Handle chanHandle);
                                   /* control function, default is NULL */
} CHAN_Attrs;
```

This function uses the UTL_assert() macro to confirm that the chanHandle and chanAttrs specified are not NULL.

**Constraints**

- May be called during run-time execution to change the state or control function for a channel.

## CHAN_setup — Sets up the CHAN module

**Syntax**

```
status = CHAN_setup( algrfInternalHeap, algrfExternalHeap,
                     sscrInternalHeap, bucketCnt, bucketBuf, bucketSize);
```

**Parameters**

| | | |
|---|---|---|
| `Int` | `algrfInternalHeap;` | /* Internal heap ID given to ALGRF_setup */ |
| `Int` | `algrfExternalHeap;` | /* External heap ID given to ALGRF_setup */ |
| `Int` | `sscrInternalHeap;` | /* Internal heap ID given to SSCR_setup */ |
| `Uns` | `bucketCnt;` | /* Number of buckets to create */ |
| `Ptr` | `bucketBuf[];` | /* Scratch buffers to use in system */ |
| `Uns` | `bucketSize[];` | /* Size of scratch buffers to use in system */ |

**Return Value** `Bool` /* Success or failure. TRUE (1) = success, else failure. */

**Description** CHAN_setup() sets up the CHAN module. This call allows you to specify which heaps should be used when the CHAN module allocates memory. It also allows you to specify how this channel should use the scratch buckets created by the SSCR module.

The algrfInternalHeap and algrfExternalHeap parameters are passed to ALGRF_setup(). the sscrInternalHeap parameter is passed to SSCR_setup(). The SSCR internal heap is used to allocate the shared internal scratch memory requested by the XDAIS algorithms.

Refer to the SSCR_setup() section for descriptions of the bucketCnt, bucketBuf, and bucketSize parameters.

This function returns that status returned by SSCR_setup().

**Constraints**

- Must be called after CHAN_init() during the program initialization and startup phase.

**Example** This example is from appThreads.c in RF5.

```
CHAN_setup( INTERNALHEAP, EXTERNALHEAP, INTERNALHEAP, SCRBUCKETS, NULL, NULL );
```

## CHAN_unregCell — Release memory allocated for ICC arrays

**Syntax** `CHAN_unregCell( cellHandle);`

**Parameters** `ICELL_Handle cellHandle;` /* Cell to unregister */

**Return Value** `Void`

**Description** CHAN_unregCell() frees the memory allocated by the corresponding call to CHAN_regCell(). It frees the memory for the inputIcc and outputIcc arrays pointed to by the ICELL_Obj structure for this cell.

**Constraints**

- May only be called for a cell that has been registered with CHAN_regCell().

# 7 ICC Module

**Name** ICC – Manage inter-cell communication assignment

**Synopsis** `#include <icc.h>`

### Types and Constants

```
#define ICC_USERSTART  32

typedef enum ICC_ObjType {          /* ICC types */
   ICC_NULLOBJ,
   ICC_LINEAROBJ,                   /* only type supported by API functions */
   ICC_USEROBJ=ICC_USERSTART,
   ICC_MAXTYPES
} ICC_ObjType;

typedef struct ICC_Obj *ICC_Handle;

typedef struct ICC_Obj  {
  Ptr            buffer;          // Pointer to the buffer
  Uns            nmaus;           // Size of the buffer
  ICC_ObjType    objType;         // Type of ICC
} ICC_Obj;

typedef struct {
   ICC_Obj  obj;
} ICC_LinearObj, *ICC_LinearHandle;
```

### Functions and Macros

```
ICC_exit()                        /* Exits the ICC module */
ICC_getBuf()                      /* Gets buffer and buffer size for ICC object */
ICC_init()                        /* Initializes ICC module */
ICC_linearCreate()                /* Creates a linear ICC object */
ICC_linearDelete()                /* Deletes specified linear ICC object */
ICC_setBuf()                      /* Sets buffer and buffer size for ICC object */
```

**Description** The ICC module manages inter-cell communication. When the cells of a channel execute their algorithm's function, ICCs are used instead of passing input/output buffers as parameters in the execution call. This module manages one or more ICC objects of type ICC_Obj.

The ICC_Obj structure includes a type specifier. Currently, the only type for which API functions are provided is ICC_LINEAROBJ, which is simply a linear buffer. These types may be extended to support more complex types of signal data or to handle debugging and validation of the buffer contents.

Each cell has an array of input and output ICC objects. These input and output ICC objects are set up by CHAN_regCell().

Using ICC objects allows a channel to have a flexible data flow. For example, consider the data flow shown in 0.

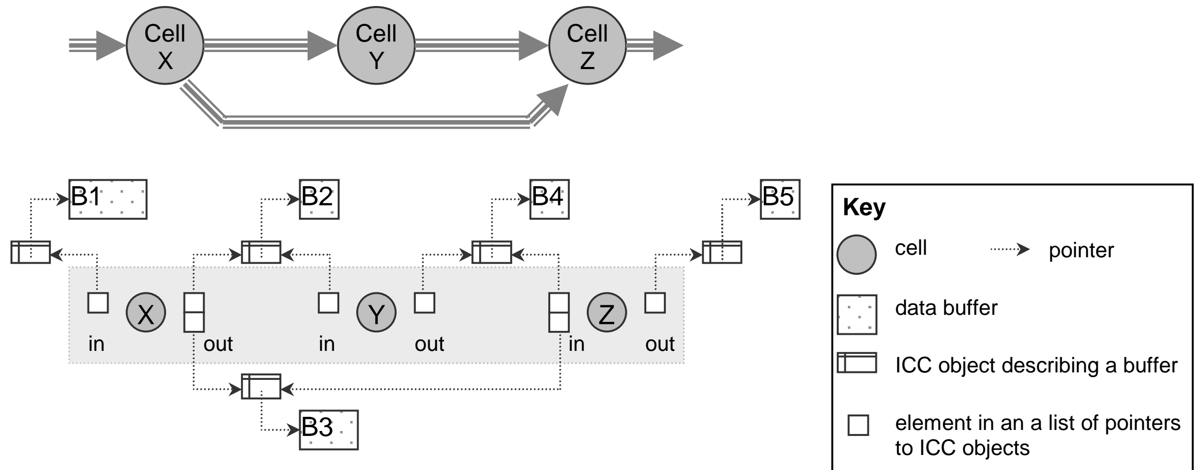


**Figure 8. ICC Data Flow Example**

There are three cells in this channel. The first cell has two outputs. By using five linear ICC objects (corresponding to the five arrows), this can easily managed. The pseudo-code to set up this data flow would be as follows:

```
inX[0]  = ICC_linearCreate( B1,  sizeof( B1 ) )
outX[0] = ICC_linearCreate( B2,  sizeof( B2 ) )
outX[1] = ICC_linearCreate( B3,  sizeof( B3 ) )

inY[0]  = outX[0]
outY[0] = ICC_linearCreate( B4,  sizeof( B4 ) )

inZ[0]  = outY[0]
inZ[1]  = outX[1]
outZ[0] = ICC_linearCreate( B5,  sizeof( B5 ) )

