Application Report
SPRA732 - March 2001

# Real-Time Analysis in an eXpressDSP-Compliant Algorithm

*Randy Wu* *Americas 3P DSP Software Technical Staff*

**ABSTRACT**

In addition to the IALG interface that is required for an algorithm to conform to the TMS320™ DSP Algorithm Standard, an additional interface can be implemented to enable consumers to take advantage of the Real Time Analysis (RTA) capabilities of DSP/BIOS libraries. The Real-time Trace Control Interface (IRTC), defines an interface, that when implemented, allows a module's various RTA modes to be enabled, disabled, and controlled in real time by the consumer of the algorithm.

An eXpressDSP™-compliant algorithm which is DSP/BIOS enabled will allow the consumer of the algorithm to retrieve data for debugging purposes in real-time. This allows the system integrator of a specific application to easily troubleshoot problems incurred after integrating an eXpressDSP-compliant algorithm into the system. By knowing what to expect from the algorithm in terms of real-time debug data, the system integrator can easily analyze whether the algorithm is working as expected. In the event that the algorithm is not working as expected, the system integrator will be able to gain insight into what is happening inside the black box. The consumer will also be able to benchmark certain portions of the algorithm which make calls to APIs of the DSP/BIOS libraries that can measure performance of the code, such as number of instruction cycles and execution time.

This application note describes the IRTC interface from two perspectives. The Producer (algorithm developer) perspective deals with the methods of embedding DSP/BIOS APIs in the algorithm and implementing the IRTC interface. The Consumer (user) perspective deals with how the IRTC interface is used in a typical DSP application (framework).

**Contents**

**List of Figures**



# 1 Introduction

DSP/BIOS is TI's fully scalable and extensible real-time multitasking kernel, which includes APIs for non-intrusive real-time event and statistics gathering. DSP/BIOS gives developers of applications for DSP chips the ability to develop and analyze embedded real-time software. DSP/BIOS includes a small firmware real-time library, the DSP/BIOS API for using real-time library services, and easy-to-use tools for configuration and for real-time tracing and analysis.

The TMS320 DSP Algorithm Standard Rules and Guidelines allow the algorithm developer to utilize the real-time analysis modules of DSP/BIOS. These modules consist of the LOG and STS APIs. It is important to realize that none of the LOG and STS operations has any semantic effect on the execution of an algorithm. In other words, it is possible to implement all of these operations as aliases to a function that simply returns and the operation of the algorithm will be unaffected. DSP/BIOS calls, in general, are non-intrusive and incur virtually no overhead.

This application note focuses on how to implement the IRTC interface which will allow the consumer of an eXpressDSP-compliant algorithm to enable, disable, and control the DSP/BIOS APIs that have been embedded in the algorithm by the developer. In the event that the consumer needs to benchmark or troubleshoot the algorithm, the developer can embed DSP/BIOS calls into the algorithm so that the consumer can get accurate real-time analysis while introducing virtually no extra overhead into the system.

For further details on all the DSP/BIOS APIs, please refer to the *TMS320C5000 User's Guide* (SPRU326) or *TMS320C6000 DSP/BIOS User's Guide* (SPRU303).

# 2 Producer: Embedding DSP/BIOS Into the Algorithm for Real-Time Analysis

## 2.1 Message Log Data

The Message Log manager of DSP/BIOS is used to capture events in real time while the target program executes. The consumer of the algorithm can use the system log or create user-defined logs. Here, we use the *LOG_printf()* function of the LOG Module to append a formatted message, similar to the format of the standard *printf()* function, to a continuous message log.

```
#include <clk.h>
#include <log.h>
extern LOG_Obj trace;
LOG_printf(&trace, “Started execution of block #1 at time %d…”,(Arg)CLK_getltime());
. . . <“algorithm code for block #1”> . . .
LOG_printf(&trace, “Finished execution of block #1 at time %d…”,(Arg)CLK_getltime());
```

