

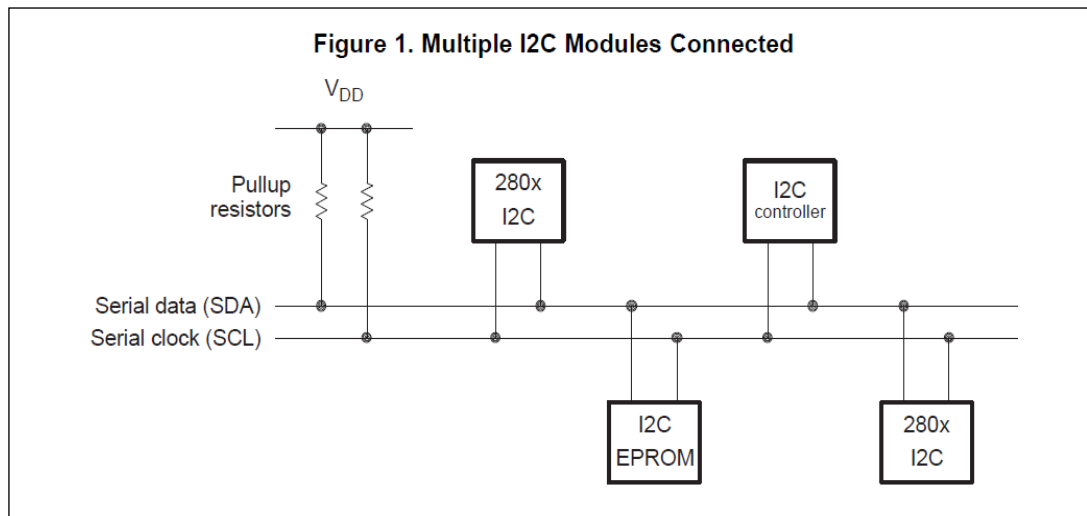
---

# F2833x Inter Integrated Circuit

---

## Introduction

This module discusses the features and operation of the inter-integrated circuit (I2C) module that is available on the F2833x digital signal controller (DSC). The I2C module provides an interface between DSCs and devices compliant with Philips Semiconductors Inter-IC bus (I2C-bus) specification version 2.1 and connected by way of an I2C-bus. External components attached to this 2-wire serial bus can transmit and/or receive data between 1-bit and 8-bits to/from the F2833x through the I2C module. This student guide assumes the reader is somewhat familiar with the I2C-bus specification.



Each device connected to an I2C-bus is identified by a unique address. It can operate as either a transmitter or a receiver, depending on the function of the device. A device connected to the I2C - bus can also be considered as the master or the slave when performing data transfers. A master device is the device that initiates a data transfer on the bus and generates the clock signals to permit that transfer. During this transfer, any device addressed by this master is considered to be a slave.

The I2C module supports the multi-master mode, in which one or more devices are capable of controlling an I2C-bus and can be connected to the same I2C-bus.

For data communication, the I2C module has a serial data pin (SDA) and a serial clock pin (SCL). The SDA and SCL pins both are bidirectional. They must each be connected to a positive 3.3 V supply voltage using pull-up resistors. When the bus is free, both pins are high. The driver of these two pins has an open-drain configuration to perform the required wired-AND function. The F2833x includes internal pull-up resistors for SDA and SCL, which can be enabled during the setup of the GPIO - pins.

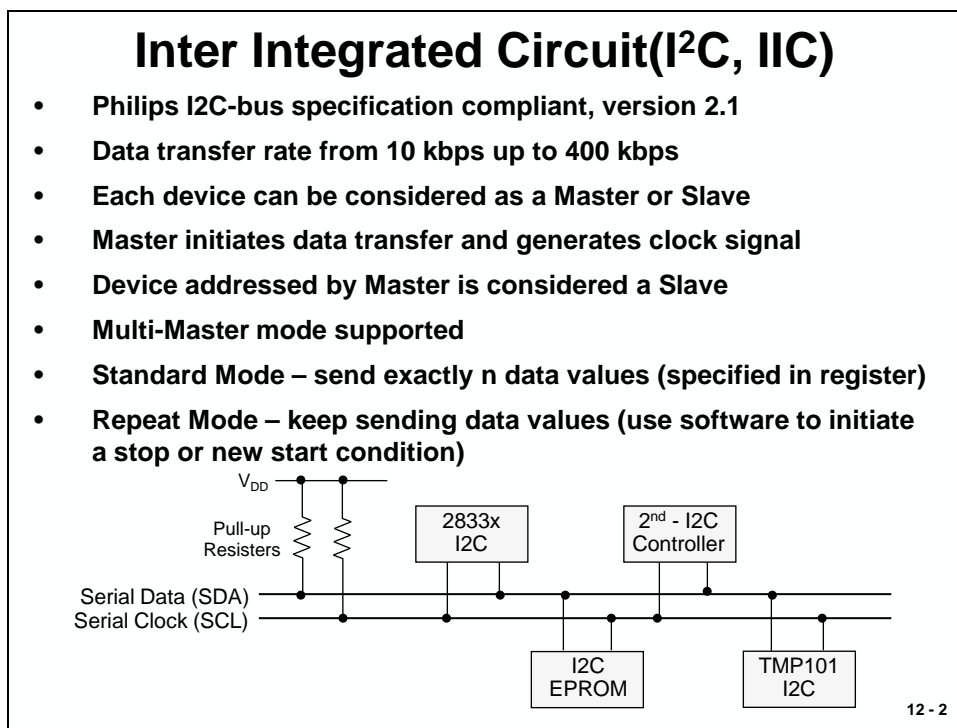
# Module Topics

<b>F2833x Inter Integrated Circuit .....</b>	<b>12-1</b>
<i>Introduction.....</i>	<i>12-1</i>
<i>Module Topics.....</i>	<i>12-2</i>
<i>Basic I2C Features.....</i>	<i>12-4</i>
<i>F2833x I2C Block Diagram.....</i>	<i>12-5</i>
<i>I2C Clock Generation .....</i>	<i>12-6</i>
<i>I2C Operating Modes.....</i>	<i>12-8</i>
Master / Slave modes .....	12-8
Input and Output Voltage Levels .....	12-8
Data Validity.....	12-9
Serial Data Formats .....	12-10
<i>Arbitration.....</i>	<i>12-11</i>
<i>I2C Interrupts.....</i>	<i>12-12</i>
<i>I2C Module Registers.....</i>	<i>12-13</i>
I2C Mode Register.....	12-13
I2C Interrupt Enable Register .....	12-17
I2C Status Register .....	12-17
I2C Interrupt Source Register .....	12-18
I2C Clock Register.....	12-18
I2C Slave Address Register .....	12-19
I2C Own Address Register .....	12-20
I2C Data Count Register .....	12-20
I2C Data Registers .....	12-21
<i>I2C FIFO Buffers .....</i>	<i>12-21</i>
I2C TX-FIFO Register.....	12-22
I2C RX-FIFO Register.....	12-22
<i>Temperature Sensor TMP100 .....</i>	<i>12-23</i>
TMP100 Register Structure .....	12-25
Temperature Register.....	12-26
Configuration Register.....	12-26
TMP100 Timing Diagrams .....	12-27
<i>Lab Exercise 12_1 .....</i>	<i>12-30</i>
Preface .....	12-30
Objective.....	12-30
Procedure .....	12-30
Open Files, Create Project File .....	12-30
Project Build Options.....	12-31
Preliminary Test.....	12-32
Add TMP100 and I2C Initialization Code .....	12-32
Build, Load and Run .....	12-34
<i>Lab Exercise 12_2.....</i>	<i>12-35</i>
Objective.....	12-35
Procedure .....	12-35
Open Project, Modify Source File .....	12-35
Build, Load and Run .....	12-36
Troubleshooting .....	12-38

<i>Lab Exercise 12_3</i> .....	12-40
Objective .....	12-40
Procedure.....	12-40
Open Project, Modify Source File .....	12-40
Build, Load and Run .....	12-41
<i>Lab Exercise 12_4</i> .....	12-42
Objective .....	12-42
Procedure.....	12-42
Open Project, Modify Source File .....	12-42
Build, Load and Run .....	12-44

## Basic I2C Features

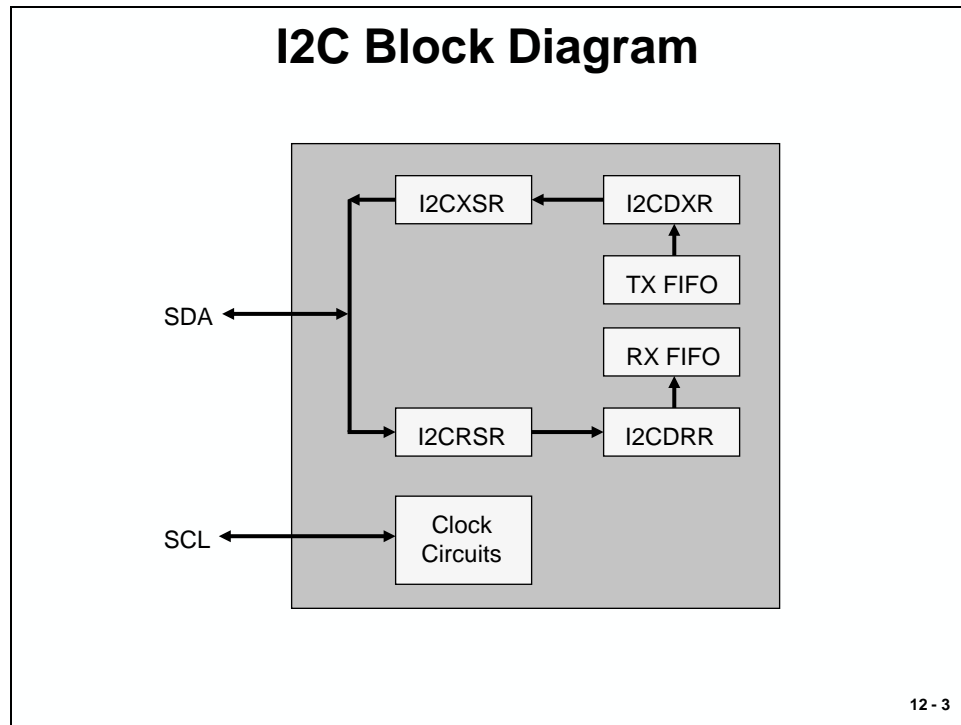
The I2C module supports any slave or master I2C-compatible device.



The I2C module has the following features:

- Compliance with the Philips Semiconductors I2C-bus specification (version 2.1):
  - Support for 8-bit format transfers
  - 7-bit and 10-bit addressing modes
  - General call
  - START byte mode
  - Support for multiple master-transmitters and slave-receivers
  - Support for multiple slave-transmitters and master-receivers
  - Combined master transmit/receive and receive/transmit mode
  - Data transfer rate of from 10 kbps up to 400 kbps (Philips Fast-mode rate)
- One 16-byte deep receive FIFO and one 16-byte deep transmit FIFO
- An interrupt can be generated as a result of one of the following conditions:
  - transmit-data ready,
  - receive-data ready,
  - register-access ready,
  - no-acknowledgment received,
  - arbitration lost,
  - stop condition detected,
  - addressed as slave.
- Free data format mode

## F2833x I2C Block Diagram



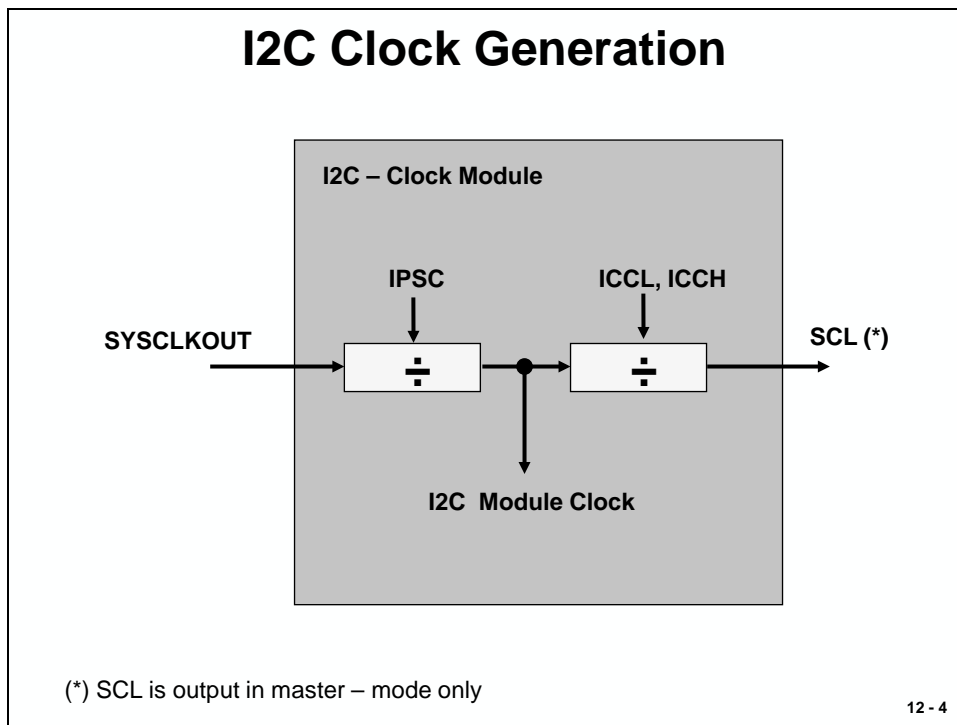
The I2C module consists of the following primary blocks:

- A serial interface: one data pin (SDA) and one clock pin (SCL)
- Data registers and FIFOs to temporarily hold receive data and transmit data traveling between the SDA pin and the CPU
- Control and status registers
- A peripheral bus interface to enable the CPU to access the I2C module registers and FIFOs.
- A clock synchronizer to synchronize the I2C input clock (from the DSP clock generator) and the clock on the SCL pin, and to synchronize data transfers with masters of different clock speeds required.
- A pre-scaler to divide down the input clock that is driven to the I2C module
- A noise filter on each of the two pins, SDA and SCL
- An arbitrator to handle arbitration between the I2C module (when it is a master) and another master
- Interrupt generation logic, so that an interrupt can be sent to the CPU.

