UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

SE 423: Introduction to Mechatronics

Lecture 4:

Marius Juston

Monday, February 2nd, 2026

Reminders

Office Hours:

- **Monday:** 4-6 PM, Neel
- **Tuesday:** 11-1 PM, Lakshmi
- **Tuesday:** 1-3 PM, Samuel
- **Tuesday:** 2-5 PM, Dan & Marius

Lab: Starting Lab 2 this week. This is also a 1-week lab!

Homeworks (This week):

- All check-offs for homework 1 are due **February 3, 5PM (The end of office hours)**
- Answers and code submissions for homework 1 are due **February 4, 9AM**

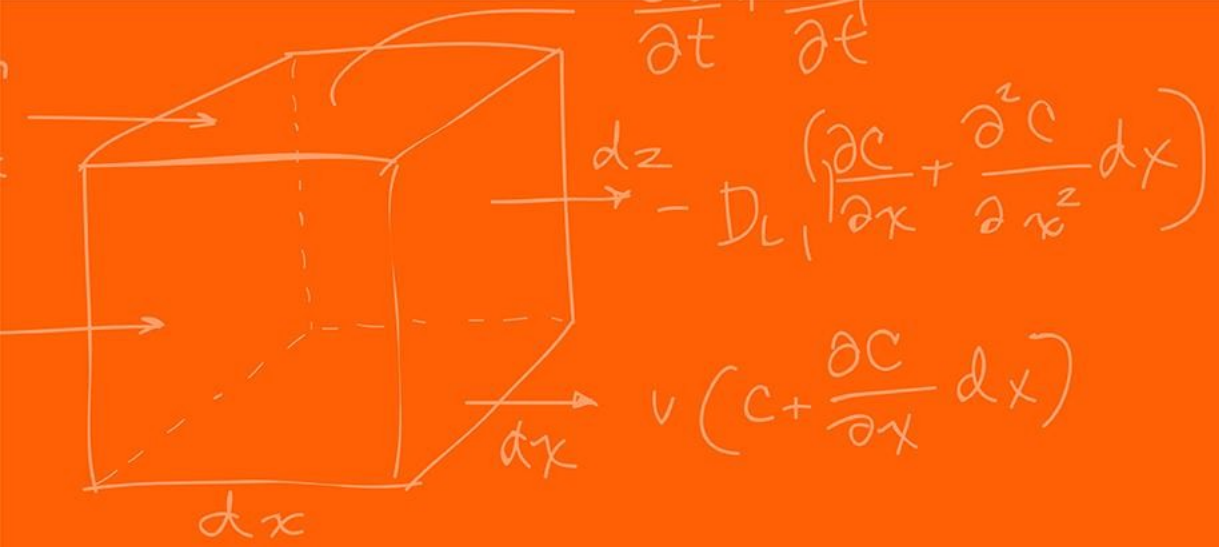
LabVIEW (This week):

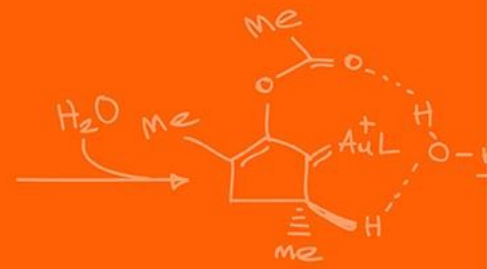
- All check-offs for LabVIEW 1 are due **February 5, 5PM**

FIRST ASSIGNMENTS ARE DUE THIS WEEK!

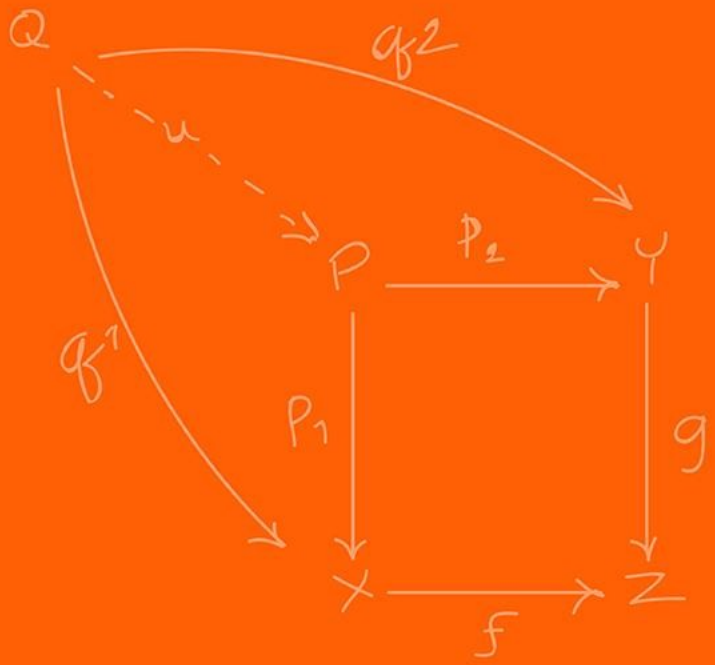
Questions?

Homework, Lab, LabVIEW, quiz questions?
(Enjoyed the first actual lab?)





Integers



Integers Size

The C2000 MCU is a 32-bit CPU, meaning its registers and Arithmetic Logic Unit (ALU) are optimized for 32-bit operations, though it can also handle 16-bit data types. As seen in the Homework the minimum data size is 16-bits!

In most programming classes you will have learned that 1-byte = 8-bits; however, in the C2000, given that TMS320C28x char is 16 bits and the ANSI/ISO (C programming standard) further stipulates that when sizeof is applied to char, the result is 1. So, 1-byte = 16-bits for this system!!!

Table 6-1. TMS320C28x C/C++ COFF and EABI Data Types

			Range	
Type	**Size**	**Representation**	**Minimum**	**Maximum**
char, signed char	16 bits	ASCII	-32 768	32 767
unsigned char	16 bits	ASCII	0	65 535
_Bool	16 bits	Binary	0 (false)	1 (true)
short	16 bits	Binary	-32 768	32 767
unsigned short	16 bits	Binary	0	65 535
int, signed int	16 bits	Binary	-32 768	32 767
unsigned int	16 bits	Binary	0	65 535
long, signed long	32 bits	Binary	-2 147 483 648	2 147 483 647
unsigned long	32 bits	Binary	0	4 294 967 295
long long, signed long long	64 bits	Binary	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
unsigned long long	64 bits	Binary	0	18 446 744 073 709 551 615
enum (1)	varies	Binary	varies	varies
float (2)	32 bits	IEEE 32-bit	1.19 209 290e-38(3)	3.40 282 35e+38
double (COFF)	32 bits	IEEE 32-bit	1.19 209 290e-38(3)	3.40 282 35e+38 (COFF)
double (EABI)	64 bits	IEEE 64-bit	2.22 507 385e-308(3)	1.79 769 313e+308
long double	64 bits	IEEE 64-bit	2.22 507 385e-308(3)	1.79 769 313e+308
pointers (4)	32 bits	Binary	0	0xFFFFFFFF

(1) For details about the size of an enum type, see Section 6.3.1.
(2) It is recommended that 32-bit floating point values for COFF be declared as `float` , not as `double` .
(3) Figures are minimum precision.
(4) Even though pointers are 32-bits, the compiler assumes that the addresses of global variables and functions are within 22-bits.

Signed Integers

How do we represent negative numbers in a system that only understands 0 and 1?

What is the problem with just setting a single bit to represent positive or negative?

The problem is that you just use a single bit to define the positive or negative, i.e.

0b101 = -1 vs 0b001 = 1

What happens at 0?

0b100 = -0 vs 0vb000 = 0

Which doesn't make much sense, we waste a single number for a -0!