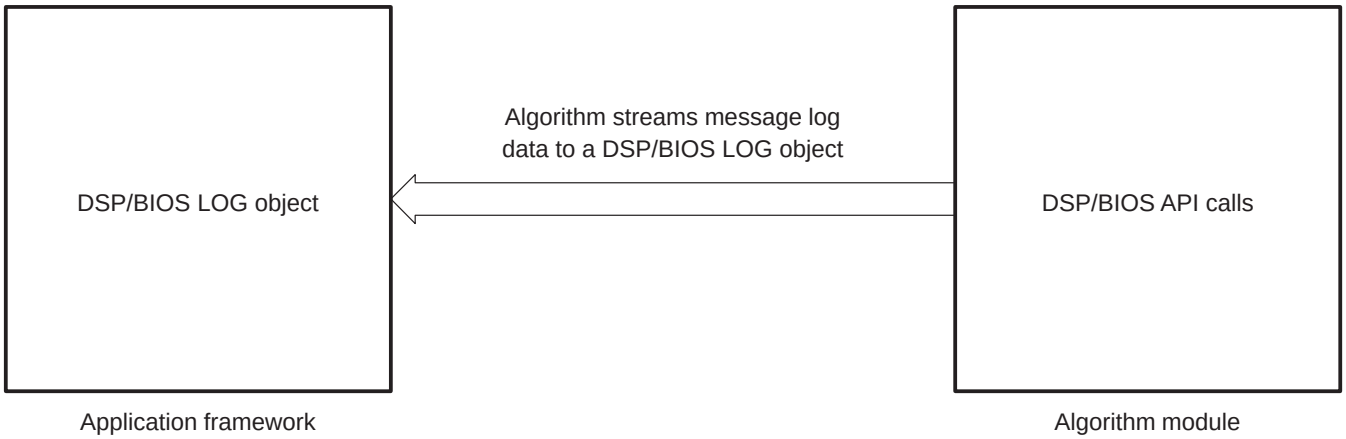


**Figure 1. DSP/BIOS LOG Message Data**

## 2.2 Statistics Accumulator Data

The STS module of DSP/BIOS manages objects called statistics accumulators. Each STS object accumulates the following statistical information about an arbitrary 32-bit-wide data series:

- Count — the number of values in an application-supplied data series
- Total — the sum of the individual data values in this series
- Maximum — the largest value already encountered in this series

For example, portions of the algorithm can be benchmarked by using paired calls to *STS_set()* and *STS_delta()* that pass the value provided by *CLK_gethtime():*

```
#include <clk.h>
#include <sts.h>
extern STS_Obj sts;
STS_reset(&sts);
STS_set(%sts, CLK_gethtime());
. . . <”processing to be benchmarked”> . . .
STS_delta(%sts, CLK_gethtime());
```

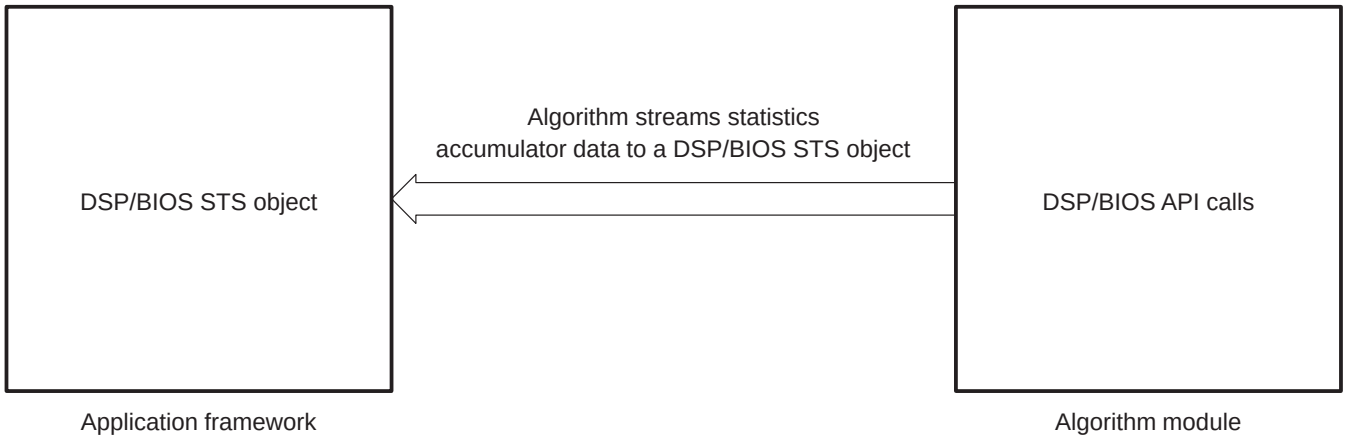


**Figure 2. DSP/BIOS STS Accumulator Data**

In essence, the DSP/BIOS API calls can be inserted anywhere in the algorithm code where it is useful to obtain debug information during program execution.

For further details on all the DSP/BIOS APIs, please refer to the *TMS320C5000 DSP/BIOS User's Guide* (SPRU326) or *TMS320C6000 DSP/BIOS User's Guide* (SPRU303)

# 3 Producer: Implementing the Real-time Trace Control Interface (IRTC)

The real-time trace control interface (IRTC) defines an interface, that when implemented, allows a module's various trace modes to be enabled, disabled, and controlled in real-time. This interface is a standard API supplied as anTMS320 DSP Algorithm Standard system header file, but the actual levels of debug and algorithm-specific IRTC functions must be implemented by the algorithm developer. This interface does not supersede the IALG interface; it exists as an additional interface of the eXpressDSP-compliant algorithm.