CHAN_regCell( cellX, inX, 1, outX, 2)
CHAN_regCell( cellY, inY, 1, outY, 1)
CHAN_regCell( cellZ, inZ, 2, outZ, 1)
```

Figure 9 summarizes the sequence in which ICC module functions may be called.

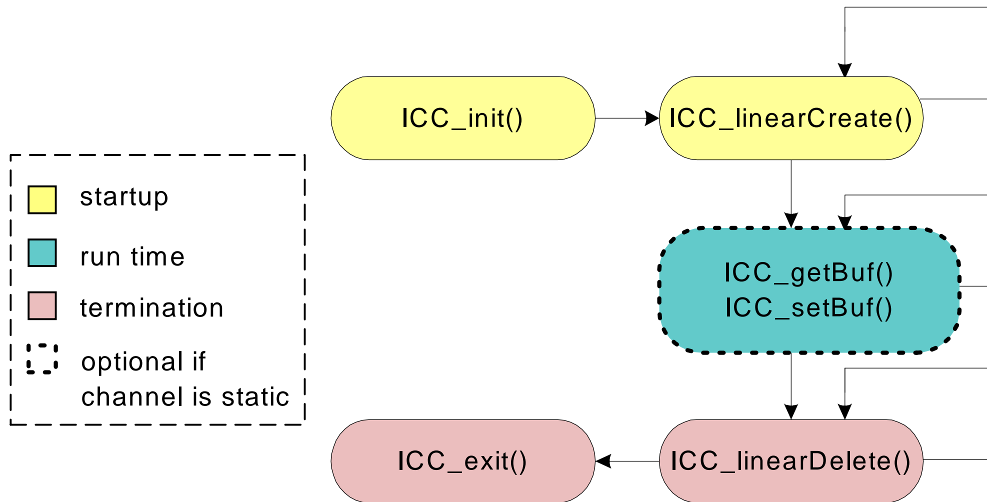


**Figure 9. ICC Function Calling Sequence**

## ICC_exit

Exits the ICC module

**Syntax** `Void ICC_exit( );`

**Parameters** `Void;`

**Return Value** `Void`

**Description** ICC_exit() shuts down use of the ICC module. Currently, it performs no actions.

**Constraints**

- No ICC module functions may be called after ICC_exit() has been called.

## ICC_getBuf

Gets buffer and buffer size for ICC object

**Syntax** `ICC_getBuf(iccHandle, &buffer, &nmaus);`

**Parameters**

| | | |
|---|---|---|
| `ICC_Handle` | `iccHandle;` | /* Pointer to ICC object */ |
| `Ptr` | `*buffer;` | /* Place to return buffer pointer */ |
| `Uns` | `*nmaus;` | /* Size of buffer */ |

**Return Value** `Void`

**Description** ICC_getBuf() is an inline macro that returns the buffer and size of the buffer maintained by the specified ICC object.

## ICC_init — Initializes ICC module

**Syntax** `Void ICC_init( );`

**Parameters** `Void;`

**Return Value** `Void`

**Description** ICC_init() initializes the ICC module. Currently, it performs no actions.

**Constraints**

- This function should be called before any other ICC module functions.

## ICC_linearCreate — Creates a linear ICC object

**Syntax** `addr = ICC_linearCreate (buffer, nmaus);`

**Parameters**

| | | |
|---|---|---|
| `Ptr` | `buffer;` | /* Buffer to be used in ICC object */ |
| `Uns` | `nmaus;` | /* Size of buffer */ |

**Return Value** `ICC_LinearHandle`

**Description** ICC_linearCreate() creates a linear ICC object from the DSP/BIOS segment. It initializes the fields of the ICC object using the buffer and nmaus parameters. This function does not allocate the buffer; the caller must supply the buffer.

This function returns NULL if the memory allocation for the ICC object fails. If successful, this function returns a handle to the new object.

**Constraints**

- This function (or a similar function if implemented) must be called before calls to CHAN_regCell().

**Example** This example is from thrProcess.c in RF5.

```
inputIcc = (ICC_Handle)ICC_linearCreate( thrProcess.bufInput[chanNum],
                  FRAMELEN * sizeof( Sample ) );
```

## ICC_linearDelete

Deletes specified linear ICC object

**Syntax**

```
status = ICC_linearDelete(linearIccHandle);
```

**Parameters**

```
LinearHandle  linearIccHandle;          /* Object to delete */
```

**Return Value** `Bool`

**Description** ICC_linearDelete() deletes the specified linear ICC object. It does not free the buffer associated to the object.

This function returns FALSE if the call to MEM_free() fails.

This function uses the UTL_assert() macro to confirm that the linearIccHandle specified is not NULL and that the specified object is of type ICC_LINEAROBJ.

**Constraints**

- This function can only be called for ICC objects created with ICC_linearCreate().

## ICC_setBuf

Sets buffer and buffer size for ICC object

**Syntax**

```
ICC_setBuf(iccHandle, buffer, nmaus);
```

**Parameters**

```
ICC_Handle    iccHandle;          /* Pointer to ICC object */
Ptr           buffer;             /* New buffer pointer */
Uns           nmaus;              /* Size of buffer */
```

**Return Value** `Void`

**Description** ICC_setBuf() is an inline macro that sets the buffer and size of the buffer maintained by the specified ICC object. This function may be used at runtime to change the size or address of the buffer.

# 8 ICELL Interface

**Name** ICELL – Cell container for algorithm interface

**Synopsis** `#include <icell.h>`

**Types and Constants**

```
typedef struct ICELL_Obj {
    Int          size;            /* Number of MAUs in the structure          */
    String       name;            /* User chosen name                         */
    ICELL_Fxns  *cellFxns;        /* Pointer to cell v-table function         */
    Ptr          cellEnv;         /* Pointer to user-defined cell env. struct */
    IALG_Fxns   *algFxns;         /* Pointer to alg v-table functions         */
    IALG_Params *algParams;       /* Pointer to alg parameters                */
    IALG_Handle  algHandle;       /* Handle of alg managed by cell            */
    Uns          scrBucketIndex;  /* Scratch bucket for XDAIS scratch memory  */
    ICC_Handle  *inputIcc;        /* Array of input ICC objects               */
    Uns          inputIccCnt;     /* # of ICC objects in the input array      */
    ICC_Handle  *outputIcc;       /* Array of output ICC objects              */
    Uns          outputIccCnt;    /* # of ICC objects in the output array     */
} ICELL_Obj;

typedef struct ICELL_Obj *ICELL_Handle;

typedef struct ICELL_Fxns {
   Bool (*cellClose   )(ICELL_Handle handle);
   Int  (*cellControl)(ICELL_Handle handle, IALG_Cmd cmd, IALG_Status *status);
   Bool (*cellExecute)(ICELL_Handle handle, Arg arg);
   Bool (*cellOpen    )(ICELL_Handle handle);
} ICELL_Fxns;
```

**Functions and Macros**

None

**Description** RF5 provides a cell interface called ICELL. The ICELL structures are defined by the interface. There are no CELL module function calls. See *Reference Framework 5: An Extensive Framework for Medium- to High-Complexity Systems* (SPRA795) for a description of how the ICELL interface is used.

The ICELL interface is similar to the IALG interface in the XDAIS specification. That is, the structures for the interface are specific in a header file. There must be an implementation of the structures for an algorithm. One major difference between ICELL and IALG is that algorithm providers are required to implement the IALG interface. In contrast, the application designer implements the ICELL interface for each algorithm used in an RF5 application.

Your application must create implementations of the ICELL_Obj and ICELL_Fxns structures for each algorithm.

The ICELL_Obj data structure contains all the pertinent information about an individual cell. Table 3 describes the fields in the ICELL_Obj structure.

**Table 3. ICELL_Obj Elements**

| Name | Type | Description |
|---|---|---|
| size | Int | The sizeof() of the ICELL_Obj structure. |
| name | String | A user-selected name for this cell. This string typically identifies the algorithm performed. In the default RF5 application, this name is sent to a log by calls to CHAN_open() via a UTL module function. |
| cellFxns | ICELL_Fxns | Reference to the ICELL_Fxns table defined for this cell. This structure is described in Table 4. |
| cellEnv | Ptr | This structure is user-defined. Each cell has its own cellEnv pointer, which can be used to maintain cell-specific information. Each cell may have a different structure definition. For example, if an algorithm has mutually-exclusive runtime functions, such as apply1 and apply2, the cellExecute function could determine which function to execute based on a field in the cellEnv structure. Another use of the cellEnv structure might be to store DMA handles used by the cell (this is not for the algorithm's DMA use). In the cellOpen function, the DMA channel could be allocated and stored in the cellEnv structure. Then the cellExecute function could use the DMA handle. |
| algFxns | IALG_Fxns * | Algorithm functions table provided by the algorithm. |
| algParams | IALG_Params * | Algorithm parameters structure provided by the algorithm. |
| algHandle | IALG_Handle | Handle of the algorithm instance. This element should initially be set to NULL. Its value is set within CHAN_open() by a call to ALGRF_createScratchSupport(). |
| scrBucketIndex | Uns | Scratch memory bucket to be used by the SSCR module to create shared scratch buffers for groups of cells. Generally, all cells in channels executed by tasks at the same priority level should have the same scrBucketIndex. |
| inputIcc | ICC_Handle | Array of input ICC objects. Its value is set by calls to CHAN_regCell(). |
| inputIccCnt | Uns | Number of input ICC objects in the array. |
| outputIcc | ICC_Handle | Array of output ICC objects. Its value is set by calls to CHAN_regCell(). |
| outputIccCnt | Uns | Number of output ICC objects in the array. |

The other data structure is the ICELL_Fxns. This structure contains the interface definition of the cell functions. As with other interfaces (e.g. IALG), it is up to the user to implement the ICELL_Fxns. It is not required that you implement the cellClose, cellControl, and cellOpen functions. The cellExecute function is required.

**Table 4. ICELL_Fxns Functions**

| Prototype | Description |
|---|---|
| `Bool (*cellClose)( ICELL_Handle cellHandle)` | Function to close cell resources. Not used to free the algorithm's resources. This function is optional. |
| `Int (*cellControl)(ICELL_Handle cellHandle, IALG_Cmd cmd, IALG_Status *status)` | Function to call the algorithm's control functions. This function is optional. |
| `Bool (*cellExecute)(ICELL_Handle cellHandle, Arg arg)` | Function to call the algorithm's runtime functions. This function is required.<br>The arg parameter is passed to this function by the call to CHAN_execute(). This user-defined structure may be different for each channel (but must be the same for all cells in a channel). This parameter allows you to pass channel-level information. For example, the result of one algorithm could affect the desired processing of a later algorithm in the channel. The arg structure could communicate this result. Another example would be to use the arg parameter to store a semaphore handle used to lock the scratch buffer in the "locking mechanism" method of SSCR. |
| `Bool (*cellOpen)( ICELL_Handle cellHandle)` | Function to open cell resources. Not used to create the algorithm's resources. This function is optional. |

For example, the cellFir.c file in RF5 creates the ICELL_Fxns table as follows:

