xDAIS DSKT2 User's Guide

Literature Number: SPRUEV5A
September 2007

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 TI 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. Information of third parties may be subject to additional restrictions.

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.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
Low Power Wireless	www.ti.com/lpw	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

Preface

About This Guide

The purpose of this document is to describe DSKT2 module xDAIS algorithm support and configuration APIs.

Additional Documents and Resources

You can use the following sources to supplement this user's guide.

- *Techniques for Implementing Shared Relocatable Buffers Using the TMS320 DSP Algorithm Standard* (SPRA790)
- *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352)
- *TMS320 DSP Algorithm Standard API Reference* (SPRU360)
- *Reference Framework RF5 Channel Infrastructure Design Document, Version 0.9*
- *Reference Frameworks for eXpressDSP Software: API Reference* (SPRA147A)

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `mono-spaced font`. Examples use **`bold`** for emphasis, and interactive displays use **`bold`** to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).
- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets. Unless the square brackets are in a **bold** typeface, do not enter the brackets themselves.

Acronyms and Definitions

- **DSKT2** – a new module that exports functions that automate the operations necessary for instantiating, activating, and controlling xDAIS algorithms within the framework.
- **IALG Interface** – a set of standard interface functions exported by an xDAIS algorithm (algAlloc, algActivate, algControl, algDeactivate, algFree, algInit, algMoved, algNumAlloc).
- **Scratch Memory** – an xDAIS-defined scheme for sharing memory between xDAIS algorithms. Typically the framework does the following:

  a) Calls the algorithm's algActivate function to allow it to initialize scratch memory buffers from persistent memory.

  b) Calls the algorithm's processing function(s).

  c) Calls the algorithm's algDeactivate function to allow it to save the appropriate scratch memory contents back to persistent memory.

  Once this cycle is complete, the framework can repeat the cycle for another algorithm, which may use the same physical scratch memory during its processing.

Contents

1 Using the DSKT2 Interface 1-1

This chapter describes the DSKT2 interface.

1.1 Introducing the DSKT2 Interfaces 1-2
1.2 DSKT2 Calling Sequence 1-3
1.3 Configuring DSKT2 to Map Algorithm Data Memory Segments 1-4
1.3.1 RTSC Configuration of DSKT2 1-4
1.3.2 RTSC Configuration Example 1-7
1.3.3 Non-RTSC Configuration of DSKT2 1-9
1.4 DSKT2 Support for Shared Algorithm Scratch Memory 1-11
1.4.1 xDAIS Scratch Memory Support Overview 1-11
1.4.2 Scratch Groups for Arranged Sharing of Scratch Memory 1-12
1.5 Hardware Dependencies 1-14
1.6 Runtime Optimization of Algorithm Activation and Deactivation 1-15
1.7 Context Switching During xDAIS Callback Functions 1-15
1.8 DSKT2 IALG Extension: Providing Information to Algorithms 1-16

2 The DSKT2 API 2-1

This chapter provides additional information about the DSKT2 API.

2.1 Memory Requirements 2-2
2.2 Mandated Calling Sequences 2-2
2.3 DSKT2 APIs 2-3
2.3.1 DSKT2_createAlg 2-4
2.3.2 DSKT2_createAlg2 2-6
2.3.3 DSKT2_createAlgExt 2-8
2.3.4 DSKT2_activateAlg 2-9
2.3.5 DSKT2_deactivateAlg 2-10
2.3.6 DSKT2_deactivateAll 2-12
2.3.7 DSKT2_freeAlg 2-12
2.3.8 DSKT2_controlAlg 2-13

Chapter 1

Using the DSKT2 Interface

This chapter describes the DSKT2 interface.

Topic **Page**

1.1 Introducing the DSKT2 Interfaces 1–2
1.2 DSKT2 Calling Sequence 1–3
1.3 Configuring DSKT2 to Map Algorithm Data Memory Segments . . . 1–4
1.4 DSKT2 Support for Shared Algorithm Scratch Memory 1–11
1.5 Hardware Dependencies 1–14
1.6 Runtime Optimization of Algorithm Activation and Deactivation . 1–15
1.7 Context Switching During xDAIS Callback Functions 1–15
1.8 DSKT2 IALG Extension: Providing Information to Algorithms . . . 1–16

1.1 Introducing the DSKT2 Interfaces

The xDAIS library provides services to support the creation, initialization, control, and deletion of xDAIS algorithm instance objects.

The primary purpose of the DSKT2 library is to automate the standard algorithm operations that use an algorithm's IALG methods. A significant part of the work required to instantiate and use an algorithm is algorithm-independent. This work includes using an algorithm's IALG methods to instantiate the algorithm, get its memory requests, allocate memory for the algorithm, and activate/deactivate scratch memory. DSKT2 provides API interfaces (described in Section 2.3, *DSKT2 APIs*) to perform these and other tasks.

DSKT2 also introduces two primary features that result in fine-grained memory configuration and optimized memory management and use:

- ❏ You can define multiple memory heap segments and configure a mapping from an algorithm's memory requests to a preferred heap segment designated for the request's memory-space attribute.
- ❏ You can transparently share scratch memory assignments of algorithm instances that belong to the same scratch-group ID.

By adopting DSKT2, you can realize the following benefits:

- ❏ You can reduce the DSP application footprint by not duplicating functionality that is provided by DSKT2.
- ❏ Having all algorithm memory allocation performed inside DSKT2 provides the ability to retain a certain level of control of DSP-side memory allocations. Without this centralized allocation of algorithm memory, each algorithm developer could implement different allocation policies, for example grabbing all on-chip memory for their own algorithms, without consideration for other algorithms concurrently running on the DSP.

1.2 DSKT2 Calling Sequence

The following code example uses the typical calling sequence for DSKT2 APIs:

```
#include <std.h>
#include <ti/sdo/fc/dskt2/dskt2.h>
#include <sys.h>
#include <usescratch_ti.h>

Void smain(Int argc, Char * argv[])
{
    IUSESCRATCH_Handle  alg;
    IUSESCRATCH_Params  params = IUSESCRATCH_PARAMS;
    IALG_Fxns          *fxns;
    Int                 scratchId = 0;
    Int                 status = USESCRATCH_SOK;

   /* IALG_Fxns for an algorithm that uses scratch memory */
    fxns = (IALG_Fxns *)&USESCRATCH_TI_IUSESCRATCH;

    /* Create alg */
    alg = (IUSESCRATCH_Handle)DSKT2_createAlg(scratchId,
            fxns, NULL, (IALG_Params *)&params);
    if (alg == NULL) {
        SYS_abort("Memory allocation failed\n");
    }

    /* Activate alg before calling its process function */
    DSKT2_activateAlg(scratchId, (IALG_Handle)alg);

    /* Call alg's processing function */
    status = alg->fxns->process((IALG_Handle)alg);

    /* Deactivate the alg */
    DSKT2_deactivateAlg(scratchId, (IALG_Handle)alg1);

    /* Free alg */
    DSKT2_freeAlg(scratchId, (IALG_Handle)alg);
}
```

1.3 Configuring DSKT2 to Map Algorithm Data Memory Segments

For DSKT2 to fully honor algorithm memory requests it must know the following:

- ❏ What DSP/BIOS memory segments are available to allocate from?
- ❏ What are the attributes of the available memory segments?