The following code sample is the interface definition for the standard IRTC interface as defined by Texas Instruments:

```
/* ======== irtc.h ======== */
/* System header file as defined in the XDAIS Developers Kit directory */
/*--------------------------*/
/*    TYPES AND CONSTANTS    */
/*--------------------------*/
#define IRTC_ENTER      0     /* Enable ALL Levels of Real Time Analysis */
#define IRTC_CLASS1     1     /* Used for (optional) different levels of debug */
#define IRTC_CLASS2     2     /* as defined by the algorithm developer */
#define IRTC_CLASS3     3
#define IRTC_CLASS4     4
#define IRTC_CLASS5     5
#define IRTC_CLASS6     6
#define IRTC_CLASS7     7
```

```
/*
 *  ======== IRTC_Handle ========
 *  Handle to a testable instance object
 */
typedef struct IALG_Obj *IRTC_Handle;
```

```
/*
 *  ======== IRTC_Mask ========
 */
typedef LgUns IRTC_Mask;
```

```
/*
 *  ======== IRTC_Fxns ========
 */
typedef struct IRTC_Fxns {
    Void        *implementationId;
    Void        (*rtcBind)(LOG_Obj *log, STS_Obj *sts);
    IRTC_Mask   (*rtcGet)(IRTC_Handle);
    Void        (*rtcSet)(IRTC_Handle, IRTC_Mask mask);
} IRTC_Fxns;
```

## 3.1 *IRTC_Fxns* V-table

A module implements the IRTC interface if it defines and initializes a global structure of type *IRTC_Fxns*. For the most part, this means that every function defined in this structure must be implemented (and assigned to the appropriate field in this structure). It is important to note that the first field of the *IRTC_Fxns* structure is a Void* pointer. This field must be initialized to a value that uniquely identifies the module implementation. This same value must be used in all interfaces implemented by the module. Since all eXpressDSP-compliant algorithms must implement the IALG interface, it is sufficient for eXpressDSP-compliant algorithm modules to set this field to the address of the module‘s declared *IALG_Fxns* structure.

```
/*
 *  ======== FIR_TI_IRTC ========
 *  This structure defines the IRTC interface for the FIR_TI module.
 */
IRTC_Fxns FIR_TI_IRTC = {
    &FIR_TI_IALG,       /* module ID */
    FIR_TI_rtcBind,     /* rtcBind */
    FIR_TI_rtcGet,      /* rtcGet */
    FIR_TI_rtcSet       /* rtcSet */
};
```

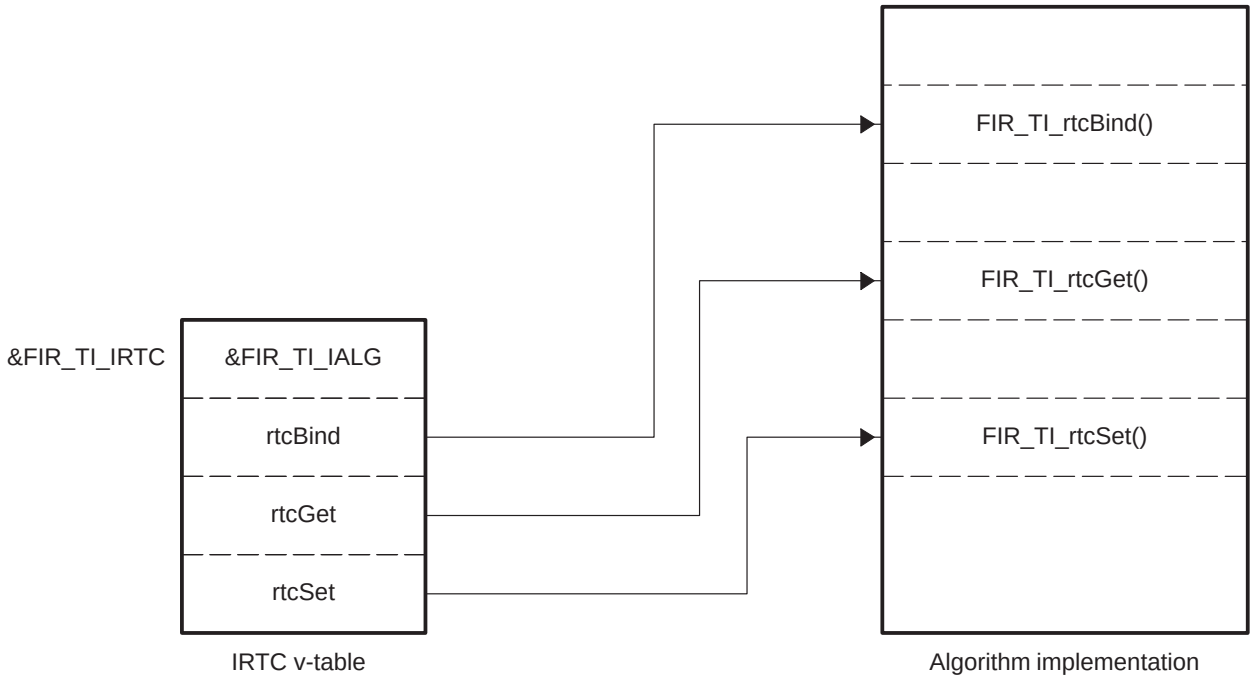


**Figure 3. *IRTC_Fxns* V-table**

## 3.2 Adding the *biosMask* Field to the Algorithm Object Definition

The algorithm object definition must include a field of type *IRTC_Mask* to store the current trace mask settings for each instance. For example, an additional structure field called *biosMask* can be added to the existing object definition as another piece of context information.

```
#include <irtc.h>
typedef struct FIR_TI_Obj {
     IALG_Obj alg; /* IALG object MUST be the first field */
     Int *workBuf; /* pointer to on–chip scratch memory */
     . . . ;
     IRTC_Mask biosMask; /* current trace mask */
} FIR_TI_Obj;
```