```
ICELL_Fxns FIR_CELLFXNS = {   /* v-table for this cell */
    NULL,              /* cellClose */
    FIR_cellControl, /* cellControl */
    FIR_cellExecute, /* cellExecute */
    NULL               /* cellOpen */
};
```

# 9 SCOM Module

**Name** SCOM – Synchronized COMmunication module

**Synopsis** `#include <scom.h>`

### Types and Constants

```
typedef struct SCOM_Attrs {        /* SCOM object creation attributes */
       Char          dummy;        /* no attributes at present */
} SCOM_Attrs;

typedef struct SCOM_Obj SCOM_Obj, *SCOM_Handle;       /* SCOM object handle */

extern SCOM_Attrs SCOM_ATTRS;      /* default SCOM creation attributes */
```

### Functions and Macros

```
SCOM_create()                      /* Create an SCOM queue */
SCOM_delete()                      /* Delete an SCOM queue */
SCOM_exit()                        /* Exit from the SCOM module */
SCOM_getMsg()                      /* Get a message from an SCOM queue */
SCOM_init()                        /* Initialize the SCOM module  */
SCOM_open()                        /* Get reference to existing SCOM queue by name */
SCOM_putMsg()                      /* Put a message to an SCOM queue */
```

**Description** This module implements message passing among tasks. It lets the user create any number of named, synchronized queues, and put messages to such queues and receive messages from them. Messages are buffers of arbitrary sizes.

The SCOM module manages SCOM queue objects. Each SCOM queue internally uses a queue object (QUE) and a semaphore object (SEM). The structure of the SCOM queue is intended to be private within the SCOM module. The application should not reference elements of the SCOM queue object.

A queue can hold an arbitrary message, but the first field of the message must be a QUE_elem. For example, here is the SCOM message structure in appThreads.h:

```
typedef struct ScomBufChannels {
    QUE_Elem   elem;
    Sample     *bufChannel[ NUMCHANNELS ];
} ScomBufChannels;
```

Figure 10 summarizes the sequence in which SCOM module functions may be called for a single SCOM queue. (The SCOM_init() function is called only once for the entire module.) Once a queue has been created by a thread, the other thread using this queue for communication performs SCOM_open().

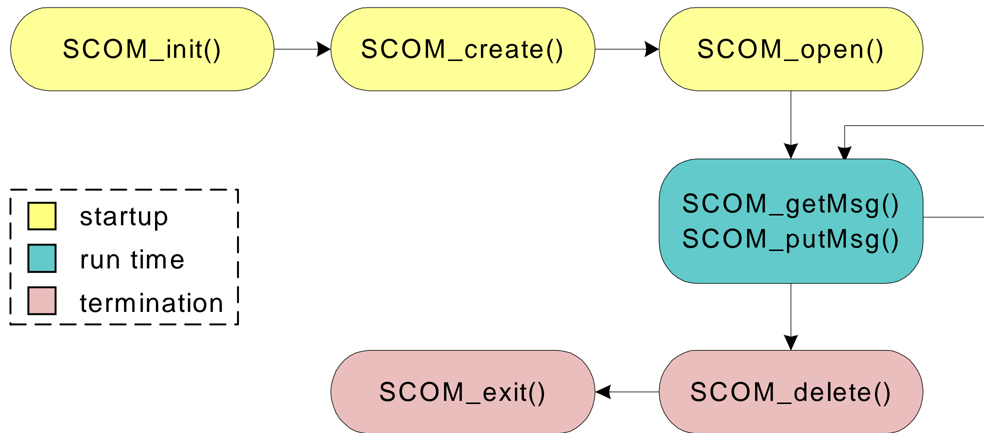


**Figure 10. SCOM Function Calling Sequence**

The number of messages that can be waiting in an SCOM queue is unlimited. See *Reference Framework 5: An Extensive Framework for Medium- to High-Complexity Systems* (SPRA795) for a description of how SCOM queues can be used.

## SCOM_create

Create a new SCOM queue under given name

**Syntax**

```
scomQueue = SCOM_create(queueName, *attrs);
```

**Parameters**

| | | |
|---|---|---|
| `String` | `queueName;` | /* Name of SCOM queue to create */ |
| `SCOM_Attrs` | `*attrs;` | /* SCOM object attributes; only NULL supported */ |

**Return Value** `SCOM_Handle` /* handle of new SCOM queue */

**Description** SCOM_create() creates an SCOM queue object with the specified name. It uses MEM_alloc() to allocate memory for the object. It also uses QUE_new() and SEM_new() to initialize the queue and semaphore objects used internally by the SCOM module.

This function returns a handle to the new SCOM object. If memory cannot be allocated successfully, this function returns NULL. If you do not want to make the handle returned by SCOM_create() global, you can later use SCOM_open() to get the handle for an SCOM object using the queueName.

This function uses the UTL_assert() macro to confirm that the queueName specified is not NULL.

**Constraints**

- This function must be called before any functions that attempt to use the SCOM object.

**Example** This example is from thrProcess.c in RF5.