2’s Complement

- This utilizes Modular Arithmetic to derive that:

$$-x = \sim x + 1$$

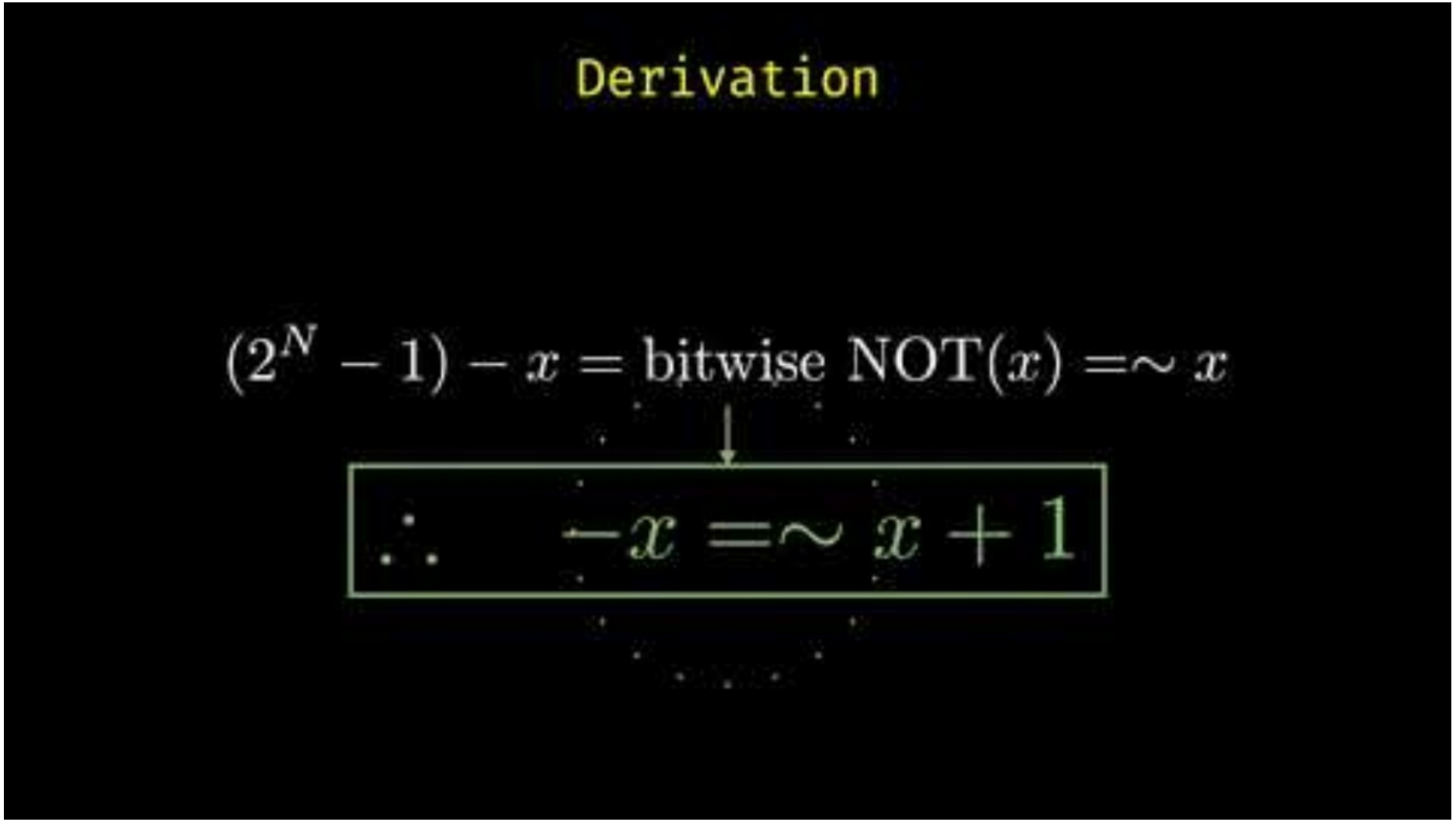
Extremely simple and CPU operation wisely efficient!

Achieves full range of:

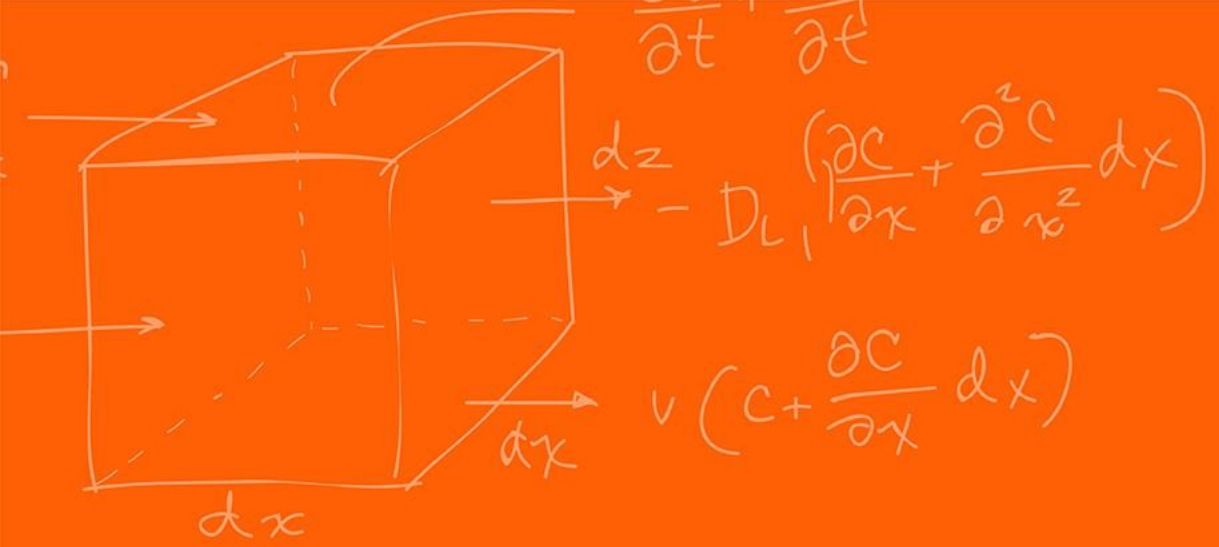
$$[-2^{N-1}, 2^{N-1}-1]$$

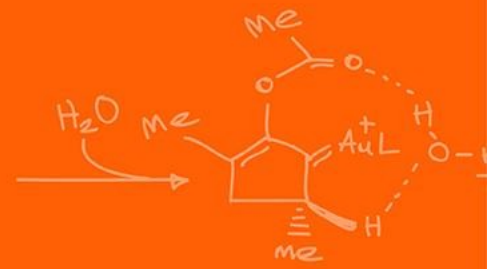
Most significant bit (the left-most bit in this case), represents the sign (0=positive, 1=negative)

2’s Complement

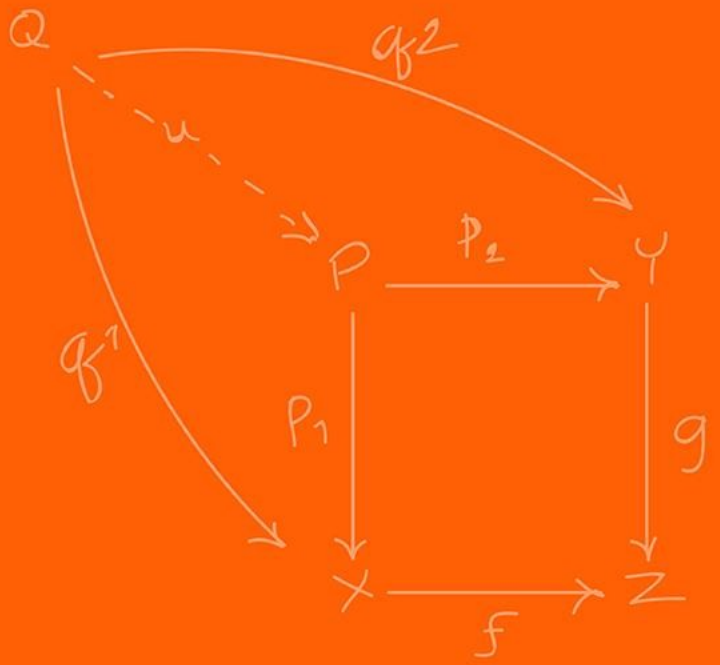


Additional resource: https://www.youtube.com/watch?v=lKTsv6iVxV4





String



ASCII

Remember that everything in the CPU is binary. So how do we define what a `a`, `b`, `<` represents as a character?

The ASCII character set! It is a 7-bit mapping, 0-127, with a 8-bit character extended edition (not the same as UNICODE).

In C a character is a single letter and represented as `x`, you CANNOT use “x” for a character in C, in Python there is no such thing as a char, just strings!

Strings “abcd” in C represent an array of characters char*. The most important character is 0x00 (NULL), which represents a sentence stopper.

In C, a string is a contiguous (where the points are sequentially next to each other) block of memory containing ASCII character, terminated with NULL. Without it the CPU would read forever until a segfaul occurs (reading restricted memory)!