This section describes the configuration of this information for the DSKT2 module.

There are two ways to configure DSKT2 parameters.

- ❏ You can use a low-level C language and linker command file based approach to directly modify global DSKT2 parameters.
- ❏ You can use XDC tooling to configure the RTSC module, DSKT2. The XDC tooling approach results in the generation of the same low-level C-based global variables, so the type of configuration technology used does not matter to the underlying DSKT2 library implementation.

You will still need to configure some DSP/BIOS heaps that will be used by the DSKT2 module, using the Tconf language and configuration files used for DSP/BIOS. (See SPRU007.)

For example, if you want to define a DSP/BIOS heap that will be used by DSKT2, you may have something like the following in your TCF file:

```
// Create a heap in external memory and give it a label
var EXTMEM = prog.module("MEM").create("EXTMEM")
EXTMEM.createHeap = true;
EXTMEM.enableHeapLabel = true;
EXTMEM.heapLabel = prog.extern("EXTMEM_HEAP");
```

Note that the heap must be given a label so that it can be referenced by DSKT2.

1.3.1 RTSC Configuration of DSKT2

Follow these steps to use RTSC to configure DSKT2:

1) The first statement related to DSKT2 in your RTSC configuration (CFG) file should get access to the DSKT2 module as follows:

```
var DSKT2 = xdc.useModule('ti.sdo.fc.dskt2.DSKT2');
```

2) To allow DSKT2 to use external scratch memory, add the following statement:

```
DSKT2.ALLOW_EXTERNAL_SCRATCH = true;
```

Setting the ALLOW_EXTERNAL_SCRATCH property to "true" means that if a scratch request in internal memory cannot be granted AND there is insufficient memory in persistent internal memory to allocate for the request, then DSKT2 allocates using external memory.

If you set this property to "false", then DSKT2_createAlg fails if there is insufficient scratch memory and insufficient internal persistent memory to satisfy the request.

3) Next, your CFG file should specify the heap that DSKT2 will use by default to allocate internal objects. This is the name of a heap label that has been defined in a TCF file. For example, if you defined the heap label "EXTMEM_HEAP" as in Section 1.3, *Configuring DSKT2 to Map Algorithm Data Memory Segments*, then in your CFG file, you could specify that DSKT2 use heap for allocating its internal objects as follows.

```
DSKT2.DSKT2_HEAP = "_EXTMEM_HEAP";
```

Notice that you add a leading underscore ("_") to the name of the heap label, since prog.extern() generates a "C" name, and DSKT2_HEAP is the assembly name of the heap.

4) Then, you map IALG memory space types to specific heaps. In the following statements, _L1D_HEAP and _EXTMEM_HEAP are heap labels that have been assigned to DSP/BIOS MEM segments

```
DSKT2.DARAM0 = "_L1D_HEAP";
DSKT2.DARAM1 = "_L1D_HEAP";
DSKT2.DARAM2 = "_L1D_HEAP";

DSKT2.SARAM0 = "_L1D_HEAP";
DSKT2.SARAM1 = "_L1D_HEAP";
DSKT2.SARAM2 = "_L1D_HEAP";

DSKT2.ESDATA = "_EXTMEM_HEAP";
DSKT2.IPROG = "_EXTMEM_HEAP";
DSKT2.EPROG = "_EXTMEM_HEAP";
```

5) Next, you link in the DSKT2 library. The following statement links in the debug library of DSKT2. You can set the debug property to "false" to link in the non-debug DSKT2 library.

```
DSKT2.debug = true;
```

6) Then, you assign sizes to the scratch groups. There can be up to 20 scratch groups for each of the memory types, DARAM and SARAM. In the following statements, the scratch group 0 has been assigned a size of 512 MAUs.

```
DSKT2.DARAM_SCRATCH_SIZES = [0x200,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0,0];
DSKT2.SARAM_SCRATCH_SIZES = [0x200,0,0,0,0,0,0,0,0,0,
                             0,0,0,0,0,0,0,0,0];
```

If you assign the same heap label to both DSKT2.DARAM0 and DSKT2.SARAM0, the values in DSKT2.SARAM_SCRATCH_SIZES are ignored; algorithms that request scratch memory from SARAM are granted scratch memory in DARAM if memory is available.

7) Finally, you name a function to implement cache writeback and invalidation. This function must have the following signature in your C code:

```
(*DSKT2_CacheWBInvFxn)(Ptr blockPtr, size_t byteCnt,
                       Bool wait)
```

For C6000 platforms, DSKT2 calls this function to ensure that the algorithm instance memory is not resident in a cache when it is instantiated. This is necessary for compliance with the xDAIS C6000 DMA Rules and Guidelines.

The default value of this parameter is "BCACHE_wbInv", so you don't need to set this on C6000 platforms. If you are configuring this for a C55, then you would set it to null as follows:

```
DSKT2.cacheWritebackInvalidateFxn = null;
```

1.3.2 RTSC Configuration Example

For example, the combined example.tcf file for the DSKT2 Tconf configuration might look like the following:

```
/* ======== example.tcf ======== */

// DaVinci platform
var platform = "ti.platforms.evmDM6446";
var params = null;

/* load the platform */
utils.loadPlatform(platform, params);

/* Enable BIOS features needed */
bios.enableRealTimeAnalysis(prog);
bios.enableMemoryHeaps(prog);
bios.enableTskManager(prog);

var DDR = prog.module("MEM").instance("DDR2");

/*
 *  Create external memory segment for this (simulated) board
 *  Enable heaps in it and define the label for heap usage.
 */
DDR.base             = 0x83F00000;
DDR.len              = 0x0FFE00;   // may be much bigger
DDR.space          = "code/data"; // so we can put code here
DDR.createHeap       = true;
DDR.enableHeapLabel  = true;
DDR["heapLabel"]     = prog.extern("EXTMEM_HEAP");
DDR.heapSize         = 0xc0000;
DDR.comment          = "DDR";

/*
 *  Enable heaps in L1DSRAM (internal L1 cache ram, fixed
 *  size) and define the label for heap usage.
 */
bios.L1DSRAM.createHeap       = true;
bios.L1DSRAM.enableHeapLabel  = true;
bios.L1DSRAM["heapLabel"]     = prog.extern("L1D_HEAP");
bios.L1DSRAM.heapSize         = 0x4000;
```