Slide 12-3 shows the four registers used for transmission and reception in non-FIFO mode. The CPU writes data for transmission to I2CDXR and reads received data from I2CDRR. When the I2C module is configured as a transmitter, data written to I2CDXR is copied to I2CXSR and shifted out on the SDA pin one bit at a time. When the I2C module is configured as a receiver, received data is shifted into I2CRSR and then copied to I2CDRR.

## I2C Clock Generation

As shown in Slide 12-4, the I2C input clock is equivalent to the CPU clock (SYSCLKOUT) and is then divided twice more inside the I2C module to produce the module clock and the master SCL clock.



The I2C module clock determines the frequency at which the I2C module operates. A programmable pre-scaler in the I2C module divides down the I2C input clock to produce the module clock. To specify the divide-down value, initialize the IPSC field of the pre-scaler register, I2CPSC. The resulting frequency should be in the range of 7 - 12 MHz and is given by:

$$I2C\_Module\_Clock = \frac{SYSCLKOUT}{(IPSC + 1)}$$

IPSC must be initialized only while the I2C module is in the reset state (IRS = 0 in I2CMDR). The pre-scaled frequency takes effect only when IRS is changed to 1. Changing the IPSC value while IRS = 1 has no effect.

The master clock appears on the SCL pin when the I2C module is configured to be a master on the I2C-bus. This clock controls the timing of communication between the I2C module and a slave. As shown in slide 12-4, a second clock divider in the I2C module divides down the module clock to produce the master clock. The clock divider uses the ICCL value of I2CCLKL to divide down the low portion of the module clock signal and uses the ICCH value of I2CCLKH to divide down the high portion of the module clock signal.

Example for I2C-clock calculation:

The period of the master clock ( $T_{MASTER}$ ) is a multiple of the period of the I2C module clock:

$$T_{MASTER} = \frac{(IPSC + 1)[(ICCL + d) + (ICCH + d)]}{SYSCLKOUT}$$

Parameter d is a systematic offset, which depends on the device type.

Example: Set I2C-Master clock to 50 kHz for a 150 MHz device

(1) Set I2C module clock to 10MHz:

$$10MHz = \frac{100MHz}{(IPSC + 1)}; \quad IPSC = 14$$

(2) Set I2C Master clock to 20μs; use d = 5

$$20\mu s = \frac{(14 + 1)[(ICCL + 5) + (ICCH + 5)]}{150MHz}$$

$$ICCL + ICCH = 190;$$

To produce an I2C master clock with a duty cycle of 50% set:

- IPSC = 14
- ICCL = 95
- ICCH = 95

The following table give some more options for the I2C clock unit:

SYSCLKOUT	100 MHz	100MHz	150MHz	150MHz
I2C-clock	IPSC	ICCL / ICCH	IPSC	ICCL / ICCH
50 kHz	9	95 / 95	14	95 / 95
100 kHz	9	45 / 45	14	45 / 45
400 kHz	9	10 / 5	14	10 / 5

## I2C Operating Modes

### Master / Slave modes

The I2C module has four basic operating modes to support data transfers as a master and as a slave.

If the I2C module is a master, it begins as a master-transmitter and typically transmits an address for a particular slave. When giving data to the slave, the I2C module must remain a master-transmitter. To receive data from a slave, the I2C module must be changed to the master-receiver mode.

When the I2C module is a slave, it begins as a slave-receiver and typically sends acknowledgment when it recognizes its slave address from a master. If the master is sending data to the I2C module, the module must remain a slave-receiver. If the master has requested data from the I2C module, the module must be changed to the slave-transmitter mode.

I2C Operating Modes	
Operating Mode	Description
Slave-receiver mode	Module is a slave and receives data from a master (all slaves begin in this mode)
Slave-transmitter mode	Module is a slave and transmits data to a master (can only be entered from slave-receiver mode)
Master-receiver mode	Module is a master and receives data from a slave (can only be entered from master-transmit mode)
Master-transmitter mode	Module is a master and transmits to a slave (all masters begin in this mode)

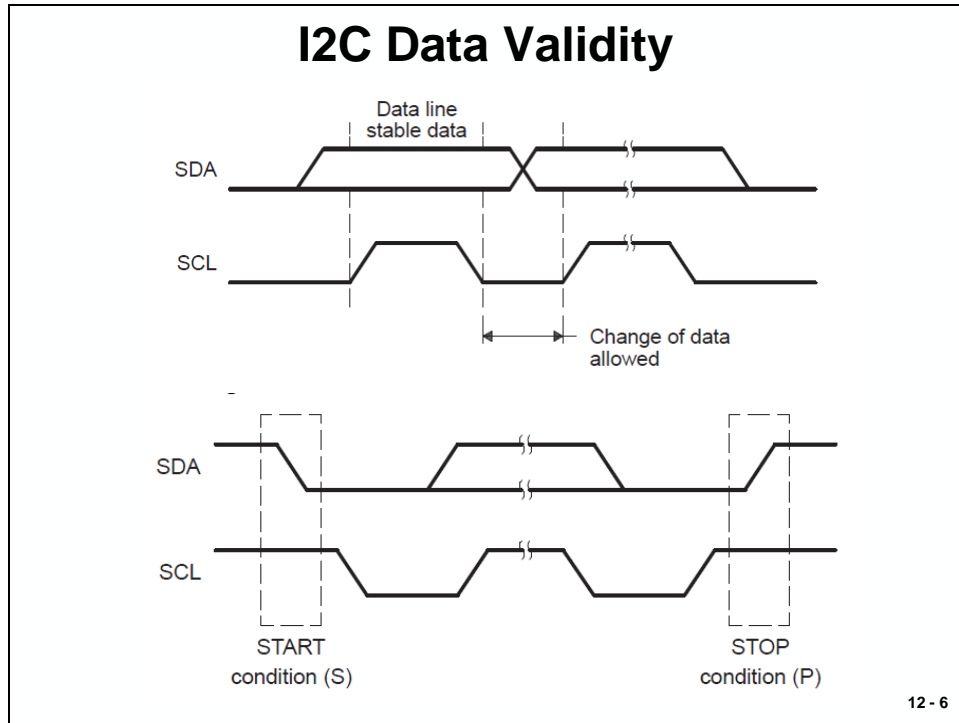
12 - 5

### Input and Output Voltage Levels

One clock pulse is generated by the master device for each data bit transferred. Due to a variety of different technology devices that can be connected to the I2C-bus, the levels of logic 0 (low) and logic 1 (high) are not fixed and depend on the associated level of  $V_{DD}$ . For details, see the data manual for your particular F2833x.

## Data Validity

The data on SDA must be stable during the high period of the clock (see Slide 12-6). The high or low state of the data line, SDA, should change only when the clock signal on SCL is low.



START and STOP conditions can be generated by the I2C module when the module is configured to be a master on the I2C-bus.

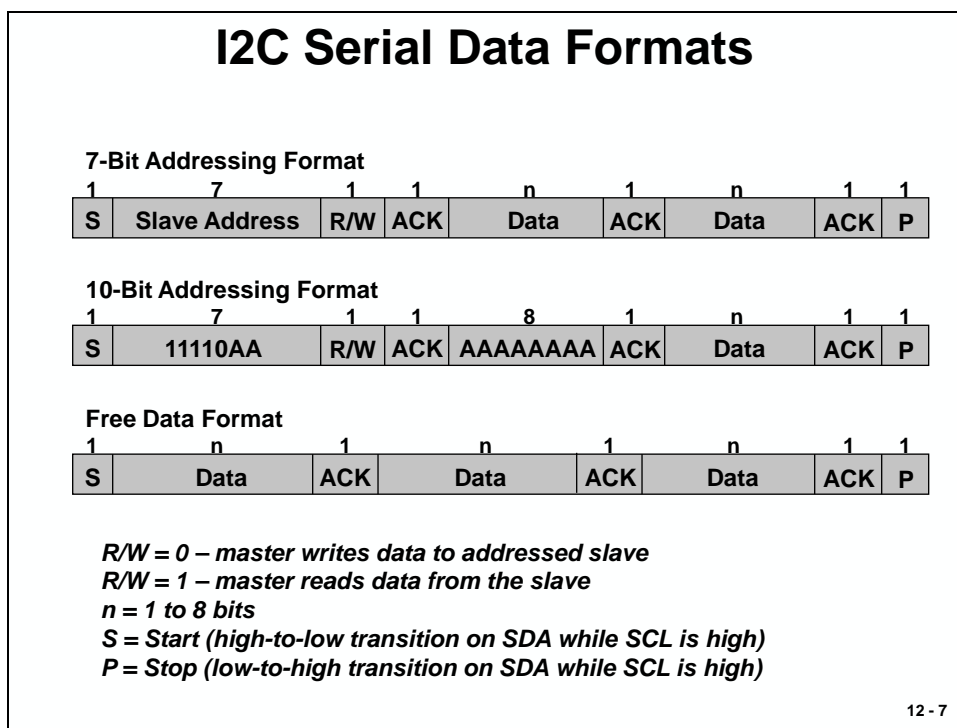
- The START condition is defined as a high-to-low transition on the SDA line while SCL is high. A master drives this condition to indicate the start of a data transfer.
- The STOP condition is defined as a low-to-high transition on the SDA line while SCL is high. A master drives this condition to indicate the end of a data transfer

After a START condition and before a subsequent STOP condition, the I2C-bus is considered busy, and the bus busy (BB) bit of I2CSTR is 1. Between a STOP condition and the next START condition, the bus is considered free, and BB is 0.

For the I2C module to start a data transfer with a START condition, the master mode bit (MST) and the START condition bit (STT) in I2CMDR must both be 1. For the I2C module to end a data transfer with a STOP condition, the STOP condition bit (STP) must be set to 1.

## Serial Data Formats

I2C is programmable to operate in different addressing formats, selected by bits “FDF” and “XA” in register I2CMDR.



FDF	XA	Format
1	X	Free Data Format
0	0	7-Bit Addressing Format
0	1	10-Bit Addressing Format

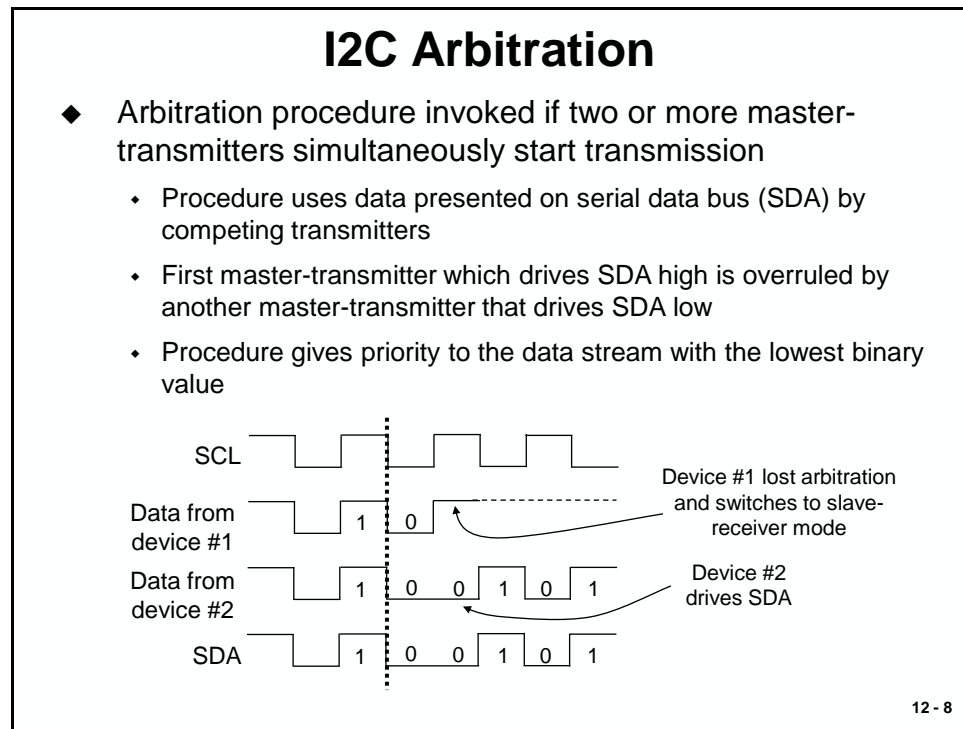
In the 7-bit addressing format, which is often used, the first byte after a START condition (S) consists of a 7-bit slave address followed by an R/W bit. R/W determines the direction of the data:

- R/W = 0: The master writes (transmits) data to the addressed slave.
- R/W = 1: The master reads (receives) data from the slave.

An extra clock cycle dedicated to acknowledgment (ACK) is inserted after each byte. If the ACK bit is inserted by the slave after the first byte from the master, it is followed by n bits of data from the transmitter (master or slave, depending on the R/W bit). N is a number from 1 to 8 determined by the bit count (BC) field of I2CMDR. After the data bits have been transferred, the receiver inserts an ACK bit.

## Arbitration

If two or more master-transmitters attempt to start a transmission on the same bus at approximately the same time, an arbitration procedure is invoked. The arbitration procedure uses the data presented on the serial data bus (SDA) by the competing transmitters. Slide 12-8 illustrates the arbitration procedure between two devices. The first master-transmitter, which releases the SDA line high, is overruled by another master-transmitter that drives SDA low. The arbitration procedure gives priority to the device that transmits the serial data stream with the lowest binary value. Should two or more devices send identical first bytes, arbitration continues on the subsequent bytes.



If the I2C module is the losing master, it switches to the slave-receiver mode, sets the arbitration lost (AL) flag, and generates the arbitration-lost interrupt request.

If during a serial transfer the arbitration procedure is still in progress when a repeated START condition or a STOP condition is transmitted to SDA, the master-transmitters involved must send the repeated START condition or the STOP condition at the same position in the format frame. Arbitration is not allowed between:

- A repeated START condition and a data bit
- A STOP condition and a data bit
- A repeated START condition and a STOP condition

## I2C Interrupts

The I2C module generates the interrupt requests described in Slide 12-9. All interrupt sources are multiplexed through an arbiter to a single I2C interrupt request to the CPU. Each interrupt request has a flag bit in the status register (I2CSTR) and an enable bit in the interrupt enable register (I2CIER). When one of the specified events occurs, its flag bit is set. If the corresponding enable bit is 0, the interrupt request is blocked. If the enable bit is 1, the request is forwarded to the CPU as an I2C interrupt.

I2C Interrupts	
Interrupt Source	Description
XRDYINT	Transmit ready condition: The data transmit register (I2CDXR) is ready to accept new data because the previous data has been copied from I2CDXR to the transmit shift register (I2CXSR).
RRDYINT	Receive ready condition: The data receive register (I2CDRR) is ready to be read because data has been copied from the receive shift register (I2CRSR) to I2CDRR.
ARDYINT	Register-access ready condition: The I2C module registers are ready to be accessed because the previously programmed address, data, and command values have been used.
NACKINT	No-acknowledgment condition: The I2C module is configured as a master-transmitter and did not received acknowledgment from the slave-receiver.
ALINT	Arbitration-lost condition: The I2C module has lost an arbitration contest with another master-transmitter.
SCDINT	Stop condition detected: A STOP condition was detected on the I2C bus.
AASINT	Addressed as slave condition: The I2C has been addressed as a slave device by another master on the I2C bus.
I2CFIFO	see FIFO - Registers

12 - 9

The I2C interrupt is one of the maskable interrupts of the CPU. Like any other maskable interrupt request, if it is properly enabled, the CPU executes the corresponding interrupt service routine (I2CINT1A\_ISR). The I2CINT1A\_ISR for the I2C interrupt can determine the interrupt source by reading the interrupt source register, I2CISRC. Then the I2CINT1A\_ISR can branch to the appropriate subroutine.

After the CPU reads I2CISRC, the following events occur:

- (1) The flag for the source interrupt is cleared in I2CSTR. Exception: The ARDY, RRDY, and XRDY bits in I2CSTR are not cleared when I2CISRC is read. To clear one of these bits, write a 1 to it.
- (2) The arbiter determines which of the remaining interrupt requests has the highest priority, writes the code for that interrupt to I2CISRC, and forwards the interrupt request to the CPU.

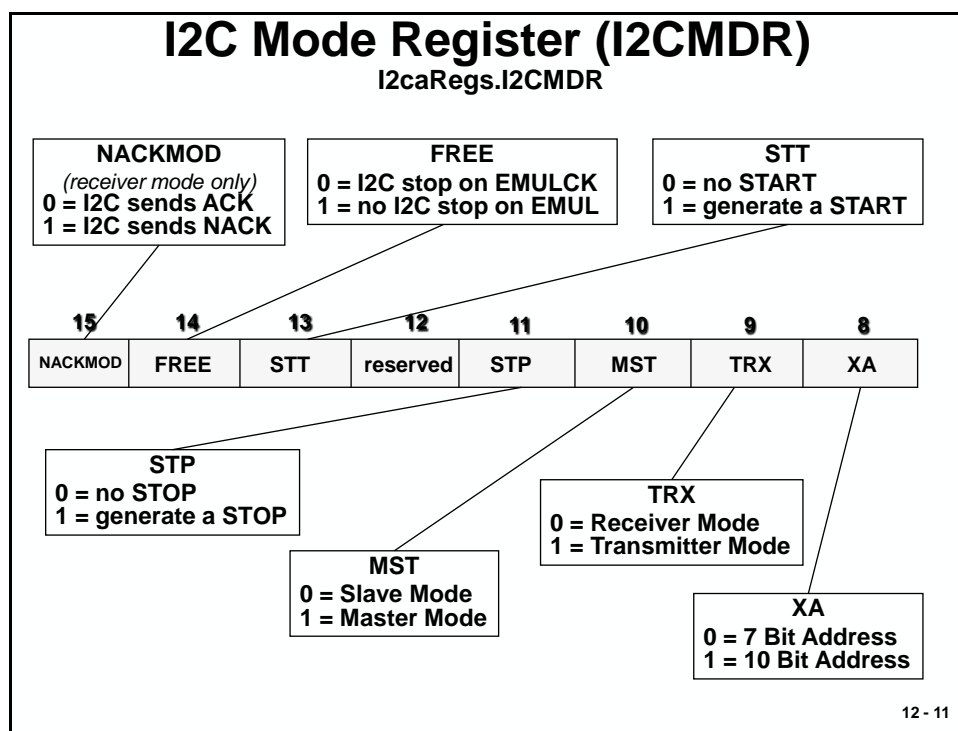
In addition to the seven basic I2C interrupts, the transmit and receive FIFOs each have the ability to generate an additional interrupt (I2CINT2A). The FIFOs can be configured to generate an interrupt after transmitting/receiving a defined number of bytes, up to 16. These two interrupt sources are ORed together into a single maskable CPU interrupt. The interrupt service routine can then read the FIFO interrupt status flags to determine from which source the interrupt came. See the I2C transmit FIFO register (I2CFFTX) and the I2C receive FIFO register (I2CFFRX) descriptions.

## I2C Module Registers

I2C Registers	
Register Name	Description
I2COAR	I2C own address register
I2CIER	I2C interrupt enable register
I2CSTR	I2C status register
I2CCLKL	I2C clock low-time divide register
I2CCLKH	I2C clock high-time divide register
I2CCNT	I2C data count register
I2CDRR	I2C data receive register
I2CSAR	I2C slave address register
I2CDXR	I2C data transmit register
I2CMDR	I2C mode register
I2CISRC	I2C interrupt source register
I2CEMDR	I2C extended mode register
I2CPSC	I2C prescaler register
I2CFFTX	I2C FIFO transmit register
I2CFFRX	I2C FIFO receive register

12 - 10

## I2C Mode Register



12 - 11

The I2C mode register (I2CMDR) is a 16-bit register that contains the control bits of the I2C module. The bit fields of I2CMDR are shown in slides 12-11 and 12-12.

**NACKMOD**

This bit is only applicable when the I2C module is acting as a receiver.

- 0 In the slave-receiver mode: The I2C module sends an acknowledge (ACK) bit to the transmitter during each acknowledge cycle on the bus. The I2C module only sends a no-acknowledge (NACK) bit if you set the NACKMOD bit.  
In the master-receiver mode: The I2C module sends an ACK bit during each acknowledge cycle until the internal data counter counts down to 0. At that point, the I2C module sends a NACK bit to the transmitter. To have a NACK bit sent earlier, you must set the NACKMOD bit.
- 1 In either slave-receiver or master-receiver mode: The I2C module sends a NACK bit to the transmitter during the next acknowledge cycle on the bus. Once the NACK bit has been sent, NACKMOD is cleared. Note: To send a NACK bit in the next acknowledge cycle, NACKMOD must be set before the rising edge of the last data bit.

**FREE**

This bit controls the action taken by the I2C module when a debugger breakpoint is encountered.

- 0 When the I2C module is a master:  
If SCL is low when the breakpoint occurs, the I2C module stops immediately and keeps driving SCL low, whether the I2C module is the transmitter or the receiver. If SCL is high, the I2C module waits until SCL becomes low and then stops.  
When I2C module is slave:  
A breakpoint forces the I2C module to stop when the current transmission/reception is complete.
- 1 The I2C module runs free; that is, it continues to operate when a breakpoint occurs.

**STT**

START condition bit (only applicable when the I2C module is a master). The RM, STT, and STP bits determine when the I2C module starts and stops data transmissions (see Table 6). Note that the STT and STP bits can be used to terminate the repeat mode, and that this bit is not writable when IRS = 0.

- 0 In the master mode, STT is automatically cleared after the START condition has been generated.
- 1 In the master mode, setting STT to 1 causes the I2C module to generate a START condition on the I2C-bus.

**STP**

STOP condition bit (only applicable when the I2C module is a master). In the master mode, the RM, STT, and STP bits determine when the I2C module starts and stops data transmissions. Note that the STT and STP bits can be used to terminate the repeat mode, and that this bit is not writable when IRS=0.

- 0 STP is automatically cleared after the STOP condition has been generated.
- 1 STP has been set to generate a STOP condition when the internal data counter of the I2C module counts down to 0.

**MST**

Master mode bit. MST determines whether the I2C module is in the slave mode or the master mode. MST is automatically changed from 1 to 0 when the I2C master generates a STOP condition.

- 0 Slave mode. The I2C module is a slave and receives the serial clock from the master.
- 1 Master mode. The I2C module is a master and generates the serial clock on the SCL pin.

**TRX**

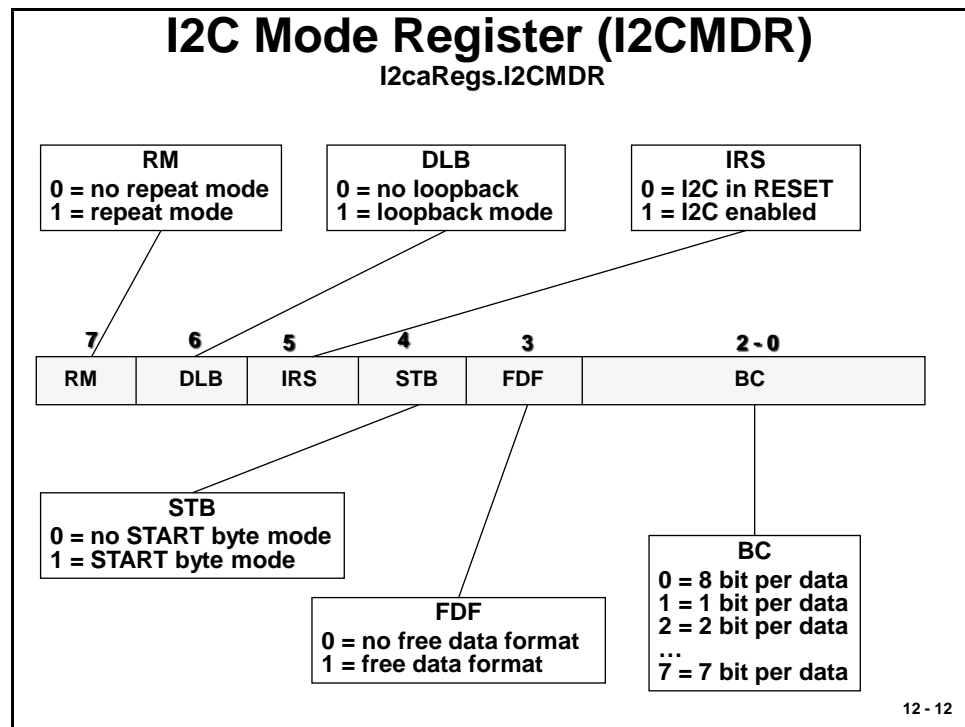
Transmitter mode bit. When relevant, TRX selects whether the I2C module is in the transmitter mode or the receiver mode.

- 0 Receiver mode. The I2C module is a receiver and receives data on the SDA pin.
- 1 Transmitter mode. The I2C module is a transmitter and transmits data on the SDA pin.

**XA**

Expanded address enable bit.

- 0 7-bit addressing mode (normal address mode). The I2C module transmits 7-bit slave addresses (from bits 6-0 of I2CSAR).
- 1 10-bit addressing mode (expanded address mode). The I2C module transmits 10-bit slave addresses (from bits 9-0 of I2CSAR).

**RM**

Repeat mode bit (only applicable when the I2C module is a master-transmitter). The RM, STT, and STP bits determine when the I2C module starts and stops data transmissions.

- 0 Non-repeat mode. The value in the data count register (I2CCNT) determines how many bytes are received / transmitted by the I2C module.

- 1 Repeat mode. A data byte is transmitted each time the I2CDXR register is written to (or until the transmit FIFO is empty when in FIFO mode) until the STP bit is manually set. The value of I2CCNT is ignored. The ARDY bit/interrupt can be used to determine when the I2CDXR (or FIFO) is ready for more data, or when the data has all been sent and the CPU is allowed to write to the STP bit.

**DLB**

Digital loopback mode bit.

- 0 Digital loopback mode is disabled.  
1 Digital loopback mode is enabled. For proper operation in this mode, the MST bit must be 1.

**IRS**

I2C module reset bit.

- 0 The I2C module is in reset/disabled. When this bit is cleared to 0, all status bits (in I2CSTR) are set to their default values.  
1 The I2C module is enabled. This has the effect of releasing the I2C bus if the I2C peripheral is holding it.

**STB**

START byte mode bit. This bit is only applicable when the I2C module is a master. As described in version 2.1 of the Philips Semiconductors I2C-bus specification, the START byte can be used to help a slave that needs extra time to detect a START condition. When the I2C module is a slave, it ignores a START byte from a master, regardless of the value of the STB bit.

- 0 The I2C module is not in the START byte mode.  
1 The I2C module is in the START byte mode.

**FDF**

Free data format mode bit.

- 0 Free data format mode is disabled. Transfers use the 7-/10-bit addressing format selected by the XA bit.  
1 Free data format mode is enabled.

**BC**

Bit count bits. BC defines the number of bits (1 to 8) in the next data byte that is to be received or transmitted by the I2C module. The number of bits selected with BC must match the data size of the other device. Notice that when BC = 000b, a data byte has 8 bits. BC does not affect address bytes, which always have 8 bits.

000	8 bits per data byte
001	1 bit per data byte
010	2 bits per data byte
011	3 bits per data byte
100	4 bits per data byte
101	5 bits per data byte
110	6 bits per data byte
111	7 bits per data byte

## I2C Interrupt Enable Register

I2C Interrupt Enable Register							
I2caRegs.I2CIER							
15	14	13	12	11	10	9	8
reserved	reserved	reserved	reserved	reserved	reserved	reserved	reserved
7	6	5	4	3	2	1	0
reserved	AAS	SCD	XRDY	RRDY	ARDY	NACK	AL

1 = enable interrupt source      0 = disable

AAS: an I2C - master has addressed the DSC as slave  
 SCD: Stop condition detected  
 XRDY: Transmit data ready for new data  
 RRDY: Receive data available  
 ARDY: I2C register access ready  
 NACK: No – Acknowledge received  
 AL: Arbitration lost during competition

12 - 13

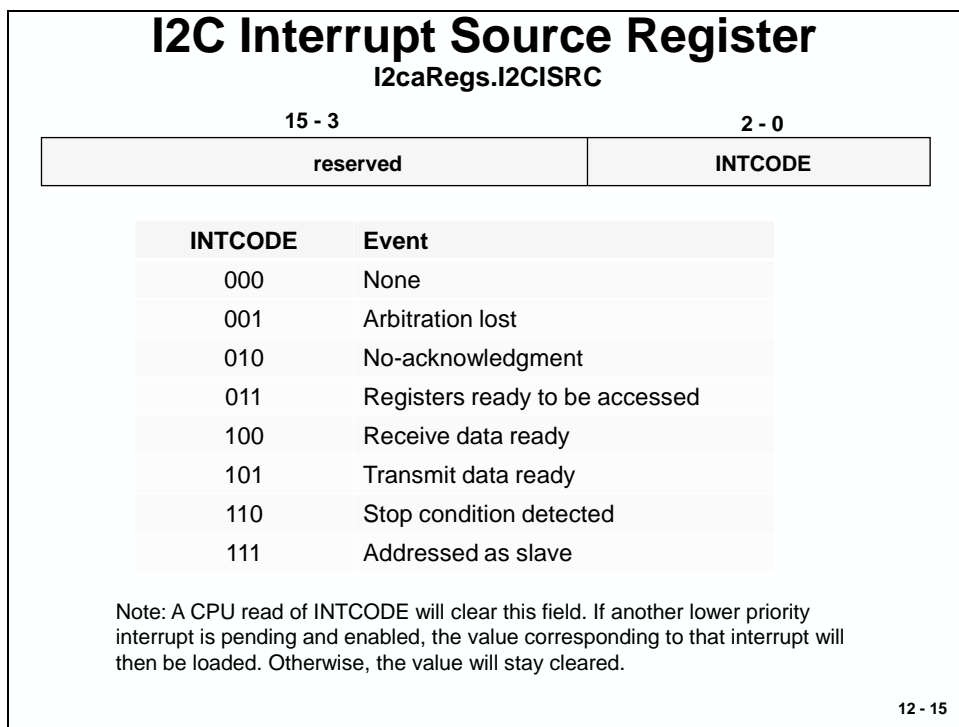
## I2C Status Register

I2C Status Register							
I2caRegs.I2CSTR							
15	14	13	12	11	10	9	8
reserved	SDIR	NACKSNT	BB	RSFULL	XSMT	AAS	AD0
7	6	5	4	3	2	1	0
reserved	reserved	SCD	XRDY	RRDY	ARDY	NACK	AL

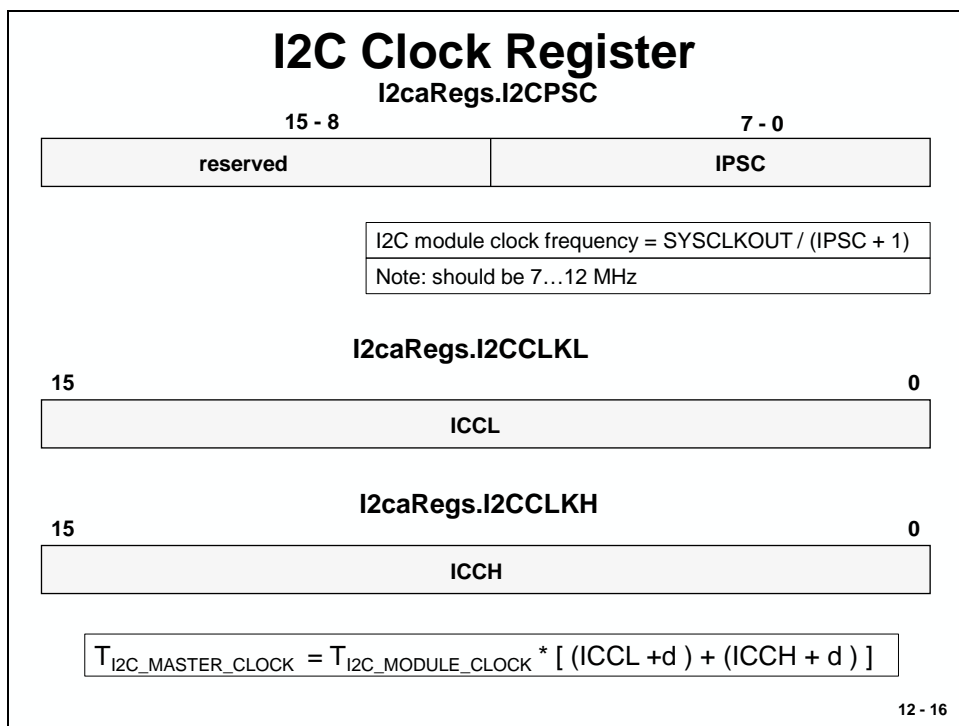
SDIR: 1 = I2C is addressed as a slave transmitter; 0 = as receiver  
 NACKSNT: 1 = a NACK was sent as receiver from I2C  
 BB: 1 = Bus is busy, a START has been sent  
 RSFULL: 1 = Overrun of the receiver detected  
 XSMT: 0 = underflow of transmit shift register detected  
 AAS: 1 = I2C was addressed as a slave  
 AD0: 1 = a general call (address 0) has been detected  
 SCD: 1 = a STOP condition has been detected  
 XRDY: 1 = Transmit register I2CDXR ready for new data  
 RRDY: 1 = Receive data available in I2CDRR  
 ARDY: 1 = a previous cycle has completed  
 NACK: 0 = ACK received; 1 = NACK received  
 AL: 1 = Arbitration lost during competition by 2 masters

12 - 14

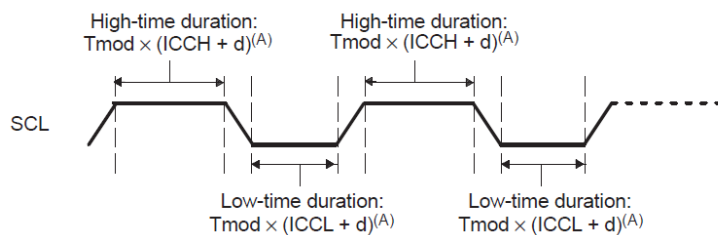
## I2C Interrupt Source Register



## I2C Clock Register



## I2C Master Clock



(A):

IPSC	Value for d
0	7
1	6
greater than 1	5

12 - 17

## I2C Slave Address Register

### I2C Slave Address Register

I2caRegs.I2CSAR

15 - 10	9 - 0
reserved	SAR

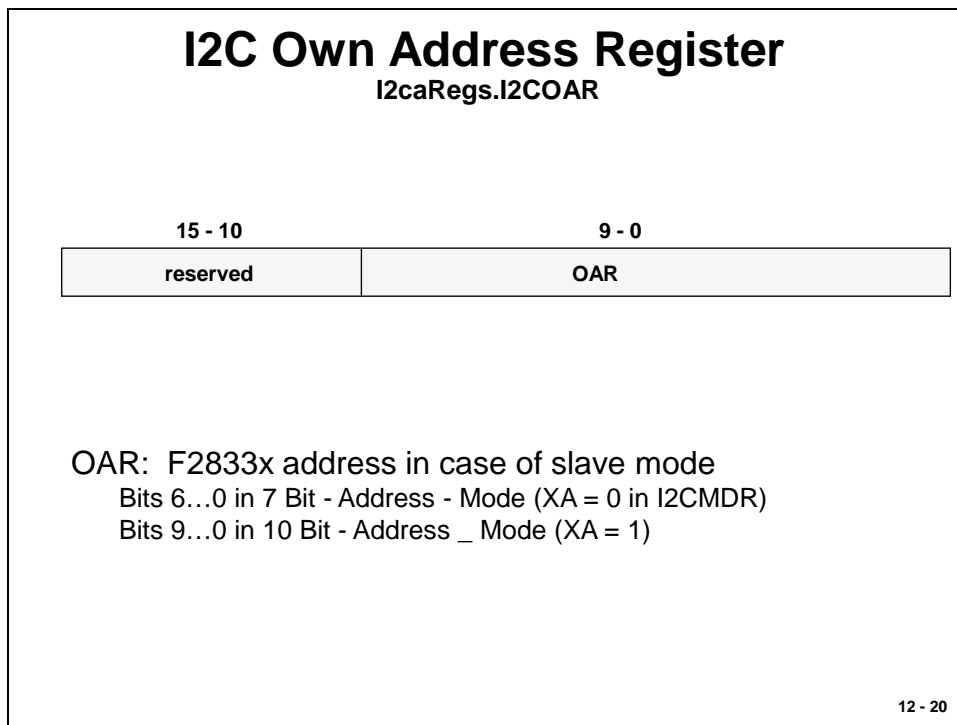
SAR: next slave address that will be transmitted

Bits 6...0 in 7 Bit - Address - Mode (XA = 0 in I2CMDR)

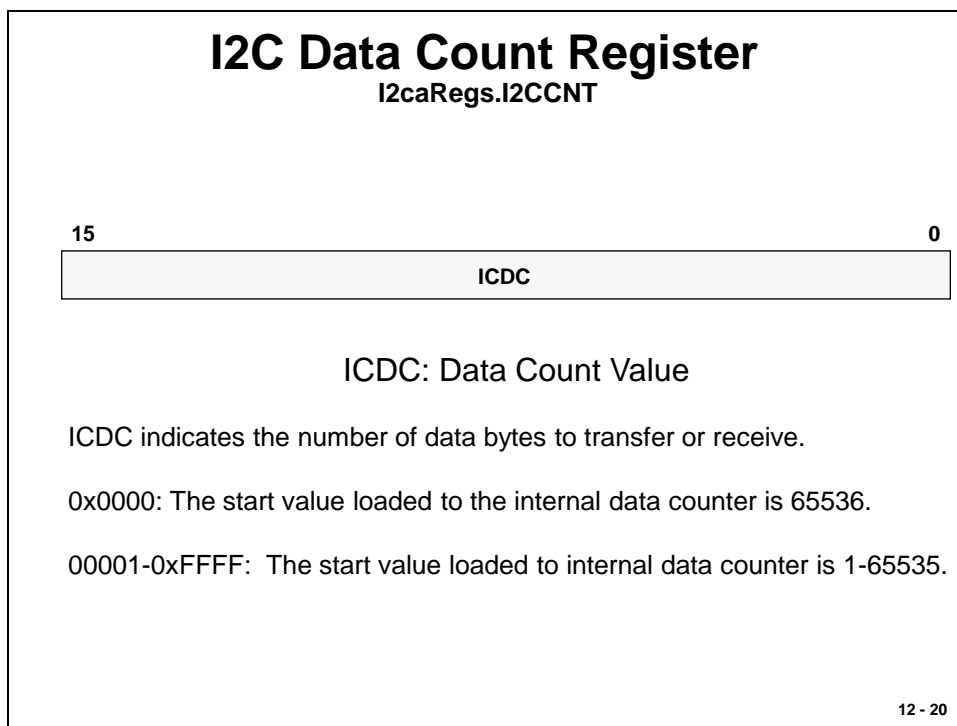
Bits 9...0 in 10 Bit - Address \_ Mode (XA = 1)

12 - 19

## I2C Own Address Register



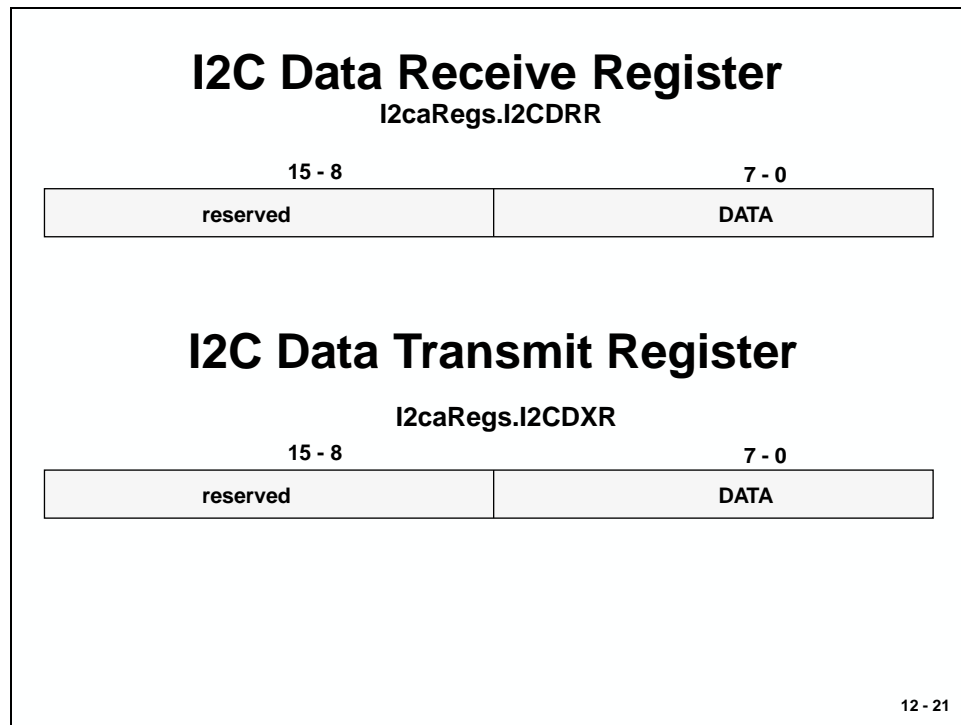
## I2C Data Count Register



---

## I2C Data Registers

To read or write data from / to I2C, we use the lower 8 bits of two data registers:



## I2C FIFO Buffers

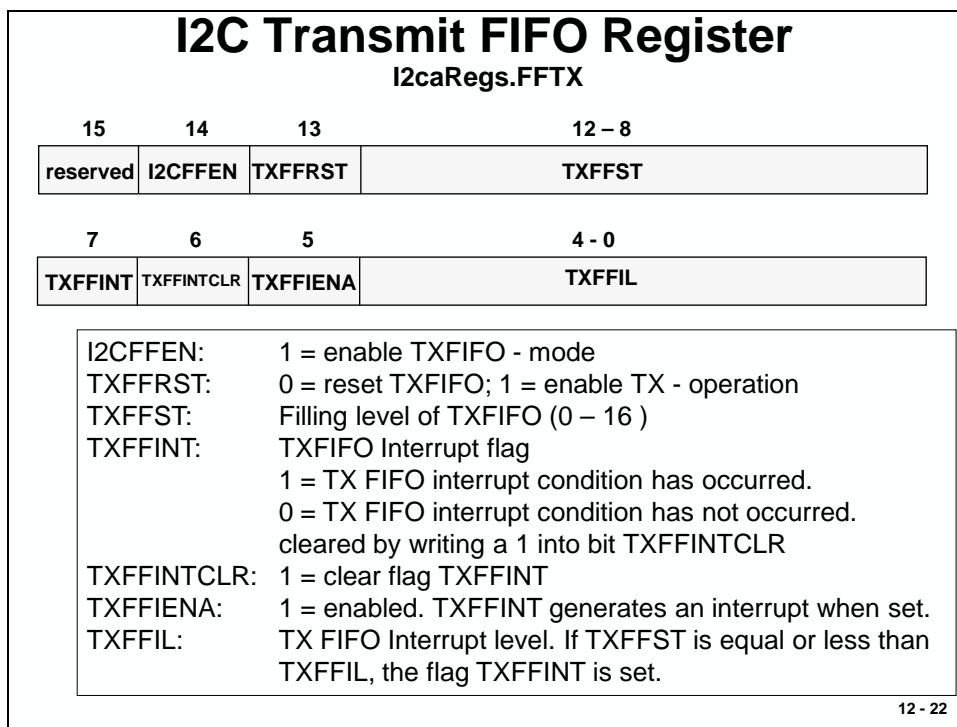
The I2C module of the F2833x is enhanced by a set of FIFO-buffers for register I2CDRR and I2CDXR. The FIFO-buffers, each of them 16 levels deep, can be used to buffer up to 16 characters, before they are transmitted into I2C (TXFIFO) or after they have been received (RXFIFO). This greatly reduces the workload for the CPU to service the I2C.

Note that the FIFO-units have no individual register names or address spaces; once the FIFOs are enabled, a repeated write into register I2CDXR will indirectly use this background transmit buffer to store the data. Values from this background FIFO are loaded into the foreground automatically as soon as register I2CDXR is ready for new data.

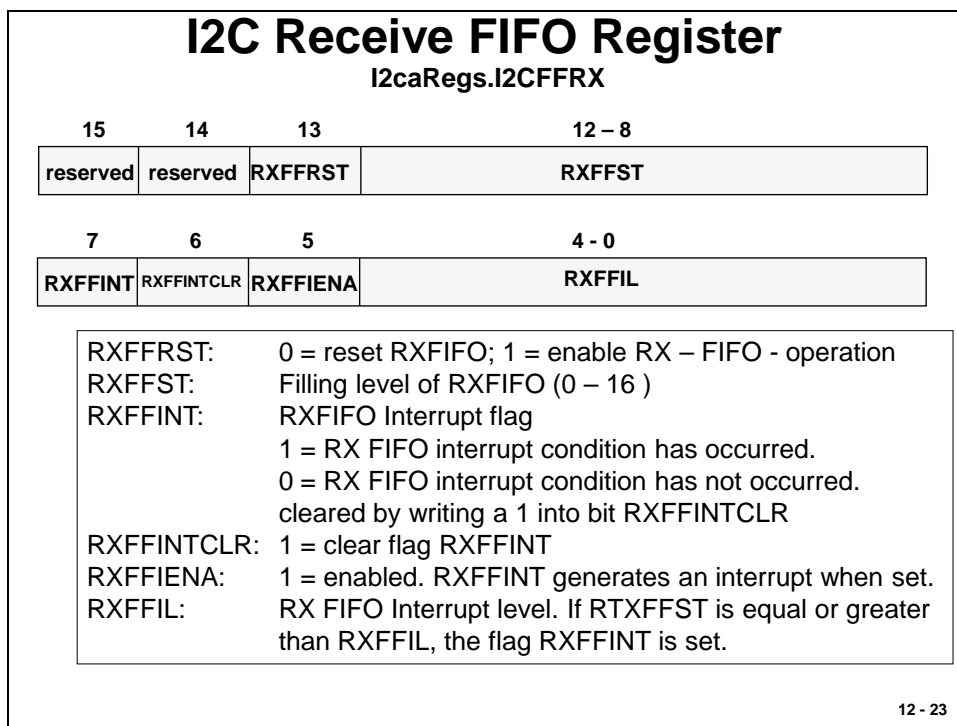
The same principle applies to the I2CDRR register for data, received by the I2C input channel. By defining a threshold for an interrupt request, when a certain number of FIFO entries are consumed, the number of interrupts for I2C can be reduced.

The FIFO-option is disabled by default. Since we will use the FIFO in lab exercise 12\_3, we will have to discuss the two control registers in the following two slides:

## I2C TX-FIFO Register



## I2C RX-FIFO Register



## Temperature Sensor TMP100

To exercise the I2C module of the F2833x we need to connect external I2C-device(s). The Peripheral Explorer Board is equipped with a Texas Instruments temperature sensor TMP100 (or TMP 100) - see datasheet literature number “SBOS231G”.

### Temperature Sensor TMP101

- **Digital Interface: I2C Serial 2-Wire**
- **Resolution: 9- to 12-Bits, User-Selectable**
  - 9 – Bit: 0.5 °C; 12 – Bit: 0.0625 °C
- **Accuracy:**
  - $\pm 2.0^{\circ}\text{C}$  from  $-25^{\circ}\text{C}$  to  $+85^{\circ}\text{C}$  (max)
  - $\pm 3.0^{\circ}\text{C}$  from  $-55^{\circ}\text{C}$  to  $+125^{\circ}\text{C}$  (max)
- **Low quiescent current of 45 $\mu\text{A}$ , 0.1 $\mu\text{A}$  Standby**
- **Power supply range: 2.7V to 5.5V**
- **Tiny SOT23-6 package**

12 - 25

The TMP100 and TMP101 are two-wire, serial output temperature sensors available in SOT23-6 packages. Requiring no external components, the TMP100 and TMP101 are capable of reading temperatures with a resolution of 0.0625°C. The TMP100 and TMP101 feature I2C interface compatibility, with the TMP100 allowing up to eight devices on one bus.

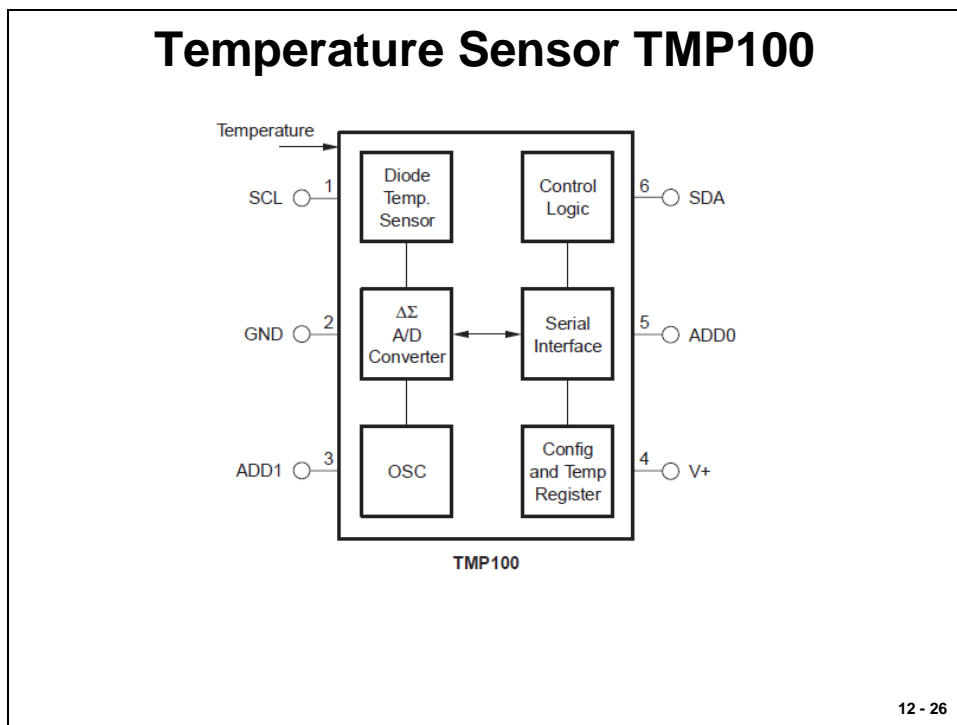
The TMP101 offers SMBus alert function with up to three devices per bus. The TMP100 and TMP101 are ideal for extended temperature measurement in a variety of communication, computer, consumer, environmental, industrial, and instrumentation applications.

The TMP100 and TMP101 are specified for operation over a temperature range of  $-55^{\circ}\text{C}$  to  $+125^{\circ}\text{C}$ .

The following Slide 12-26 shows the physical pin out of the device. Signals SCL and SDA are the I2C clock and data lines discussed above. Signal V+ is connected to +3.3V. Pins “ADD0” and “ADD1” are code pins to define the device slave address:

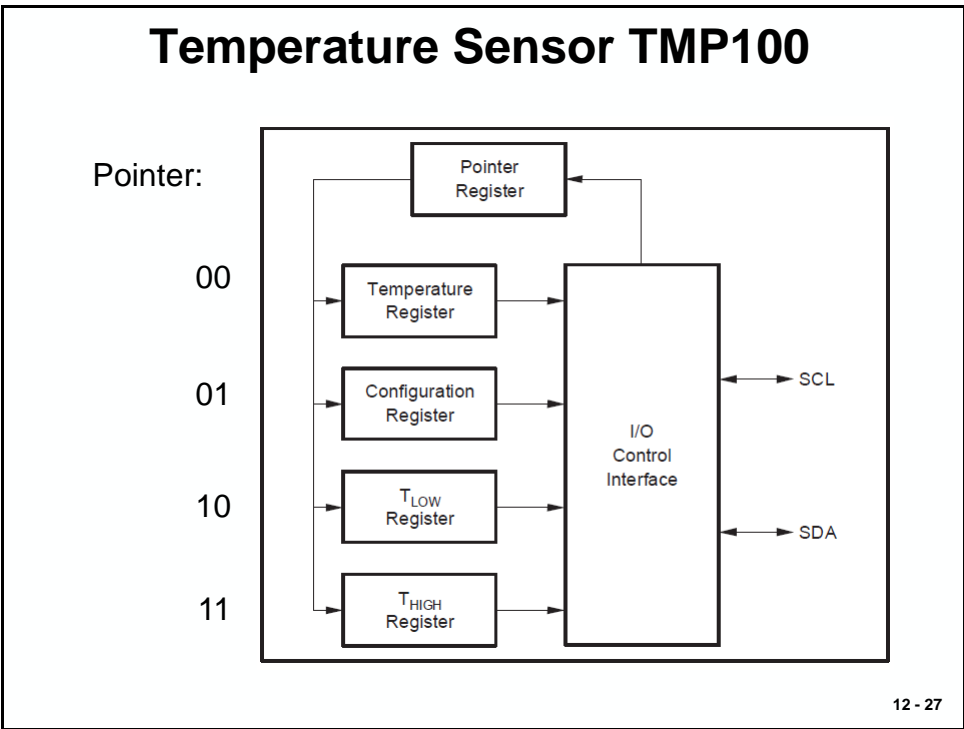
ADD1	ADD0	Slave address
0	0	0x48
0	Float	0x49
0	1	0x4A
1	0	0x4C
1	Float	0x4D
1	1	0x4E
Float	0	0x4B
Float	1	0x4F

At the Peripheral Explorer Board pins ADD0 and ADD1 are fixed to 0.



## TMP100 Register Structure

The TMP100 must be initialized by a set of 5 internal registers:



The 8-bit Pointer Register of the TMP100 and TMP101 is used to address a given data register. The Pointer Register uses the two LSBs to identify which of the data registers should respond to a read or write command.

The Pointer Register has the following layout:

P7	P6	P5	P4	P3	P2	P1	P0
0	0	0	0	0	0	Register -	Bits

Using the “Register-Bits”, one of the registers available in the TMP100 and TMP101 can be pre-selected. The Power-up Reset value of P1/P0 is 00.

P1	P0	Register
0	0	Temperature Register
0	1	Configuration Register
1	0	Low Temperature Threshold Register
1	1	High Temperature Threshold Register

## Temperature Register

The Temperature Register of the TMP100 or TMP101 is a 12-bit read-only register that stores the output of the most recent conversion. Two bytes must be read to obtain the data:

# Temperature Sensor TMP100

Table 3. Byte 1 of Temperature Register

D7	D6	D5	D4	D3	D2	D1	D0
T11	T10	T9	T8	T7	T6	T5	T4

Table 4. Byte 2 of Temperature Register

D7	D6	D5	D4	D3	D2	D1	D0
T3	T2	T1	T0	0	0	0	0

Table 5. Temperature Data Format

TEMPERATURE (°C)	DIGITAL OUTPUT (BINARY)	HEX
128	0111 1111 1111	7FF
127.9375	0111 1111 1111	7FF
100	0110 0100 0000	640
80	0101 0000 0000	500
75	0100 1011 0000	4B0
50	0011 0010 0000	320
25	0001 1001 0000	190
0.25	0000 0000 0100	004
0.0	0000 0000 0000	000
-0.25	1111 1111 1100	FFC
-25	1110 0111 0000	E70
-55	1100 1001 0000	C90
-128	1000 0000 0000	800

12 - 28

12 - 28

## Configuration Register

The Configuration Register is an 8-bit read/write register used to store bits that control the operational modes of the temperature sensor. Read/write operations are performed MSB first. The format of the Configuration Register for the TMP100 and TMP101 is shown in Slide 12-29.

TMP100 Configuration Register							
7	6	5	4	3	2	1	0
OS/ALERT	R1	R0	F1	F0	POL	TM	SD
<p>OS/ALERT:      write 1: single temperature conversion                           write 0: continuous temperature conversion                           read 1: temperature above THIGH                           read 0: temperature below TLOW</p> <p>R1, R0:        Resolution 9 bit (0,0) ... 12 bit (1,1)</p> <p>F1,F0:        activate ALERT after number of consecutive faults (1,2,4,6)</p> <p>POL:          Polarity of ALERT (0 or 1)</p> <p>TM:           write 0: Comparator Mode                           (ALERT stays active as long as condition is true)                           write 1: Interrupt Mode                           (ALERT is cleared by a read instruction of any reg)</p> <p>SD:           write 1: shutdown                           write 0: active mode</p>							
							12 - 29

The power-up/reset value of the Configuration Register is with all bits equal to 0.

## TMP100 Timing Diagrams

The I2C Bus Timing is based on different bus conditions:

### Bus Idle:

Both SDA and SCL lines remain HIGH.

### Start Data Transfer:

A change in the state of the SDA line, from HIGH to LOW, while the SCL line is HIGH, defines a START condition. Each data transfer is initiated with a START condition.

### Stop Data Transfer:

A change in the state of the SDA line from LOW to HIGH while the SCL line is HIGH defines a STOP condition. Each data transfer is terminated with a repeated START or STOP condition.

### Data Transfer:

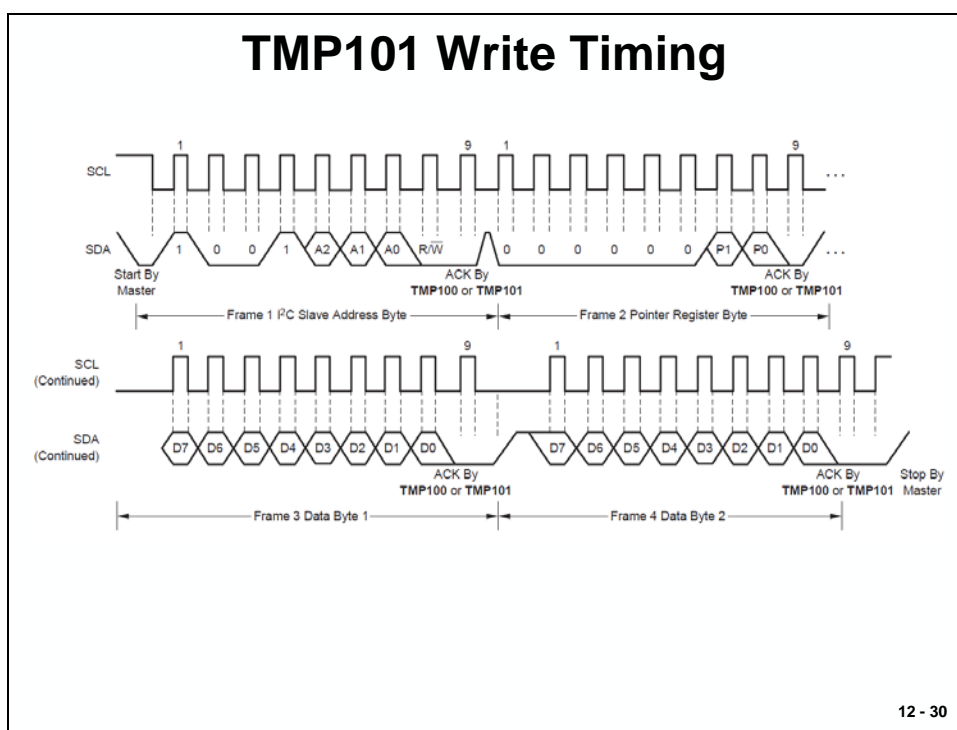
The number of data bytes transferred between a START and a STOP condition is not limited and is determined by the master device. The receiver acknowledges the transfer of data.

### Acknowledge and Not-Acknowledge:

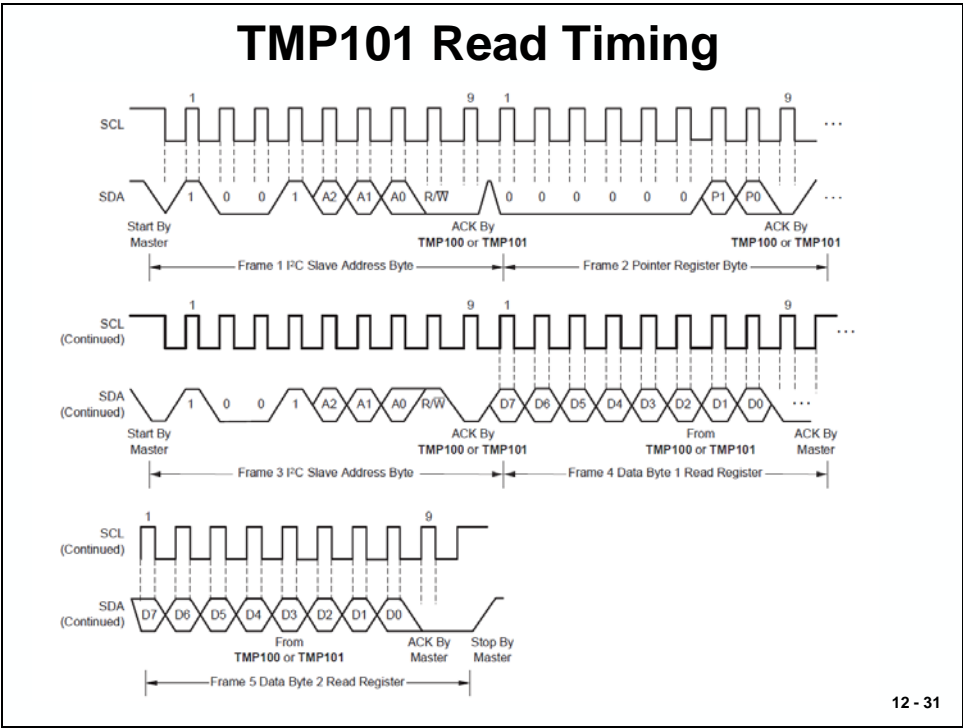
Each receiving device, when addressed, is obliged to generate an Acknowledge bit. A device that acknowledges must pull down the SDA line during the Acknowledge clock pulse in such a way that the SDA line is stable LOW during the HIGH period of the Acknowledge clock pulse. Setup and hold times must be taken into account. On a master receive, the termination of the data transfer can be signalled by the master generating a Not-Acknowledge (NACK) on the last byte that has been transmitted by the slave.

Note: Data books on I2C sometimes state that “the master does NOT acknowledge”. This means that “the master performs Not-Acknowledge (NACK)”, rather than skipping the acknowledge part of the cycle.

## TMP100 / TMP101 Write Timing



TMP100 / TMP101 Write Timing



# Lab Exercise 12\_1

## Preface

The Peripheral Explorer Board is equipped with an external temperature sensor TMP100 (device U9). During the early stages of this textbook the first version of the Peripheral Explorer Board was not equipped with such an I2C- device.

Here is the description what to do, in case if your Peripheral Explorer Board does not include a TMP101 or TMP100. Connect the following four pins of the TMP101 with wires to the headers of the Peripheral Explorer Board:

Pin TMP101	Header of Peripheral Explorer
<b>1: SCL</b>	<b>J15:1 (I2C - SCL)</b>
<b>2: GND</b>	<b>J12:3 (RS232 - GND)</b>
<b>4: V+</b>	<b>J12:2 (RS232 - V33)</b>
<b>6: SDA</b>	<b>J15:2(I2C - SDA)</b>

The two I2C-signals are multiplexed at GPIO32 (SDA) and GPIO33 (SCL). To guarantee the voltage levels for the two signals we need external pull-up - resistors of 4.7 kOhm between the signal line SCL and 3.3V and between SDA and 3.3V. Note: The F2833x is equipped with internal pull-up- resistors. However, their resistance is not low enough to guarantee the timing of an I2C-bit period.

## Objective

The objective of Lab 12\_1 is to initialize the I2C interface and to read the current temperature from the external device TMP100. For simplification we will use a watch window to monitor the current value of integer variable “temperature”. Note that the result 16-bit register of the TMP100 has 8 integer bits and 8 binary fraction bits; so if we display this value as I8Q8-number (type: int, Radix: Q8) we can immediately verify the temperature value.

## Procedure

### Open Files, Create Project File

1. Using Code Composer Studio, create a new project, called **Lab12.pjt** in C:\DSP2833x\_V4\Labs (or in another path that is accessible by you; ask your teacher or a technician for an appropriate location!).
2. A good point to start with is the source code of Lab6.c, which produces a hardware based time period using CPU core timer 0. Open file Lab6.c from C:\DSP2833x\_V4\Labs\Lab6 and save it as Lab12\_1.c in C:\DSP2833x\_V4\Labs\Lab12.

3. Define the size of the C system stack. In the project window, right click at project “Lab12” and select “Properties”. In category “C/C++ Build”, “C2000 Linker”, “Basic Options” set the C stack size to 0x400.

Link some of the source code files, provided by Texas Instruments, to the project:

4. In the C/C++ perspective, right click at project “Lab12” and select “**Link Files to Project**”. Go to folder “C:\tidcs\c28\dsp2833x\v131\DSP2833x\_headers\source” and link:

- **DSP2833x\_GlobalVariableDefs.c**

From C:\tidcs\c28\dsp2833x\v131\DSP2833x\_common\source link:

- **DSP2833x\_PieCtrl.c**
- **DSP2833x\_PieVect.c**
- **DSP2833x\_DefaultIsr.c**
- **DSP2833x\_CpuTimers.c**
- **DSP2833x\_SysCtrl.c**
- **DSP2833x\_CodeStartBranch.asm**
- **DSP2833x\_ADC\_cal.asm**
- **DSP2833x\_usDelay.asm**

From C:\tidcs\c28\dsp2833x\v131\DSP2833x\_headers\cmd link:

- **DSP2833x\_Headers\_nonBIOS.cmd**

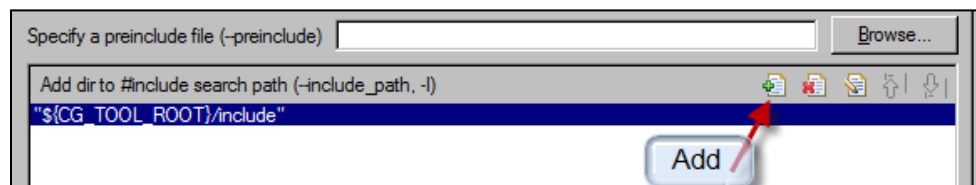
## Project Build Options

5. We have to extent the search path of the C-Compiler for include files. Right click at project “Lab12” and select “Properties”. Select “C/C++ Build”, “C2000 Compiler”, “Include Options”. In the box: “Add dir to #include search path”, add the following lines:

**C:\tidcs\C28\dsp2833x\v131\DSP2833x\_headers\include**

**C:\tidcs\c28\DSP2833x\v131\DSP2833x\_common\include**

Note: Use the “Add” Icon to add the new paths:



Close the Property Window by Clicking <OK>.

## Preliminary Test

6. So far, we have just created a new project “Lab12.pjt” with the same functionality as in Lab6. A good step would be to rebuild Lab12, load the code into the controller and verify the binary counter at LED’s LD1 to LD4 of the Peripheral Explorer Board.

The LEDs should be updated by the counter in 100 milliseconds time steps.

If not: Debug!

## Add TMP100 and I2C Initialization Code

7. Now let us add code to initialize the I2C and the TMP100.

The TMP100 is addressed as I2C-Slave. If pin ADDR0 is floating, its address is hexadecimal 0x49. The initialization is based on TMP100 internal registers with the following addresses:

- Temperature Register: 0
- Configuration Register: 1
- Temperature Low Register: 2
- Temperature High Register: 3

To allow a simple addressing of these registers, add the following macros at the beginning of Lab12\_1.c:

```
#define TMP100_SLAVE          0x48
#define POINTER_TEMPERATURE   0
#define POINTER_CONFIGURATION 1
#define POINTER_T_LOW         2
#define POINTER_T_HIGH        3
```

8. At the beginning of “main()”, remove variable “counter”. Define a global integer variable “temperature”. Note: it is good software practice to write out the word “temperature” in full, rather than using the abbreviation “temp”. This is because the abbreviation could mean either “temporary” or “temperature”.
9. In local function “Gpio\_select()” change register GPBMUX1 to enable lines GPIO32 and GPIO33 for I2C operation. In register GPBPUD enable the internal pull-up - resistors for lines GPIO32 and GPIO33. In register GPBQSEL1 set lines GPIO32 and GPIO33 to asynchronous input.
10. In main, after the function call of “Gpio\_select()”, add a function call of a new function “I2CA\_Init()”.
11. At the end of “main()”, add the definition of the new function “I2CA\_Init()” with void both as input and return parameter type. In “I2CA\_Init()” perform the following:
- Reset the I2C-module (Register I2CMDR, bit IRS)
  - Set the slave address register to 0x49 (Register I2CSAR)
  - Initialize the I2C module clock to 10MHz. If SYSCLKOUT is 150 MHz, set Register I2CPSC to 14:

$$10MHz = \frac{SYSCLKOUT}{(PSC + 1)};$$

- Set low and high phase of the I2C-clock signal to 50% each. As an example, we will use an I2C-clock frequency of 50 kHz (clock period = 20μs).

$$20\mu s = \frac{(14 + 1)[(ICCL + 5) + (ICCH + 5)]}{150MHz}$$

The equation above results in ICCL = ICCH = 95. Initialize registers I2CCLKL and I2CCLKH accordingly.

- Finally take the I2C module out of reset (Register I2CMDR bit IRS).
  - At the beginning of “Lab12\_1.c” add a prototype for the new local function “I2CA\_Init()”.
12. In the endless while(1)-loop of function "main()", remove all lines which are related to variable “counter” and to the monitoring with LEDs LD1 to LD4.
  13. After the watchdog service code lines in the while(1)-loop of “main()”, add code to read the current temperature from TMP100:
    - Set register “**I2CCNT**” to 2 to read a 2 byte temperature information from TMP100
    - Initialize register “**I2CMDR**”:
      - Bit15 = 0; no NACK in receiver mode
      - Bit14 = 1; FREE on emulation halt
      - Bit13 = 1; STT generate START
      - Bit12 = 0; reserved
      - Bit11 = 1; STP generate STOP
      - Bit10 = 1; MST master mode
      - Bit9 = 0; TRX master-receiver mode
      - Bit8 = 0; XA 7-bit address mode
      - Bit7 = 0; RM non-repeat mode, I2CCNT determines # of bytes
      - Bit6 = 0; DLB no loopback mode
      - Bit5 = 1; IRS I2C module enabled
      - Bit4 = 0; STB no start byte mode
      - Bit3 = 0; FDF no free data format
      - Bit2...0 = 0; BC 8 bit per data byte
  14. Install a wait loop until the 1<sup>st</sup> byte has been received from TMP100:
 

```
while(I2caRegs.I2CSTR.bit.RRDY == 0);
```
  15. Read the upper 8 bits of temperature:
 

```
temperature = I2caRegs.I2CDRR << 8;
```
  16. Wait for the 2<sup>nd</sup> byte and add it as the 8 lower bits to temperature:
 

```
while(I2caRegs.I2CSTR.bit.RRDY == 0);  
temperature += I2caRegs.I2CDRR;
```

## Build, Load and Run

17. Click the “Rebuild Active Project ” button or perform:

**Project → Rebuild All (Alt +B)**

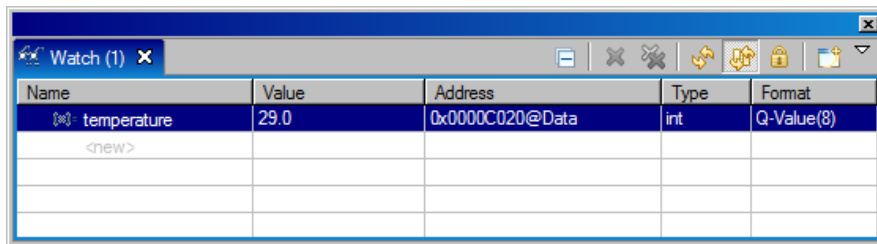
and watch the tools run in the build output window. If you get errors or warnings debug as necessary.

18. Load the output file in the debugger session:

**Target → Debug Active Project**

and switch into the “Debug” perspective.

19. Verify that in the debug perspective the window of the source code “Lab12\_1.c” is high-lighted and that the blue arrow for the current Program Counter position is placed under the line “void main(void)”.
20. Open a watch window and enter variable “temperature”. With a left mouse click into column “Format”, select “Q-Value(8)”. Activate “Continuous Refresh” button in the Watch Window.



Name	Value	Address	Type	Format
temperature	29.0	0x0000C020@Data	int	Q-Value(8)
<new>				

Now start a “Real Time Run”:

**→ Scripts → Realtime Emulation Control → Run\_Realtime\_with\_Restart**

The Variable “temperature” should display the current ambient temperature with a resolution of 0.5 °C (the example above shows 29.0 °C).

21. Stop the code- execution:

**→ Scripts → Realtime Emulation Control → Full Halt**

## Lab Exercise 12\_2

### Objective

In Lab12\_1 we used the TMP100 in a basic scenario with a resolution of 9 bits (or 0.5 °C) only. However, the TMP100 is able to operate with a resolution of 12 bits (or 1/16 °C). This high resolution must be initialized in the configuration register of the TMP100. This is the task for Lab12\_2.

### Procedure

#### Open Project, Modify Source File

1. If not still open from Lab12\_1, re-open project Lab12.pjt in C:\DSP2833x\_V4\Labs.
2. Open file “Lab12\_1.c” and save it as “Lab12\_2.c”
3. Exclude file “Lab12\_1.c” from build. Use a right mouse click at file “Lab12\_1.c”, and enable “Exclude File(s) from Build”.
4. In function main, after the function call of “I2CA\_Init()”, add I2C-code to address the configuration register of the TMP100:
  - Set register “**I2CCNT**” to 2 to send a 2-byte command (configuration register address, followed by configuration data) to TMP100.
  - Load register “**I2CDXR**” with the configuration register address:  
**I2caRegs.I2CDXR = POINTER\_CONFIGURATION;**
  - Initialize register “**I2CMDR**”:
    - Bit15 = 0; no NACK in receiver mode
    - Bit14 = 1; FREE on emulation halt
    - Bit13 = 1; STT generate START
    - Bit12 = 0; reserved
    - Bit11 = 1; STP generate STOP
    - Bit10 = 1; MST master mode
    - Bit9 = 1; TRX master-transmitter mode
    - Bit8 = 0; XA 7-bit address mode
    - Bit7 = 0; RM non-repeat mode, I2CCNT defines # of bytes
    - Bit6 = 0; DLB no loopback mode
    - Bit5 = 1; IRS I2C module enabled
    - Bit4 = 0; STB no start byte mode
    - Bit3 = 0; FDF no free data format
    - Bit2...0 = 0; BC 8 bit per data byte
  - Install a wait loop until the 1<sup>st</sup> byte has been transmitted to TMP100:  
**while(I2caRegs.I2CSTR.bit.XRDY == 0);**

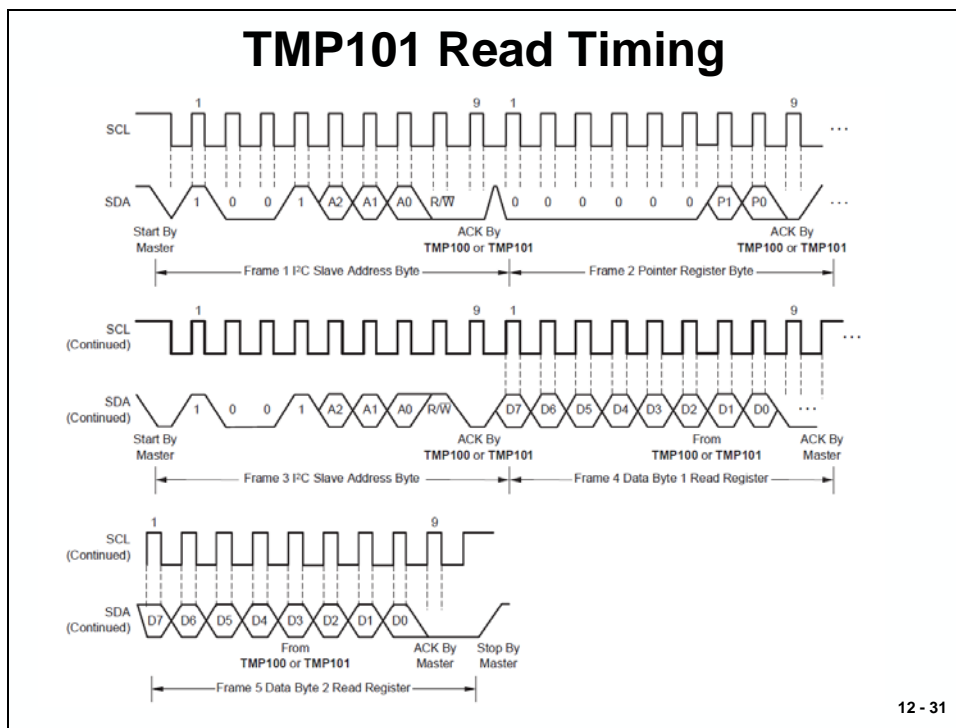
- Load register “**I2CDXR**” with the configuration data (0x60, see Slide 12-29) to initialize the temperature measurement with 12-bit resolution.
- Wait for the successful generation of the stop-condition:

```
while(I2caRegs.I2CSTR.bit.SCD == 0);
```

- Clear the stop condition flag:

```
I2caRegs.I2CSTR.bit.SCD = 1;
```

5. In the endless while(1)- loop of “main()” we have to change the code to read the TMP100 temperature register. According to the “read temperature” time diagram (Slide 12-31) we have to generate a 5-byte I2C frame (slave address, temperature register address, slave address, read temperature high, read temperature low). Note that there are two “Start By Master” conditions in this sequence. Also, we have to transmit the first two bytes as Master-Transmitter and then to switch into Master-Receiver-Mode.



Whilst the second half of the required code is identical to the code from Lab12\_1, we have to add the code to generate byte 1 and 2 of diagram 12-31. In the while(1)-loop before the line “I2caRegs.I2CCNT = 2”, add:

```
I2caRegs.I2CCNT = 1; // 1 byte message  
I2caRegs.I2CDXR = POINTER_TEMPERATURE;  
I2caRegs.I2CMDR.all = 0x6620; // master-receiver, START, STOP  
while(I2caRegs.I2CSTR.bit.ARDY == 0); // wait for STOP condition
```

## Build, Load and Run

6. Click the “Rebuild Active Project ” button or perform:

**Project → Rebuild All (Alt +B)**

and watch the tools run in the build output window. If you get errors or warnings debug as necessary.

7. Load the output file in the debugger session:

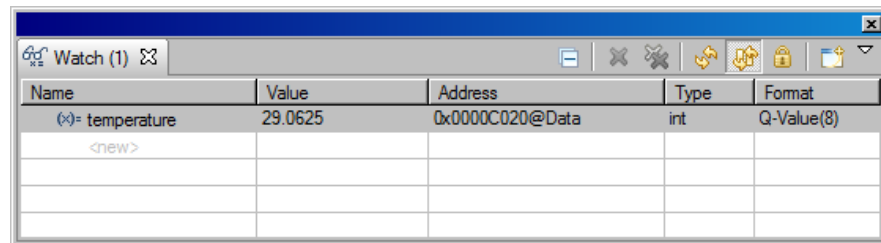
**Target → Debug Active Project**

and switch into the “Debug” perspective.

8. Verify that in the debug perspective the window of the source code “Lab12\_2.c” is high-lighted and that the blue arrow for the current Program Counter position is placed under the line “void main(void)”.
9. Perform a real time run.

**Target → Run**

10. Open a watch window and enter variable “temperature”. With a left mouse click into column “Format”, select “Q-Value(8)”. Activate “Continuous Refresh” button in the Watch Window.



Now start a “Real Time Run”:

**→ Scripts → Realtime Emulation Control → Run\_Realtime\_with\_Restart**

Variable “temperature” should display the current ambient temperature with a resolution of 1/16 °C (the example above shows 29.0625 °C).

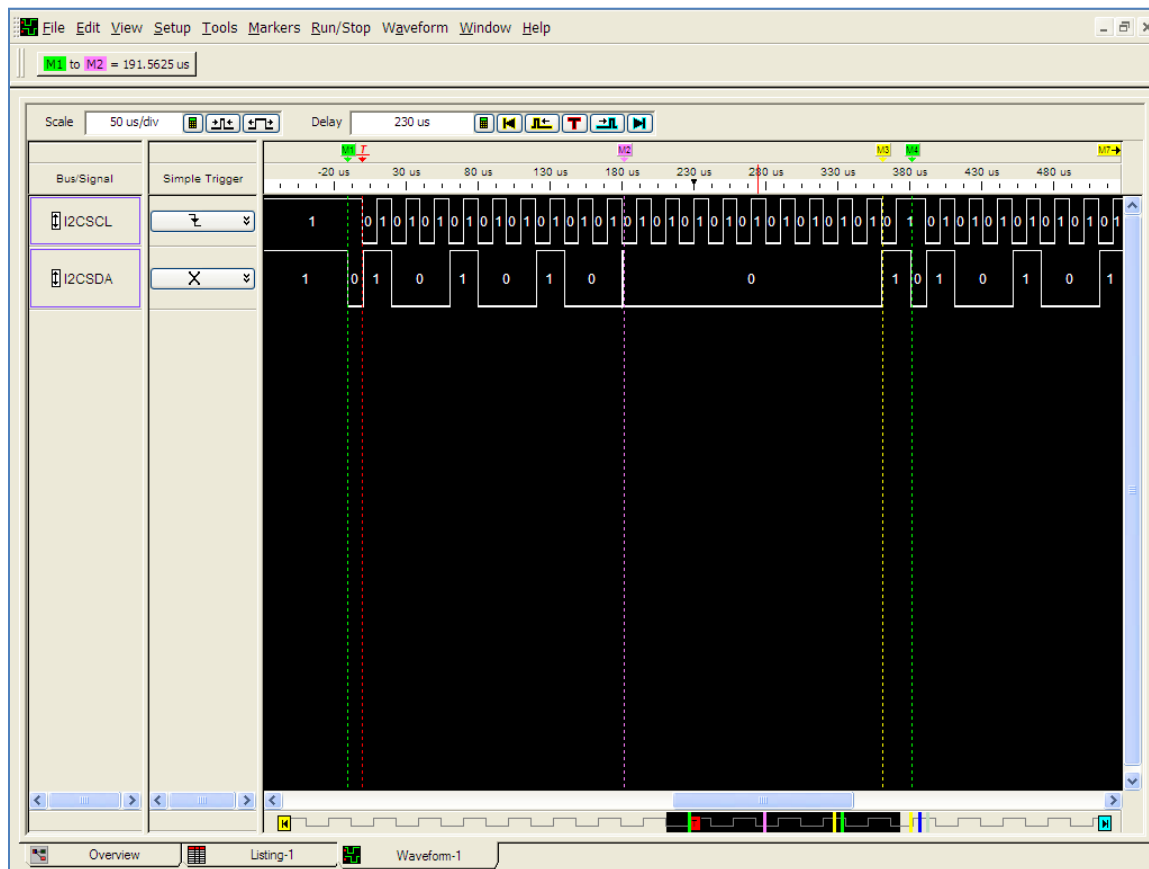
11. Stop the code - execution:

**→ Scripts → Realtime Emulation Control → Full Halt**

## Troubleshooting

If your variable “temperature” does not show correct numbers but the code is running as expected, then it might be useful to measure the signals SCL and SDA with an oscilloscope or logic analyzer.

The following image is a screenshot of a logic analyzer measurement of bytes 1 and 2 of an I2C-frame “Read -Timing” according to Slide 12-31 (see above) after the 1<sup>st</sup> START-Condition.



What follows is a description of the screenshot above from left to right:

- M1 (green marker): START Condition
- M2 (pink marker): 100100100  
 | | | | | | | | \_\_\_\_ ACK by TMP101  
 | | | | | | | | \_\_\_\_ Write  
 | | | | | | | | \_\_\_\_ device address 0x49
- M3 (yellow marker): 00000000  
 | | | | | | | | \_\_\_\_ ACK by TMP101  
 | | | | | | | | \_\_\_\_ pointer to temperature register: 0x00
- M4 (green marker): 2<sup>nd</sup> START Condition



## Lab Exercise 12\_3

### Objective

In Lab12\_1 and Lab12\_2 we used the TMP100 with a temperature resolution of 9 bits (or 0.5 °C) or 12 bits (1/16 °C). For the I<sup>2</sup>C-communication we installed a non FIFO data transmission, which leads to an increasing CPU load, especially when one would use a more demanding slave device, such as an EEPROM, ADC or DAC. To reduce the CPU load we should try to setup the FIFO-buffered operating mode of the I<sup>2</sup>C-interface. This is the objective of Lab12\_3.

### Procedure

#### Open Project, Modify Source File

1. If not still open from Lab12\_2, re-open project Lab12.pjt in C:\DSP2833x\_V4\Labs.
2. Open file “Lab12\_2.c” and save it as “Lab12\_3.c”
3. Exclude file “Lab12\_2.c” from build. Use a right mouse click at file “Lab12\_2.c”, and enable “Exclude File(s) from Build”.
4. First we have to change function “I2CA\_Init()”. Add new code to initialize registers I2CFFTX and I2CFFRX at the end of this function directly in front of the last code line to take the I2C out of reset:
  - For register I2CFFTX:
    - First reset the whole register to zero.
    - Next, set the transmit interrupt level (bit field TXFFIL) to zero.
    - Enable the FIFOs (bit I2CFFEN).
    - Enable the FIFO-transmit support (bit TXFFRST)
  - For register I2CFFRX:
    - First reset the whole register to zero.
    - Next, set the receive interrupt level (bit field RXFFIL) to 2, because we will receive a 2 byte temperature message from the TMP100.
    - Enable the FIFO-receiver support (bit RXFFRST)
5. Change the code in function “main()”. Before we enter the endless while(1)-loop, we already have some lines to address the TMP100 configuration register. This command consists of a 2-byte message from the F2833x to the TMP100. In Lab12\_2 we first wrote byte “POINTER\_CONFIGURATION” into register I2CDXR, then we waited until the first byte had been transmitted, before we wrote the next byte “0x60” (12-bit resolution mode) into register I2CDXR. For now, since we have enabled the transmit FIFO, there is no need to wait. We can write the two bytes directly one after another:

**I2caRegs.I2CCNT = 2;**

```

I2caRegs.I2CDXR = POINTER_CONFIGURATION;
I2caRegs.I2CDXR = 0x60;
I2caRegs.I2CMDR.all = 0x6E20;

```

6. In Lab12\_2 we read the temperature value from TMP100 in a 2-byte sequence at the end of the while(1)-loop. First we waited until the first byte was received (register I2CSTR bit RRDY), then we copied the information into variable “temperature” and finally we waited for another RRDY flag before we read the remaining byte and added it to “temperature”. For the new lab 12\_3 we initialized the receive FIFO to set the interrupt flag “RXFFINT” after 2 bytes have been received. Using this new flag we can simplify the wait construction to a single line and read the two temperature bytes directly one after another:

```

while(I2caRegs.I2CFFRX.bit.RXFFINT == 0);
I2caRegs.I2CFFRX.bit.RXFFINTCLR = 1;
temperature = I2caRegs.I2CDRR << 8; //read upper 8 Bit (integers)
temperature += I2caRegs.I2CDRR;      //add lower 8 Bit (fractions)

```

## Build, Load and Run

7. Click the “Rebuild Active Project ” button or perform:

**Project → Rebuild All (Alt +B)**

8. Load the output file in the debugger session:

**Target → Debug Active Project**

and switch into the “Debug” perspective. Verify that in the debug perspective the window of the source code “Lab12\_3.c” is high-lighted and that the blue arrow for the current Program Counter position is placed under the line “void main(void)”.

9. Perform a real time run.

**→ Scripts → RealTime Emulation Control → Run\_Realtime\_with\_Restart**

10. Open a watch window and enter variable “temperature”. With a left mouse click into column “Format”, select “Q-Value(8)”. Activate “Continuous Refresh” button in the Watch Window.

Name	Value	Address	Type	Format	
(x)= temperature	27.625	0x0000C020@Data	int	Q-Value(8)	
<new>					

Variable “temperature” should display the current ambient temperature with a resolution of 1/16 °C (the example above shows 27.625 °C).

11. Stop the code - execution:  
**→ Scripts → Realtime Emulation Control → Full Halt**

## Lab Exercise 12\_4

### Objective

As final laboratory exercise we will use the I2C-interrupt system to start “follow-up” - activities. You might have noticed that in “Lab12\_1.c” to “Lab12\_3.c” we used while-loops to wait until the I2C-interface had finished previous parts of a data frame. This was simple and easy; but we wasted CPU performance with this technique. Now we will activate interrupt services to replace such while-loops.

The I2C interface has two groups of interrupts, (1) basic interrupts, described in Slide 12-13 and (2) FIFO-interrupts. Basic Interrupts are wired to Peripheral Interrupt Expansion (PIE) 8.1; FIFO - Interrupts are wired to PIE 8.2

### Procedure

#### Open Project, Modify Source File

1. If not still open from Lab12\_3, re-open project Lab12.pjt in C:\DSP2833x\_V4\Labs.
2. Open file “Lab12\_3.c” and save it as “Lab12\_4.c”
3. Exclude file “Lab12\_3.c” from build. Use a right mouse click at file “Lab12\_3.c”, and enable “Exclude File(s) from Build”.
4. Edit file “Lab12\_4.c”. First we have to change function “I2CA\_Init()”. Since we will use the RXFIFO - interrupt after receiving two temperature bytes from the TMP100, we have to enable this interrupt source. Add the following line:

**I2caRegs.I2CFFRX.bit.RXFFIENA = 1;**

As a basic I2C-interrupt we will use the “Access Ready” - signal (ARDY), which is generated, when the first two bytes of the “TMP100 Read Timing” I2C - data frame (see Slide 12-31) are transmitted. Add the following line:

**I2caRegs.I2CIER.bit.ARDY = 1;**

5. In function “main()”, before we enter the endless while(1)-loop we have to enable two more PIE - interrupt lines for I2C-basic (8.1) and I2C-receiver-FIFO (8.2):

**PieCtrlRegs.PIEIER8.bit.INTx1 = 1;           // i2c - basic**

**PieCtrlRegs.PIEIER8.bit.INTx2 = 1;           // i2c - FIFO**

Also, the register IER must now allow lines INT1 and INT8:

**IER |=0x81;**

In Lab12\_3.c we used only one interrupt source, CPU-Timer 0. Now we have three, which requires that we load two more addresses of interrupt service routines into the PieVectTable. At the appropriate spot in your code, add:

```
PieVectTable.I2CINT2A = &i2c_fifo_isr;
PieVectTable.I2CINT1A = &i2c_basic_isr;
```

6. Change the type of variable “temperature” from a local variable in “main()” to a global variable.
7. At the beginning of “Lab12\_4.c”, add two prototypes for new interrupt service routines:

```
interrupt void i2c_fifo_isr(void);
interrupt void i2c_basic_isr(void);
```

8. At the end of “Lab12\_4.c” add a new interrupt function “i2c\_fifo\_isr()”. There are two possible interrupt sources, a receiver FIFO-level and a transmitter FIFO-level interrupt. We will use the receiver FIFO only. However, it is good practice to verify which one of the two sources is active. In case the receiver interrupt is active, we will find bit “RXFFINT” is set. We will use this bit in an if-condition to perform the following activities:

- Read two times the I2CDRR - register to get the temperature values
- Clear the RXFFINT - flag by setting bit RXFFINTCLR
- Acknowledge the PIE - Interrupt of PIE - group 8.

The code in this interrupt service should look like:

```
unsigned int i;
if (I2caRegs.I2CFFRX.bit.RXFFINT == 1) // RX-FIFO - interrupt
{
    i = I2caRegs.I2CDRR << 8; // read upper 8 bit (integers)
    i += I2caRegs.I2CDRR; // add lower 8 bit (fractions)
    temperature = i; // update temperature
    I2caRegs.I2CFFRX.bit.RXFFINTCLR = 1; // clear ISR
}
PieCtrlRegs.PIEACK.all = PIEACK_GROUP8;
```

9. At the end of “Lab12\_4.c” add a new interrupt function “i2c\_basic\_isr()”. This function is shared between all basic I2C - interrupt sources. Register “I2CISRC” (see Slide 12-15) contains a code number for the current source of the interrupt. Although we have enabled only 1 of these basic interrupts (ARDY), it is good practice and will be very important later, when you enable more than one basic source, to make a local copy of this register. The reason is that the first read of this register will clear it automatically.

In Lab12\_4 we wait for ARDY (code number 3, see Slide 12-15), which is set after the first two bytes of the “TMP100 Read Timing” (Slide 12-31) are transmitted. At this moment we have to switch I2C from Master-Transmitter into Master-Receiver via register I2CMDR. Since the TMP100 will send two temperature bytes, we also set register I2CCNT = 2.

The whole body of function “i2c\_basic\_isr()” should look like:

```
unsigned int IntSource;
IntSource = I2caRegs.I2CISRC.all;
if (IntSource == 3) // ARDY was source of int
```

```

{
    I2caRegs.I2CCNT = 2;      // read 2 byte temperature
    I2caRegs.I2CMDR.all = 0x6C20; // Master-Receiver-Mode
}
PieCtrlRegs.PIEACK.all = PIEACK_GROUP8;

```

10. In the endless while(1) - loop of “main()”, remove the wait construction, which waits until bit “ARDY” is set.

After that line, remove also the code to initialize registers I2CCNT and I2CMDR. We moved this code in procedure step 9 into interrupt service routine “i2c\_basic\_isr()”, which is now called automatically by ARDY.

Remove also the following lines, where we waited until bit “RXFFINT” was set and the lines to read the temperature values. We moved this code in procedure step 8 into interrupt service routine “i2c\_fifo\_isr()”. This function is now called automatically after two bytes have been received (RXFFINT).

## Build, Load and Run

11. Click the “Rebuild Active Project ” button or perform:

**→ Project → Rebuild All (Alt +B)**

12. Load the output file in the debugger session:

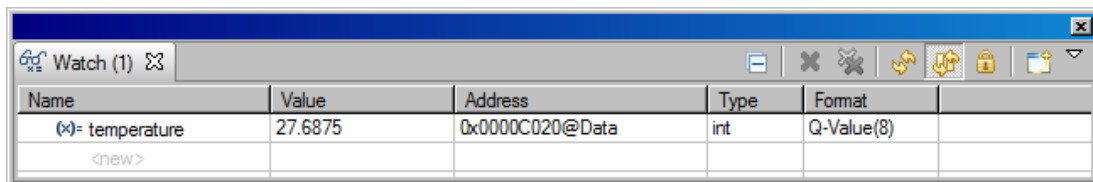
**→ Target → Debug Active Project**

and switch into the “Debug” perspective. Verify that in the debug perspective the window of the source code “Lab12\_3.c” is high-lighted and that the blue arrow for the current Program Counter position is placed under the line “void main(void)”.

13. Perform a real time run.

**→ Scripts → RealTime Emulation Control → Run\_Realtime\_with\_Restart**

14. Open a watch window and enter variable “temperature”. With a left mouse click into column “Format”, select “Q-Value(8)”. Activate “Continuous Refresh” button in the Watch Window.



Variable “temperature” should display the current ambient temperature with a resolution of 1/16 °C (the example above shows 27.6875 °C).

15. Stop the code - execution:

**→ Scripts → Realtime Emulation Control → Full Halt**