## 3.3 *IRTC_Mask* Settings

Specific levels of debug functionality can be defined as constants to be used with the *IRTC_Mask* data type defined in the IRTC API header file. This data type will be defined as one of the fields in the vendor-specific IALG object as part of the state information for each algorithm instance. This mask setting can be read/written during the lifetime of the algorithm instance. The algorithm developer is free to use the existing levels of debug, as defined in the IRTC API, or create additional levels of debug as needed.

```
#include <irtc.h>
if (firHandle–>biosMask == IRTC_CLASS4)
{
     /* Execute the following if current mask setting is set to Class 4 */
     LOG_printf(&trace, “IRTC Class 4 Debug Information”);
     . . . “additional DSP/BIOS API calls for real–time analysis” . . .
}
```

## 3.4 *rtcBind()*

The *rtcBind()* function must be implemented to set the module's current output log and/or statistics accumulator. For any module that implements the IRTC interface, there is just one output log and statistics accumulator. All instances of a single module use a single output log and/or statistics accumulator. The first argument to *rtcBind()* is a pointer to a LOG object, which must be defined by the consumer of the algorithm. The second argument is a pointer to an STS object.

```
/* This module’s output trace log */
LOG_Obj *FIR_TI_rtcLog = NULL;
/* This module’s statistics accumulator */
STS_Obj *FIR_TI_rtcSts = NULL;
Void FIR_TI_rtcBind(LOG_Obj *log, STS_Obj *sts)
{
     /* set current output log and statistics accumulator pointers */
     FIR_TI_rtcLog = log;
     FIR_TI_rtcSts = sts;
}
```

The pointer to the LOG and STS objects must be defined by the application prior to calling *rtcBind().* They must be valid pointers to objects managed by the LOG and STS modules. No other module method must currently be run on any instance; i.e. this method never preempts any other method or any instance managed by the implementing module. All subsequent output trace is redirected to the message log referenced by the log pointer. This function basically sets the module's internal *LOG_Obj* pointer to point to the same LOG object, which was pre-defined by the application. The same scenario applies to the STS object as well. If there is no LOG or STS object, the calling application should pass in NULL for those parameters.

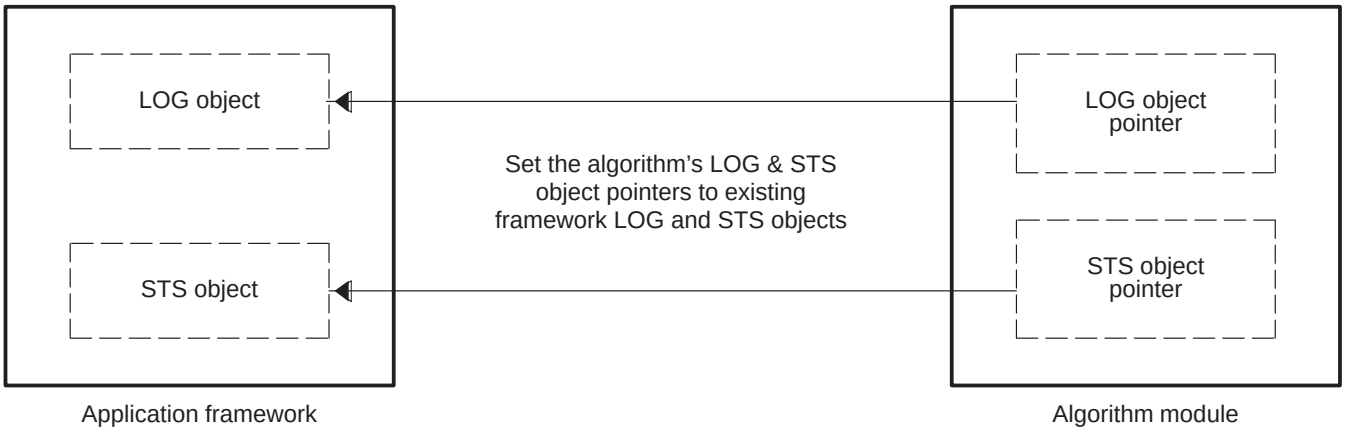


**Figure 4. Module Object Pointers**

## 3.5 *rtcGet()*

The *rtcGet()* function must be implemented to get the current trace mask setting. For any module that implements the IRTC interface, the module instance object is also the trace instance object. The first (and only) argument to *rtcGet()* is a trace instance handle. The *IRTC_Handle* must be a valid handle for the module's trace instance object.

```
IRTC_Mask FIR_TI_rtcGet(IRTC_Handle handle)
{
     FIR_TI_Obj *inst = (FIR_TI_Obj *)handle;
     /* return current trace mask */
     return (inst–>biosMask);
}
```