And, the combined example.cfg file for the DSKT2 RTSC configuration might look like the following:

```
/*  ======== example.cfg ========
 *  Example configuration of DSKT2 module
 */

// Get the DSKT2 module.
var DSKT2 = xdc.useModule('ti.sdo.fc.dskt2.DSKT2');

// If a scratch request in internal memory cannot be granted
// AND there is insufficient persistent internal memory to
// allocate for the request, THEN DSKT2 uses external memory.
DSKT2.ALLOW_EXTERNAL_SCRATCH = true;

// Set the heap that the DSKT2 will use to allocate internal
// objects. This is a heap label defined in a .tcf file.
DSKT2.DSKT2_HEAP = "_EXTMEM_HEAP";

// Map IALG memory space types to specific heaps assigned to
// DSP/BIOS MEM segments.
DSKT2.DARAM0 = "_L1D_HEAP";
DSKT2.DARAM1 = "_L1D_HEAP";
DSKT2.DARAM2 = "_L1D_HEAP";

DSKT2.SARAM0 = "_L1D_HEAP";
DSKT2.SARAM1 = "_L1D_HEAP";
DSKT2.SARAM2 = "_L1D_HEAP";

DSKT2.ESDATA = "_EXTMEM_HEAP";
DSKT2.IPROG = "_EXTMEM_HEAP";
DSKT2.EPROG = "_EXTMEM_HEAP";

// Link in the debug library of DSKT2.
DSKT2.debug = true;

// Assign sizes to scratch groups.
DSKT2.DARAM_SCRATCH_SIZES = [0x200,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];
DSKT2.SARAM_SCRATCH_SIZES = [0x200,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0];

// Function implementing cache writeback and invalidate.
// For C6000 platforms the default is "BCACHE_wbInv",
// so we don't need to set this on C6000 platforms.
// For a C55x, we would set it to null, as follows:
//    DSKT2.cacheWritebackInvalidateFxn = null;
```

1.3.3 Non-RTSC Configuration of DSKT2

It is also possible to configure DSKT2 without using RTSC tooling. In this case, you still need to configure DSP/BIOS heaps in a TCF file, as described in Section 1.3.1, *RTSC Configuration of DSKT2*. You will also need a C source file and linker command file to build your application.

To match the RTSC configuration of DSKT2 shown in the previous section, your C file should contain the following:

```
/*  ======== dskt2_config.c ========
 *  DSKT2 scratch config file to initialize pre-allocated
 *  scratch heap size for each node priority level.  */
#include <std.h>
#include <ti/sdo/fc/dskt2/dskt2.h>
#include <ti/sdo/fc/dskt2/_dskt2.h>

#include <bcache.h>

#ifdef _6x_
far DSKT2_CacheWBInvFxn DSKT2_cacheWBInvFxn = BCACHE_wbInv;
#else
DSKT2_CacheWBInvFxn DSKT2_cacheWBInvFxn = NULL;
#endif

Uns _DSKT2_DARAM_SCRATCH_SIZES[DSKT2_NUM_SCRATCH_GROUPS] = {
    0x200       /* 0 */
    0,          /* 1 */
    0,          /* 2 */
    0,          /* 3 */
    0,          /* 4 */
    0,          /* 5 */
    0,          /* 6 */
    0,          /* 7 */
    0,          /* 8 */
    0,          /* 9 */
    0,          /* 10 */
    0,          /* 11 */
    0,          /* 12 */
    0,          /* 13 */
    0,          /* 14 */
    0,          /* 15*/
    0,          /* 16 */
    0,          /* 17 */
    0,          /* 18 */
    0,          /* 19 */
};
```

```
Uns _DSKT2_SARAM_SCRATCH_SIZES[DSKT2_NUM_SCRATCH_GROUPS] = {
    0x200,      /* 0 */
    0,          /* 1 */
    0,          /* 2 */
    0,          /* 3 */
    0,          /* 4 */
    0,          /* 5 */
    0,          /* 6 */
    0,          /* 7 */
    0,          /* 8 */
    0,          /* 9 */
    0,          /* 10 */
    0,          /* 11 */
    0,          /* 12 */
    0,          /* 13 */
    0,          /* 14 */
    0,          /* 15*/
    0,          /* 16 */
    0,          /* 17 */
    0,          /* 18 */
    0,          /* 19 */
};
```

The linker command file needs to link in the DSP/BIOS generated linker command file and the DSKT2 library, and define the mapping of IALG memory spaces to DSP/BIOS heaps. The following linker command file code illustrates this.

```
/*
 * example.cmd
 * Linker command file for non RTSC DSKT2 configuration.
 */

/* Link in DSP/BIOS generated linker command file */
-l examplecfg.cmd

/* Link in DSKT2 library */
-l dskt2.a64P
```

```
/*
 * Mapping of IALG mem spaces to BIOS heaps
 * (Note: there is no DSKT2 mapping of IALG_EXTERNAL)
 */
__DSKT_DARAM0 = _L1D_HEAP;      /* IALG_DARAM0 */
__DSKT_DARAM1 = _L1D_HEAP;      /* IALG_DARAM1 */
__DSKT_DARAM2 = _L1D_HEAP;      /* IALG_DARAM2 */
__DSKT_SARAM0 = _EXTMEM_HEAP;   /* IALG_SARAM0 */
__DSKT_SARAM1 = _EXTMEM_HEAP;   /* IALG_SARAM1 */
__DSKT_SARAM2 = _EXTMEM_HEAP;   /* IALG_SARAM2 */
__DSKT_ESDATA = _EXTMEM_HEAP;   /* IALG_ESDATA */
__DSKT_IPROG  = _EXTMEM_HEAP;   /* IALG_IPROG */
__DSKT_EPROG  = _EXTMEM_HEAP;   /* IALG_EPROG */

/* DSKT2 heap for allocating internal objects */
__DSKT2_HEAP  = _EXTMEM_HEAP;
```

1.4 DSKT2 Support for Shared Algorithm Scratch Memory

The xDAIS standard includes provisions to allow algorithms to share memory buffers and reduce the overall application memory footprint.

1.4.1 xDAIS Scratch Memory Support Overview

TMS320 DSP Algorithm Standard (xDAIS) compliant algorithms request memory blocks from their housing application frameworks. Each requested memory block is designated as either "persistent" or "scratch".

Scratch memory is defined as a type of memory that is freely used by an algorithm without regard to its prior contents, that is, no assumptions about the content can be made by the algorithm and the algorithm is free to leave it in any state. The algorithm instance initializes its scratch buffers when the application activates the instance by granting it exclusive access to the scratch region and calling its IALG activation function, algActivate().