```
/* create named SCOM queues for receiving messages from other tasks */
scomReceiveFromRx = SCOM_create( "scomToProcessFromRx", NULL );
scomReceiveFromTx = SCOM_create( "scomToProcessFromTx", NULL );
```

## SCOM_delete — Delete the SCOM queue

**Syntax** `status = SCOM_delete(scomQueue);`

**Parameters** `SCOM_Handle scomQueue;` /* handle of SCOM queue to be deleted */

**Return Value** `Bool` /* Success or failure. TRUE (1) = success, else failure */

**Description** SCOM_delete() frees the memory allocated for the SCOM_Obj by SCOM_create(). It also removes the element for this object from the master queue used internally by the SCOM module.

This function uses the UTL_assert() macro to confirm that the scomQueue specified is not NULL.

**Constraints**

- The specified queue may not be referenced by other function calls after it is deleted.

## SCOM_exit — Finalize use of the SCOM module

**Syntax** `SCOM_exit( );`

**Parameters** `Void;`

**Return Value** `Void`

**Description** SCOM_exit() shuts down use of the SCOM module. Currently, it performs no actions.

**Constraints**

- No SCOM module functions may be called after SCOM_exit() has been called.

## SCOM_getMsg — Receive a message from a synchronized queue

**Syntax** `msg = SCOM_getMsg(scomQueue, timeout);`

**Parameters**

| | | |
|---|---|---|
| `SCOM_Handle` | `scomQueue;` | /* SCOM queue to get message from */ |
| `Uns` | `timeout;` | /* semaphore timeout in system clock ticks*/ |

**Return Value** `Ptr` /* pointer to message buffer */

**Description** SCOM_getMsg() gets a message from the specified SCOM queue. If no message is available within the specified timeout, this function returns NULL. If timeout is zero, this function does not block.

This function uses the UTL_assert() macro to confirm that the scomQueue specified is not NULL.

**Constraints**

- This function gets a message only if one has been placed by SCOM_putMsg().

**Example** This example is from thrProcess.c in RF5.

```
// get a full buffer from Rx
scombufRx = SCOM_getMsg( scomReceiveFromRx, SYS_FOREVER );
```

## SCOM_init — Initialize use of the SCOM module

**Syntax** `SCOM_init( );`

**Parameters** `Void;`

**Return Value** `Void`

**Description** SCOM_init() initializes a master queue used internally by the SCOM module.

**Constraints**

- This function must be called before any other SCOM module functions.

## SCOM_open — Get a reference to an existing SCOM queue by name

**Syntax** `scomQueue = SCOM_open(queueName);`

**Parameters** `String` `queueName;` /* ID (name) of SCOM queue to be found */

**Return Value** `SCOM_Handle` /* handle of SCOM queue */

**Description** SCOM_open() returns a handle to the SCOM object specified by the queueName parameter. If the specified name is not found, this function returns NULL.

SCOM_open() returns a handle to an SCOM object that was created by an earlier call to SCOM_create(). The call to SCOM_open() is not necessary if you pass the handle returned by SCOM_create() to any functions that need to use it or make the handle global.

This function uses the UTL_assert() macro to confirm that the queueName specified is not NULL.

**Constraints**

- This function must be called after SCOM_create().

**Example** This example is from thrRxSplit.c in RF5.

```
// open the SCOM queues (your own for receiving and another for sending)
scomReceive = SCOM_open( "scomRxSplit"      );
scomSend    = SCOM_open( "scomToProcessFromRx" );
UTL_assert( scomReceive != NULL );
UTL_assert( scomSend    != NULL );
```

## SCOM_putMsg

Place the message on a synchronized queue

**Syntax** `SCOM_putMsg(scomQueue, msg);`

**Parameters**

| | |
|---|---|
| `SCOM_Handle scomQueue;` | /* SCOM queue to put message to */ |
| `Ptr msg;` | /* message buffer to be freed */ |

**Return Value** `Void`

**Description** SCOM_putMsg() puts a message into the specified SCOM queue.

This function uses the UTL_assert() macro to confirm that the scomQueue specified is not NULL.

**Constraints**

- This function must be called after the scomQueue has been created.

**Example** This example is from thrProcess.c in RF5.

```
// send the message describing full output buffers to Tx
SCOM_putMsg( scomSendToTx, scombufTx );

// send the message describing consumed input buffers to Rx
SCOM_putMsg( scomSendToRx, scombufRx );
```

# 10 SSCR Module

**Name** SSCR – Shared scratch memory module

**Synopsis** `#include <sscr.h>`

### Functions and Macros

```
SSCR_createBuf()          /* Create or return scratch buffer */
SSCR_deleteBuf()          /* Delete scratch buffer */
SSCR_exit()               /* Exit from SSCR module */
SSCR_getBuf()             /* Get size of and pointer to scratch buffer */
SSCR_init()               /* Initialize SSCR module */
SSCR_prime()              /* Determine worst-case scratch usage */
SSCR_setup()              /* Set up SSCR module */
```

**Description** The SSCR module manages overlaying of on-chip scratch memory requested by XDAIS algorithms. Scratch memory is memory the algorithm needs during run-time execution. An algorithm uses scratch memory without regard to its prior contents. Persistent memory is memory that can be safely written to, knowing that the contents will be unchanged between successive invocations by the application. This distinction enables optimization by overlaying scratch memory for several algorithms on the same physical memory as shown in Figure 11.

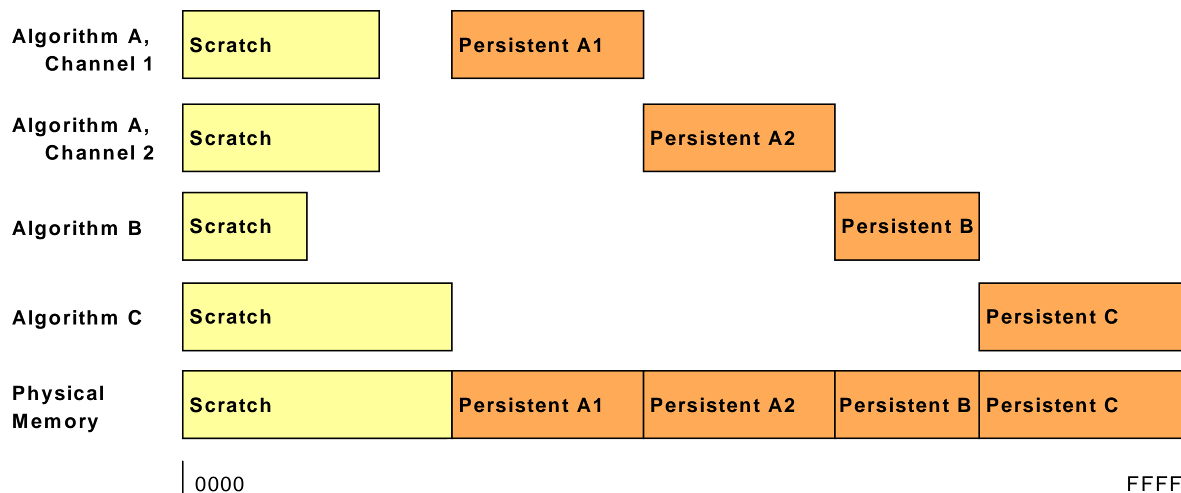


**Figure 11. Scratch vs. Persistent Memory Allocation**

Since an algorithm cannot block while executing a run-time execution function, scratch memory can be shared among other algorithms. However, since a higher-priority thread can interrupt a lower-priority thread, special protection must be provided for the shared memory.

The SSCR module allows you to provide protection in one of two ways:

- **Priority level.** A separate scratch buffer is used for each thread priority level that uses scratch buffers. Since DSP/BIOS is principally a priority-based (as opposed to time-sliced) pre-emptive real-time kernel, this method is safe.

- **Locking mechanisms.** One scratch buffer is shared by all algorithms. Locking mechanisms (such as semaphores or HWI_enable/HWI_disable calls) must be used to protect against preemption whenever the scratch buffer is accessed.

The priority level method provides lower latency but uses a larger total scratch buffer allocation. The application designer should decide whether to minimize latency or memory use.

The SSCR module manages one or more "bucket" objects. Each bucket maintains one scratch buffer. In the priority level method, each bucket corresponds to a priority level. In the locking mechanism method, there is only one bucket.

Figure 12 summarizes the sequence in which SSCR module functions may be called.

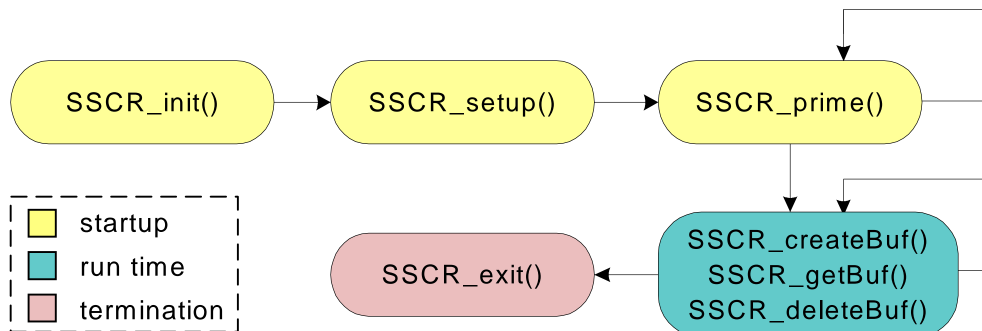


**Figure 12. SSCR Function Calling Sequence**

| **SSCR_createBuf** | Create the scratch buffer (or return it if already allocated) |
|---|---|

**Syntax** `addr = SSCR_createBuf(scrBucketIndex, *scrSize);`

**Parameters**

| | |
|---|---|
| `Uns scrBucketIndex;` | /* bucket index */ |
| `Uns *scrSize;` | /* size of scratch buffer returned */ |

**Return Value** `Ptr` /* address of scratch buffer returned */

**Description** This function creates and returns a pointer to the scratch buffer associated with the specified scrBucketIndex. The size of the buffer, which is determined by the analysis performed by SSCR_prime(), is also returned in scrSize.

If you are using the CHAN module, the CHAN_open() function calls SSCR_createBuf().

The first time SSCR_createBuf() is called for a particular scrBucketIndex, it allocates the buffer the heap specified by SSCR_setup(). Subsequent calls to SSCR_createBuf() for the same scrBucketIndex simply return the address of the buffer; that is, it is not allocated again.

If the call to SSCR_setup() specified the bucketBuf[] and bucketSize[] arrays, the SSCR_createBuf() function returns the buffer address and size specified for this scrBucketIndex.

To prevent corruption of the bucket list by two separate threads, this function disables HWI interrupts while accessing the bucket list.

This function returns NULL and scrSize = 0 if the scratch buffer size required is 0 or if the buffer allocation fails.

This function uses the UTL_assert() macro to confirm that the scrSize parameter is not NULL and the scrBucketIndex is less than the bucketCnt specified in the call to SSCR_setup().

**Constraints**

- Must be called after SSCR_prime() has been called for all algorithms in the application.

**Example** This example is from chan_open.c in RF5.