## 3.6 *rtcSet()*

The *rtcSet()* function must be implemented to set a new trace mask for a trace instance object. For any module that implements the IRTC interface, the module instance object is also the trace instance object, since the instance object contains the mask setting as part of its state information. The first argument to *rtcSet()* is a trace instance handle and the second argument is the new setting for this instance. The *IRTC_Handle* must be a valid handle for the module's trace instance object.

```
Void FIR_TI_rtcSet(IRTC_Handle handle, IRTC_Mask mask)
{
     FIR_TI_Obj *inst = (FIR_TI_Obj *)handle;
     /* set current trace mask */
     inst–>biosMask = mask;
     . . . ; /* enable trace modes indicated by the mask */
}
```

After returning from this method, the trace levels specified in mask are enabled for this instance. This instance emits the specified trace information during its execution. Although the execution time for the module's methods may change slightly, all of the module's methods continue to execute as they would have if *rtcSet()* had not been called.

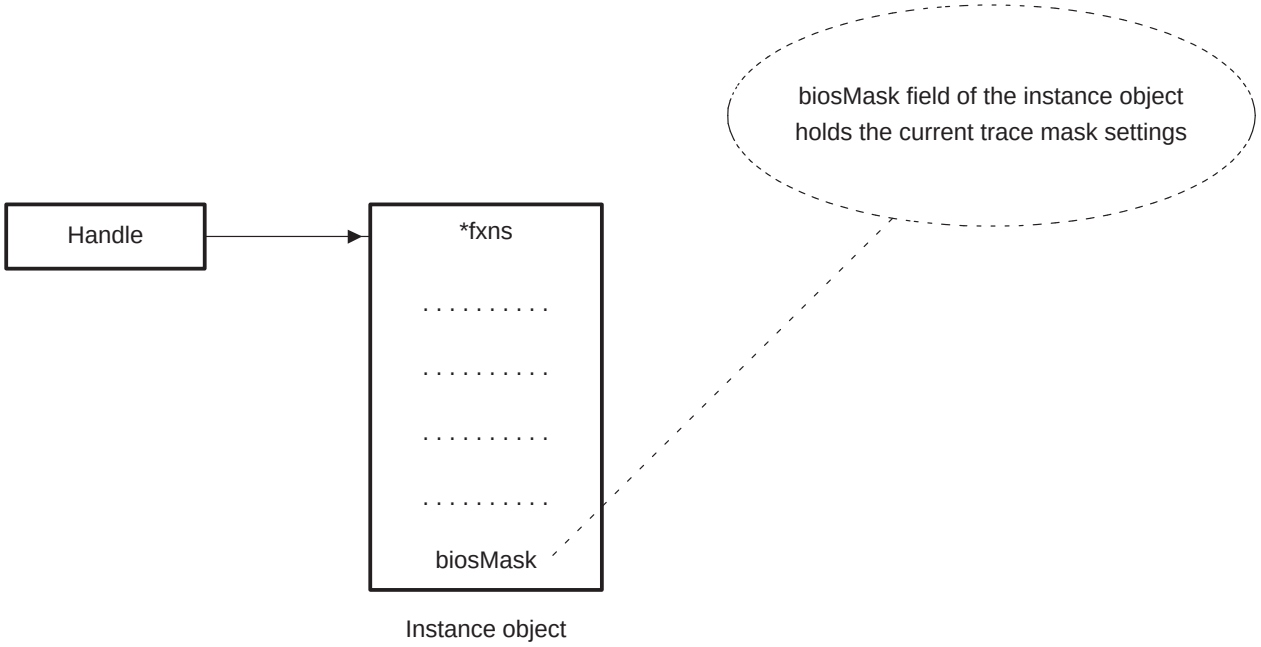


**Figure 5. Setting and Retrieving Current Mask Settings**

## 3.7 What if the Consumer Is Not Using DSP/BIOS?

An algorithm cannot assume that the application will always incorporate DSP/BIOS and take advantage of the Real-Time Analysis information generated by the algorithm. As such, all DSP/BIOS *LOG_printf()* calls embedded in the algorithm should check to make sure there exists a pointer to a valid *LOG_Obj*. All STS API calls should check for the existence of a valid pointer of type *STS_Obj* as well. For example, this can be achieved by defining a macro called *FIR_TI_trace*.

```
/*
 *  ======== FIR_TI_trace ========
 *  Our equivalent of ”printf”
 */
extern LOG_Obj *FIR_TI_rtcOut;
#define FIR_TI_trace(f, a1, a2)                         \
    if (FIR_TI_rtcOut != NULL) {                        \
        LOG_printf(FIR_TI_rtcOut, (f), (a1), (a2));     \
    }
```

If the consumer/framework is not interested in using the real-time analysis feature(s) of the algorithm, it can simply pass a NULL value for the *LOG_Obj* parameter. Since the *FIR_TI_trace* macro checks for a valid *LOG_Obj* pointer before it calls *LOG_printf(),* the algorithm should run without crashing.