During initialization of its scratch buffers in algActivate() the algorithm can only access its static memory and what's saved in its persistent instance memory. The application calls algDeactivate() when it wants to use/free up the scratch area granted to the instance. The algorithm saves to its persistent memory any information in its scratch buffers that it will need later during re-activation to re-initialize its scratch buffers.

After the standard algorithm initialization call to algInit(), all compliant algorithm instances with scratch buffers are in either one of these two states: activated or deactivated. It is in activated state if no algDeactivate

calls have been issued since the last algActivate call. Algorithm is in deactivated state if it has received no algActivate calls since algInit or since the last algDeactivate.

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

The basic rule of sharing a system overlay scratch area is that at any given time at most one algorithm instance sharing the overlay area can be activated—all other instances must be de-activated.

A xDAIS application framework is always in charge as to where to allocate scratch memory and decides which groups of algorithm instances (if any) will share a common “scratch” overlay region and when a particular algorithm instance gets activated or deactivated.

1.4.2 Scratch Groups for Arranged Sharing of Scratch Memory

Scratch groups form the basis for arranging multiple algorithm instances to share "scratch memory". Each scratch group is associated with an ID. It is the responsibility of the application framework to ensure mutually exclusive operation of algorithm instances having the same scratch group ID. No algorithm instance is allowed to preempt another algorithm instance’s processing stage belonging to the same scratch group.

One way to assign scratch IDs if algorithms are run from DSP/BIOS tasks is to use the task priority as the scratch ID for this algorithm. This technique can be used so long as the application doesn’t change task priorities at run-time.

Using a “task priority level” based protection approach, algorithms that share the same scratch buffers run at the same priority level, and preemption is avoided. Therefore, multiple algorithms running at the same task priority level can share the same physical addresses for their scratch buffers and are ensured exclusive access to the shared buffer when their processing functions get called.

Here’s an overview of how this works for DSKT2: OEMs configure the size of scratch memory to be supported for each group ID. For example, this size could be set to the largest amount of scratch needed by any of the algorithms to be instantiated with the specified scratch ID.

For example, the following statements configure two scratch buffers—one of 2048 MAU and another of 1024 MAU. These are designated as the default sizes of the shared scratch memory area allocated in the IALG_DARAM memory spaces.

```
// Get the DSKT2 module
var DSKT2 = xdc.useModule('ti.sdo.fc.dskt2.DSKT2');

// Assign sizes 2048 and 1024 to scratch groups 5 and 6,
// respectively, for IALG_DARAM memory spaces.
DSKT2.DARAM_SCRATCH_SIZES = [0,0,0,0,0, 2048, 1024,
        0,0,0,0,0,0,0,0,0,0,0,0,0];
```

These scratch buffers are shared among algorithm instances configured with scratch group IDs 5 and 6, respectively. The following actions occur:

1) When the *first* xDAIS algorithm created with the specified scratch ID requests scratch memory, DSKT2 dynamically allocates "shared" scratch buffers of the maximum size configured by the OEM for the specified scratch ID and the amount requested by the algorithm.

2) DSKT2 uses the OEM-configured mappings of IALG memory spaces to designated DSP/BIOS heap segments when determining which system heaps to use for creating the shared scratch buffers.

3) DSKT2 pieces out individual scratch buffers from shared buffers, to satisfy each individual scratch buffer requested by the algorithm. The memory allocator processes each request for scratch in the memTab[] by assigning a slice of the shared scratch buffer with adjustments for alignment. If the algorithm requests more scratch than is allocated in the shared scratch buffer, DSKT2 allocates as much of the scratch memory as it can from any other shared scratch buffer available to the same scratch group. All other requests are fulfilled as non-shared private memory based on the following policy:

    a) If the algorithm requests scratch memory in IALG_DARAM0, the DSKT2 allocator first tries to satisfy the request using the shared IALG_DARAM scratch buffer. If it cannot, it tries to satisfy the request using the IALG_SARAM shared scratch of the same scratch group. If both attempts fail, it attempts to dynamically allocate the buffer in one of the OEM configured “internal” system heaps. If those attempts also fail:

        i) DSKT2 indicates failure if “Allow External Memory for IALG_SCRATCH requests” configuration is not enabled. That is, if DSKT2.ALLOW_EXTERNAL_SCRATCH is configured to be false.

        ii) Otherwise, DSKT2 attempts to allocate memory in external heap.

b) If the algorithm requests scratch memory in IALG_DARAM1 or in IALG_DARAM2, and the actual mapped DSP/BIOS heap is different than the heap for IALG_DARAM0, the allocator first tries to satisfy the request by attempting dynamic allocation in the requested memory space as configured by OEM. If that fails, any existing shared scratch buffers at the same priority level in IALG_DARAM0 or IALG_SARAM0 are tried respectively. If still not satisfied, DSKT2, attempts to dynamically allocate the buffer in one of the OEM configured “internal” system heaps. If those attempts also fail:

   i) DSKT2 indicates failure if “Allow External Memory for IALG_SCRATCH requests” configuration is not enabled.

   ii) Otherwise, DSKT2 attempts to allocate memory in “external” heap.

c) Scratch memory requests in IALG_SARAM0, IALG_SARAM1, or IALG_SARAM2 are handled similarly to their DARAM counterparts as outlined in steps (a) and (b)

4) As each new algorithm requesting scratch memory at a given scratch group is instantiated, the scratch is pieced out from the previously allocated shared buffer, and the reference count for the buffer is incremented.

5) When freeing algorithm instance memory (via DSKT2_freeAlg), any shared scratch buffer is not immediately freed, but those allocated outside the shared buffers are dynamically freed. Each time an algorithm using scratch is deleted, the reference count for the shared scratch buffer at the given priority level is decremented.

6) When the last algorithm using scratch at a given scratch group is deleted, the shared scratch buffer at that scratch group is freed by DSKT2, as it is no longer needed.

1.5 Hardware Dependencies

DSKT2 does not reference hardware-specific configuration directly. Instead, the DSP/BIOS configuration tools are used in conjunction with the DSP/BIOS MEM module for creating and configuring multiple system heaps and dynamic memory allocation and freeing. DSP/BIOS TSK APIs are called for implementing critical sections. If you have linked with the debug library, LOG_printf calls are used for limited real-time trace messages.

DSKT2 is designed to be modular and independent—all current DSP/BIOS dependencies can be implemented independently by custom application frameworks.

1.6 Runtime Optimization of Algorithm Activation and Deactivation

A benefit of using the DSKT2-mandated calling sequences outlined in Section 1.2, *DSKT2 Calling Sequence* is that the implementation of the DSKT2_activateAlg and DSKT2_deactivateAlg APIs can transparently maintain runtime state information to minimize real activation/de-activation of the algorithm instances.