```
/* get scratch buffer and size of the buffer */
scratchBuf = SSCR_createBuf(cellHandle->scrBucketIndex, &scratchSize);

/* create algorithm using the scratch buffers returned from the SSCR module */
cellHandle->algHandle = ALGRF_createScratchSupport(cellHandle->algFxns,
                 NULL, cellHandle->algParams, scratchBuf, scratchSize);
```

## SSCR_deleteBuf

Delete the scratch buffer

**Syntax** `status = SSCR_deleteBuf(scrBucketIndex);`

**Parameters** `Uns scrBucketIndex;` /* bucket index */

**Return Value** `Bool` /* Success or failure. TRUE (1) = success, else failure. */

**Description** SSCR_deleteBuf() is used to delete a scratch buffer if it is no longer in use. The SSCR module keeps track of how many algorithms use a particular buffer. A call to SSCR_deleteBuf() subtracts one from this count. If the count reaches zero, the scratch buffer is freed.

If the buffer was specified in the call to SSCR_setup(), this function does not delete the buffer. No attempt is made to free the buffer if the buffer size is 0.

If you are using the CHAN module, the CHAN_close() function calls SSCR_deleteBuf().

To prevent corruption of the bucket list by two separate threads, this function disables HWI interrupts while accessing the bucket list.

This function returns FALSE if the buffer cannot be freed.

This function uses the UTL_assert() macro to confirm that the scrBucketIndex is less than the bucketCnt specified in the call to SSCR_setup().

**Example** This example is from chan_close.c in RF5.

```
/* release the scratch buffers */
if (SSCR_deleteBuf(cellHandle->scrBucketIndex) == FALSE) {
   return (FALSE);
}
```

## SSCR_exit

Exit the SSCR module

**Syntax** `SSCR_exit( );`

**Parameters** `Void;`

**Return Value** `Void`

**Description** SSCR_exit() frees the internal bucket list created by SSCR_setup(). It also resets internal global variables to their initial values.

If you are using the CHAN module, the CHAN_exit() function calls SSCR_exit().

**Constraints**

- No SSCR module functions may be called after SSCR_exit() has been called.

## SSCR_getBuf

Get the size of and pointer to the scratch buffer

**Syntax** `addr = SSCR_getBuf(scrBucketIndex, *scrSize);`

**Parameters**

```
Uns scrBucketIndex;      /* bucket index */
Uns *scrSize;            /* size of scratch buffer returned */
```

**Return Value**

```
Ptr                      /* address of scratch buffer returned */
```

**Description** SSCR_getBuf() returns the pointer to and size of the scratch buffer associated with the scrBucketIndex specified.

If there is no scratch buffer allocated for the scrBucketIndex, this function returns NULL and sets scrSize to 0. To prevent corruption of the bucket list by two separate threads, this function disables HWI interrupts while accessing the bucket list.

This function uses the UTL_assert() macro to confirm that the scrSize parameter is not NULL and the scrBucketIndex is less than the bucketCnt specified in the call to SSCR_setup().

**Constraints**

- Must be called after SSCR_createBuf().

## SSCR_init

Initialize the SSCR module

**Syntax** `SSCR_init( );`

**Parameters** `Void;`

**Return Value** `Void`

**Description** SSCR_init() initializes the SSCR module. Currently, it performs no actions.

If you are using the CHAN module, the CHAN_init() function calls SSCR_init().

**Constraints**

- This function should be called before any other SSCR module functions.

## SSCR_prime

Determine worst-case scratch usage for an algorithm instance

**Syntax** `status = SSCR_prime(scrBucketIndex, *fxns, *params);`

**Parameters**

| | | |
|---|---|---|
| `Uns` | `scrBucketIndex;` | /* scratch bucket index level */ |
| `IALG_Fxns` | `*fxns;` | /* pointer to algorithm functions */ |
| `IALG_Params` | `*params;` | /* pointer to parameters for algorithm */ |

**Return Value** `Bool` /* Success or failure. TRUE (1) = success, else failure. */

**Description** SSCR_prime() is used to determine the largest scratch buffer size required by an algorithm that uses the specified scrBucketIndex. If the specified algorithm has a larger scratch buffer requirement that algorithms previously specified in calls to SSCR_prime(), the size for the specified scratch bucket is increased to the larger size. The SSCR module keeps track of the worst-case requirement for each scrBucketIndex level.

This function does not allocate scratch buffers. Instead, SSCR_createBuf() allocates buffers with the sizes determined by calls to SSCR_prime().

A program should call this function for each algorithm it uses. If you are using the CHAN module, the CHAN_regCell() function calls SSCR_prime().

To prevent corruption of the bucket list by two separate threads, this function disables HWI interrupts while accessing the bucket list.

If params is NULL, the algorithm's default parameter structure is used.

If the call to SSCR_setup() specified buffers and sizes, SSCR_prime() still computes the worst-case requirements. If the computed requirement for this bucket is greater than the size specified in SSCR_setup(), SSCR_prime() returns FALSE.

This function also returns FALSE if either of the following occurs:

- The algorithm has more than 16 memTab records.
- The algorithm's algAlloc() function returns a value less than 1.

This function uses the UTL_assert() macro to confirm that the fxns parameter is not NULL and the scrBucketIndex is less than the bucketCnt specified in the call to SSCR_setup().

**Constraints**

- Must be called after SSCR_setup() and before any calls to SSCR_createBuf(), SSCR_getBuf(), or SSCR_deleteBuf().

**Example** This example is from chan_regcell.c in RF5.