If there are less than 7 bits, just put 0s in front of the binary representation

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	*[NULL]*	48	30	110000	60	**0**	96	60	1100000	140	`
1	1	1	1	*[START OF HEADING]*	49	31	110001	61	**1**	97	61	1100001	141	**a**
2	2	10	2	*[START OF TEXT]*	50	32	110010	62	**2**	98	62	1100010	142	**b**
3	3	11	3	*[END OF TEXT]*	51	33	110011	63	**3**	99	63	1100011	143	**c**
4	4	100	4	*[END OF TRANSMISSION]*	52	34	110100	64	**4**	100	64	1100100	144	**d**
5	5	101	5	*[ENQUIRY]*	53	35	110101	65	**5**	101	65	1100101	145	**e**
6	6	110	6	*[ACKNOWLEDGE]*	54	36	110110	66	**6**	102	66	1100110	146	**f**
7	7	111	7	*[BELL]*	55	37	110111	67	**7**	103	67	1100111	147	**g**
8	8	1000	10	*[BACKSPACE]*	56	38	111000	70	**8**	104	68	1101000	150	**h**
9	9	1001	11	*[HORIZONTAL TAB]*	57	39	111001	71	**9**	105	69	1101001	151	**i**
10	A	1010	12	*[LINE FEED]*	58	3A	111010	72	**:**	106	6A	1101010	152	**j**
11	B	1011	13	*[VERTICAL TAB]*	59	3B	111011	73	**;**	107	6B	1101011	153	**k**
12	C	1100	14	*[FORM FEED]*	60	3C	111100	74	**<**	108	6C	1101100	154	**l**
13	D	1101	15	*[CARRIAGE RETURN]*	61	3D	111101	75	**=**	109	6D	1101101	155	**m**
14	E	1110	16	*[SHIFT OUT]*	62	3E	111110	76	**>**	110	6E	1101110	156	**n**
15	F	1111	17	*[SHIFT IN]*	63	3F	111111	77	**?**	111	6F	1101111	157	**o**
16	10	10000	20	*[DATA LINK ESCAPE]*	64	40	1000000	100	**@**	112	70	1110000	160	**p**
17	11	10001	21	*[DEVICE CONTROL 1]*	65	41	1000001	101	**A**	113	71	1110001	161	**q**
18	12	10010	22	*[DEVICE CONTROL 2]*	66	42	1000010	102	**B**	114	72	1110010	162	**r**
19	13	10011	23	*[DEVICE CONTROL 3]*	67	43	1000011	103	**C**	115	73	1110011	163	**s**
20	14	10100	24	*[DEVICE CONTROL 4]*	68	44	1000100	104	**D**	116	74	1110100	164	**t**
21	15	10101	25	*[NEGATIVE ACKNOWLEDGE]*	69	45	1000101	105	**E**	117	75	1110101	165	**u**
22	16	10110	26	*[SYNCHRONOUS IDLE]*	70	46	1000110	106	**F**	118	76	1110110	166	**v**
23	17	10111	27	*[END OF TRANS. BLOCK]*	71	47	1000111	107	**G**	119	77	1110111	167	**w**
24	18	11000	30	*[CANCEL]*	72	48	1001000	110	**H**	120	78	1111000	170	**x**
25	19	11001	31	*[END OF MEDIUM]*	73	49	1001001	111	**I**	121	79	1111001	171	**y**
26	1A	11010	32	*[SUBSTITUTE]*	74	4A	1001010	112	**J**	122	7A	1111010	172	**z**
27	1B	11011	33	*[ESCAPE]*	75	4B	1001011	113	**K**	123	7B	1111011	173	**{**
28	1C	11100	34	*[FILE SEPARATOR]*	76	4C	1001100	114	**L**	124	7C	1111100	174	**|**
29	1D	11101	35	*[GROUP SEPARATOR]*	77	4D	1001101	115	**M**	125	7D	1111101	175	**}**
30	1E	11110	36	*[RECORD SEPARATOR]*	78	4E	1001110	116	**N**	126	7E	1111110	176	**~**
31	1F	11111	37	*[UNIT SEPARATOR]*	79	4F	1001111	117	**O**	127	7F	1111111	177	*[DEL]*
32	20	100000	40	*[SPACE]*	80	50	1010000	120	**P**					
33	21	100001	41	**!**	81	51	1010001	121	**Q**					
34	22	100010	42	**"**	82	52	1010010	122	**R**					
35	23	100011	43	**#**	83	53	1010011	123	**S**					
36	24	100100	44	**$**	84	54	1010100	124	**T**					
37	25	100101	45	**%**	85	55	1010101	125	**U**					
38	26	100110	46	**&**	86	56	1010110	126	**V**					
39	27	100111	47	**'**	87	57	1010111	127	**W**					
40	28	101000	50	**(**	88	58	1011000	130	**X**					
41	29	101001	51	**)**	89	59	1011001	131	**Y**					
42	2A	101010	52	*****	90	5A	1011010	132	**Z**					
43	2B	101011	53	**+**	91	5B	1011011	133	**[**					
44	2C	101100	54	**,**	92	5C	1011100	134	****					
45	2D	101101	55	**-**	93	5D	1011101	135	**]**					
46	2E	101110	56	**.**	94	5E	1011110	136	**^**					
47	2F	101111	57	**/**	95	5F	1011111	137	**_**					

printf and sprintf

We want to write content to the output stream (the stream can be your console output, serial communication, a file, a string). The function sprintf, outputs to a buffer, printf outputs to the default output stream.

```
serial_printf(&SerialA,"Num Timer2:%ld Num SerialRX: %ld\r\
              n",CpuTimer2.InterruptCount,numRXA);
```

Defined in header <stdio.h>

`int printf( const char* format, ... );`	(1)	(until C99)
`int printf( const char* restrict format, ... );`		(since C99)
`int fprintf( FILE* stream, const char* format, ... );`	(2)	(until C99)
`int fprintf( FILE* restrict stream, const char* restrict format, ... );`		(since C99)
`int sprintf( char* buffer, const char* format, ... );`	(3)	(until C99)
`int sprintf( char* restrict buffer, const char* restrict format, ... );`		(since C99)
`int snprintf( char* restrict buffer, size_t bufsz, const char* restrict format, ... );`	(4)	(since C99)
`int printf_s( const char* restrict format, ... );`	(5)	(since C11)
`int fprintf_s( FILE* restrict stream, const char* restrict format, ... );`	(6)	(since C11)
`int sprintf_s( char* restrict buffer, rsize_t bufsz, const char* restrict format, ... );`	(7)	(since C11)
`int snprintf_s( char* restrict buffer, rsize_t bufsz, const char* restrict format, ... );`	(8)	(since C11)

printf and sprint - format

The format options are available here. Given that you need to format you string based on your pattern, the runtime is non-deterministic and printing usually takes a “very long” (in computer speed) time.

A format specifier follows this prototype:

%[flags][width][.precision][length]specifier

```c
/* printf example */
#include <stdio.h>
int main()
{
   printf ("Characters: %c %c \n", 'a', 65);
   printf ("Decimals: %d %ld\n", 1977, 650000L);
   printf ("Preceding with blanks: %10d \n", 1977);
   printf ("Preceding with zeros: %010d \n", 1977);
   printf ("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
   printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
   printf ("Width trick: %*d \n", 5, 10);
   printf ("%s \n", "A string");
   return 0;
}
```

printf and sprint - format

The format options are available here. Given that you need to format you string based on your pattern, the runtime is non-deterministic and printing usually takes a “very long” (in computer speed) time.

A format specifier follows this prototype:

%[flags][width][.precision][length]specifier

```c
/* printf example */
#include <stdio.h>
int main()
{
   printf ("Characters: %c %c \n", 'a', 65);
   printf ("Decimals: %d %ld\n", 1977, 650000L);
   printf ("Preceding with blanks: %10d \n", 1977);
   printf ("Preceding with zeros: %010d \n", 1977);
   printf ("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
   printf ("floats: %4.2f %+.0e %E \n", 3.1416, 3.1416, 3.1416);
   printf ("Width trick: %*d \n", 5, 10);
   printf ("%s \n", "A string");
   return 0;
}
```

printf and sprint – format – specifier & length

specifier	Output	Example
d ***or*** i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

