CC1110/ CC2430/ CC2510 Debug and Programming Interface Specification

Rev. 1.2

1 Introduction

The debug interface implements a proprietary two-wire serial interface that is used for in-circuit debugging. Through this debug interface it is possible to perform an erasure of the entire flash memory, control which oscillators are enabled, stop and start execution of the user program, execute supplied instructions on the 8051 core, set code breakpoints, and single step through instructions in the code. This document describes the programming interface for the following Chipcon devices:

Device Name	Flash memory size(Kbytes)
CC1110	32 / 16 / 8
CC2430	128 / 64 / 32
CC2431	128 / 64 / 32
CC2510	32 / 16 / 8
CC2511	32 / 16 / 8

2 Hardware interface

The debug interface uses an SPI-like two-wire interface consisting of the bi-directional Debug Data (P2_1) and Debug Clock (P2_2) input pin. Data is driven on the bi-directional Debug Data pin at the positive edge of Debug Clock and data is sampled on the negative edge of this clock.

Debug commands are sent by an external host and consist of 1 to 4 output bytes from the host and an optional input byte read by the host. Figure 1 shows a timing diagram of data on the debug interface.
The first byte of the debug command is a command byte and is encoded as follows:

- bits 7 to 3 : instruction code
- bit 2 : return input byte to host
- bits 1 to 0 : number of output bytes from host following instruction code byte

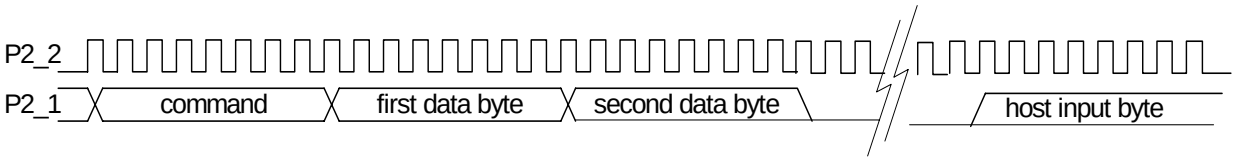


Figure 1: Debug interface timing diagram

2.1 Debug Interface AC Characteristics

T_A= -40°C to 85°C, VDD=3.0V if nothing else stated.

Parameter		Min	Typ	Max	Unit	Condition/Note
Debug clock period:	CC2430, CC2431	31.25			ns	See item 1 Figure 2
	CC1110, CC2510	38.46				
	CC2511	41.67				
Debug data setup		5			ns	See item 2 Figure 2
Debug data hold		5			ns	See item 3 Figure 2
Clock to data delay				10	ns	See item 4 Figure 2, load = 10 pF
RESET_N inactive after P2_2 rising		10			ns	See item 5 Figure 2

Table 1: Debug Interface AC Characteristics

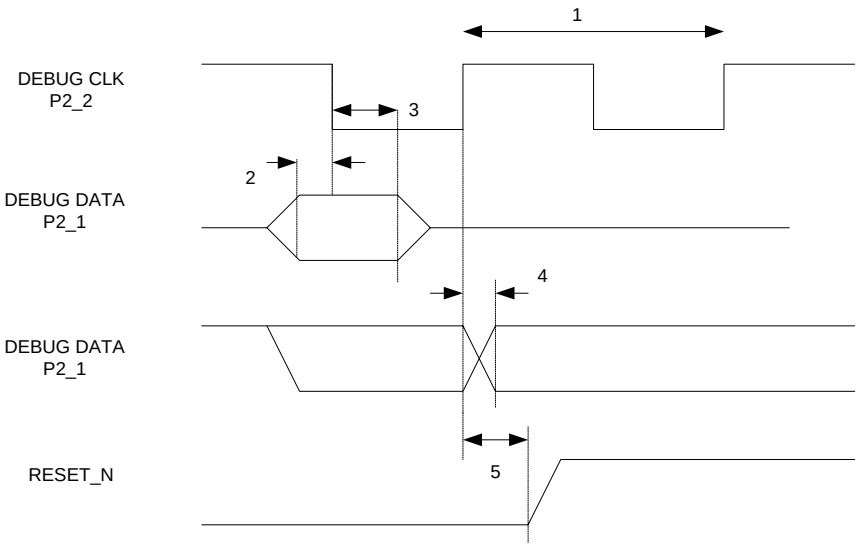


Figure 2: Debug Interface AC Characteristics

2.2 Debug Lock Bit

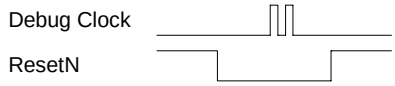
For software code security the Debug Interface may be locked. When the Debug Lock bit, DBGLOCK, is set all debug commands except CHIP_ERASE, READ_STATUS and GET_CHIP_ID are disabled and will not function.

The CHIP_ERASE command is used to clear the Debug Lock bit.

2.3 Debug Init

Debug mode is entered by forcing two rising edge transitions on pin P2_2 (Debug Clock) while the RESET_N input is held low.

`DEBUG_INIT()`
Resets the chip for debug mode.



2.4 Debug Commands

The debug commands are shown in Table 2. Some of the debug commands are described in further detail in the following sections.

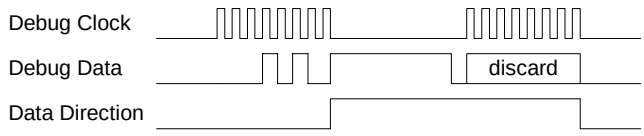
Command	Instruction code	Description
CHIP_ERASE	0001 0x00	Perform flash chip erase (mass erase) and clear lock bits. If any other command, except READ_STATUS, is issued, then the use of CHIP_ERASE is disabled.
WR_CONFIG	0001 1x01	Write configuration data.
RD_CONFIG	0010 0100	Read configuration data. Returns value set by WR_CONFIG command.
GET_PC	0010 1000	Return value of 16-bit program counter. Returns 2 bytes regardless of value of bit 2 in instruction code
READ_STATUS	0011 0x00	Read status byte.
SET_HW_BRKPNT	0011 1x11	Set hardware breakpoint
HALT	0100 0100	Halt CPU operation
RESUME	0100 1100	Resume CPU operation. The CPU must be in halted state for this command to be run.
DEBUG_INSTR	0101 01xx	Run debug instruction. The supplied instruction will be executed by the CPU without incrementing the program counter. The CPU must be in halted state for this command to be run.
STEP_INSTR	0101 1100	Step CPU instruction. The CPU will execute the next instruction from program memory and increment the program counter after execution. The CPU must be in halted state for this command to be run.
STEP_REPLACE	0110 01xx	Step and replace CPU instruction. The supplied instruction will be executed by the CPU instead of the next instruction in program memory. The program counter will be incremented after execution. The CPU must be in halted state for this command to be run.
GET_CHIP_ID	0110 1000	Return value of 16-bit chip ID and version number. Returns 2 bytes regardless of value of bit 2 of instruction code

Table 2: Debug Commands

2.4.1 CHIP_ERASE()

Erases the entire flash memory, including lock bits.

Debug command header = 0x14.

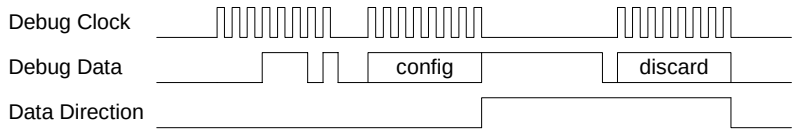


2.4.2 WR_CONFIG(IN: config_8)

Writes the debug configuration byte, which contains the following bits:

- 0x08 - TIMERS_OFF
- 0x04 - DMA_PAUSE
- 0x02 - TIMER_SUSPEND
- 0x01 - SEL_FLASH_INFO_PAGE

Debug command header = 0x1D.

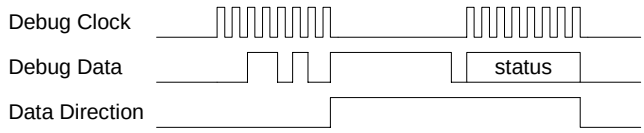


2.4.3 READ_STATUS(OUT: status_8)

Reads the debug status byte, which contains the following bits:

- 0x80 - CHIP_ERASE_DONE
- 0x40 - PCON_IDLE
- 0x20 - CPU_HALTED
- 0x10 - POWER_MODE_0
- 0x08 - HALT_STATUS
- 0x04 - DEBUG_LOCKED
- 0x02 - OSCILLATOR_STABLE
- 0x01 - STACK_OVERFLOW

Debug command header = 0x34.

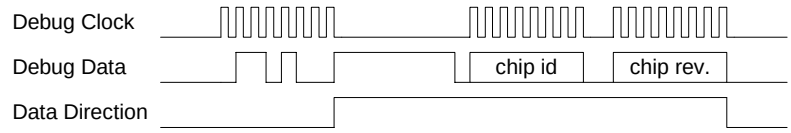


The READ_STATUS command is used e.g. for polling the status of flash chip erase after a CHIP_ERASE command or oscillator stable status required for debug commands HALT, RESUME, DEBUG_INSTR, STEP_REPLACE and STEP_INSTR.

2.4.4 GET_CHIP_ID(OUT: chip_id_8, chip_rev_8)

Writes the debug configuration byte, where bit 0 selects the flash information page (containing the lock bits).

Debug command header = 0x68.



The GET_CHIP_ID command returns the Chip_ID and version number. Chip ID and version number is also accessible for the MCU in the XDATA address range. Table 5 list Chip ID for the Chipcon devices. Version number normally corresponds to the letter describing the revision of the device: 0x01 = A, 0x02 = B, 0x03 = C.

There is one common Chip ID for all memory configurations.

Bit	Name	Reset	R/W	Description
7:0	CHIPID[7:0]	0xXX	R	Chip identification number, see table for chip

Table 3: Register CHIPID

Bit	Name	Reset	R/W	Description
7:0	VERSION[7:0]	0xXX	R	Chip revision number

Table 4: Register CHVER

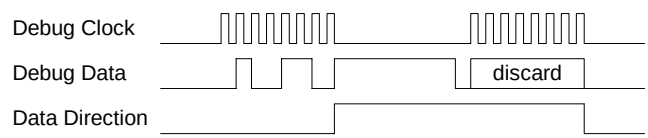
Device Name	Chip ID
CC1110	0x01
CC2430	0x85
CC2431	0x89
CC2510	0x81
CC2511	0x91

Table 5: Chip ID for Chipcon devices

2.4.5 HALT()

Halts the CPU

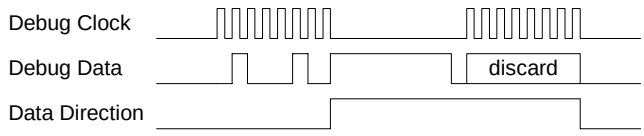
Debug command header = 0x44.



2.4.6 RESUME()

Starts/resumes the CPU

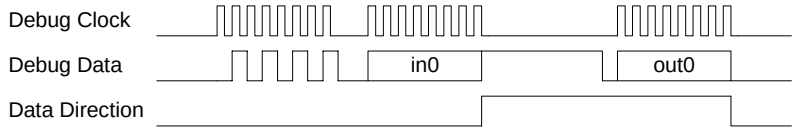
Debug command header = 0x4C.



2.4.7 DEBUG_INSTR(IN: in0_8, OUT: out0_8);

Executes a 1-byte 8051 instruction on the CPU, without changing the program counter (unless the debug instruction is a jump operation).

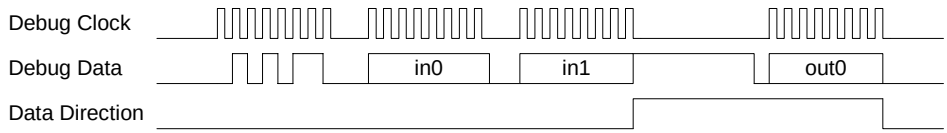
Debug command header = 0x55.



2.4.8 DEBUG_INSTR(IN: in0_8, in1_8, OUT: out0_8);

Executes a 2-byte 8051 instruction on the CPU, without changing the program counter (unless the debug instruction is a jump operation).

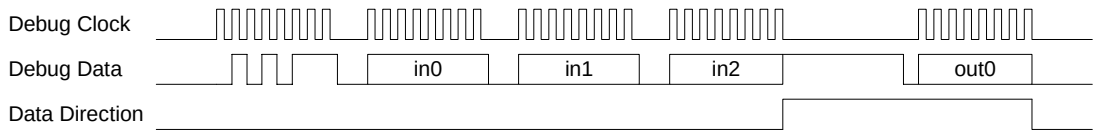
Debug command header = 0x56.



2.4.9 DEBUG_INSTR(IN: in0_8, in1_8, in2_8, OUT: out0_8);

Executes a 3-byte 8051 instruction on the CPU, without changing the program counter (unless the debug instruction is a jump operation).

Debug command header = 0x57.



3 Flash Programming

Programming of the on-chip flash is performed via the debug interface. The external host must initially send instructions using the DEBUG_INSTR debug command to perform the flash programming with the Flash Controller as described below. The sequences are based on the commands listed in chapter 2.4. For complete description of the flash controller, please see the flash controller section of each datasheet.

```
#define LOBYTE(w)           ((BYTE)(w))
#define HIBYTE(w)           ((BYTE)(((WORD)(w) >> 8) & 0xFF))
```

3.1.1 READ_CODE_MEMORY(IN: address_16, bank_8, count_16, OUT: outputArray_8)

Reads from the specified bank in CODE memory into outputArray, byte by byte.

address < 0x8000: Linear address = address
address >= 0x8000: Linear address = (address & 0x7FFF) + (bank * 0x8000)

```
DEBUG_INSTR(IN: 0x75, 0xC7, (bank * 16) + 1, OUT: Discard);                 MOV MEMCTR, (bank * 16) + 1;
DEBUG_INSTR(IN: 0x90, HIBYTE(address), LOBYTE(address), OUT: Discard);     MOV DPTR, address;
for (n = 0; n < count_16; n++) {
    DEBUG_INSTR(IN: 0xE4, OUT: Discard);                                    CLR A;
    DEBUG_INSTR(IN: 0x93, OUT: outputArray[n]);                             MOVC A, @A+DPTR; (outputArray[n] = A)
    DEBUG_INSTR(IN: 0xA3, OUT: Discard);                                    INC DPTR;
}
```

3.1.2 READ_XDATA_MEMORY(IN: address_16, count_16, OUT: outputArray_8)

Reads from XDATA memory into inputArray, byte by byte.

```
DEBUG_INSTR(IN: 0x90, HIBYTE(address), LOBYTE(address), OUT: Discard);     MOV DPTR, address;
for (n = 0; n < count; n++) {
    DEBUG_INSTR(IN: 0xE0, OUT: outputArray[n]);                             MOVX A, @DPTR; (outputArray[n] = A)
    DEBUG_INSTR(IN: 0xA3, OUT: Discard);                                    INC DPTR;
}
```

3.1.3 WRITE_XDATA_MEMORY(IN: address_16, count_16, inputArray_8)

Writes data from inputArray into XDATA memory, byte by byte.

```
DEBUG_INSTR(IN: 0x90, HIBYTE(address), LOBYTE(address), OUT: Discard);       MOV DPTR, address;
for (n = 0; n < count; n++) {
DEBUG_INSTR(IN: 0x74, inputArray[n], OUT: Discard);                        MOV A, #inputArray[n];
    DEBUG_INSTR(IN: 0xF0, OUT: Discard);                                       MOVX @DPTR, A;
    DEBUG_INSTR(IN: 0xA3, OUT: Discard);                                       INC DPTR;
}
```

3.1.4 SET_PC(IN: address_16)

Modifies the program counter value

```
DEBUG_INSTR(IN: 0x02, HIBYTE(address), LOBYTE(address), OUT: Discard);       LJMP address;
```

3.1.5 CLOCK_INIT()

Initializes the 32 MHz crystal oscillator

Important:

- The loop can lock up if the operation fails (due to communication or chip errors). If polling is not desirable, then wait for 1 ms instead.

```
DEBUG_INSTR(IN: 0x75, 0xC6, 0x00);                                        MOV CLKCON, #00H;
  do {
      DEBUG_INSTR(IN: 0xE5, 0xBE, OUT: sleepReg);                          MOV A, SLEEP; (sleepReg = A)
  } while (!(sleepReg & 0x40));
```

3.1.6 WRITE_FLASH_PAGE(IN: address_17, inputArray_8, erase_page_1)

Writes a single flash page by loading the image into XDATA memory, together with an assembly routine that performs the actual update. This is done by using unified mapping.

The **marked** section, which performs page erasure, should only be included in the routine when the erase_page_1 = 1. **The pseudo-code does not refer to this parameter!**

```
routine_8[] = {
    0x75, 0xAD, ((address >> 8) / FLASH_WORD_SIZE) & 0x7E,
    0x75, 0xAC, 0x00,
    0x75, 0xAE, 0x01,

    0xE5, 0xAE,
    0x20, 0xE7, 0xFB,

    0x90, 0xF0, 0x00,

    0x7F, HIBYTE(WORDS_PER_FLASH_PAGE),
    0x7E, LOBYTE(WORDS_PER_FLASH_PAGE),
    0x75, 0xAE, 0x02,

    0x7D, FLASH_WORD_SIZE,
    0xE0,
    0xA3,
    0xF5, 0xAF,
    0xDD, 0xFA,

    0xE5, 0xAE,
    0x20, 0xE6, 0xFB,
    0xDE, 0xF1,
    0xDF, 0xEF,

    0xA5
};
```

```
//                  MOV FADDRH, #imm;
//                  MOV FADDRL, #00;
//                  MOV FLC, #01H; // ERASE
//                  ; Wait for flash erase to complete
// eraseWaitLoop:   MOV A, FLC;
//                  JB ACC_BUSY, eraseWaitLoop;
//                  ; Initialize the data pointer
//                  MOV DPTR, #0F000H;
//                  ; Outer loops
//                  MOV R7, #imm;
//                  MOV R6, #imm;
//                  MOV FLC, #02H; // WRITE
//                      ; Inner loops
// writeLoop:           MOV R5, #imm;
// writeWordLoop:           MOVX A, @DPTR;
//                          INC DPTR;
//                          MOV FWDATA, A;
//                      DJNZ R5, writeWordLoop;
//                      ; Wait for completion
// writeWaitLoop:       MOV A, FLC;
//                      JB ACC_SWBSY, writeWaitLoop;
//                  DJNZ R6, writeLoop;
//                  DJNZ R7, writeLoop;
//                  ; Done, fake a breakpoint
//                  DB 0xA5;
```

```
WRITE_XDATA_MEMORY(IN: 0xF000, FLASH_PAGE_SIZE, inputArray_8);
WRITE_XDATA_MEMORY(IN: 0xF000 + FLASH_PAGE_SIZE, sizeof(routine), routine);
DEBUG_INSTR(IN: 0x75, 0xC7, 0x51, OUT: Discard);                              MOV MEMCTR, (bank * 16) + 1;
SET_PC(0xF000 + FLASH_PAGE_SIZE);
RESUME();
   do {
       READ_STATUS(OUT: statusByte);
   } while (!(statusByte & CPU_HALTED));
```

3.1.7 READ_FLASH_PAGE(IN: linearAddress_17, OUT: outputArray_8)

Reads one page from flash memory

```
READ_CODE_MEMORY(address & 0xFFFF, (linearAddress >> 15) & 0x03, FLASH_PAGE_SIZE, outputArray);
```

3.1.8 MASS_ERASE_FLASH()

Initiates a mass erase, which blanks out the entire flash memory and lock bits, and then waits for the operation to complete.
The initial NOP ensures that the status byte has been updated

Important:

- The loop can lock up if the operation fails (due to communication or chip errors). If polling is not desirable, then wait for 20 ms instead.

```
DEBUG_INSTR(IN: 0x00, OUT: Discard);                                    NOP;
CHIP_ERASE();
do {
    READ_STATUS(OUT: statusByte);
} while (!(statusByte & CHIP_ERASE_DONE));
```

3.1.9 PROGRAM_FLASH(IN: imageArray_8, OUT: verificationIsOk_1)

Programs the entire flash memory and verifies it.

The logic in this pseudo-code requires that the size of the image array matches the total size of the flash memory in the device to be programmed (e.g. 128 kB for the CC2430F128). It also requires unused addresses to be set to 0xFF.

```
DEBUG_INIT();
CLOCK_INIT();
MASS_ERASE_FLASH();
verificationArray_8[FLASH_PAGE_SIZE];
verificationIsOk = 1;
  for (p = 0; p < (FLASH_SIZE / FLASH_PAGE_SIZE); p++) {
    pageAddress = p * FLASH_PAGE_SIZE;
    memset(verificationArray, 0xFF, FLASH_PAGE_SIZE);
    if (memcmp(verificationArray, &inputArray[pageAddress], FLASH_PAGE_SIZE)) {
        WRITE_FLASH_PAGE(IN: pageAddress, &inputArray[pageAddress], 0);
        READ_FLASH_PAGE(IN: pageAddress, verificationArray);
        if (memcmp(verificationArray, &inputArray[pageAddress], FLASH_PAGE_SIZE)) {
            verificationIsOk = 0;
            return;
        }
    }
}
```

3.1.10 Flash Write Timing

The Flash Controller contains a timing generator, which controls the timing sequence of flash write and erase operations. The timing generator uses the information set in the Flash Write Timing register, `FWT.FWT[5:0]`, to set the internal timing. `FWT.FWT[5:0]` must be set to a value according to the currently selected CPU clock frequency.

The value set in the `FWT.FWT[5:0]` shall be set according to the CPU clock frequency by the following equation.

$$FWT = \frac{21000 * F_{CPU}}{16 * 10^9}$$

F_{CPU} is the CPU clock frequency. The initial value held in `FWT.FWT[5:0]` after a reset is 0x2A which corresponds to 32 MHz CPU clock frequency.

The FWT values for common CPU clock frequencies are given in Table 6.

CPU clock frequency (MHz)	FWT
12	0x10
13	0x11
16	0x15
24	0x20
26	0x23
32	0x2A

Table 6: Flash timing (FWT) values

4 Document history

Version	Data	Description/Changes
1.2	22-12-2006	Removed classification “Chipcon internal and partners with NDA”, added CC1110, CC2510, CC2511 data.
1.1	27-04-2006	Bug fix in WRITE_FLASHPAGE, changed value in HALT and RESUME, deleted chapter READ_FLASH_PAGE (was written twice).
1.0		Initial

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.

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