```
/* determine worst-case scratch requirements */
if (SSCR_prime(cellHandle->scrBucketIndex,
               cellHandle->algFxns,
               cellHandle->algParams) == FALSE) {
    return (FALSE);
}
```

## SSCR_setup

Set up the SSCR module

**Syntax**

```
status = SSCR_setup(heapId, bucketCnt, bucketBuf[],bucketSize[]);
```

**Parameters**

| | |
|---|---|
| `Int heapId;` | /* memory segment ID for buffer allocation */ |
| `Uns bucketCnt;` | /* number of buckets to create */ |
| `Ptr bucketBuf[];` | /* optional array of bucket buffers */ |
| `Uns bucketSize[];` | /* optional array of bucket sizes */ |

**Return Value** `Bool` /* Success or failure. TRUE (1) = success, else failure. */

**Description** SSCR_setup() sets up the SSCR module. A program should call this function once. If you are using the CHAN module, the CHAN_setup() function calls SSCR_setup().

The heapId parameter specifies where the SSCR module allocates buffers. The bucket list used internally by the SSCR module is allocated from the segment for DSP/BIOS objects.

The bucketCnt parameter specifies the number of buckets to be used by the application. For the "priority level" method, there should be one bucket for each priority level at which algorithms are executed. For the "locking mechanism" method, this value should be set to 1.

The bucketBuf[] and bucketSize[] arrays are optional parameters. If you want to specify the buffers and sizes for the buckets rather than letting the SSCR module determine the size and allocate the buffers, use these parameters. The arrays must the same length specified by bucketCnt.

If the bucketBuf[] and bucketSize[] arrays are NULL, the SSCR module manages the buffers internally by determining the sizes needed in SSCR_prime() and allocating the buffers in SSCR_createBuf().

This function returns FALSE if the internal bucket list cannot be allocated.

This function uses the UTL_assert() macro to confirm that if the bucketSize[] array is specified, the bucketBuf[] array is also not NULL.

**Constraints**

- Must be called after SSCR_init() and before SSCR_prime().

**Example** This example is from chan_setup.c in RF5.

```
/* Set-up the SSCR module */
return (SSCR_setup(internalHeap, bucketCnt, bucketBuf, bucketSize));
```

# 11 UTL Module

**Name** UTL – Utility module for debugging and diagnostics

**Synopsis** `#include <utl.h>`

### Types and Constants

```
/* numeric value from 0 to 70 indicating level of debugging */
UTL_DBGLEVEL

/* debugging class: if defined to be 0 or 1 the class is explicitly turned off or on */
UTL_LOGERROR      /* class: LOG_printf of custom error messages */
UTL_LOGWARNING    /* class: LOG_printf of custom warning messages */
UTL_LOGMESSAGE    /* class: LOG_printf of custom general messages */
UTL_LOGDEBUG      /* class: LOG_printf of custom debugging messages */
UTL_ASSERT        /* class: assertions */
UTL_STS           /* class: STS time measurement utilities */
UTL_ALGMEM        /* class: showing algorithms' memory reqs. and heap usage */
```

### Functions and Macros

```
class: LOG_printf of custom error messages
UTL_logError()    /* print error message with zero arguments */
UTL_logError0()   /* print error message with zero arguments */
UTL_logError1()   /* print error message with one argument */
UTL_logError2()   /* print error message with two arguments */

class: LOG_printf of custom warning messages
UTL_logWarning()  /* print warning message with zero arguments */
UTL_logWarning0() /* print warning message with zero arguments */
UTL_logWarning1() /* print warning message with one argument */
UTL_logWarning2() /* print warning message with two arguments */

class: LOG_printf of custom general messages
UTL_logMessage()  /* print general message with zero arguments */
UTL_logMessage0() /* print general message with zero arguments */
UTL_logMessage1() /* print general message with one argument */
UTL_logMessage2() /* print general message with two arguments */

class: LOG_printf of custom debugging messages
UTL_logDebug()    /* print debug message with zero arguments */
UTL_logDebug0()   /* print debug message with zero arguments */
UTL_logDebug1()   /* print debug message with one argument */
UTL_logDebug2()   /* print debug message with two arguments */

class: any of the classes above
UTL_setLogs()     /* set default LOG objects for errors/warnings/messages/debugging */

class: assertions
UTL_assert()      /* verify that a condition is true, halt target otherwise */
```

```
class: STS time measurement utilities
UTL_stsDefine()      /* declare a UTL STS object for use with UTL_sts* macros */
UTL_stsStart()       /* start timer */
UTL_stsStop()        /* stop timer */
UTL_stsPeriod()      /* measure period */
UTL_stsPhase()       /* measure phase */
UTL_stsReset()       /* reset a UTL STS object */

class: showing algorithms' memory reqs. and heap usage
UTL_showAlgMem()      /* report algorithm instance's memory usage */
UTL_showAlgMemName() /* report algorithm instance's memory usage */
UTL_showHeapUsage()  /* report dynamic heap usage */
```

**Description** The module provides several classes of features. Each debugging feature is implemented via a macro (or set of macros). The macro expands to code if the feature class is enabled; otherwise it expands to nothing. Features are enabled using the -d preprocessor option. There are two ways to enable classes of features:

- To enable or disable a class individually, define a flag name as 0 (disabled) or 1 (enabled). These names are shown in the "Flag Name" column in Table 5.
- To enable all classes up to a particular level, define UTL_DBGLEVEL with a numeric debugging level. The values for UTL_DBGLEVEL are shown in the "Debugging Level" column in Table 5. A debugging level enables all classes up to that level. For example, if you set UTL_DBGLEVEL to 20, both error and warning messages are enabled.

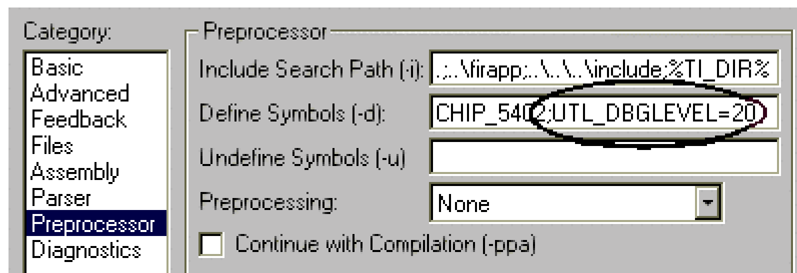


**Figure 13. Setting UTL_DBGLEVEL**

The individual and group methods can be used in combination to set a general level and then enable or disable individual classes.

**Table 5. UTL Module Classes**

| Class | Description | Flag Name | Level |
|---|---|---|---|
| | All classes disabled | | 0 |
| Error messages | Print errors to a LOG object via UTL_logError() | UTL_LOGERROR | 10 |
| Warning messages | Print warnings to a LOG object via UTL_logWarning() | UTL_LOGWARNING | 20 |
| General messages | Print general messages to a LOG object via UTL_logMessage() | UTL_LOGMESSAGE | 30 |
| Debug messages | Print debug messages to a LOG object via UTL_logDebug() | UTL_LOGDEBUG | 40 |
| Assertions | Halt execution if the condition in UTL_assert() fails | UTL_ASSERT | 50 |
| Time statistics | Store real-time parameters in STS objects with UTL_sts*() | UTL_STS | 60 |
| Algorithm memory usage | Report heap usage with UTL_showAlgMem(), UTL_showAlgMemName(), UTL_showHeapUsage() | UTL_ALGMEM | 70 |

Typically, levels 40-70 are used during development, and levels 0-30 are used in deployment mode.

Reference Framework Levels 3 and 5 are supplied with UTL_DBGLEVEL set to 70, which enables all debug levels. Reference Framework Level 1, however, uses a UTL_DBGLEVEL of 60 since it has no need for heap usage or algorithm memory use debugging.

Details on these classes are provided in the descriptions of individual macros. For background on the UTL module, see Section 3, *About the UTL Module*, page 10.

**Examples**

- -d"UTL_DBGLEVEL=30"
  Enables error, warning, and general messages (macros for those features are turned into actual code) and disables all others (their macros expand to nothing).
- -d"UTL_DBGLEVEL=20 -d"UTL_ASSERT=1"
  Enables error/warning messages and assertions.
- -d"UTL_DBGLEVEL=70 -d"UTL_STS=0"
  Enables all classes except STS.

The following pages list and describe the UTL macros in alphabetic order.

## UTL_assert

Halt execution if condition is false

**Syntax** `UTL_assert( condition )`

**Parameters** `Expression condition` /* condition under which target is halted */

**Description** This is an assertion macro for exceptions. It halts the target if the condition specified as the argument is false. For example, the following makes sure handle (pointer) algFIR is not NULL.