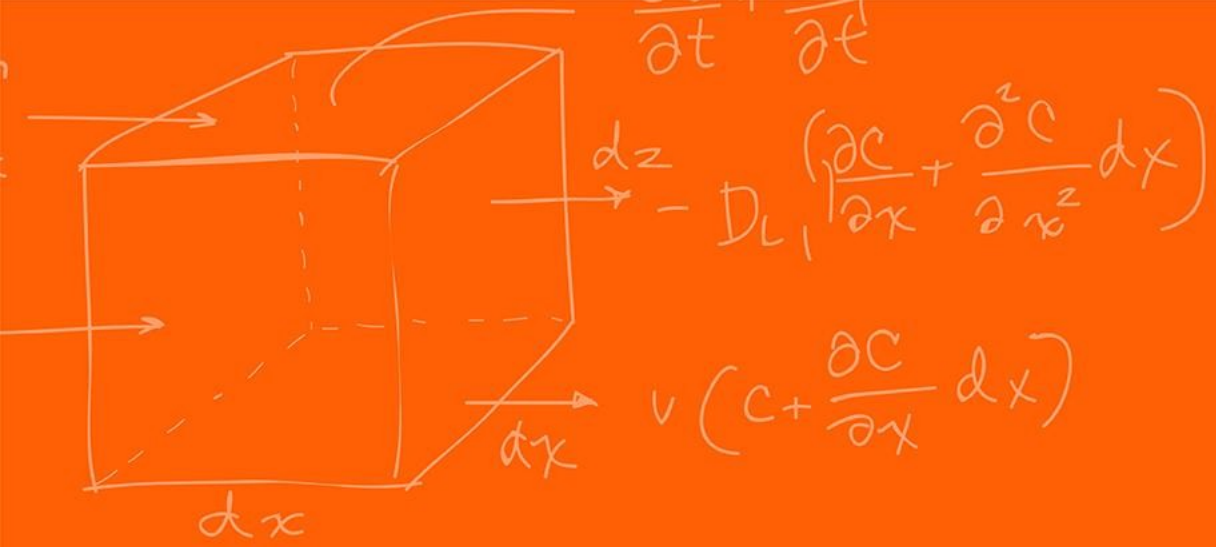
	specifiers						
length	d i	u o x X	f F e E g G a A	c	s	p	n
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

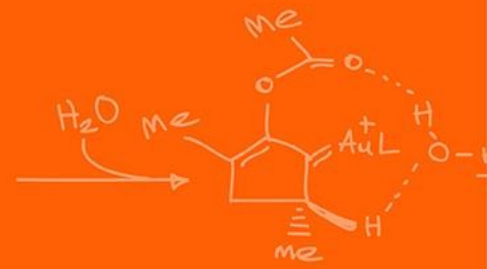
printf and sprint – format – additional flags

flags	**description**
-	Left-justify within the given field width; Right justification is the default (see ***width*** sub-specifier).
+	Forces to preceed the result with a plus or minus sign (`+` or `-`) even for positive numbers. By default, only negative numbers are preceded with a `-` sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with `o`, `x` or `X` specifiers the value is preceeded with `0`, `0x` or `0X` respectively for values different than zero. Used with `a`, `A`, `e`, `E`, `f`, `F`, `g` or `G` it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
`0`	Left-pads the number with zeroes (`0`) instead of spaces when padding is specified (see ***width*** sub-specifier).

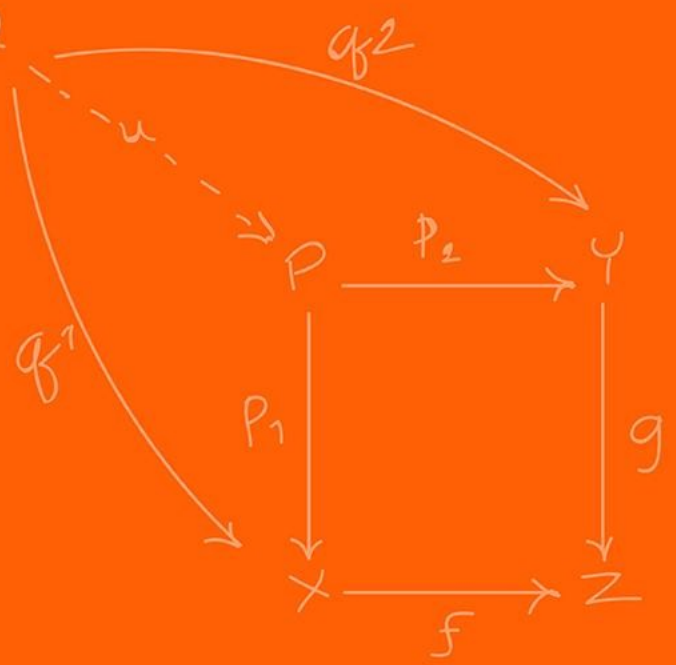
width	**description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The ***width*** is not specified in the ***format*** string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	**description**
.number	For integer specifiers (`d`, `i`, `o`, `u`, `x`, `X`): ***precision*** specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A ***precision*** of `0` means that no character is written for the value `0`. For `a`, `A`, `e`, `E`, `f` and `F` specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For `g` and `G` specifiers: This is the maximum number of significant digits to be printed. For `s`: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for ***precision***, `0` is assumed.
.*	The ***precision*** is not specified in the ***format*** string, but as an additional integer value argument preceding the argument that has to be formatted.





RS-232 Serial Port



SCI

An RS-232 serial port is a standard for serial communication, sending data one bit at a time

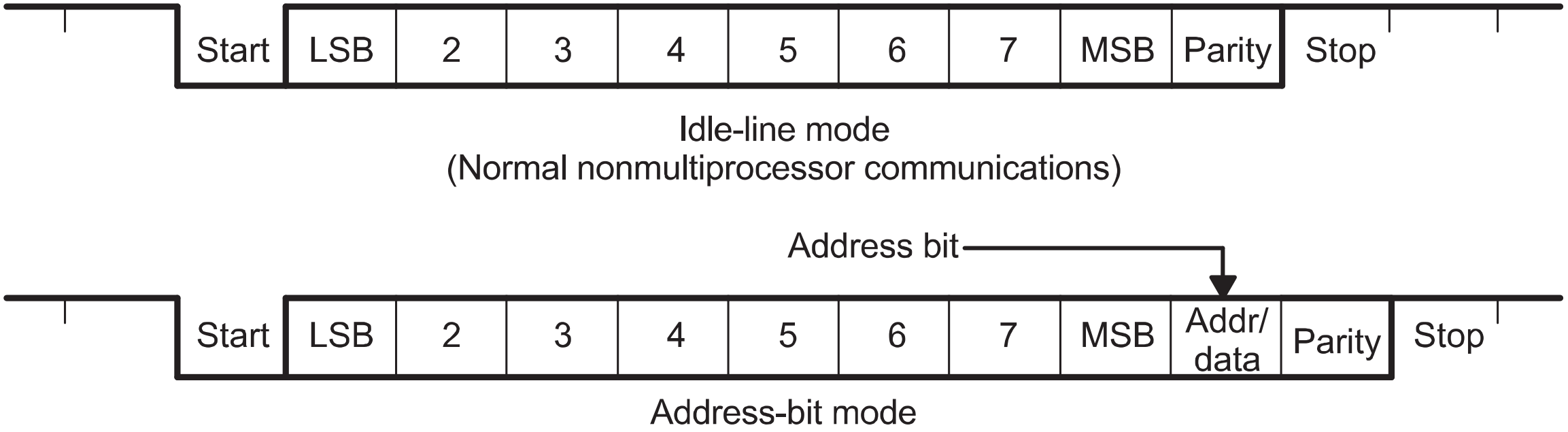
SCI is the hardware peripheral inside the C2000 MCU that implements the UART (Universal Asynchronous Receiver-Transmitter) protocol.

The Protocol:

- **Asynchronous**: No shared clock wire. Timing relies on agreed-upon Baud Rate.
- **Frame**: Start Bit (Logic 0) → Data Bits (LSB first) → Stop Bit (Logic 1).

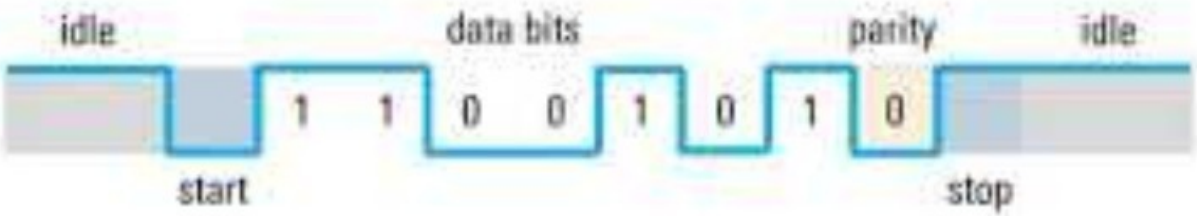
```
init_serialSCIA(&SerialA,115200);
```

Figure 19-3. Typical SCI Data Frame Formats



UART frame format

- UART frames consist of:
  - Start / stop bits
  - Data bits
  - Parity bit (optional)
- High voltage ("mark") = 1, low voltage ("space") = 0



SCI

TTL (Transistor-Transistor Logic):

The native language of the MCU (GPIO pins).