# 4 Consumer: Using the Real-Time Trace Control Interface (IRTC)

## 4.1 Creating the RTC Interface

The consumer can write an RTC interface to include concrete APIs which are extensions to the IRTC Interface. An additional RTC Descriptor, a structure of type *RTC_Desc*, can be defined to store the pointer to the *IRTC_Fxns* v-table, pointer to the instance object, and the current mask settings all in one convenient location.

The RTC module defined in this section is a Framework/Consumer Level convenience layer. The RTC module is generic and applicable to ALL algorithms that implement the IRTC interface. This layer helps the system integrator/consumer to easily create and configure IRTC-enabled algorithm instance objects.

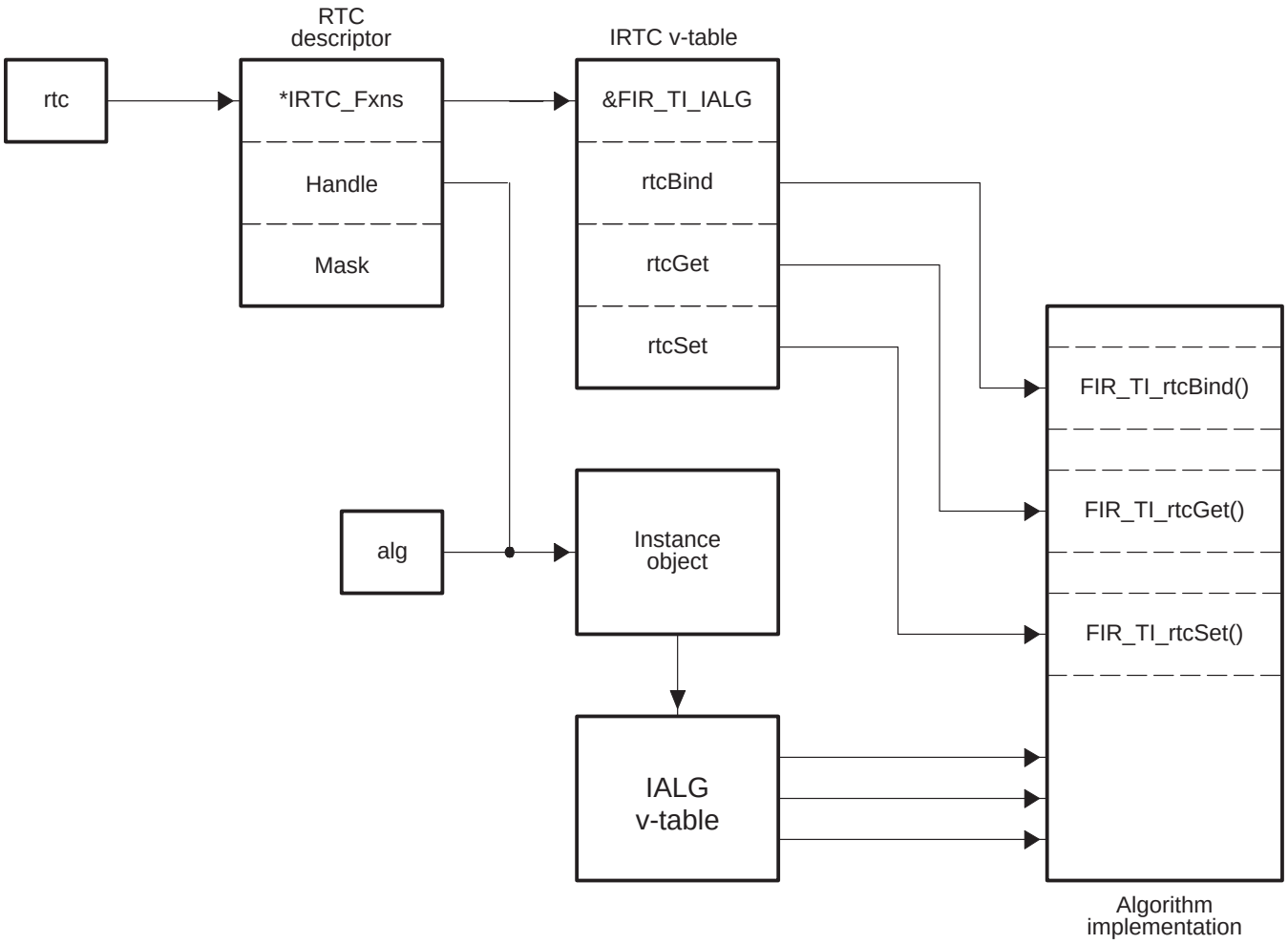


**Figure 6. RTC Descriptor**

```
#define RTC_ENTER       (1 << IRTC_ENTER)
#define RTC_WARNING     (1 << IRTC_CLASS1)
typedef struct RTC_Desc {
    IRTC_Fxns   *fxns;      /* test functions */
    IRTC_Handle handle;     /* test object */
    IRTC_Mask   mask;       /* current mask setting */
} RTC_Desc;
```

```
IRTC_Mask RTC_get(RTC_Desc *rtc)
{
    if (rtc && rtc–>fxns)
        return(rtc–>fxns–>rtcGet(rtc–>handle));
    else
        return(NULL);
}
```

```
Void RTC_set(RTC_Desc *rtc, IRTC_Mask mask)
{
    if (rtc && rtc–>fxns) {
        rtc–>fxns–>rtcSet(rtc–>handle, (IRTC_Mask)mask);
        rtc–>mask = mask;
    }
}
```

```
Void RTC_init(Void)
{
}
```

```
Void RTC_exit(Void)
{
}
```

```
Void RTC_bind(IRTC_Fxns *fxns, LOG_Obj *log, STS_Obj *sts)
{
    if (fxns != NULL) {
        fxns–>rtcBind(log, sts);
    }
}
```

```
Void RTC_delete(RTC_Desc *rtc)
{
    if (rtc != NULL)
        free(rtc);
}
```

```
RTC_Desc *RTC_create(RTC_Desc *obj, ALG_Handle alg, IRTC_Fxns *fxns)
{
    if (obj != NULL) {
        /* verify that alg and rtc fxns are from same concrete implementation */
        if (fxns->implementationId == alg->fxns->implementationId) {
            obj->handle = (IRTC_Handle)alg;
            obj->fxns = fxns;
            return (obj);
        }
        obj->fxns = NULL;
    }
    return (NULL);
}
```

```
Void RTC_disable(const RTC_Desc *rtc)
{
    if (rtc && rtc->fxns)
        rtc->fxns->rtcSet(rtc->handle, (IRTC_Mask)0);
}
```

```
Void RTC_enable(const RTC_Desc *rtc)
{
    if (rtc && rtc->fxns)
        rtc->fxns->rtcSet(rtc->handle, rtc->mask);
}
```

## 4.2 Using the RTC Interface

The following code sample shows a framework and typical usage of the RTC interface in a DSP application:

```
#include <std.h>
#include <fir.h>
#include <alg.h>
#include <log.h>
#include <ialg.h>
#include <rtc.h>
#include <fir_ti.h>
```

```
#define FRAMELEN    (sizeof (input) / sizeof (Int))
#define FILTERLEN   (sizeof (coeff) / sizeof (Int))