```
UTL_assert( algFIR != NULL );
```

If an assertion fails, we interrupt the execution flow and preserve the state of the system.

On a 'C54x, *if* CCStudio is connected to the target, we place a word in program memory (0xf4f0) that causes the target to halt and CCStudio shows the line of code where the failed assertion is. If you are using 'C54x without CCStudio, you should define, in your build options, UTL_ASSERTCCS to be 0.

On platforms where a software breakpoint is not available, we disable interrupts and run into an infinite loop, waiting for the user to halt the target manually.

UTL_assert()is useful when a condition must be fulfilled in order for the application to run correctly, but the condition may not be met during earlier development stages. To reduce code complexity, you can use the UTL_assert() macro rather than making all such functions return FALSE if they fail.

**Example** In RF3, the thrProcess thread's initialization procedure does not return FALSE if algorithm instantiation fails. Instead, it simply does the following:

```
UTL_assert( thrAudioproc[i].algFIR != NULL );
```

RF1 and RF3 also use UTL_assert as follows to assert that pipes the threads operate with have free frames. If such an assertion fails, the thread was posted in error. This may indicate there is a mistake in the pipe's configuration parameters.

```
UTL_assert( PIP_getReaderNumFrames( thrRxSplit.pipIn ) > 0 );
```

When the application is built for a release, there is no need to test these must-be-true conditions, so UTL_assert can be turned off. You need not remove them from the code; you may need them in the future if you modify the application.

## UTL_logDebug

Send debug message to log

**Syntax** `UTL_logDebug[0,1,2]( format[, arg1[, arg2]] )`

**Parameters**

| | | |
|---|---|---|
| `String` | `format` | /* printf-style format string */ |
| `Arg` | `arg1, arg2` | /* values for format string tokens */ |

**Description** If enabled, this macro performs a LOG_printf of the given parameters to the LOG object for debugging messages specified by UTL_setLogs(). The format, arg1, and arg2 parameters correspond to the LOG_printf parameters.

The suffix in the function name (nothing, 0, 1, or 2) determines how many parameters the formatted output has (none, none, one, or two, respectively).

**Constraints**

- Should only be called after UTL_setLogs() has been called.

**Example** If UTL_LOGDEBUG is defined as 1 (UTL_DBGLEVEL >= 40), the following macro expands to LOG_printf( UTL_logDebugHandle, "Current framesize: %d", fs ); where UTL_logDebugHandle contains the address of the LOG object for debugging messages as determined by the call to UTL_setLogs().

```
UTL_logDebug1( "Current framesize: %d", fs );
```

## UTL_logError

Send error message to log

**Syntax** `UTL_logError[0,1,2]( format[, arg1[, arg2]] )`

**Parameters**

| | | |
|---|---|---|
| `String` | `format` | /* printf-style format string */ |
| `Arg` | `arg1, arg2` | /* values for format string tokens */ |

**Description** If enabled, this macro performs a LOG_printf of the given parameters to the LOG object for errors specified by UTL_setLogs(). The format, arg1, and arg2 parameters correspond to the LOG_printf parameters.

The suffix in the function name (nothing, 0, 1, or 2) determines how many parameters the formatted output has (none, none, one, or two, respectively).

**Constraints**

- Should only be called after UTL_setLogs() has been called.

**Example** If UTL_LOGERROR is defined as 1 (or UTL_DBGLEVEL >= 10), the following macro expands to LOG_printf( UTL_logErrorHandle, "Error! Framesize overflow: %d", fs ); where UTL_logErrorHandle is the address of the LOG object for errors as specified by the call to UTL_setLogs().

```
UTL_logError1( "Error! Framesize overflow: %d", fs );
```

## UTL_logMessage

Send general message to log

**Syntax** `UTL_logMessage[0,1,2]( format[, arg1[, arg2]] )`

**Parameters**

| | | |
|---|---|---|
| `String` | `format` | /* printf-style format string */ |
| `Arg` | `arg1, arg2` | /* values for format string tokens */ |

**Description** If enabled, this macro performs a LOG_printf of the given parameters to the LOG object for general messages specified by UTL_setLogs(). The format, arg1, and arg2 parameters correspond to the LOG_printf parameters.

The suffix in the function name (nothing, 0, 1, or 2) determines how many parameters the formatted output has (none, none, one, or two, respectively).

**Constraints**

- Should only be called after UTL_setLogs() has been called.

**Example** If UTL_LOGMESSAGE is defined as 1 (UTL_DBGLEVEL >= 30), the following macro expands to LOG_printf( UTL_logMessageHandle, "Current framesize: %d", fs ); where UTL_logMessageHandle contains the address of the LOG object for general messages as determined by the call to UTL_setLogs().

```
UTL_logMessage1( "Current framesize: %d", fs );
```

## UTL_logWarning

Send warning message to log

**Syntax** `UTL_logWarning[0,1,2]( format[, arg1[, arg2]] )`

**Parameters**

| | | |
|---|---|---|
| `String` | `format` | /* printf-style format string */ |
| `Arg` | `arg1, arg2` | /* values for format string tokens */ |

**Description** If enabled, this macro performs a LOG_printf of the given parameters to the LOG object for warnings specified by UTL_setLogs(). The format, arg1, and arg2 parameters correspond to the LOG_printf parameters.

The suffix in the function name (nothing, 0, 1, or 2) determines how many parameters the formatted output has (none, none, one, or two, respectively).

**Constraints**

- Should only be called after UTL_setLogs() has been called.

**Example** If UTL_LOGWARNING is defined as 1 (UTL_DBGLEVEL >= 20), the following macro expands to LOG_printf( UTL_logWarningHandle, "Warning! Empty frame. Framesize: %d", fs ); where UTL_logWarningHandle contains the address of the LOG object for warnings as determined by the call to UTL_setLogs().

```
UTL_logWarning1( "Warning! Empty frame. Framesize: %d", fs );
```

## UTL_setLogs

Specify LOG objects to use for other macros

**Syntax**

```
UTL_setLogs( error_LOG,
             warning_LOG,
             message_LOG,
             debug_LOG );
```

**Parameters**

| | | |
|---|---|---|
| `LOG_Handle` | `error_LOG` | /* address of LOG object for errors */ |
| `LOG_Handle` | `warning_LOG` | /* address of LOG object for warnings */ |
| `LOG_Handle` | `message_LOG` | /* address of LOG object for messages */ |
| `LOG_Handle` | `debug_LOG` | /* address of LOG object for debug messages */ |

**Description** This macro sets which LOG objects are used by the UTL_log* macros. Error, warning, message, and debug messages can use the same LOG objects or different ones. Assertions and algorithm memory usage reports use the LOG object for debugging messages. (Time statistics use STS objects instead of LOG objects.)

There's no conceptual difference in printing errors, warnings, diagnostic messages, and debugging messages—only the syntax and the debugging class are different.

This macro expands to nothing if no class that uses log objects is enabled.

**Example** This macro causes the same LOG object to be used for all types of messages.

```
UTL_setLogs( &logTrace, &logTrace, &logTrace, &logTrace );
```

## UTL_showAlgMem

Show algorithm's memory requirements

**Syntax** `UTL_showAlgMem( handle );`

**Parameters** `IALG_Handle handle` /* XDAIS algorithm instance handle */

**Description** The UTL_showAlgMem macro reports XDAIS algorithm memory use. It provides size and memory type information for each allocated memory segment of the given algorithm, as well as the total size broken down by memory type.

If you set the variable UTL_ALGMEMVERBOSE to 1 in your Build Options, this macro also shows the actual memory address of each segment.

**Example** This macro reports memory use for the algFIR algorithm.