- Logic 1: 3.3V (or 5V).
- Logic 0: 0V (Ground).

RS-232 (Reference Standard 232):

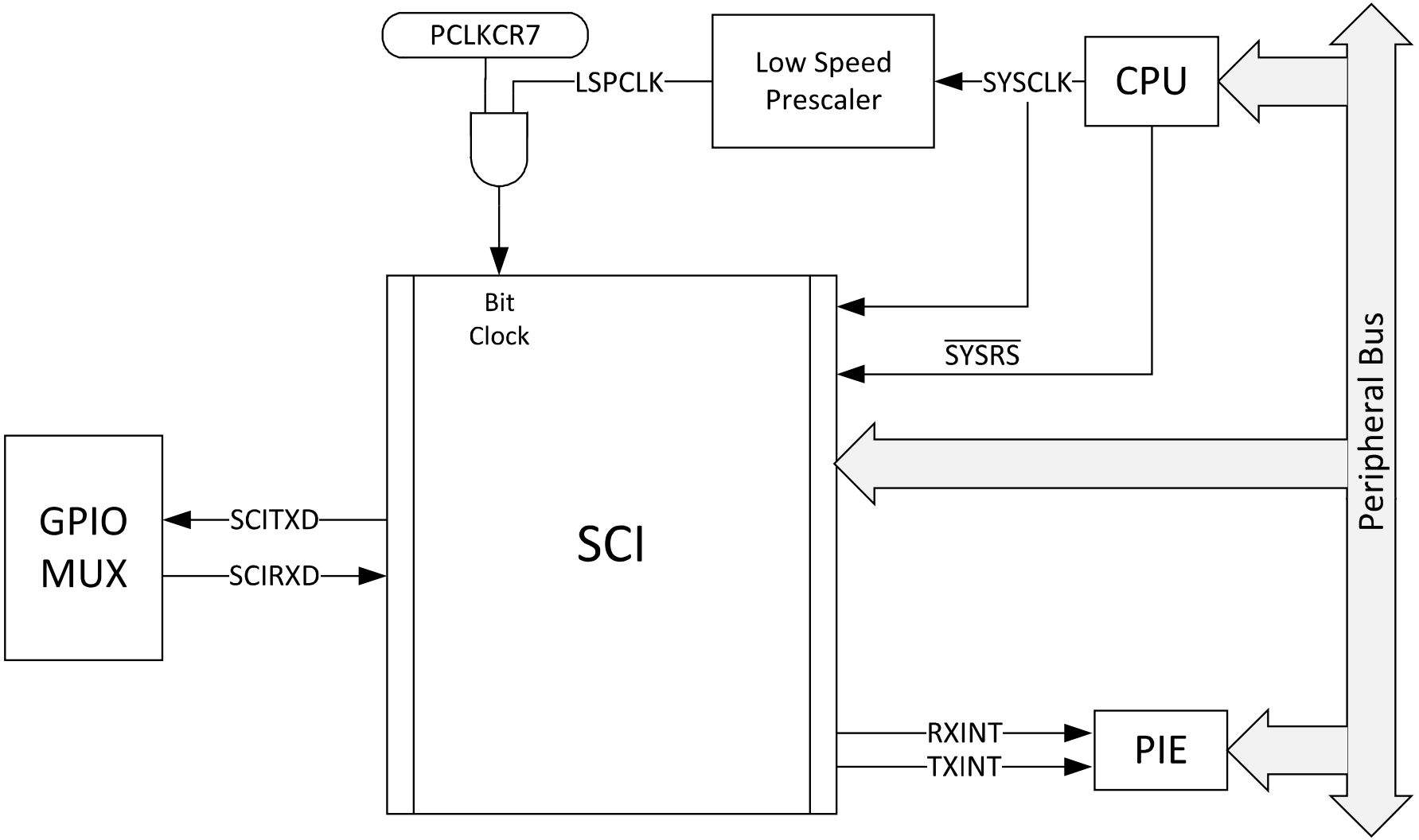
Voltage Levels: Uses bipolar signaling to resist noise.

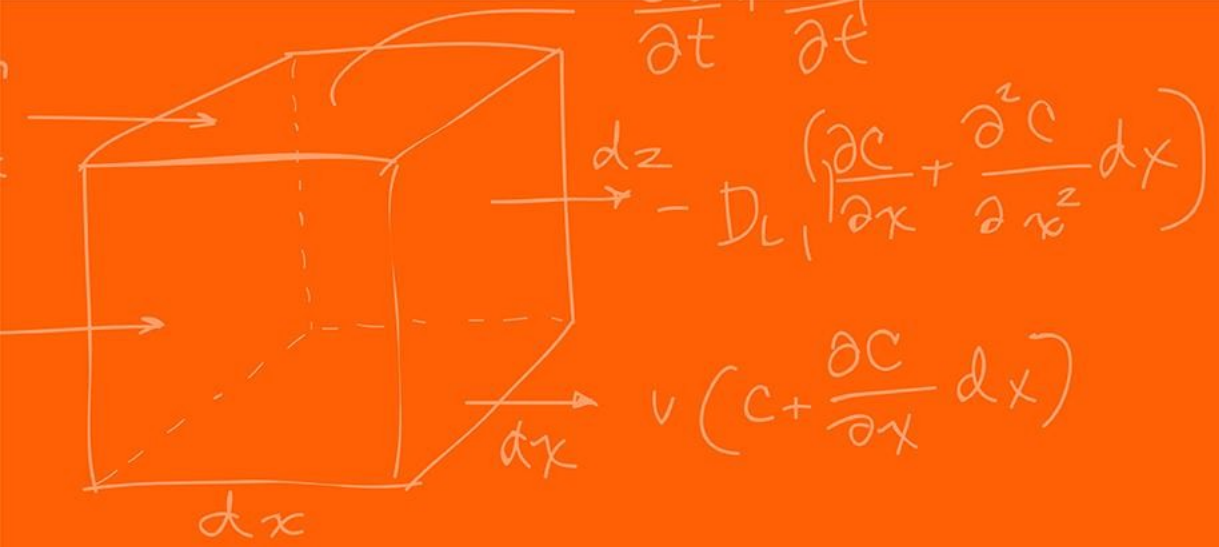
- Logic 0: +3V to +15V.
- Logic 1: -3V to -15V.

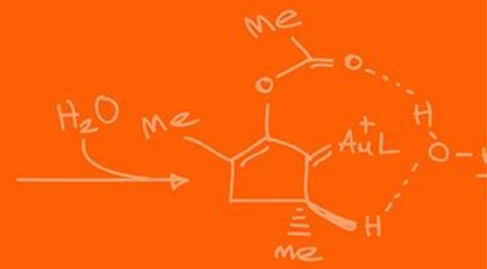
Notice that RS-232 Logic 1 is negative voltage. This is an inverted logic compared to TTL.

Hardware: You cannot connect a C2000 GPIO directly to a PC's RS-232 port. You need a transceiver to invert and boost the voltages. (this is done inside the red board)