extern LOG_Obj trace;
extern STS_Obj stat;
Int coeff[] = {1, 2, 3, 4, 4, 3, 2, 1};
Int input[] = {1, 0, 0, 0, 0, 0, 0};
Int output[FRAMELEN];
```

```
Int main(Int argc, String argv[])
{
    FIR_Params firParams = FIR_PARAMS;
    ALG_Handle alg;
    RTC_Desc *rtc;

    ALG_init();
    FIR_init();
    RTC_init();

    /* set the module's LOG and STS Object pointers to existing LOG/STS Objects */
    RTC_bind(&FIR_TI_IRTC, &trace, &stat);

    /* create an instance of a FIR algorithm */
    firParams.filterLen = FILTERLEN;
    firParams.frameLen = FRAMELEN;
    firParams.coeffPtr = coeff;
    alg = ALG_create((IALG_Fxns *)&FIR_TI_IFIR, NULL, (IALG_Params *)&firParams);
    /* if the instance creation succeeded, create a trace descriptor */
    rtc = (RTC_Desc *)malloc(sizeof(RTC_Desc));
    if (alg != NULL && RTC_create(rtc, alg, &FIR_TI_IRTC) != NULL) {
        RTC_set(rtc, RTC_ENTER);                    /* enable trace */
        FIR_apply((FIR_Handle)alg, input, output);  /* filter data */
        RTC_delete(rtc);       /* delete rtc descriptor */
        ALG_delete(alg);        /* delete alg instance */
    }
    RTC_exit();
    FIR_exit();
    ALG_exit();
}
```

- *RTC_init()* is called during system startup to perform any run-time initialization necessary for the RTC module as a whole.
- *RTC_bind()* sets the output log and statistics accumulator of a module. This operation is typically called during system initialization, and it must not preempt any other operation supported by the implementing module. The second and third arguments must be valid pointers to pre-existing, externally defined LOG and STS objects by the consumer of the algorithm. If either of the LOG or STS objects is not to be used, the calling application should pass in NULL for the corresponding parameter.
- *RTC_create()* initializes the trace descriptor structure. This structure is initialized using the algorithm object and a pointer to a module's IRTC implementation functions. The algorithm object structure pointer must be a pointer to a previously created algorithm object (e.g., via *ALG_create()*).

- *RTC_set()* sets the current trace mask for an algorithm instance specified by the descriptor pointer rtc. The first argument to *RTC_set()* is a trace descriptor initialized via *RTC_create()*. The second argument is a mask representing the level of trace and diagnostic information that should be produced in real time by the trace object. Of course, the specific levels of trace will be defined by the algorithm developer.
- *RTC_delete()* deletes the trace descriptor, rtc, which was initialized by *RTC_create()*.
- *RTC_exit()* runs during system shutdown to perform any run-time finalization necessary for the RTC module as a whole.

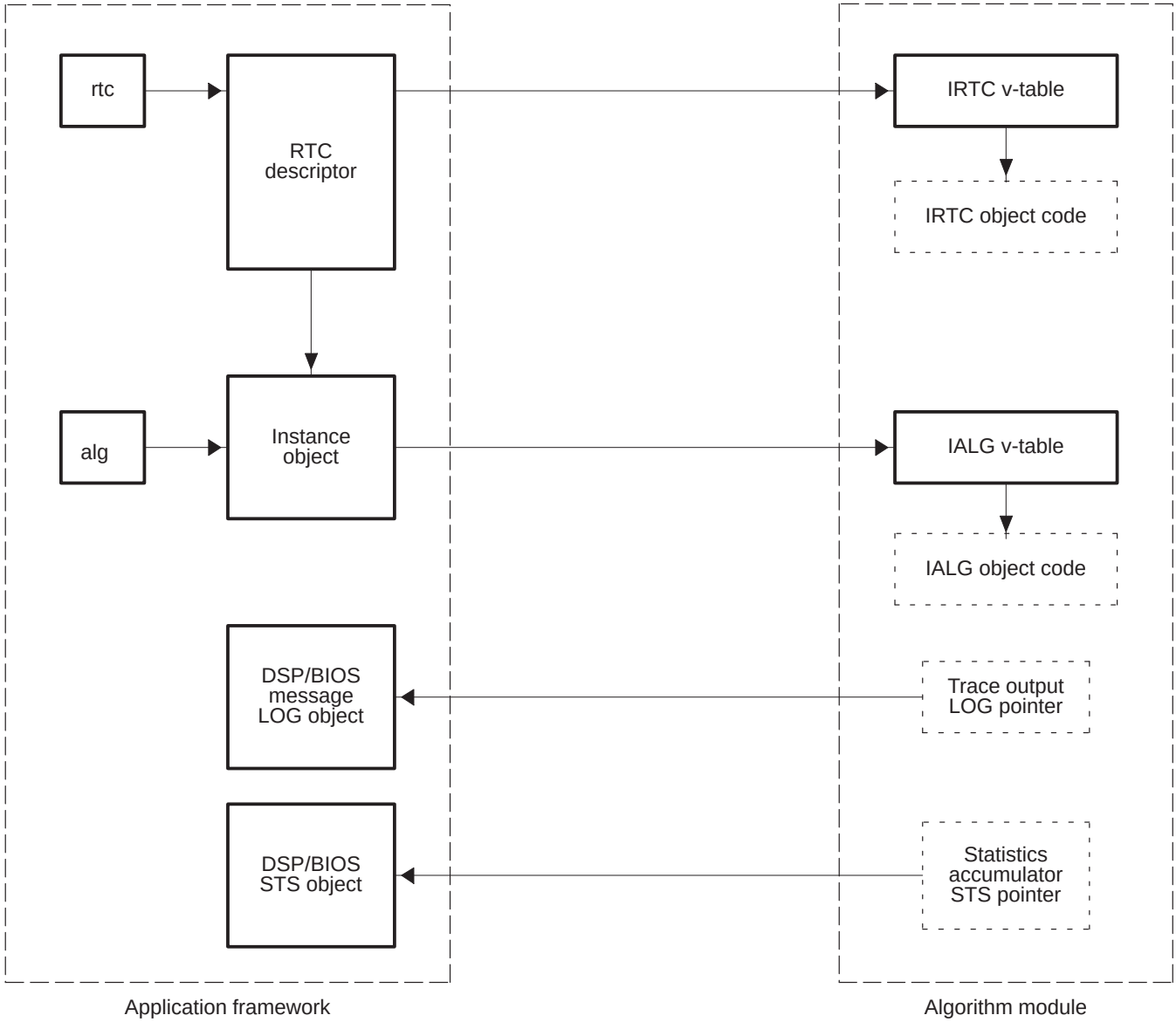


**Figure 7. Algorithm Instance and RTA Object Configuration Overview**

# 5 Conclusion

It is recommended that all eXpressDSP-compliant algorithms take advantage of the capabilities of TI's DSP/BIOS libraries by incorporating the Real-Time Analysis APIs into the algorithm. The addition of DSP/BIOS calls in an algorithm incurs virtually no extra overhead, and enables the consumer of the algorithm to obtain real-time analysis data without halting the target processor. The IRTC interface is easily created by defining the *IRTC_Fxns* v-table, *IRTC_Mask* values, and implementing the three required methods, *rtcBind(), rtcGet(),* and *rtcSet()*. A DSP system integrator could then easily write a generic RTC framework to use the IRTC interface of an eXpressDSP-compliant algorithm.

# 6 References

1. *TMS320 DSP Algorithm Standard Rules and Guidelines* (SPRU352).
2. *TMS320 DSP Algorithm Standard API Reference* (SPRU360).
3. *TMS320C5000 DSP/BIOS User's Guide* (SPRU326).
4. *TMS320C6000 DSP/BIOS User's Guide* (SPRU303).

## IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

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

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

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. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's 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, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products. www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265