Since DSKT2 can track state at runtime to determine when there is no actual “sharing” of scratch buffers it can transparently avoid unnecessary calls to IALG algActivate and algDeactivate functions. Actual de-activation of the “current” algorithm is deferred by implementing DSKT2_deactivateAlg “optimistically”. When DSKT2_activateAlg needs to activate an algorithm it checks if the instance is already active within the same scratch group, if it is already active nothing needs to be done. If another algorithm (identified by unique IALG_handles) is currently active, DSKT2_activateAlg de-activates the other (current active) instance and activates the given algorithm instance. It is sufficient for DSKT2 to keep track of only the “currently active” algorithm for each scratch group and a single test.

You can use DSKT2_deactivateAll to perform the deactivation without deferral.

1.7 Context Switching During xDAIS Callback Functions

In order to adopt a task-priority based scratch buffer sharing, we impose certain restrictions on callback functions that can be called by xDAIS algorithms. To maintain coherence of algorithm scratch buffers, callback functions are not allowed to issue any operations that may result in a context switch that may lead to the preemption of current task by another task at the same priority level. These callback functions could be defined by the xDAIS spec (for example the ACPY2 or ACPY3 DMA APIs) or they may be proprietary xDAIS-compliant algorithm framework APIs.

The restriction is required since there is no mechanism available for the callback function to be able to do algorithm deactivation and then re-activation (algActivate and algDeactivate functions cannot be called during any of its algorithm processing calls, that is, during an intermediate stage of execution). A framework or callback function cannot call algDeactivate or algDeactivate calls, which would be one way to ensure the integrity of instance scratch buffers. xDAIS algorithms are developed under the assumption that they are operationally not preemptable. If they do get preempted, their persistent *and* scratch memory must be saved and restored by the framework, making the preemption transparent to the algorithm. Additionally, algorithms implement algActivate/algDeactivate

knowing that they can only be called at well-specified steady states, not at arbitrary execution points within any one of its processing or control functions.

Finally, if a callback function has to share an algorithm's scratch buffer during the execution of the callback function, it can save and then restore the shared scratch before resuming back to the algorithm. This approach is a fair one, as the burden is on the callback side.

1.8 DSKT2 IALG Extension: Providing Information to Algorithms

In anticipation of formalization of a future xDAIS spec enhancement involving the IALG interface, the DSKT2 framework provides the actual physical memory space information for each memory buffer it grants to the algorithm during the algInit() call. Algorithms that are designed to exploit this feature will be able to utilize the provided IALG_MemSpace information to optimize or fine tune its operation or optionally return “failure” status to indicate inability to ensure proper operation with the provided memory.

xDAIS-compliant algorithms use algAlloc() to provide information about what type of memory space they want each buffer to be allocated on. However, they are expected to function correctly even if they don’t get the exact memory space they requested. Applications, for example, due to scarcity of internal memory may decide to allocate some of the buffers in external memory even thought the algorithm’s request was for internal memory.

The only risk involved is that algorithms designed to exploit this enhancement may not operate correctly when: (1) they are deployed in non-DSKT2 frameworks, and (2) they expect and rely on the memory space designation information to be passed by the framework, and (3) when the framework allocates memory in a memory space other than what the algorithm requested. This risk can be minimized by disclosing the information and impact properly to the algorithm developers.

Chapter 2

The DSKT2 API

This chapter provides additional information about the DSKT2 API.

Topic **Page**

2.1 Memory Requirements 2–2
2.2 Mandated Calling Sequences 2–2
2.3 DSKT2 APIs 2–3

2.1 Memory Requirements

All instance memory for the created algorithm instances uses the configuration provided DSP/BIOS memory heaps.

All internal DSKT2 objects are allocated on the configuration-provided DSKT2_HEAP, during the first call to one of the DSKT2_createAlg APIs—DSKT2_createAlg(), DSKT2_createAlg2(), or DSKT2_createAlgExt().

Typically, you configure DSKT2_HEAP to map to a DSP/BIOS heap in external memory. This helps save internal memory for algorithm buffers.

DSKT2 initialization does not occur until the first call to DSKT2_createAlg, and some memory internal to DSKT2 is allocated at this point. This memory is never freed, making it appear as if there is a memory leak in the algorithm. You can see this memory difference if you use the DSP/BIOS MEM_stat API to measure the heap sizes before and after the first call to DSKT2_createAlg.

2.2 Mandated Calling Sequences

To ensure protection of scratch memory shared by each “Scratch Group”, each algorithm instance must be prepared to gain “exclusive access” to its scratch memory via a DSKT2_activateAlg call. After the algorithm’s processing stage is completed, DSKT2_deactivateAlg must be called to relinquish its exclusive access to the shared scratch.

The algInit() function called through DSKT2_createAlg must not access its scratch buffers, since DSKT2_activateAlg has not yet been called, and the algorithm is not considered to be in the "active" state at this point.

2.3 DSKT2 APIs

The new DSKT2 APIs are described in the subsections that follow. The following table shows the corresponding IALG functions called by each DSKT2 API:

Table 2-1. DSKT2 APIs

DSKT2 Function	IALG Function(s)
DSKT2_createAlg	algNumAlloc, algAlloc, algInit
DSKT2_createAlg2	algNumAlloc, algAlloc, algInit
DSKT2_createAlgExt	algNumAlloc, algAlloc, algInit
DSKT2_freeAlg	algNumAlloc, algFree
DSKT2_controlAlg	algControl
DSKT2_activateAlg	algActivate
DSKT2_deactivateAlg	algDeactivate
DSKT2_deactivateAll	algDeactivate

2.3.1 DSKT2_createAlg

```
IALG_Handle DSKT2_createAlg(
        Int           scratchMutexId,
        IALG_Fxns    *fxns,
        IALG_Handle   parent,
        IALG_Params  *params);
```

Implementation

The DSKT2_createAlg function creates and initializes a xDAIS algorithm instance object. It uses the algorithm's IALG interface functions (passed in fxns) to query the algorithm for its memory needs, allocate the memory for the algorithm, and call the algorithm's algInit function to let the new algorithm instance object initialize itself using the allocated memory.

On success, the function returns the IALG_Handle of the new algorithm instance that has been created. On failure, the function returns NULL, and all memory allocated during the call (that used for algorithm query, and portions of algorithm memory that were successfully allocated during the function) is freed before DSKT2_createAlg returns.