```
UTL_showAlgMem( thrAudioproc[i].algFIR );
```

## UTL_showAlgMemName

Show algorithm's memory requirements

**Syntax** `UTL_showAlgMemName( handle, algName );`

**Parameters**

| | | |
|---|---|---|
| `IALG_Handle` | `handle` | /* XDAIS algorithm instance handle */ |
| `String` | `algName` | /* name of algorithm */ |

**Description** The UTL_showAlgMemName macro reports XDAIS algorithm memory use. It provides size and memory type information for each allocated memory segment of the given algorithm, as well as the total size broken down by memory type.

This macro is similar to UTL_showAlgMem, except that you can pass it the name of the algorithm as a string to be shown in the output.

If you set the variable UTL_ALGMEMVERBOSE to 1 in your Build Options, this macro also shows the actual memory address of each segment.

**Example** This macro reports memory use for an algorithm in an RF5 cell.

```
UTL_showAlgMemName(cellHandle->algHandle, cellHandle->name);
```

## UTL_showHeapUsage

Show size and percentage use of a heap segment

**Syntax** `UTL_showHeapUsage( segment );`

**Parameters** `Int segment` /* heap segment ID */

**Description** The UTL_showHeapUsage macro shows the size and use of the given segment.

**Example** This macro shows the size and use of the INTERNALHEAP segment.

```
extern far Int INTERNALHEAP;
...
UTL_showHeapUsage( INTERNALHEAP );
```

## UTL_stsDefine

Create structure for use by UTL_sts* macros

**Syntax** `UTL_stsDefine( sts );`

**Parameters** `STS_Obj sts` /* Name of STS object */

**Description** There are several utility functions for measuring periods, execution times, and phases between periodic events. The only prerequisite for using an STS object with any UTL_sts* function is to use this macro in any of the .c modules of the application. An application can use this macro multiple times, each time for a different STS object.

This macro allocates a space for an internal structure used by UTL_sts* macros and initializes that object.

In order to use the UTL_sts* macros, the configuration file must contain STS objects (such as stsTime0) that can be used for time/period measurements with UTL_sts* functions. These must also be declared as externs in the C code.

**Example** This macro enables the use of UTL_sts* functions for the stsTime0 object.

```
extern far STS_Obj stsTime0;
...
UTL_stsDefine( stsTime0 );
```

## UTL_stsPeriod

Measure time difference between two calls

**Syntax** `UTL_stsPeriod( sts );`

**Parameters** `STS_Obj sts` /* Name of STS object */

**Description** Measures the difference in time between two calls to this procedure (for the same STS object), applies it to the STS object and returns the difference.

**Constraints**

- Should only be called after UTL_stsDefine() has been called.

**Example** This statement records periods between successive invocations of the function that calls it, and stores them in the stsTime0 STS object. If the maximum value shown in the Statistics View is ever significantly more than the expected value, the real-time deadline has been missed one or more times.

```
UTL_stsPeriod( stsTime0 );
```

In RF3, this statement is used at the beginning of thrRxSplitRun() procedure in the thrRxSplit.c file to record periods between successive invocations of the thrRxSplitRun() procedure, and stores them in STS object stsTime0. If you run RF1 and open the Statistics View in CCStudio, you see that stsTime0's average is exactly 10 ms—the duration of one frame in RF3—with a very small variance, just a few dozen cycles for the maximum and the minimum. If the maximum significantly differs from the expected value, for example 10.2 ms, the real-time deadline has been missed one or more times.

## UTL_stsPhase

Measure phase difference between two STS objects

**Syntax** `UTL_ stsPhase( stsSrc1, stsSrc2, stsDst );`

**Parameters**

| | | |
|---|---|---|
| `STS_Obj` | `stsSrc1, stsSrc2` | /* Name of STS objects to compare */ |
| `STS_Obj` | `stsDest` | /* Name of STS object to contain result */ |

**Description** Examines the absolute difference in timestamp values for STS objects stsSrc1 and stsSrc2, stores it in stsDst and returns that value. Objects stsSrc1 and stsSrc2 must have been used in a UTL_stsPeriod() function. For instance, if data transmit functions for two channels call UTL_stsPeriod( stsCh0 ) and UTL_stsPeriod( stsCh0 ) respectively, UTL_stsPhase( stsCh0, stsCh0, stsPh01 ) returns the phase difference between the two channels and update stsPh01 accordingly.

**Constraints**

- Should only be called after UTL_stsDefine() has been called.

**Example** The following example records the phase difference between two calls to UTL_stsPeriod that store their results in stsTime0 and stsTime1. It stores the phase difference in stsTime2.

```
UTL_stsPhase( stsTime0, stsTime1, stsTime2 );
```

In RF3, this function is used in the thrTxJoinRun() function to record the phase difference between calls to UTL_stsPeriod() in thrTxJoinRun() and thrRxSplitRun().

## UTL_stsReset

Reset STS object

**Syntax** `UTL_stsReset( sts );`

**Parameters** `STS_Obj` `sts` /* Name of STS object */

**Description** Resets the STS object in question.

**Constraints**

- Should only be called after UTL_stsDefine() has been called.

## UTL_stsStart

Start measuring execution time

**Syntax** `UTL_stsStart( sts );`

**Parameters** `STS_Obj sts` /* Name of STS object */

**Description** Start measuring execution time.

For accurate results, it is recommended that you disable interrupts before calling UTL_stsStart() and enable them after returning from UTL_stsStop().

**Constraints**

- Should only be called after UTL_stsDefine() has been called.

**Example** The following example starts measuring execution time for the stsTime1 object.

```
UTL_stsStart( stsTime1 );
```

## UTL_stsStop

Stop measuring execution time

**Syntax** `UTL_stsStop( sts );`

**Parameters** `STS_Obj sts` /* Name of STS object */

**Description** Stop measuring execution time.

For accurate results, it is recommended that you disable interrupts before calling UTL_stsStart() and enable them after returning from UTL_stsStop().

**Constraints**

- Should only be called after UTL_stsDefine() and UTL_stsStart() have been called.

**Example** The following example starts measuring execution time for the stsTime1 object.

```
UTL_stsStop( stsTime1 );
```

## 12 References

For additional information, see the following sources:

**Product Documentation**

- *TMS320 DSP/BIOS User's Guide* (SPRU423)
- *TMS320C5000 DSP/BIOS API Reference Guide* (SPRU404)
- *TMS320C6000 DSP/BIOS API Reference Guide* (SPRU403)
- *DSP/BIOS TextConf User's Guide* (SPRU007)
- *DSP/BIOS Driver Developer's Guide* (SPRU616)
- *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352)
- *TMS320 DSP Algorithm Standard API Reference* (SPRU360)

**Application Notes**

- *Reference Frameworks for eXpressDSP Software: A White Paper* (SPRA094)
- *Reference Frameworks for eXpressDSP Software: RF1, A Compact Static System* (SPRA791)
- *Reference Frameworks for eXpressDSP Software: RF3, A Flexible, Multi-Channel, Multi-Algorithm, Static System* (SPRA793)
- *Reference Frameworks for eXpressDSP Software: RF5, An Extensive, High-Density System* (SPRA795)
- *Using IDMA2-Based XDAIS Algorithms in eXpressDSP RF5* (SPRA842)
- *A Video, Networking, Audio System on the C64x NVDK Using eXpressDSP RF5* (SPRA844)
- *A Multi-Channel Motion Detection System Using eXpressDSP RF5 NVDK Adaptation* (SPRA904)
- *The TMS320 DSP Algorithm Standard - A White Paper* (SPRA581)

**Web Resources**

- http://www.dspvillage.com

**IMPORTANT NOTICE**

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2003, Texas Instruments Incorporated