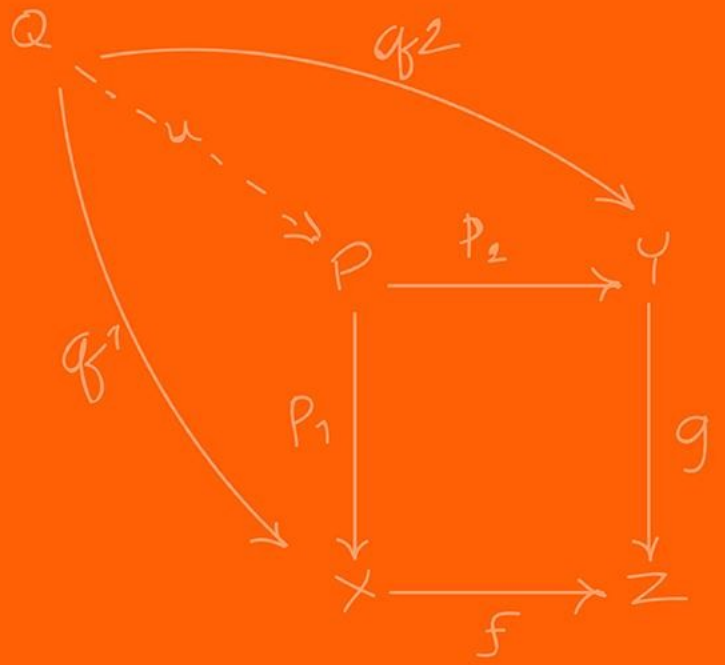
Figure 19-1. SCI CPU Interface



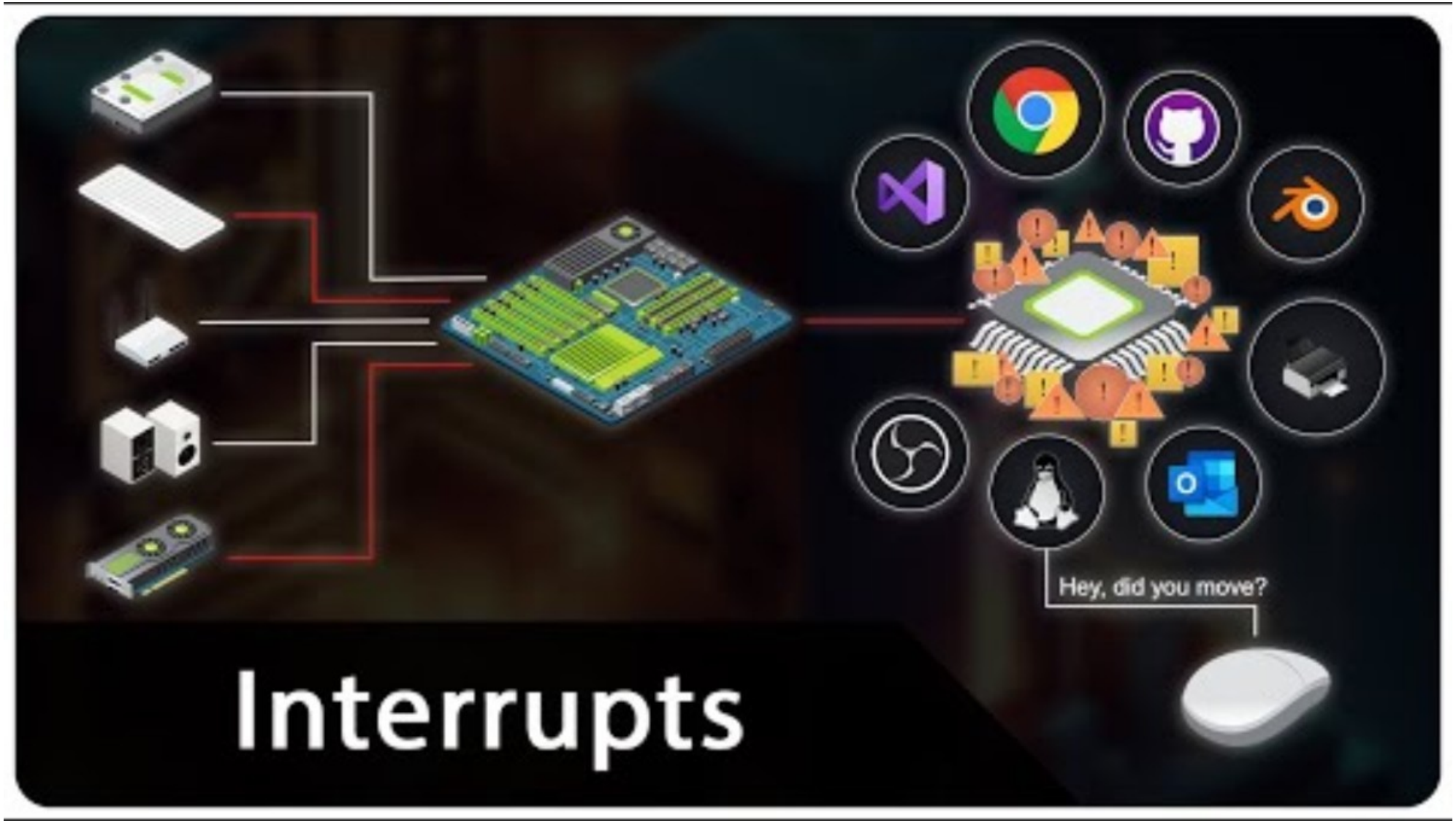




CPU Interrupts



CPU Interrupts (Modern computers)



Hardware Interrupts (Timer)

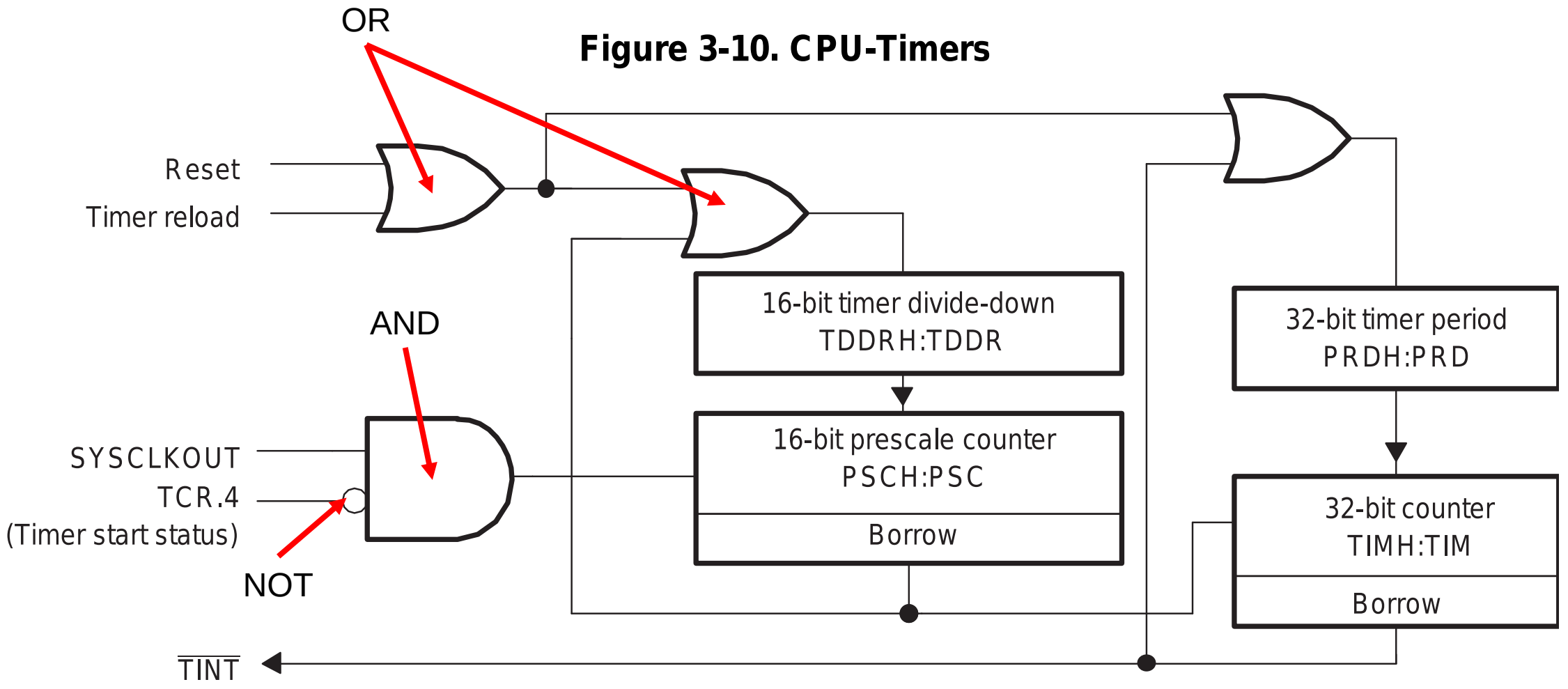
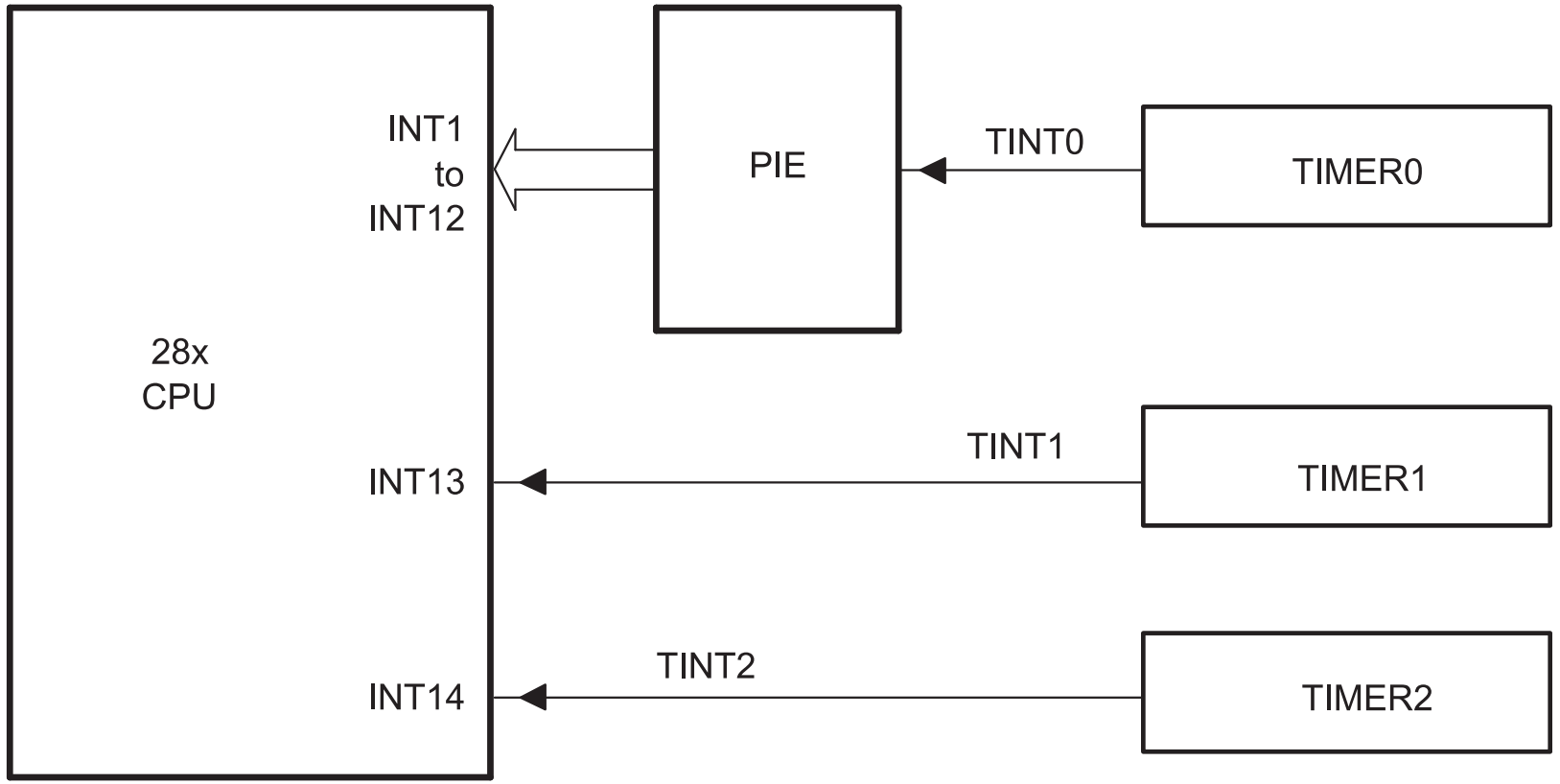