Parameters

Int scratchMutexId	scratchMutexId associates the created instance with a Scratch Group. Values = 0-19: instances created with same ID share a common scratch memory buffer. The caller ensures algorithms created with the same "scratchMutexId" do not execute simultaneously. Value = -1: Disables scratch sharing when creating this algorithm instance.
IALG_Fxns *fxns	Pointer to the algorithm's IALG_Fxns table.
IALG_Handle parent	Handle of parent algorithm (optional).
IALG_Params *params	Pointer to an IALG_Params structure.

Return

non-NULL	IALG_Handle for the new instance object
NULL	Instance creation failed.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- ❑ fxns is a valid pointer to an IALG_Fxns structure (containing the vtable for a xDAIS-compliant algorithm.)

Postconditions

The following conditions are true immediately after returning from this method:

- If scratchMutexId is "-1", all instance scratch memory is allocated as persistent (i.e. not shared) and instance operation does not require mutual exclusion.
- With the exception of any initialization performed by algActivate and (IDMA2) dmaInit, all of the instance's persistent memory is initialized and the object is ready to be used.

Comments

DSKT2_createAlg performs initialization necessary to complete the run-time creation of an algorithm's instance object. After a successful return from DSKT2_createAlg, the algorithm's instance object can be activated via a DSKT2_activateAlg (as well as dmaInit), if IDMA2 is implemented by the algorithm) before it can be used to process data.

The parent argument is a handle to another algorithm instance object. This parameter is often NULL, indicating that no parent object exists. This parameter allows clients to create a shared algorithm instance object and pass it to other algorithm instances. For example, a parent instance object might contain global read-only tables that are used by several instances of a vocoder.

The params argument is a pointer to algorithm-specific parameters that are necessary for the creation and initialization of the instance object. This pointer points to the same parameters passed to the algorithm's IALG algAlloc function. However, this pointer may be NULL. In this case, algorithm's IALG function algInit, must assume default creation parameters.

DSKT2_createAlg tries to dynamically allocate instance memory based on the IALG_MemSpace attribute of the requested memory. Global DSKT2 configuration settings allow OEM to designate a memory heap for each IALG_MemSpace. DSKT2_createAlg attempts to allocate memory in the requested space, but may search for alternative heaps when preferred heap is not large enough.

2.3.2 DSKT2_createAlg2

```
IALG_Handle DSKT2_createAlg2(
        Int          scratchMutexId,
        IALG_Fxns   *fxns,
        IALG_Handle  parent,
        IALG_Params *params,
        Int          extHeapId);
```

Implementation

DSKT2_createAlg2 performs the same actions and has the same requirements and consequences as DSKT2_createAlg. The difference is that it also has an extHeapId input parameter.

When you use the DSKT2_createAlg2 API, all IALG memory requests in IALG_ESDATA type memory are allocated in the memory segment identified by the extHeapId parameter, rather than from the DSP/BIOS memory heap that was mapped to IALG_ESDATA.

This API was created specifically for multi-processor applications in which DSP algorithms are launched from a GPP (General Purpose Processor). In such cases, it may not be known in advance which DSP algorithms will be run. Rather than having to configure an external heap in the DSP image that meets the worst case scenario, the GPP can allocate and map a buffer to the DSP's memory space on the fly. The new heap size and base address information can then be passed to the DSP program, which can create a heap with the DSP/BIOS MEM_define API. The new heap ID can then be passed as the extHeapId argument to DSKT2_createAlg2.

The DSKT2_createAlg2 function creates and initializes a xDAIS algorithm instance object. It uses the algorithm's IALG interface functions (passed in fxns) to query the algorithm for its memory needs, allocate the memory for the algorithm, and call the algorithm's algInit function to let the new algorithm instance object initialize itself using the allocated memory.

On success, the function returns the IALG_Handle of the new algorithm instance that has been created. On failure, the function returns NULL, and all memory allocated during the call (that used for algorithm query and portions of algorithm memory that were successfully allocated during the function) is freed before DSKT2_createAlg2 returns.

Parameters

Int scratchMutexId — scratchMutexId is used to associate the created instance with a Scratch Group.
Values = 0-19: instances created with same ID share a common scratch memory buffer. The caller must ensure that algorithms

		created with the same "scratchMutexId" do not execute simultaneously. Value = -1: Disables scratch sharing when creating this algorithm instance.
	IALG_Fxns *fxns	Pointer to the algorithm's IALG_Fxns table.
	IALG_Handle parent	Handle of parent algorithm (optional).
	IALG_Params *params	Pointer to an IALG_Params structure.
	Int extHeapId	Segment ID of the memory heap to be used for all allocations in memory space IALG_ESDATA. This segment will be used instead of the default external memory segment.
Return	non-NULL	IALG_Handle for the new instance object
	NULL	Instance creation failed.

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined.

- ❏ fxns is a valid pointer to an IALG_Fxns structure (containing the vtable for a xDAIS-compliant algorithm).
- ❏ extHeapId must be greater than or equal to 0.

Postconditions

The following condition is true immediately after returning from this method:

- ❏ If scratchMutexId is "-1", all instance scratch memory is allocated as persistent (i.e. not shared) and instance operation does not require mutual exclusion.
- ❏ With the exception of any initialization performed by algActivate and (IDMA2) dmaInit, all of the instance's persistent memory is initialized and the object is ready to be used.

Comments

DSKT2_createAlg2 attempts to use the heap specified by extHeapId for all allocations in the IALG_MemSpace, IALG_ESDATA. This holds for any IALG_MemSpace that has been configured with the same memory segment as IALG_ESDATA. For example, if IALG_ESDATA and IALG_SARAM3 have both been configured to use the heap in the memory segment EXTMEM, then DSKT2_createAlg2 attempts to allocate memory requested in the spaces IALG_SARAM3 and IALG_ESDATA, from the heap specified by extHeapId.

2.3.3 DSKT2_createAlgExt

```
IALG_Handle DSKT2_createAlgExt(
              Int          scratchMutexId,
              IALG_Fxns   *fxns,
              IALG_Handle  parent,
              IALG_Params *params);
```

Implementation

DSKT2_createAlgExt performs the same actions and has the same requirements and consequences as DSKT2_createAlg. The difference is that all IALG memory requests are allocated in the DSP/BIOS memory heap that was mapped to IALG_ESDATA.

The DSKT2_createAlgExt function creates and initializes an xDAIS algorithm instance object. It uses the algorithm's IALG interface functions (passed in fxns) to query the algorithm for its memory needs (only size and alignment are taken into consideration), allocate the memory for the algorithm in the external heap, and call the algorithm's algInit function to let the new algorithm instance object initialize itself using the allocated memory.

On success, the function returns the IALG_Handle of the new algorithm instance that has been created. On failure, the function returns NULL, and all memory allocated during the call is freed before DSKT2_createAlgExt returns.

Parameters

Int scratchMutexId	scratchMutexId is used to associate the created instance with a Scratch Group. Values = 0-19: instances created with same ID can share common scratch resources. Values not in the range 0-19: Scratch resources cannot be shared by this algorithm instance. In the case of memory, this is not an issue, since all the algorithm's memory will be allocated as persistent, in the external heap.
IALG_Fxns *fxns	Pointer to the algorithm's IALG_Fxns table.
IALG_Handle parent	Handle of parent algorithm (optional).
IALG_Params *params	Pointer to an IALG_Params structure.

Return

non-NULL	IALG_Handle for the new instance object
NULL	Instance creation failed.

Preconditions The following condition must be true prior to calling this method; otherwise, its operation is undefined.

- ❏ fxns is a valid pointer to an IALG_Fxns structure (containing the vtable for a xDAIS-compliant algorithm).

Postconditions The following condition is true immediately after returning from this method:

- ❏ With the exception of any initialization performed by algActivate and (IDMA3) dmaInit, all of the instance's persistent memory is initialized and the object is ready to be used.

2.3.4 DSKT2_activateAlg

```
Void DSKT2_activateAlg(
            Int          scratchMutexId,
            IALG_Handle  alg);
```

Implementation The DSKT2_activateAlg function prepares a xDAIS algorithm instance object to start using its scratch memory. Unless the instance is already active, DSKT2_activateAlg uses the algorithm's IALG interface function algActivate (accessed via IALG_Handle alg) to initialize the algorithm instance's scratch buffers from persistent data memory.

DSKT2_activateAlg must be called before any processing or control methods of the algorithm instance, alg, can be called.

Parameters

Int scratchMutexId	scratchMutexId is used to associate the instance with a Scratch Group. Value must be the same ID used when creating this algorithm instance.
IALG_Handle alg	IALG_Handle of the algorithm instance to be activated.

Return nothing

Preconditions The following conditions must be true prior to calling this method; otherwise, its operation is undefined:

- ❏ alg must be a valid handle for the algorithm's instance object returned by an earlier call to DSKT2_createAlg or DSKT2_createAlg2.
- ❏ If alg uses DMA (IDMA2) dmaInit must be called before calling this method, ensuring that all of the instance's persistent memory is initialized and the object is ready to be used.

- ❏ No other algorithm method is currently being run on this instance. (This method never preempts any other method on the same instance.)

Postconditions

The following condition is true immediately after returning from this method:

- ❏ All of the instance's persistent and scratch memory is initialized and the object is ready to be used.

Comments

DSKT2_activateAlg performs all scratch memory initialization for an algorithm's instance object. After a successful return from DSKT2_activateAlg, the algorithm's instance object is ready to be used to process data.

DSKT2 maintains state information about current "active" algorithm instances at run-time, so that it does not perform unnecessary IALG "activation" calls when "alg" is already active. As part of this optimization it may call the algDeactivate method of the currently active algorithm instance in order to activate the this (alg) algorithm instance.

The implementation of the IALG algActivate is optional by xDAIS standard. So, the instance activation makes sense only when the method is implemented by the algorithm. The DSKT2_activateAlg method makes proper checks to ensure correct operation even if the algorithm does not implement the algActivate method.

2.3.5 DSKT2_deactivateAlg

```
Void DSKT2_deactivateAlg(
          Int          scratchMutexId,
          IALG_Handle  alg);
```

Implementation

DSKT2_deactivateAlg function prepares a xDAIS algorithm instance object to give up using its scratch memory. An object's deactivation logic involves calling the algorithm's IALG interface function algDeactivate (accessed via IALG_Handle alg) to save necessary data from the algorithm instance's scratch buffers to its persistent data memory.

DSKT2_deactivateAlg must be called after the last processing or control methods of the algorithm instance, alg, during each execute stage of its operation.

Parameters

Int scratchMutexId — scratchMutexId is used to associate the instance with a Scratch Group. Value must be the same ID used when creating this algorithm instance.

IALG_Handle alg	IALG_Handle of the algorithm instance to be deactivated.

Return nothing

Preconditions The following conditions must be true prior to calling this method; otherwise, its operation is undefined:

- ❏ alg must be a valid handle for the algorithm's instance object returned by an earlier call to DSKT2_createAlg or DSKT2_createAlg2.
- ❏ DSKT2_activateAlg must be called before calling this method.
- ❏ No other algorithm method is currently being run on this instance. (This method never preempts any other method on the same instance.)

Comments DSKT2_deactivateAlg marks an algorithm's shared scratch memory as available to other instances activation. After a successful return from DSKT2_activateAlg, the algorithm's processing or control functions cannot be called to process data.

DSKT2 maintains state information about current "active" algorithm instances at run-time, so that it does not perform unnecessary IALG "deactivation" calls. As part of this optimization it may defer the deactivation (via a call to the algDeactivate method) of this algorithm instance (alg) until a later stage, that is, when DSKT2_activateAlg is called to activate another algorithm instance.

The implementation of the IALG algDeactivate is optional by xDAIS standard. So, the instance deactivation makes sense only when the method is implemented by the algorithm. DSKT2 methods make proper checks to ensure correct operation even if the algorithm does not implement the algActivate or algDeactivate methods.

To improve performance, DSKT2 uses the concept of "lazy deactivation" to avoid unnecessary activation/deactivation operations by postponing the actual algorithm deactivation.

With "lazy deactivation" the algorithm's deactivate function is not called in DSKT2_deactivateAlg, but rather in the next DSKT2_activateAlg call for a new algorithm that shares the same scratch buffer. This way, if DSKT2_activateAlg is called for an algorithm that was just deactivated (through DSKT2_deactivateAlg), no unnecessary copying of data between persistent and scratch memory needs to be performed.

Since many times the process function of a single algorithm is called repeatedly, the deactivation only happens at the end of the sequence.

However, in some situations, it may be necessary to force the deactivation of the algorithm. For example, in power-down/wakeup situations where the algActivate function must re-initialize volatile memory. In this case, the algorithm must really be deactivated before power-down, so that the next call to DSKT2_activateAlg calls the algorithm's activate function.

In order to force the deactivation of algorithms that have been lazily deactivated, DSKT2 provides the DSKT2_deactivateAll function. This function does not deactivate any currently running algorithm, and will return the number of algorithms that are still running.

2.3.6 DSKT2_deactivateAll

```
Int DSKT2_deactivateAll();
```

Implementation All algorithms that have been deactivated lazily (that is, with DSKT2_deactivateAlg) are now really deactivated. Any algorithms that are still currently active are left as is. The number of algorithms that are still active is returned by this call.

Parameters None.

Return Int — The number of remaining active algorithms. Returns zero if no active algorithms exist.

Preconditions This function must be called with the TSK and SWI schedulers disabled.

2.3.7 DSKT2_freeAlg

```
Bool DSKT2_freeAlg(
         Int          scratchMutexId,
         IALG_Handle  alg);
```

Implementation DSKT2_freeAlg function deletes a xDAIS algorithm instance object and frees all persistent memory allocated for the instance object. A reference counting mechanism is implemented to free up instance scratch memory so that when the last instance within a scratch group is deleted all shared scratch memory allocated for the group is reclaimed.

DSKT2_freeAlg must be called during delete phase of operation to prevent memory leaks.

Parameters Int scratchMutexId — scratchMutexId is used to associate the instance with a Scratch Group. Value must be the same ID used when creating this algorithm instance.

	IALG_Handle alg	IALG_Handle of the algorithm instance to be deactivated.
Return	TRUE	Success
	FALSE	Failure

Preconditions The following conditions must be true prior to calling this method; otherwise, its operation is undefined:

- ❏ alg must be a valid handle for the algorithm's instance object returned by an earlier call to DSKT2_createAlg or DSKT2_createAlg2.

Postconditions The following condition is true immediately after returning from this method:

- ❏ If status is TRUE, then all memory allocated to the algorithm will have been freed.

Comments DSKT2_freeAlg frees an algorithm's persistent and when last member of a scratch group, its shared scratch memory. After a successful return from DSKT2_activateAlg, the IALG_Handle, alg, becomes invalid and its IALG, processing or control functions cannot be called.

DSKT2 maintains allocation information about all algorithm instances created by DSKT2_createAlg or DSKT2_createAlg2, so it does not call the instance algFree method before freeing instance memory.

2.3.8 DSKT2_controlAlg

Int DSKT2_controlAlg(IALG_Handle alg, IALG_Cmd cmd, IALG_Status *status);

Implementation DSKT2_controlAlg function is a convenience API to call a xDAIS algorithm instance's algControl function.

DSKT2_ controlAlg must be called only when the instance is in active state.

Parameters	IALG_Handle alg	IALG_Handle of the algorithm instance to be deactivated.
	IALG_Cmd cmd	IALG_Cmd structure for the control operation.
	IALG_Status *status	Pointer to IALG_Status structure for algorithm to return the status.
Return	IALG_EOK	The control operation was successful.

IALG_EFAIL	A failure occurred during the control operation or algorithm-specific return value

Preconditions

The following conditions must be true prior to calling this method; otherwise, its operation is undefined:

- ❏ Algorithm specific cmd values are always less than IALG_SYSCMD
- ❏ alg must be a valid handle for the algorithm's instance object returned by an earlier call to DSKT2_createAlg or DSKT2_createAlg2 and instance must be in "active" state.

Postconditions

The following condition is true immediately after returning from this method:

- ❏ If the control operation is successful, the return value from this operation is equal to IALG_EOK; otherwise it is equal to either IALG_EFAIL or an algorithm-specific return value
- ❏ If the cmd value is not recognized, the return value is not equal to IALG_EOK.

Comments

The implementation of the IALG algControl is optional by xDAIS standard. DSKT2 makes proper checks to ensure correct operation even if the algorithm does not implement algControl.

Index

A

activation 1-11, 1-15, 2-9
algActivate function 1-11
  avoiding unneeded calls 1-15, 2-11
algAlloc function 1-16
algDeactivate function 1-11
  avoiding unneeded calls 1-15, 2-11
algInit function 1-11, 1-16
ALLOW_EXTERNAL_SCRATCH property 1-5
  example 1-13
APIs 2-3

B

BCACHE_wbInv function 1-6, 1-9

C

C source file configuration 1-9
callback functions 1-15
calling sequence 1-3
CFG file 1-5
configuration 1-4
context switch by callback functions 1-15
controlling algorithm instance 2-13
createHeap property 1-4
creating algorithm instance 2-4, 2-6, 2-8

D

DARAM mapping 1-5
DARAM_SCRATCH_SIZES property 1-6, 1-9
  example 1-13
deactivation 1-11, 1-15, 2-10, 2-12
debug library 1-5, 1-14
debug property 1-5
DSKT2
  APIs 2-3
  configuration 1-4
  primary purpose 1-2
DSKT2_activateAlg function 2-3, 2-9
  example 1-3
  optimization 1-15
DSKT2_CacheWBInvFxn 1-6, 1-9
DSKT2_controlAlg function 2-3, 2-13
DSKT2_createAlg function 2-3, 2-4
  configuration for 1-5
  example 1-3
DSKT2_createAlg2 function 2-3, 2-6
DSKT2_createAlgExt function 2-3, 2-8
DSKT2_deactivateAlg function 2-3, 2-10
  example 1-3
DSKT2_deactivateAll function 2-3, 2-12
DSKT2_freeAlg function 1-14, 2-3, 2-12
  example 1-3
DSKT2_HEAP property 1-5, 1-11, 2-2
DSP/BIOS configuration 1-4

E

enableHeapLabel property 1-4
EPROG mapping 1-5
ESDATA mapping 1-5
external heap 2-8
extHeapId parameter 2-7

F

footprint size 1-2
freeing algorithm instance 2-12
function signature 1-6, 1-9

G

group ID 1-12

H

hardware dependencies 1-14
heap segment 1-2, 1-4, 1-5
heapLabel property 1-4

I

IALG memory space types 1-5
IALG_DARAM request 1-13, 1-14
IALG_ESDATA heap 2-8
IALG_Handle
  getting 2-4
  using 2-9
IALG_MemSpace structure 1-16
IALG_SARAM request 1-14
initialization
  of DSKT2 module 2-2
  of scratch buffers 1-11
insufficient memory 1-5, 1-13
invalidation function 1-6, 1-9
IPROG mapping 1-5

L

lazy deactivation 2-12
library for DSKT2 1-5
linker command file 1-4, 1-9
linking DSKT2 library 1-5
LOG_printf messages 1-14

M

MEM module 1-4, 1-14
memory allocation 1-13

O

optimization 1-15

P

persistent memory 1-11
preemption 1-12, 1-15
priority of tasks 1-12

R

release version 1-5
RTSC configuration 1-4, 1-8

S

SARAM mapping 1-5
SARAM_SCRATCH_SIZES property 1-6, 1-10
scratch groups 1-6, 1-12
scratch ID 1-13
scratch memory 1-2, 1-11
  configuration 1-5
scratch sizes 1-6
scratchMutexId parameter 2-4, 2-6
state of algorithm 1-11

T

task priority 1-12
TCF file 1-7
Tconf configuration 1-5, 1-7
trace messages 1-14
TSK module 1-14

U

useModule method 1-4

W

writeback function 1-6, 1-9

X

xDAIS library 1-2
xDAIS standard 1-11
XDC tooling 1-4, 1-8