Programming the TMS320VC5509 RTC Peripheral

Scott Tater *DSP Applications – Semiconductor Group*

ABSTRACT

This application report demonstrates the procedure used to program the TMS320VC5509 digital signal processor (DSP) Real Time Clock (RTC). Basic operations of the RTC, such as reading and writing the time, as well as interrupt configuration, are covered. All interaction with the peripheral is accomplished using the Chip Support Library (CSL) RTC module. An example that illustrates usage of these routines is also presented.

Contents

List of Figures

List of Tables

1 Introduction

The RTC module maintains time and date information independent of DSP operation. The RTC achieves independent operation by using a separate external clock and power source from the DSP. Figure 1 shows the RTC block diagram.

The RTC tracks time in the following formats:

- Seconds (0–59)
- Minutes (0–59)
- Hours (0–23)
- Days of the week (1–7)
- Days of the month (1–31)
- Months (1–12)
- Years with leap year correction (0–99)

The RTC provides three separate interrupt sources: periodic, alarm, and update. The periodic interrupt has a programmable period from 122 μs to 1 minute. The alarm interrupt can be set to a flexible combination of the second, minute, hour, and day of the week. The update interrupt is associated with RTC internal operation and, while the interrupt originates under internal RTC control, the user may enable or disable transmission to the central processing unit (CPU) core.

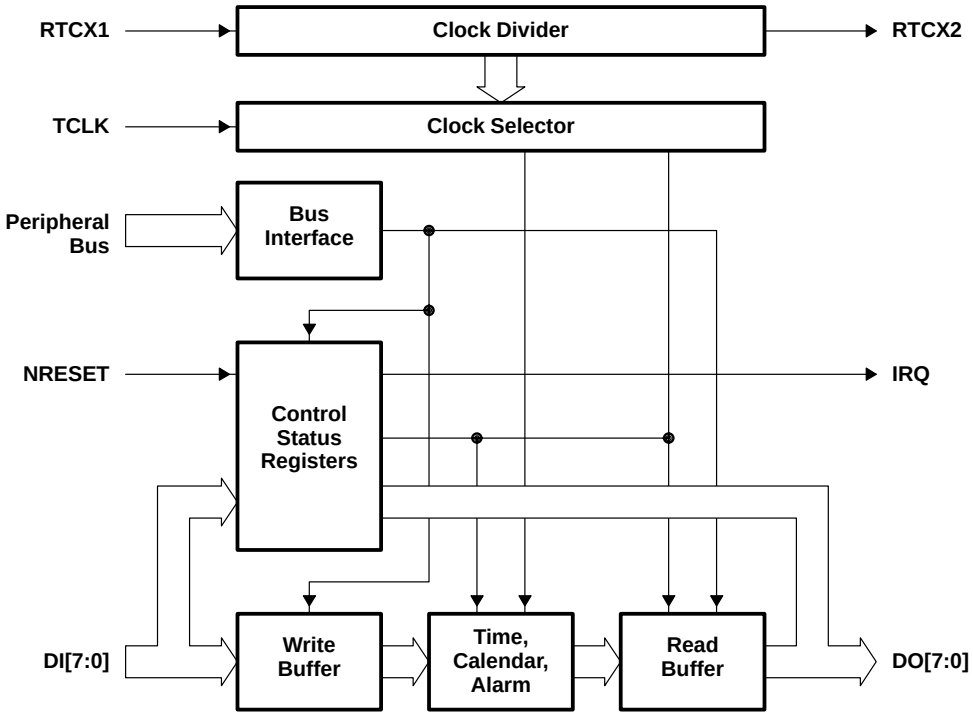

Figure 1. RTC Block Diagram

For detailed information on the RTC, including hardware implementation and operation, please refer to the RTC chapter of the *TMS320C55x DSP Peripherals Reference Guide* (SPRU317).

2 Basic Operation

Program interaction with the RTC typically involves one of the following three types of operations:

- Initializing the RTC time and alarm information using the DSP/BIOS™ Configuration tool
- Reading and writing the RTC using embedded CSL function calls
- Configuring RTC Interrupts with either embedded CSL function calls or with the DSP/BIOS Configuration tool.

The DSP/BIOS Configuration tool is used to perform the initialization of the RTC at start-up, the CSL code interface is used during run time to interact with the RTC as needed, and then the CSL or DSP/BIOS is used to implement the interrupt capabilities of the device, if desired.

2.1 Initializing the RTC Time and Alarm With the DSP/BIOS Configuration Tool

The DSP/BIOS Configuration tool provides a graphical interface to initialize the RTC at code start-up. Figure 2 shows a typical view of the Configuration tool with the RTC module expanded.

The RTC module contains two parts: the Configuration Manager and the Resource Manager. The Configuration Manager allows the user to create one or more configuration objects. Each object contains all the information needed to initialize the RTC control registers. The Resource Manager associates the RTC hardware with a particular configuration object. At run time, DSP/BIOS will automatically load the information stored in the configuration object into the RTC.

DSP/BIOS is a trademark of Texas Instruments.

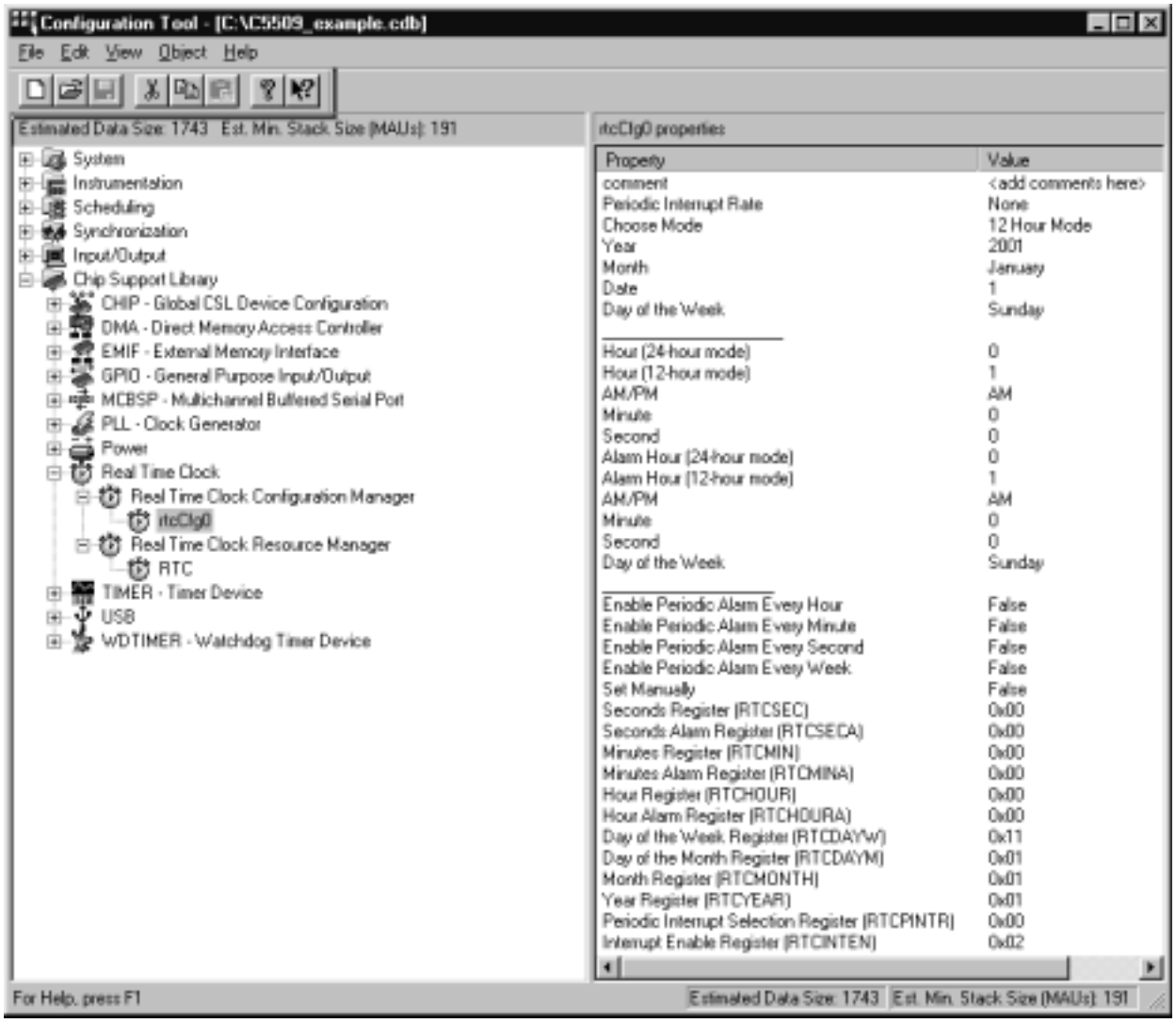

Figure 2. DSP/BIOS Configuration Tool With RTC Module

For more detailed information on using the DSP/BIOS Configuration tool, see the *TMS320 DSP/BIOS User's Guide* (SPRU423).

2.2 Reading and Writing the RTC Using Embedded Chip Support Library Function Calls

The CSL is a collection of C-callable functions, macros, and symbols used to control and access on-chip peripherals that share a standard interface. The CSL speeds development time by reducing code reinvention, and its standardized interface reduces programming errors and facilitates simple debugging. It also provides a level of hardware abstraction to facilitate porting software between different Texas Instruments DSPs.

In addition to the CSL style interface, the RTC module includes support for the ANSI C style time interface. This interface is similar to that provided in the standard include file time.h. A complete description of the RTC time.h implementation is given in section 2.2.1.

CSL support for the RTC consists of two types of objects: RTC specific data structures and support functions. The data structures are pre-defined C structures designed to store RTC register information. The support functions automate interaction with the RTC and utilize the CSL standard interface. Table 1 shows the data structures.

Table 1. RTC Data Structures

RTC Data Structure Name	...is used to
RTC_Alarm	Store information for setting the alarm time
RTC_Data	Store information for setting the date
RTC_Time	Store information for setting the time

All data is passed to the RTC in binary coded decimal form (BCD). Figure 3 shows an example of code used to declare an RTC_Time structure and store the time in BCD format. Notice that CSL supports 24-hour time format only.

```
RTC_Time myTime = {
     0x21,     // Hour (Note 24 hour time notation)
     0x58,     // Minutes
     0x30,     // Seconds
};
```

Figure 3. Data Structure Example

The RTC CSL module includes two functions used to facilitate conversion between binary and BCD forms. These functions will reduce translation errors and speed development. Table 2 lists these two utility functions, and Figure 4 shows an example of their use. Table 3 provides a BCD to decimal translation example.

Table 2. RTC Utility Functions

RTC Utility Function Name	...is used to
RTC_bcdToDec()	Convert BCD format to decimal
RTC_decToBcd()	Convert decimal to BCD format

```
minute = RTC_bcdToDec(currentTime.minute);  //Convert value from native RTC BCD
second = RTC_bcdToDec(currentTime.second);  //format to decimal

currentTime.minute = RTC_decToBcd(minute);  //Convert value from decimal to
currentTime.second = RTC_decToBcd(second);  //native RTC BCD format
```

Figure 4. Binary Coded Decimal Conversion

Table 3. Binary Coded Decimal Example

Decimal	Binary Coded Decimal
0	0x0
1	0x1
2	0x2
3	0x3
…	…
10	0x10
11	0x11
12	0x12
13	0x13
14	0x14

The majority of program interaction with the RTC will make use of a basic set of operations. These operations include performing reads and writes to the time, date, and alarm registers. Table 4 lists each of the support functions used to perform these operations.

Table 4. RTC Basic Calls

RTC Support Function Name	…is used to
RTC_getDate()	Read current date from RTC registers
RTC_getTime()	Read current time from RTC registers
RTC_setAlarm()	Set the alarm to a specific time
RTC_setDate()	Set the RTC calendar
RTC_setTime()	Set the RTC clock
RTC_start()	Start the RTC clock
RTC_stop()	Stop the RTC clock
RTC_reset()	Set the RTC to its initial state

For a complete listing of all the available RTC operations in the CSL, see the *TMS320C55x Chip Support Library API Reference Guide* (SPRU433).

The support functions work with the RTC data structures to access the RTC. Using the example code shown in Figure 3, the example in Figure 5 shows the RTC_setTime() function used to write data to the time registers.

```
// Set Time For Real Time Clock
RTC_setTime(&myTime);
```

Figure 5. RTC Support Function Example

The *TMS320C55x DSP Peripherals Reference Guide* (SPRU317) describes certain access methods to use when working with RTC register values. These methods are designed to ensure that the data transferred to or from the RTC is not corrupted by an update cycle (an update cycle occurs when the RTC changes its stored values to reflect current time). The CSL incorporates these methods and thus **a CSL programmer need only be concerned with the provided interface**, and not specific RTC access methods.

2.2.1 Using ANSI C Style Time Functions

The RTC module provides ANSI C style time functions in addition to the standard CSL style functions. These interfaces will benefit a C programmer familiar with the time.h header who wishes to either maintain continuity with existing C code, or make use of the rich time-related string manipulation provided in the header. Table 5 lists each of the ANSI C style time functions. Notice that each function is identical to its C counterpart, except for the addition of the RTC module prefix.

Table 5. RTC ANSI C Style Calls

RTC ANSI C Support Function Name	…**is used to**
RTC_asctime	Convert a time to an ASCII string
RTC_ctime	Convert calendar time to local time
RTC_difftime	Return the difference between two calendar times
RTC_gmtime	Convert calendar time to GMT
RTC_localtime	Convert calendar time to local time
RTC_mktime	Convert local time to calendar time
RTC_strftime	Format a time into a character string
RTC_time	Return the current RTC time and date

These functions maintain a similar interface, data structure, and functionality to their ANSI C counterparts.

For additional documentation on these functions, please refer to the ANSI C equivalent routines in the *TMS320C55x Optimizing C Compiler User's Guide (SPRU281).* More information about the ANSI C routines can be found in *The C Programming Language* [5].

2.3 Configuring Real Time Clock Interrupts

The Real Time Clock can provide various interrupts to the CPU in addition to keeping time. Three interrupt sources exist in the RTC:

- Alarm Interrupt
- Periodic Interrupt
- Update Cycle Interrupt

Each of the three interrupt sources share the main RTC interrupt line. Any enabled source can trigger the RTC interrupt. The alarm interrupt is based on the alarm time set in the RTC. It can be set to occur with a period of one second to one week. The periodic interrupt does not depend on the set time or alarm, and can occur with periods from one minute to 122 μs. Note that while the alarm interrupt can be set to nearly continuous values because it is based on the time format, the periodic interrupt can be set to one of 16 pre-determined values inside its period range.

The update cycle interrupt occurs once each second as the RTC updates its stored time.

The CSL simplifies the process of interrupt and interrupt service routine (ISR) configuration. Table 6 summarizes the CSL data structures and support functions used with RTC interrupts.

Table 6. CSL Structures and Functions Used With RTC Interrupts

CSL Data Structure or Function Name	…**is used to**
RTC_IsrAddr	Data structure, to hold ISR names
RTC_eventDisable()	Disable an interrupt source on the main RTC interrupt
RTC_eventEnable()	Enable an interrupt source on the main RTC interrupt
RTC_setCallback()	Bind ISR names to the CSL interrupt handler
RTC_setPeriodicInterval()	Set the interval of the periodic interrupt

The CSL function RTC_setCallback() is used to associate an interrupt service routine with each RTC interrupt source (periodic, alarm, and update). The same binding between ISR and RTC interrupt can also be accomplished with the HWI module in DSP/BIOS. Due to the underlying method of implementation of each, it is important to choose one method for binding RTC interrupts to ISRs for the entire project.

Two solutions for interrupt management are presented in sections 2.3.1 and 2.3.2, to allow maximum flexibility for code:

- Using the CSL RTC Dispatcher
- Using the DSP/BIOS HWI module without the CSL RTC Dispatcher

2.3.1 Interrupts Using the CSL RTC Dispatcher

The CSL function RTC_setCallback() supports interrupt service routines when the HWI module is not used. Configuration of the RTC Dispatcher is a two-step process:

1. Create an ISR for each active RTC interrupt source
2. Bind the individual ISRs to the RTC Dispatcher

The RTC data structure RTC_IsrAddr is used to hold the names of each specific ISR, and the function RTC_setCallback() binds the function names in the structure to the interrupt. RTC_setCallback will configure the RTC dispatcher to direct an incoming interrupt to the correct ISR automatically. Figure 6 shows an example using this function to configure RTC ISRs.

Note that the individual ISRs (myPeriodicIsr, etc.) should not be declared with the interrupt keyword when they are used with the RTC dispatcher.

```
// Declare and initialize an RTC interrupt callback function structure

RTC_IsrAddr addr = {
        myPeriodicIsr,              //function name of desired Period Interrupt ISR
        myAlarmIsr,                 //function name of desired Alarm Interrupt ISR
        myUpdateIsr                 //function name of desired Update Interrupt ISR
};

// Bind address of each interrupt service routine for RTC interrupts
RTC_setCallback(&addr);
```

Figure 6. Using RTC_setCallback() to Bind ISR Functions

2.3.2 Interrupts Using DSP/BIOS Module Without the CSL RTC Dispatcher

A program that binds the RTC interrupt using the HWI module should not also use the CSL RTC interrupt configuration objects. Instead, create an ISR and associate it with the RTC hardware interrupt signal in the HWI manager. The DSP/BIOS HWI manager will configure the system to run the specified ISR when the interrupt signal occurs.

Because most systems will only use one of the three RTC sources to trigger an interrupt, it is not recommended to decode the RTC interrupt signal to determine which of the three potential sources was the trigger. Instead, use the HWI manager to call the ISR directly. Figure 7 shows an example of the HWI manager configured to call the ISR.

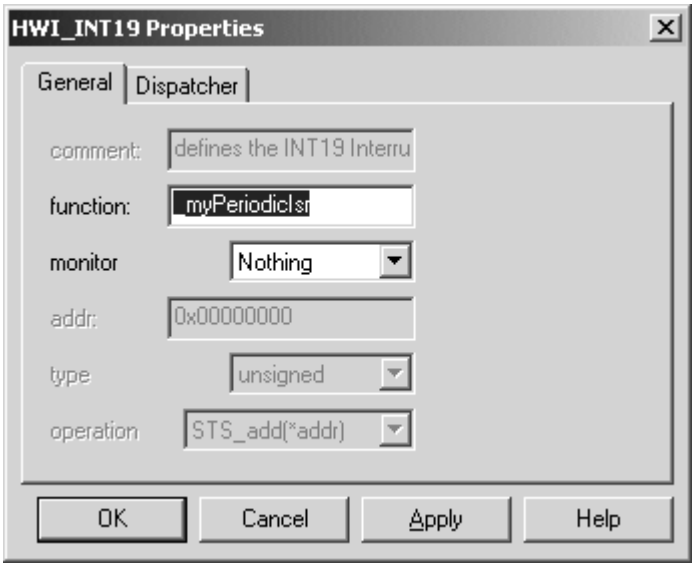

Figure 7. HWI Manager Setup for RTC

3 Examples: Programming the Real Time Clock

The accompanying zip file contains four separate example projects. These projects show how to program the RTC using the:

- Example 1: DSP/BIOS HWI Module
- Example 2: CSL RTC Module
- Example 3: CSL RTC Interrupt Dispatcher
- Example 4: CSL RTC ANSI C Style Routines

Complete source code for each example is provided in the archive file that accompanies this report. (If you do not have the archive file, it is available on the TI website at http://www.ti.com).

The source code for Example 3: CSL RTC Interrupt Dispatcher is shown in Figure 8. The code is annotated with text callouts to describe the functionality step by step. The example shows how to use configure an ISR to service an RTC interrupt using RTC_setCallback() and the associated data structures.

```
#include <csl.h>
#include <csl_rtc.h>
#include <stdio.h>

void myPeriodicIsr(void);
void myUpdateIsr(void);
extern Uint32 myVec();

volatile Uint16 rtc_cnt = 0;
volatile Uint16 counterPer = 0, counterUp = 0, sec;
int min0, min1 = 0;
int stop = 0;
int old_intm;
int eventId;

RTC_Date myDate = {
        0x02,     /* Year */
        0x02,     /* Month */
        0x28,     /* Daym */
        0x05      /* Dayw */
};

RTC_Time myTime = {
        0x1,     /* Hour */
        0x4,     /* Minutes */
        0x59,    /* Seconds */
};

RTC_IsrAddr addr = {
        myPeriodicIsr,
        NULL,
        myUpdateIsr
};

main()
{
        CSL_init();
```

Include CSL headers.

Declare and initialize CSL RTC date and time data structures.

Declare and initialize CSL RTC interrupt service routines for the periodic and update interrupts (alarm interrupt is unused).

Figure 8. Example 3: CSL RTC Interrupt Dispatcher Code

```
                                                    Configure RTC time and date.
        RTC_setTime(&myTime);
        RTC_setDate(&myDate);

        old_intm = IRQ_globalDisable();             Set interrupt table to be the one
        IRQ_setVecs((Uint32)myVec);                 defined in file vectors.asm.

        /* Clear any pending RTC interrupts */
        eventId = RTC_getEventId();
        IRQ_clear(eventId);

                                                    Place interrupt service routine
        RTC_setCallback(&addr);                     address at associated vector
                                                    location.
        /*
        RTC_setPeriodicInterval(RTC_RATE_500ms);    Set the RTC periodic interval.

        /* Enable all maskable interrupts */
        IRQ_globalEnable();

        RTC_start();                                Start RTC.

        sec = RTC_RGET(RTCSEC);                     Wait until seconds equals
                                                    zero.
         while (sec != 0) {
                sec = RTC_RGET(RTCSEC);
        }

        RTC_eventEnable(RTC_EVT_PERIODIC);          Enable alarm interrupt to start
        RTC_eventEnable(RTC_EVT_UPDATE);            one second after clock is started.

        min0 = RTC_RGET(RTCMIN);

        while (!stop)                               Allow interrupts to occur for
        {                                           one minute.
                while (RTC_FGET(RTCPINTR,UIP) != 0);
                min1 = RTC_RGET(RTCMIN);

                if ((min1 – min0) >= 1)             Disable periodic and update
                {                                   interrupts when a minute has
                        RTC_eventDisable(RTC_EVT_PERIODIC);  passed.
                        RTC_eventDisable(RTC_EVT_UPDATE);
                        stop = 1;
                }

        }

        printf(”\nRTC – testing of update and periodic interrupts – successful\n”);

        RTC_stop();                                 Stop STRC.
}
```

Figure 8. Example 3: CSL RTC Interrupt Dispatcher Code (Continued)

```
void myPeriodicIsr()
{

        ++counterPer;

        printf("\tPeriodic interrupt at: %x::%x::%x\n",
                RTC_FGET(RTCHOUR, HR), min1, RTC_RGET(RTCSEC));
}

void myUpdateIsr()
{

        ++counterUp;
        sec = RTC_RGET(RTCSEC);

        printf("Update interrupt at: %x::%x::%x\n",
                RTC_FGET(RTCHOUR, HR), min1, sec);

}
```

Figure 8. Example 3: CSL RTC Interrupt Dispatcher Code (Continued)

4 References

1. *TMS320C55x DSP Peripherals Reference Guide* (SPRU317).
2. *TMS320 DSP/BIOS User's Guide* (SPRU423).
3. *TMS320C55x Chip Support Library API Reference Guide* (SPRU433).
4. *TMS320C55x Optimizing C Compiler User's Guide* (SPRU281).
5. *The C Programming Language*, 2nd Edition, Brian W. Kernighan and Dennis M. Ritchie.
6. *TMS320VC5509 Fixed-Point DSP* (SPRS163).

Appendix A Working With the Spectrum Digital C5509 EVM

Spectrum Digital (www.spectrumdigital.com) provides a hardware evaluation module (EVM) to work with the C5509 DSP. In order to use the RTC peripheral and run the examples included with this application report, the RTC input clock (RTCLK) source needs to be configured.

Jumper JP10 is used to select the source for the RTC input clock. The clock can be driven by either the timer output pin, TOUT, or the onboard crystal. When position 1-2 is selected, the RTCLK is driven by TOUT. When the 2-3 position is used, the real-time clock is driven by a 32.768K-hertz crystal. For the example programs in this application report, **JP10 should be in position 2-3**. Note that the square solder profile on the board's underside marks pin 1. Table A–1 shows the positions and their functions.

Table A–1. JP10 RTC Clock Input Source

JP10 Position	Function
1–2	TOUT Selected as RTCLK Source
2–3	32.768K MHz Selected as RTCLK Source

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