Figure 3-10. CPU-Timers

Figure 3-11. CPU-Timer Interrupts Signals and Output Signal



A The timer registers are connected to the memory bus of the C28x processor.

B The CPU Timers are synchronized to SYSCLKOUT.

Hardware Interrupts

```
EALLOW;
PieVectTable.TIMER0_INT = &cpu_timer0_isr;
PieVectTable.TIMER2_INT = &cpu_timer2_isr;
PieVectTable.TIMER1_INT = &cpu_timer1_isr;
PieVectTable.SCIA_RX_INT = &RXAINT_recv_ready;
PieVectTable.SCIB_RX_INT = &RXBINT_recv_ready;
PieVectTable.SCIC_RX_INT = &RXCINT_recv_ready;
PieVectTable.SCID_RX_INT = &RXDINT_recv_ready;
PieVectTable.SCIA_TX_INT = &TXAINT_data_sent;
PieVectTable.SCIB_TX_INT = &TXBINT_data_sent;
PieVectTable.SCIC_TX_INT = &TXCINT_data_sent;
PieVectTable.SCID_TX_INT = &TXDINT_data_sent;
EDIS;
```

Software Interrupts

```
EALLOW;
PieVectTable.EMIF_ERROR_INT = &SWI_isr;
EDIS;
```

Interrupts

All the interrupt options are available inside the PIE Vector Table inside F2837xD_pievect.h

Table 3-2 shows the PIE group and channel assignments for each peripheral interrupt. Each row is a group, and each column is a channel within that group. When multiple interrupts are pending, the lowest-numbered channel is the lowest-numbered group is serviced first. Thus, the interrupts at the top of the table have the highest priority, and the interrupts at the bottom have the lowest priority.

Table 3-2. PIE Channel Mapping

	INTx.1	INTx.2	INTx.3	INTx.4	INTx.5	INTx.6	INTx.7	INTx.8	INTx.9	INTx.10	INTx.11	INTx.12	INTx.13	INTx.14	INTx.15	INTx.16
INT1.y	ADCA1	ADCB1	ADCC1	XINT1	XINT2	ADCD1	TIMER0	WAKE	-	-	-	-	IPC0	IPC1	IPC2	IPC3
INT2.y	EPWM1_TZ	EPWM2_TZ	EPWM3_TZ	EPWM4_TZ	EPWM5_TZ	EPWM6_TZ	EPWM7_TZ	EPWM8_TZ	EPWM9_TZ	EPWM10_TZ	EPWM11_TZ	EPWM12_TZ	-	-	-	-
INT3.y	EPWM1	EPWM2	EPWM3	EPWM4	EPWM5	EPWM6	EPWM7	EPWM8	EPWM9	EPWM10	EPWM11	EPWM12	-	-	-	-
INT4.y	ECAP1	ECAP2	ECAP3	ECAP4	ECAP5	ECAP6	-	-	-	-	-	-	-	-	-	-
INT5.y	EQEP1	EQEP2	EQEP3	-	CLB1	CLB2	CLB3	CLB4	SD1	SD2	-	-	-	-	-	-
INT6.y	SPIA_RX	SPIA_TX	SPIB_RX	SPIB_TX	MCBSPA_RX	MCBSPA_TX	MCBSPB_RX	MCBSPB_TX	SPIC_RX	SPIC_TX	-	-	-	-	-	-
INT7.y	DMA_CH1	DMA_CH2	DMA_CH3	DMA_CH4	DMA_CH5	DMA_CH6	-	-	-	-	-	-	-	-	-	-
INT8.y	I2CA	I2CA_FIFO	I2CB	I2CB_FIFO	SCIC_RX	SCIC_TX	SCID_RX	SCID_TX	-	-	-	-	-	-	UPPA (CPU1 only)	-
INT9.y	SCIA_RX	SCIA_TX	SCIB_RX	SCIB_TX	CANA_0	CANA_1	CANB_0	CANB_1	-	-	-	-	-	-	USBA (CPU1 only)	-
INT10.y	ADCA_EVT	ADCA2	ADCA3	ADCA4	ADCB_EVT	ADCB2	ADCB3	ADCB4	ADCC_EVT	ADCC2	ADCC3	ADCC4	ADCD_EVT	ADCD2	ADCD3	ADCD4
INT11.y	CLA1_1	CLA1_2	CLA1_3	CLA1_4	CLA1_5	CLA1_6	CLA1_7	CLA1_8	-	-	-	-	-	-	-	-
INT12.y	XINT3	XINT4	XINT5	-	-	VCU	FPU_OVERFLOW	FPU_UNDERFLOW	EMIF_ERROR	RAM_CORRECTABLE_ERROR	FLASH_CORRECTABLE_ERROR	RAM_ACCESS_VIOLATION	SYS_PLL_SLIP	AUX_PLL_SLIP	CLA OVERFLOW	CLA UNDERFLOW

Note: Cells marked "-" are Reserved

Enabling Interrupts

```
// Enable CPU int1 which is connected to CPU-Timer 0, CPU int13
// which is connected to CPU-Timer 1, and CPU int 14, which is connected
// to CPU-Timer 2:  int 12 is for the SWI.
IER |= M_INT1;
IER |= M_INT8;  // SCIC SCID
IER |= M_INT9;  // SCIA
IER |= M_INT12;
IER |= M_INT13;
IER |= M_INT14;

// Enable TINT0 in the PIE: Group 1 interrupt 7
PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
// Enable SWI in the PIE: Group 12 interrupt 9
PieCtrlRegs.PIEIER12.bit.INTx9 = 1;
```

Interrupts - Timers

TIMER0_INT, TIMER1_INT, TIMER2_INT:

The standard CPU countdown timers.

RTOS_INT:

A dedicated interrupt line for Real-Time Operating System context switching.

WAKE_INT:

Wakes the CPU from Low-Power (IDLE/STANDBY) modes.

Interrupts - Sensors

ADCA1_INT through **ADCD4_INT** (The ADCs):

These trigger when the Analog-to-Digital Converter finishes a batch of conversions.

Classification: **ADCx_EVT_INT** (Threshold event), **ADCx1_INT** (High Priority Loop), **ADCx2/3/4_INT** (Lower priority monitoring).

EQEP1_INT, EQEP2_INT, EQEP3_INT:

Enhanced Quadrature Encoder Pulse. These trigger on position events (e.g., Index pulse or Unit Timeout) from a motor encoder. Crucial for speed calculations.

ECAP1_INT through **ECAP6_INT**:

Enhanced Capture. Used to measure the width of incoming pulses (e.g., measuring the frequency of an incoming PWM signal from another sensor).

SD1_INT, **SD2_INT**:

Sigma-Delta Filter. These define interrupts for high-fidelity, isolated current sensing, typically used in high-voltage motor drives.

Interrupts – External Hardware Events

XINT1_INT through **XINT5_INT** (External Interrupts):

These are triggered by voltage transitions (Rising Edge, Falling Edge, or Both) on physical pins. These vectors represent the C2000's direct interface with the outside world via General Purpose Input/Output (GPIO) pins.

Interrupts – Actuation (will go into more details later)

EPWM1_TZ_INT through **EPWM12_TZ_INT** (Trip Zones):

Highest Priority Control Interrupts. These fire when a hardware comparator (CMPSS) detects an over-current or over-voltage event. The PWM hardware shuts off instantly (nanoseconds) and then informs the CPU via this interrupt.

EPWM1_INT through **EPWM12_INT**:

Standard PWM interrupts, usually triggered at TBCTR = 0 or TBCTR = PRD. Used to update duty cycles for the next cycle.

Interrupts – Communication (RX = receive, TX = transmit)

SPIA/B/C_RX_INT & SPIA/B/C_TX_INT:

Synchronous Serial Interface (High speed sensors/DACs).

I2CA/B_INT & FIFO_INT:

Inter-Integrated Circuit (EEPROMs, temperature sensors).

SCIA/B/C/D_RX_INT & TX_INT:

UART (RS-232, discussed in the lecture). Connecting to PC/MATLAB.

CANA0/1_INT & CANB0/1_INT:

Controller Area Network. The standard automotive bus for communicating with other ECUs.

MCBSPA/B_RX_INT & TX_INT: Multichannel Buffered Serial Port. (Legacy Audio/DSP protocol).

USBA_INT:

Universal Serial Bus (Bulk data transfer).

Interrupts – Errors

NMI_INT (Non-Maskable Interrupt):

The only interrupt that cannot be ignored by the CPU. Used for critical hardware failures.

ILLEGAL_INT:

Triggered if the CPU attempts to decode an opcode that does not exist. (Common pointer corruption symptom).

EMIF_ERROR_INT:

External Memory Interface error. Indicates a hardware timing violation or disconnect with external RAM/Flash.

RAM_CORRECTABLE_ERROR_INT / FLASH_CORRECTABLE_ERROR_INT:

The ECC (Error Correction Code) logic detected a bit-flip but fixed it. The ISR usually logs this event for predictive maintenance.

RAM_ACCESS_VIOLATION_INT:

The CPU tried to write to a protected memory region.

SYS_PLL_SLIP_INT / AUX_PLL_SLIP_INT:

The Phase Locked Loop (clock generator) has lost its lock. The CPU clock is unstable. The system must immediately halt PWM generation to prevent blowing up power electronics.

FPU_OVERFLOW_INT / FPU_UNDERFLOW_INT:

The Floating-Point Unit calculated a value exceeding IEEE-754 limits (creating $\pm\infty$ or NaN)

Interrupts – CPU Accelerator

DMA_CH1_INT through DMA_CH6_INT:

Direct Memory Access. Triggers when a large block of memory (e.g., an ADC buffer) has been moved without CPU intervention.

IPC0_INT through IPC3_INT:

Inter-Processor Communication. This device is Dual Core (CPU1 and CPU2). These interrupts are the "phone lines" allowing CPU1 to signal CPU2.

VCU_INT:

Viterbi Complex Unit. An accelerator for vibration analysis (FFT) and communications encoding.

Interrupts – CPU Accelerator – CLA (floating point engine)

CLA_OVERFLOW_INT / CLA_UNDERFLOW_INT:

Mathematical errors specifically within the CLA core.

CLA1_1_INT through **CLA1_8_INT:**

The Control Law Accelerator (CLA) is an independent 32-bit floating-point math engine. Called once a floating point operation is done.

Interrupts – High-Speed Peripherals

UPPA_INT: Parallel Interface Communication

This is a high-speed, multi-channel parallel bus often used to interface with high-speed ADCs or FPGAs

Interrupts – Placeholders & User Definitions

- **PIE1_RESERVED_INT** through **PIE88_RESERVED_INT:**

  The hardware architecture allocates a 12×16 grid, but not every slot is connected to a peripheral in this specific chip. These must be mapped to a "Default ISR" that essentially does nothing, to prevent system crashes if noise triggers a phantom interrupt.

USER1_INT through **USER12_INT:**

These are software traps. You can trigger these manually in code using the assembly instruction TRAP #20. They are often used by RTOS kernels to implement system calls.

Interrupts – Debugging

DATALOG_INT (Datalogging Interrupt):

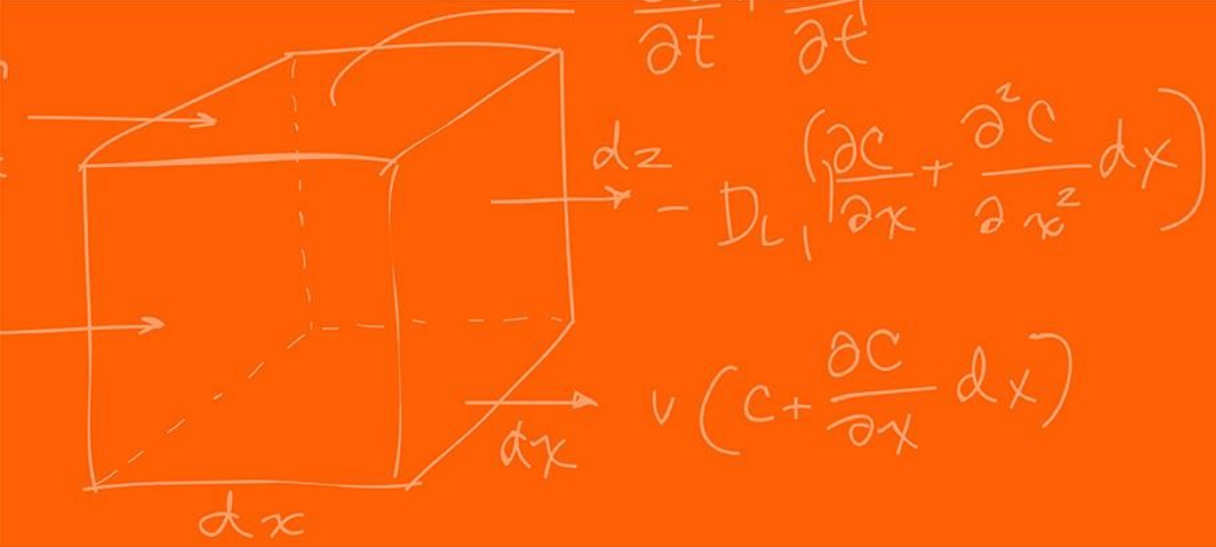
Used primarily by TI's data-logging modules to trigger a capture of internal variables at a specific sample rate. Helpful for logging data for visualization.

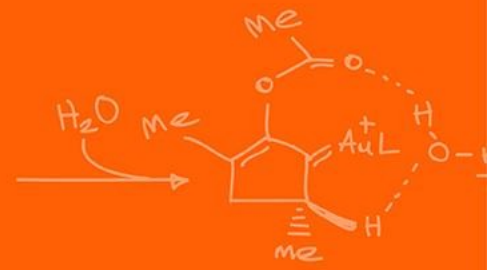
EMU_INT:

Triggered by the on-chip debug logic (JTAG). This handles hardware breakpoints or software "Halt" commands. It allows the CPU to enter a suspended state while maintaining the integrity of the peripheral registers.

Questions?

(don’t forget to submit the homework!)





The Grainger College of Engineering

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN