Euroopa Liit
Euroopa Sotsiaalfond

Eesti tuleviku heaks

Tallinna Tehnikaülikooli elektriajamite ja jõuelektroonika instituut

ENERGIA- JA GEOTEHNIKA DOKTORIKOOL II

INTENSIIVKURSUS

"MICROCONTROLLERS FOR POWER ELECTRONICS APPLICATIONS"

Prof. ILYA GALKIN, Riga Technical University

Tallinn 2011

Part I: Overview of microcontrollers MSP430

Basic information

MSP430 is a family of cost effective 16-bit microcontrollers designed for ultra low power applications, in particular, for portable measurement equipment. Other key features of MSP430 microcontrollers are: von Neumann architecture, RISC CPU with a file (set) of 16 16-bit registers, flash memory 0.5...256kB, RAM 128B...16kB, maximum clock frequency 8...25MHz, a wide set of peripheral devices to handle analogue signals and control external devices (timers, ADCs , DACs, communications etc.), 14...113pin pinout (24 packages).

What does "ultra low power consumption" mean

Ultra low power (ULP) consumption assumes:

1) Interrupt driven MCU – variety of operation modes and free/fast switching between them;

2) Multiple clock signals;

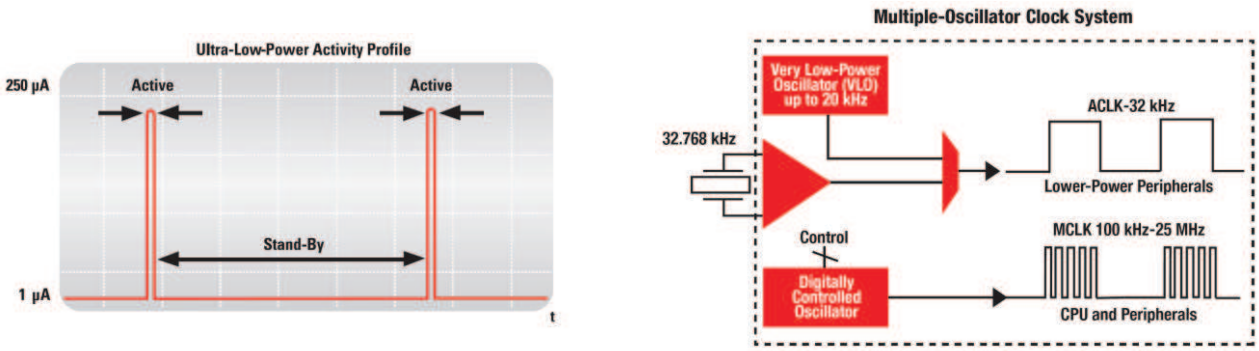
3) Low current CMOS technology.

Interrupt driven MCU

This approach means that most of the time MSP430 stays in an idle mode and its energy consumption is low. Its CPU awakes only when it is necessary (after some event), processes new data and returns back to the idle mode (Figure 1-a). Other hardware of MCU (peripheral devices) is enabled (in active or idle modes) only if it is necessary. This approach assumes the following features:

Diversity of operation modes – There must be at least two operation modes of MCU – active and idle mode. MSP430 have 6 modes – active mode and 5 low power modes (LPM0...LPM4). Low power modes differ mostly depending on the number of activated hardware modules and clock sources.

Free and fast switching between the operating modes – MCU must be able to go quickly into the active mode from one of low power modes, as well as quickly return back to the low power mode. MSP430 can "wake up" in 1μs.



a) Interrupt driven operation b) Clock system

Figure 1. Basic features of clock system of MSP430

Multiple clock signals

This means that there are several clock signals of different frequencies (Figure 1-b). This provides an opportunity to run each of peripheral devices with necessary speed and reach a tradeoff between the

functionality and energy consumption of CPU and peripheral devices. Typically the clock system of MSP430 includes 3 clock signals for hardware:

High frequency clock –supplied to CPU; typically of the maximal frequency of MCU (16MHz for MSP430G2231); can be of low accuracy and with high temperature and voltage drift;

Medium frequency clock – either divided/prescaled high frequency clock or clock obtained from a more accurate source (referenced by a crystal); typically hundreds of kHz or couple of MHz

Low frequency clock – usually high accuracy clock referenced by a crystal for those peripheral devices that operate in real time mode (often 32768Hz).

Low current technology

This means that self consumption of MSP430 is low and there are several operation modes available with different levels of power consumption:

0.1μA – power-down current (Low Power Mode 4 – LPM4 – all peripherals and CPU disabled, RAM retention, external interrupts are possible);

1.0μA – standby current at 3V and LPM3 (one clock source 32768Hz is active, some peripheral devices are on, CPU is disabled, RAM retention, external interrupts are possible);

65μA – standby current at 3V and LPM0 (two clock sources 1MHz and 32768Hz are active, all peripheral devices are on, CPU is disabled, RAM retention, external interrupts are possible);

300μA – active operation mode (CPU and all peripheral devices are enabled) at CPU clock 1MHz and supply voltage 3V;

7mA – active operation mode (CPU and all peripheral devices enabled for MSP430G2231) at CPU clock 16MHz and supply voltage 3.9V;

Architecture

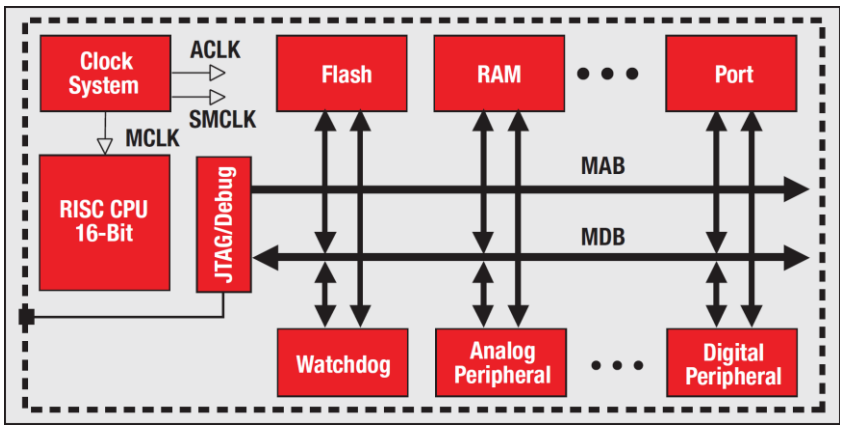


Figure 2. Basics of architecture of MSP430

Basic features of architecture of MSP430 are:

Von Neumann principle: 1) common memory address bus (MAB) and data bus (MDB) for programs and data as well as for control registers of peripheral devices (Figure 2); 2) Common data space for different segments of memory; 3) Common set of commands for all memory operations; 4) No simultaneous access.

16-bit data: 1) memory data bus is 16-bit wide (16-bit data can be processed in one sweep); 2) memory address bus for standard architecture is 16-bit wide (up to 2^{16}=64k memory cells [bytes] can be accessed); 3)

memory address bus for extended architecture (MSP430x2xx, MSP430x5xx) is 20-bit wide (up to 2^{20}=1M memory cells [bytes] can be accessed);

RISC and file type CPU: 1) Command system contains 27 instructions (easy to get started); 2) 16 16-bit CPU registers are equal data sources and data receivers; 3) CPU directly (through status flags) supports 8-bit and 16-bit signed and unsigned integer data formats.

Direct Memory Access – in some MCUs data can be transferred from a peripheral module to memory or from memory to module or between modules apart CPU

Variety of peripheral devices is integrated in CPU and available through the common data space

Peripheral devices

Analog peripheral devices

10 and 12-bit Analog to Digital Converters (ADC10 and ADC12) are successive approximation 200ksps converters equipped with 1.5/2.5 internal or external reference. ADC10 is equipped also with data transfer controller (DTC), but ADC12 with conversion-and-control buffer – features that transfer measured data directly to memory.

Sigma Delta analog to digital converters (SD16) of 16-bit resolution are equipped with an internal 1.2V reference and have up to 8 fully differential multiplexed inputs. SD16 are second-order oversampling sigma-delta modulators with selectable oversampling ratio.

Analog Pool (A-POOL) can be configured as an ADC, DAC, Comparator, Supply Voltage Supervisor (SVS) or temperature sensor.

Comparator A (Comp_A) provides high precision voltage comparison that, in turn, ensures accurate slope analog to digital conversion, supply-voltage supervision and monitoring of external signals.

Digital to Analog Converter (DAC12) is a 12-bit voltage-output module with internal or external reference. This module can be configured also for 8-bit mode. Multiple DAC12 modules can be synchronized.

Operational Amplifiers (OpAmp) - some MSP430 microcontrollers include integrated, single-supply, rail-to-rail output operational amplifiers with programmable settling times, feedback resistors and connections between multiple op amps.

Timers

Basic Timer (BT) includes 2 independent 8-bit timers that can form a 16-bit timer/counter. Both timers can be read and written by software. The BT can be configured as a real time clock and calendar that supports month length and includes leap-year correction.

Real-Time Clock (RTC_A, RTC_B) is a 32-bit hardware counter that provides clock counters with a calendar, programmable alarm and calibration.

General purpose timers A, B and D (Timer_A/Timer_B/Timer_D) are asynchronous 16-bit timers/counters that can operate in 3 operating modes and that are equipped with up to seven capture/compare registers. The timers support multiple capture/compares, PWM outputs, and interval timing.

Watchdog timer (WDT) performs a controlled system restart if the selected time interval expires (that usually happen if software of microcontroller hangs). The watchdog can be also be configured as an interval timer and can generate interrupts at selected time intervals.

Communications

The universal synchronous/asynchronous receiver/transmitter (USART) supports asynchronous UART interface or synchronous SPI. The USART of MSP430F15x/16x USART modules also support I2C. USART ensures programmable baud rate and independent interrupt capability for reception and transmission.

Universal Serial Bus USB controller is compliant with the USB 2.0 specification. It supports control, interrupt and bulk transfers at 12 Mbps. The controller supports USB suspend, resume and remote wake-up and can be configured for up to 8 input and 8 output endpoints. The controller has an integrated physical interface (PHY), a phase-locked loop (PLL) for clock generation and a power-supply controller enabling bus-powered and self-powered devices.

The universal serial communication interface (USCI) includes two independent channels that can be used simultaneously. The asynchronous channel (USCI_A) supports UART mode, SPI mode, pulse shaping for IrDA; and automatic baud-rate detection for LIN communications. The synchronous channel (USCI_B) supports I2C and SPI modes.

The universal serial interface (USI) module is a synchronous serial interface with a data length of up to 16-bits that supports SPI and I2C communication.

Other important peripheral devices

Direct Memory Access (DMA) controller transfers data from one address to another across the whole memory space with no CPU intervention. The module has up to 3 independent transfer channels.

Hardware Multiplier (MPY) supports 8/16-bit x 8/16-bit signed and unsigned multiply with optional multiply&accumulate function. The module is a peripheral device, does not affect CPU and can be accessed by the DMA.

Digital Input/Output Ports – groups of I/O pins. MSP430 have up to 12 ports. Every I/O pin can be configured as input or output and can be individually read or written. Ports P1 and P2 have interrupt capability. In some MCUs individually configurable pull-up or pull-down resistors are available.

Families of MSP430

There are several subfamilies of MSP430 microcontrollers. Their capabilities are presented in a short form in Figure 3. Briefly they can be described as follow:

MSP430F1xx – an oldest subfamily of general purpose MCUs with standard 16-bit bus system, 8MHz clock and comprehensive peripheral set;

MSP430x2xx – an advanced subfamily of general purpose MCUs with extended 20-bit bus system, 16MHz clock and improved peripheral set;

MSP430G2xx – a cost effective subfamily of MSP430x2xx;

MSP430Lxxx – a subfamily of general purpose MCUs for ultra low voltage applications; currently (2011.04.17) represented by one chip MSP430L092;

MSP430x5xx/x6xx – a high performance subfamily of general purpose MCUs with extended 20-bit bus system and 25MHz clock;

MSP430x4xx – a subfamily of MCUs with LCD support and specific peripheral hardware for metering and medical applications;

CC430 – a subfamily of MCUs with integrated RF transceiver for system-on-chip development.

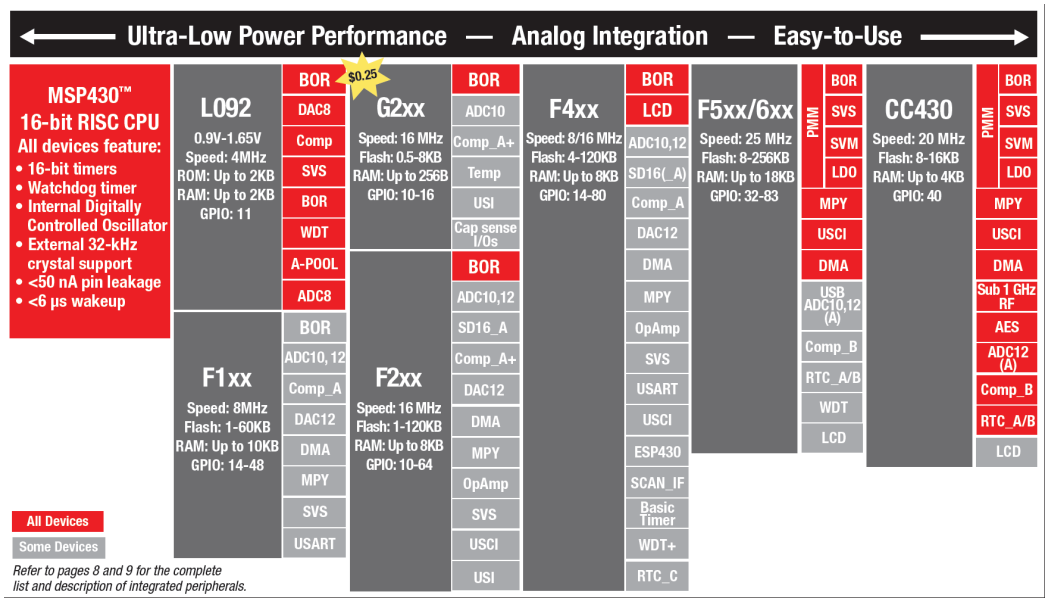


Figure 3. Review of MSP430 subfamilies

Development tools

Hardware development tools

Texas Instruments and third-party vendors provide the following groups of hardware development tools: 1) programming and debugging tools; 2) experimenter's boards and eZ430 development tools; 3) production programmers.

The first group (Table 1) includes the tools intended for development of microcontroller's software, its transfer into a microcontroller and in-circuit debugging.

Table 1. Debugging and programming tools for MSP430

Tool	MSP-FET430PIF	MSP-FET430UIF	eZ430(-F2013)	LaunchPad
Price	$20	$99	$20	$4.30
MCU interface	LPT	USB	USB	USB
PC interface	JTAG	JTAG + Spy Bi-Wire	Spy Bi-Wire	Spy Bi-Wire
Short description	Low cost interfaced programmer/ debugger	The most versatile programmer/ debugger	Low cost programmer/debugger and experimenter's board	Very low cost experiemter's board with integrated programmer/debugger (can be used as programmer/debugger for external MCUs)

The second group includes PCBs with MSP430 microcontrollers and specific hardware (for instance, RF transceivers, energy harvesters, displays etc.). One of the experimenter's boards, namely, LaunchPad Development Kit provides not only the microcontroller with some external components, but also

programming/debugging tools for them as well as for external microcontroller with Spy Bi-Wire interface at extremely competitive price $4.30

The third group includes programmers for mass-volume programming of MCUs. This group includes two devices: 1) MSP-GANG430 with serial interface provided by TI for 199$; 2) GangPro430 with USB interface provided by Elprotronic for 339$. For high-volume customers factory programming of masked ROM or flash devices is available. The ROM programming takes 8-12 weeks, but flash programming – 6-8 weeks staring from the receipt of customer's code to the first production samples.

Software development tools (for program development)

The most popular software for development of microcontroller programs is listed in Table 2. The software provides also programming and in-circuit debugging functions. It specially must be noted that "Code Compose Studio v4 MCU Core Edition" and "IAR - Kick Start for MSP430" have limitation for programming in C, while assembler programming is unlimited. Besides that, for many particular MCUs size limitation exceeds the maximal volume of program memory (and it fact does not work)

Table 2. Software development tools for MSP430

Software Tool	Description	Price	Manufacturer
Code Composer Studio v4 MCU Core Edition	Free 16KB limited Eclipse-based IDE. (Formerly CCE v3 Core Edition)	Free	Texas Instruments
Code Composer Studio v4 MCU Edition	Unrestricted Eclipse-based IDE for MSP430. (Formerly CCE v3 Professional)	$445	Texas Instruments
IAR - Kick Start for MSP430	Code limited IDE: 4KB (MSP430), 8KB (MSP430X), 16KB (eZ430)	Free	IAR Systems
IAR Embedded Workbench for MSP430	Unrestricted IDE for MSP430	???	IAR Systems
MSPGCC	Open-source GCC tool-chain for MSP430	Free	SourceForge
CrossWorks	Unrestricted IDE for Windows, Linux or MacOSX	$1500	Rowley Associates

Part II: Basic peripheral devices of MSP430

It is useful to start training on MSP430 from the hardware that is critical for MCU's operation. This regards first of the all the basic clock system, watchdog timer and digital input/output ports. In this part of the course the most necessary information on these devices is given.

Introduction to Basic Clock System

Basics of clock system

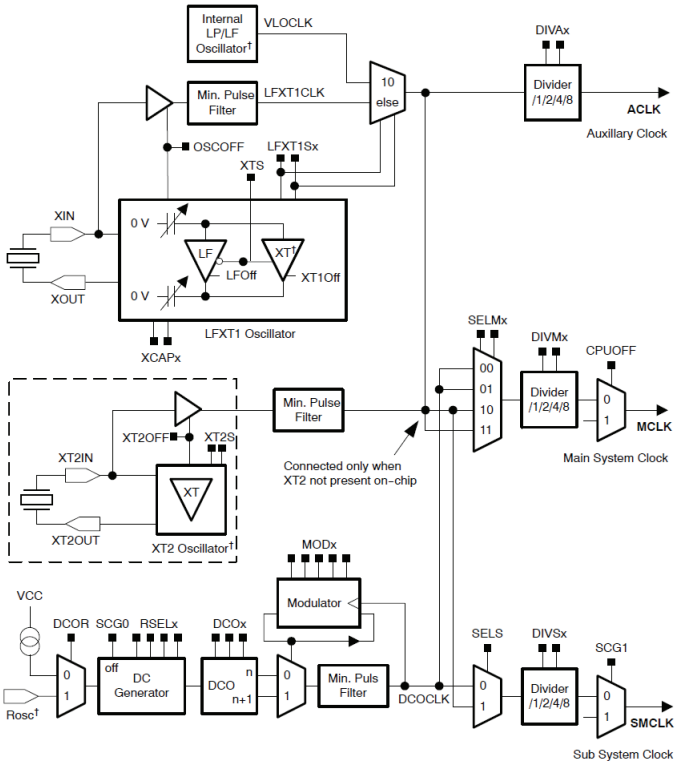


Figure 4. Generalized structure of clock generation system of MSP microcontrollers

Any MSP microcontroller includes an advanced system of clock signal generation. It contains up to 4 (depends on the particular MCU's model) initial clock sources that generates primary signals: DOCLCK, LFXT1CLK, XT2CLK and VLOCLK. Besides that clock system includes also three system clock controllers that form the main clock (MCLK), subsystem clock (SMCLK) and auxiliary clock (ACLK). The controllers include a multiplexer and divider (Figure 4). The modules of clock system are software controllable - it is possible to disable or enable them through the bits of the control registers of the clock system. The described system provides opportunity to find a reasonable trade-off between the clock frequency and power consumption for each peripheral module.

Primary clock sources

VLOCLK is a 12kHz signal generated by a very low power generator VLO. This signal has a quite high frequency temperature drift (0.5%/^{0}C) and frequency supply voltage drift (4%/V) but its generator consumes extremely low power. This clock signal is not utilized in the course.

Signals LFXT1CLK and XT2CLK are high precision signals referenced from a crystal or an active signal through XIN pin. These signals require a special startup procedure initiated and controlled by the software of microcontroller. These clock signals are also not utilized in the course.

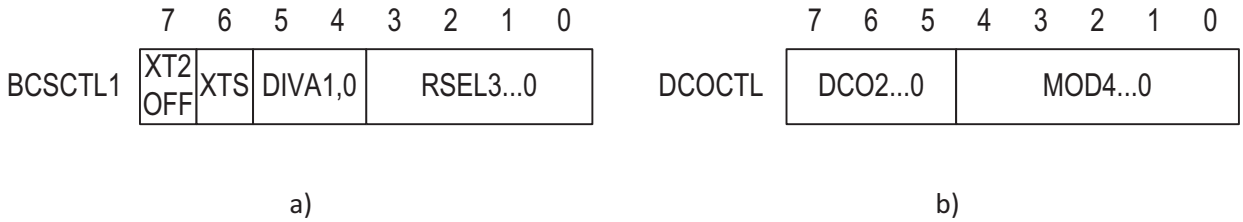


Figure 5. DCO control registers

One more primary clock signal is software programmable DCOLK. Range of frequencies for this signal is chosen by bits RSEL3...0 from 8-bit clock control register BCSCTL1 (Figure 5-a) while the particular value of the frequency – by bits DCO0...2 from DCOCTL control register (Figure 5-b).

All the peripherals of MSP430 are controlled through the dedicated control registers – specific 8-bit or 16-bit memory cells located at particular addresses. C programming of these memory cells is done utilizing special constants and system variables defined in the header file of the microcontroller. Their readings more or less correspond to the titles of the registers and their bits.

The whole range of frequencies is 100kHz…20MHz but ranges overlap. Approximate values of frequency are given in Table 3. Since the frequency is fully programmable through the bits of BCSCTL1 and DCOCTL the maximal frequency can be chosen by commands:

```
DCOCTL = DCO2 + DCO1 + DCO0
BCSCTL1 = XT2OFF + RSEL3 + RSEL2 + RSEL1 + RSEL0
```

while the minimal – by commands:

```
DCOCTL = 0
BCSCTL1 = 0
```

Here DCO… and RSEL… are C constants that in conjunction create the necessary code. These constants as well as the addresses of control registers DCOCTL (0x0056) and BCSCTL1 (0x0057) are defined in header file for the particular microcontroller (for instance “msp430g2231.h”).

Form Figure 5 is seen that the content of BCSCTL1 is not composed only of DCO control bits. For this reason frequency control bits for intermediate values of DCO frequency have to be modified through sequential logical “and” and “or” operations.

In the given course DCO generator will be run at maximal frequency of 16MHz.

Table 3. DCOCLK frequency vs. DCO and RSEL bits

DCO RSEL	0	1	2	3	4	5	6	7
0	95	103	111	120	130	140	151	163
1	119	129	139	150	162	175	189	204
2	167	180	194	210	227	245	265	286
3	238	257	278	300	324	350	378	408
4	325	352	380	410	443	478	516	558
5	460	497	537	580	626	677	731	789
6	635	686	741	800	864	933	1008	1088
7	857	926	1000	1080	1166	1260	1360	1469
8	1270	1372	1481	1600	1728	1866	2016	2177

9	1826	1972	2130	2300	2484	2683	2897	3129
10	2699	2915	3148	3400	3672	3966	4283	4626
11	3374	3644	3935	4250	4590	4957	5354	5782
12	4557	4921	5315	5740	6199	6695	7231	7809
13	6192	6687	7222	7800	8424	9098	9826	10612
14	8359	9028	9750	10530	11372	12282	13265	14326
15	12106	13074	14120	15250	16470	17788	19211	20747

System clock controllers

As it has been mentioned above basic clock system includes also system clock controllers. ACLK clock source is always connected to LFXT1 generator that is disabled by default. That is why f_{ACLK}=0. This clock source will not be used in exercises.

In this course CPU for better performance must be run at the maximal speed and f_{MCLK} must also be maximal. This can be achieved by applying the minimal divider that is default option of the clock system. No modifications of clock control registers are required therefore.

The same regards the third system clock source SMCLK that is used to synchronize PWM generating peripheral device (Timer A). Since switching frequency is chosen equal to f_{SW}=80kHz, but PWM must have at least 1% resolution then f_{SMCLK} must be at least 80kHz×100%=8MHz. Therefore it is reasonable to keep SMCLK clock source at level 16MHz that is also possible with minimal divider that, in turn, is default value.

Introduction to Watchdog

Watchdog timer is a digital counter that periodically resets the microcontroller thus interrupting faulty deadlock states. In order to avoid these periodical resets the watchdog timer must be kept hold or periodically cleared more frequently than the watchdog resets occur.

In the given course watchdog protection is not so urgent and watchdog timer will be stopped with the command:

```
WDTCTL = WDTPW + WDTHOLD
```

Otherwise it must be cleared more than once per each 32768 clock periods of SMCLK signal. This can be done with the command

```
WDTCTL = WDTPW + WDTCNTCL
```

Here WDTPW, WDTHOLD and WDTCLR are C constants. In conjunction they create the necessary binary code for watchdog control register WDTCTL shown in Figure 6. These constants as well as the address corresponding to WDTCTL (0x0120) are defined in header file for the particular microcontroller (for instance “msp430g2231.h”).

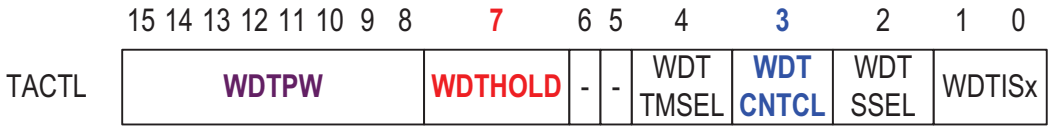


Figure 6. Watchdog control register

Introduction to digital interfacing

Basic information about digital ports

Exchange of digital information is based on the definition of digital ports. Port is a group of microcontroller's contacts (usually – 8) intended for input and output of digital signals. All contacts of the same port are tuned through the same set of control registers. The particular bits of the register control the corresponding contacts of the port.

Each port (number x) of MSP430 has the following 8-bit control registers:

PxSEL – defines connection for port's contacts: if 0 – the contact is attached to the port, if 1 – the contact is attached to some other peripheral device.

PxDIR – defines directions for port's contacts: if 0 – the contact operates as an input, if 1 – as an output.

PxOUT – defines output information for the contacts that operates as outputs.

PxIN – acquires the information from the port's contacts. If a contact operates as an output the corresponding bits of this register just read the output information. This register has not to be written.

PxREN – if a bit of this register is set, a pull-up resistor is attached to the corresponding port's contact.

Microcontrollers MSP430G2231 shown in Figure 7 have 2 ports: 8-bit port P1 and port P2 that has only two contacts 6th and 7th. By default these contacts are not attached to the port but to basic clock system and intended for external crystal connection. Therefore P2SEL initially is 11000000b (b means a binary number).

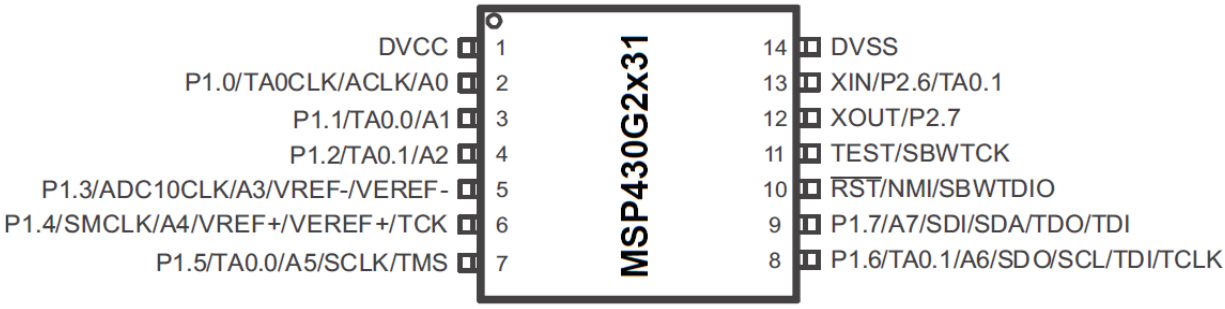


Figure 7. Pinout of MSP430G2231

Basic operations with ports

If it is necessary to modify all bits in a port control register (some must be set, some – reset) the register can be directly written by a command:

```
P1DIR = 0xF0
```

The command sets 4 most significant bits in the direction register, but 4 least significant – resets.

If it is necessary only to set some bits in the port control register use bitwise logical OR. Then one of the operand can be regarded as a mask whose 1s set bits in the other operand. For instance, 4 most significant bits can be set by the command where 4 most significant bits in the mask are 1s, but other (4 least significant bits) are 0s that generates mask 0xF0:

```
P1DIR |= 0x0F
```
or
```
P1DIR = P1DIR | 0x0F
```

If some of the control bits have to be reset, use bitwise logical AND. Then the meaning of the mask bits is different – 0s in the mask produce 0s in another operand. For example 4 most significant bits are reset by the command where 4 most significant bits in the mask are 0s, but other 4 (least significant) bits are 1s:

`P1DIR &= 0xF0` or `P1DIR = P1DIR & 0xF0`

Sometimes it is useful to clear bits by a mask where active values are 1s (like for setting bits). Then the bitwise logical AND must be applied to the inverted mask. The next command produces the same result as the previous one:

`P1DIR &= ~0x0F` or `P1DIR = P1DIR & (~0x0F)`

If is also possible to invert selected bits utilizing bitwise logical XOR. For example odd (1,3,5 and 7) bits of direction register are inverted by the command:

`P1DIR ^= 0xAA` or `P1DIR = P1DIR ^ 0xAA`

Utilizing C constants BIT0…BIT7 defined in header files often makes programs mode readable. This rule is especially effective with the port control registers. For example, the previous command can be written in another (more comprehensive way):

```
P1DIR ^= BIT7+BIT5+BIT3+BIT1
```

Simple examples of MCU programming

Setting digital outputs

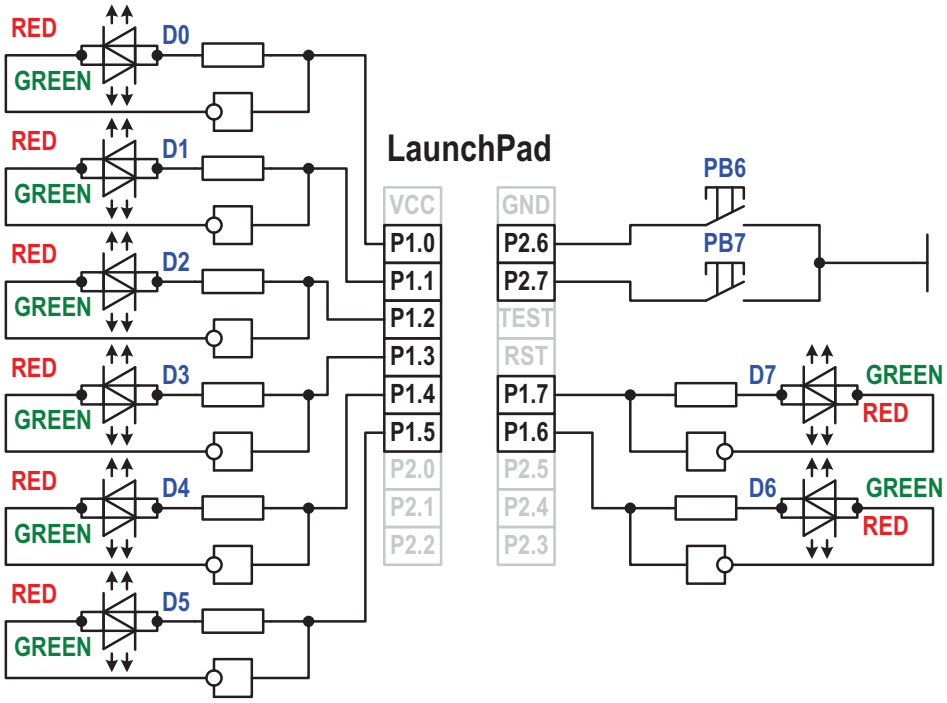


Figure 8. Training configuration of microprocessor system with LaunchPad

In order to train basic skills of programming of CPU basic clock system, watchdog and ports, a simple microprocessor system with LaunchPad development kit has been developed. As it is seen from Figure 8 besides the kit it includes 2 pushbuttons and 8 doubled LED (2 back to back connecter LEDs in the same package). The pushbuttons connect (when pressed) contacts of port P2 and ground (which assumes utilization of internal pull-up resistors). The LEDs are placed between contacts of port P1 and outputs of inverters that are connected to the same port pins. When a contact is configured as an output and

generates 1, the internal red LED is on. When the contact is output with 0, green LED is on. When the contact operates as input, both LEDs are off.

Let's define the task to indicate first 8 green lights, make 400ms delay, indicate 4 green (MSBs) and 4 red (LSBs), 400ms delay, 4 green (MSBs) and 4 red (LSBs), 400ms delay, 8 red and 400ms delay.

In the given example it is not necessary to tune the basic clock system (because time delay may be adjusted as in software loops). At the same time all pins of port P1 has to be programmed as outputs. For this reason initial commands of Program 1 regard only the watchdog timer that must be stopped and writing all ones (1) in P1DIR.

In the main loop 4 combinations of port P1 programming followed by four 400ms delays form an endless loop organized with "for" operator. Software loops for time delay are implemented as "do-while" cycle in the dedicated function "delay". Time delay can be changed writing another initial cycle variable "i" (constant "DelayCode").

Program 1. An example of simplified ports programming

```
#include "msp430g2231.h"
#define DelayCode (50000)                 //Delay Defining Number
//*****************************************************************************
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;               // Stop watchdog timer
  P1DIR = 0xFF;                           // Set P1.x to output direction

  for (;;)
  {
    P1OUT = 0x00;                         // Indicate GGGGGGGG
    delay();
    P1OUT = 0x0F;                         // Indicate GGGGRRRR
    delay();
    P1OUT = 0xF0;                         // Indicate RRRRGGGG
    delay();
    P1OUT = 0xFF;                         // Indicate RRRRRRRR
    delay();
  }
}
//*****************************************************************************
int delay(void)
{
  volatile unsigned int i;                // volatile to prevent optimization
  i = DelayCode;                          // SW Delay
  do i--;
  while (i != 0);
  return 0;
}
//*****************************************************************************
```

Program 2 shows how to solve the previous task through inverting the port bits. Some commands then can be excluded from the main cycle.

Program 2. Utilizing inversion programming ports (part)

```
  for (;;)
  {
    P1OUT = P1OUT ^ 0x0F;                  // Invert 4LSB in P1
    delay();
    P1OUT = P1OUT ^ 0xFF;                  // Invert all bits in P1
    delay();
  }
```

Acquiring digital inputs

The next fundamental task is acquiring the information from ports. A simple way to do this is utilization of bitwise logical AND of PxIN and a mask with only one 1 (constants BIT0…BIT7 are suitable) like P2IN&BIT7. Such operation produces either 0 or 1 (non zero) and can easily be used with “if” and “while” statements. Program 3 shows how to use this principle on the example of LED programming depending on the pushbutton attached to P2.7 (if not pressed – all red, if pressed – all green). The program has two specific commands related to the port P2. The first one P2SEL=0x00 disconnects the corresponding pins from the basic clock system and connects to the port while the second P2REN=BIT7+BIT6 attach pull-up resistors to pins.

Program 3. Analyzing port information with “if” statement

```
#include "msp430g2231.h"
#define DelayCode (50000)                  //Delay Defining Number
//*************************************************************************
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                // Stop watchdog timer
  P1DIR = 0xFF;                            // Set P1.x to output direction
  P2DIR = 0x00;                            // P2.6 and P2.7 inputs
  P2SEL = 0x00;                            // Connect pins to port
  P2REN = BIT7 + BIT6;                     // Connect pins to port

  for (;;)
  {
    if (P2IN & BIT7)
      P1OUT = 0x00;                        // Switch-off P1 LEDs
    else
      P1OUT = 0xFF;                        // Switch-on P1 LEDs
  }
}
//*************************************************************************
```

The above described method works well if the number of input signals is low (1 in Program 3). If this number is high the program may become bulky. For instance, if the number of signal is 3 then the number of possible branches is 8, but number of one bit conditions is 7. In such occasions it is useful to combine statements like P2IN&BIT7 with “switch” and “case”. Program 4 turns-on 4 previously described LED combinations depending on the state of 2 pushbuttons attached to MCU.

Program 4. Analyzing port information with "switch" statement

```
#include "msp430g2231.h"
#define DelayCode (50000)                   //Delay Defining Number
//*************************************************************************
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop watchdog timer
  P1DIR = 0xFF;                             // Set P1.x to output direction
  P2DIR = 0x00;                             // P2.6 and P2.7 inputs
  P2SEL = 0x00;                             // Connect pins to port
  P2REN = BIT7 + BIT6;                      // Connect pins to port

  for (;;)
  {
    switch (~P2IN & (BIT7 + BIT6))
    {
      case 0:
        P1OUT = 0x00;                       // Indicate GGGGGGGG
        break;
      case 64:
        P1OUT = 0x0F;                       // Indicate GGGGRRRR
        break;
      case 128:
        P1OUT = 0xF0;                       // Indicate RRRRGGGG
        break;
      case 192:
        P1OUT = 0xFF;                       // Indicate RRRRRRRR
        break;
    }
  }
}
//*************************************************************************
```

Summary on basic peripheral devices

There are some peripheral devices of MSP430 that cannot be left in their default state. This regards the basic clock system (it usually has to be reprogrammed to higher operation frequency), watchdog (that must be stopped or frequently reset) and I/O ports (that communicates the external hardware and must be programmed properly.

In the given course CPU clock source MCLK and clock source for peripheral devices SMCLK are kept at the highest level of 16MHz. This is done through setting DCO0…2 control bits in DCOCTL control register and RSEL0…3 bits in BCSCTL1 register keeping at the same time divider values for MCLK and SMCLK at their default minimal level 1.

The watchdog timer can be stopped by writing WDTHOLD+WDTPW into its register WDTCTL, but reset – by writing WDTCLR+WDTPW.

I/O ports are controlled through a set of control registers. The most significant registers are PxDIR (defines directions), PxOUT (defines output information) and PxIN (reads information directly from the port pins). Controlling of the ports is usually done through bitwise logical operations with port control registers.

Part III: MSP430 hardware for automatic control

Automatic control of power converters

The main content of this course is design and optimisation of a microcontroller based control system for a simple power converter. In order to simplify the example a one switch boost (step-up) DC to DC converter is taken as an object of control. Its schematic is given in Figure 9-a. When the boost converter is operating without any feedbacks it can be represented by a transfer function of its duty cycle to output voltage while other significant variables (input voltage, output current, temperature etc.) are parameters of its operation. Such block is graphically presented in Figure 9-b. The static part of the transfer function is nonlinear:

$V_o=V_d \cdot /(1–D)= V_d \cdot G$, where $G=1/(1–D)$

but dynamical part, typically, is a second order transfer function.

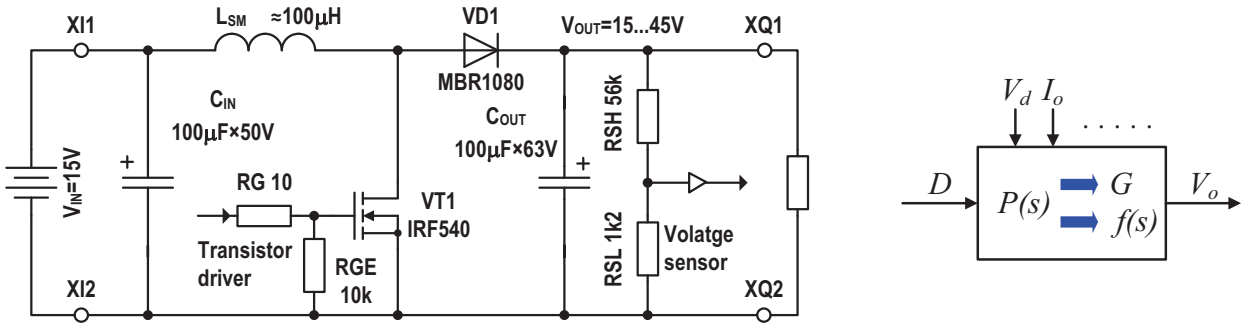


Figure 9. Boost converter as an object of regulation

In conjunction with a control system the boost converter forms a voltage stabilizer. Obviously, keeping the output voltage constant the stabilizer requires also an output voltage feedback that forms a closed loop system - Figure 10-a. Then the functions of the control system are: 1) measurement of the output voltage; 2) calculation of difference between reference signal (setpoint - SP) and measured voltage (process variable – PV); 3) calculation of duty cycle in correspondence with the structure of the regulator; 4) generation of PWM signal. The corresponding, more detailed, control system is given in Figure 10-b. The complete schematic of the proposed control system is given in Appendix A.

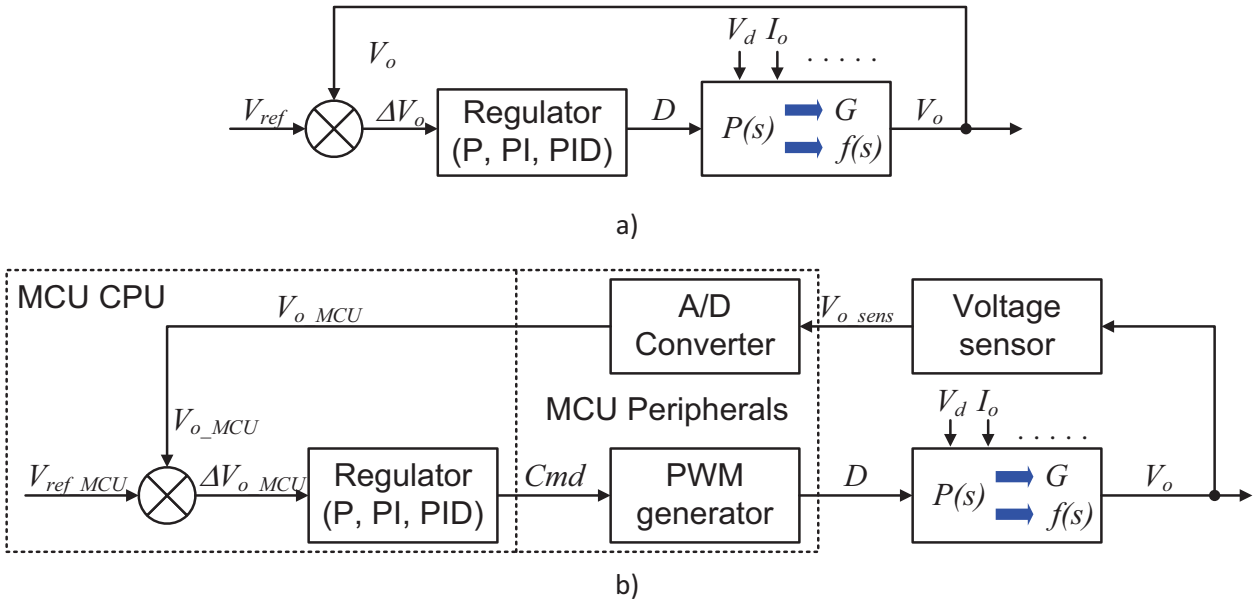


Figure 10. Voltage stabilizer with a boost converter and microprocessor controller

From Figure 10-b is seen, that the regulator (MCU) and object of regulation (boost converter) have two interface signals: pulse mode signal controlling the transistor and output voltage feedback. In order to provide them two kinds of microcontroller hardware must be studied: PWM (or FM) generator and Analog to Digital Converter.

Timer A

The peripheral device that is capable of generating PWM signals is Timer A. The sections below provide basic information on the operation of the Timer A and explain how to use it to generate pulse mode signals.

Basics of Timer A

Timer A is a programmable 16-bit counter that is capable of counting internal and external pulses (automatic increment per each pulse). If Timer A counts internal pulses of the constant and known period then it can be regarded as a timer - time counting device. If it counts external pulses of unknown and variable period then it is regarded as an event counter. Embedded compare function of Timer A makes it capable to source pulse width modulated (PWM) and frequency modulated (FM) control signals.

The counting 16-bit register of Timer A is called TAR and can be read or written at any time. There exist also other registers associated with Timer A. Some of them perform control functions while the others are auxiliary data registers attached to the timer. These registers will be introduced in the corresponding place later.

Once programmed Timer A can operate stand-alone as long as it necessary and requires software intervention only for reprogramming.

Counting modes of Timer A

There are 3 possible modes of Timer A operation: continuous, up and up/down modes.

Continuous mode

In the continuous counting mode (Figure 11) register TAR counts from 0 to the maximal 16-bit number 65535 and then instantaneously returns to 0. Then the full cycle (period) of the timer consists of 65536 periods of TA clock – T_{TACLK}.

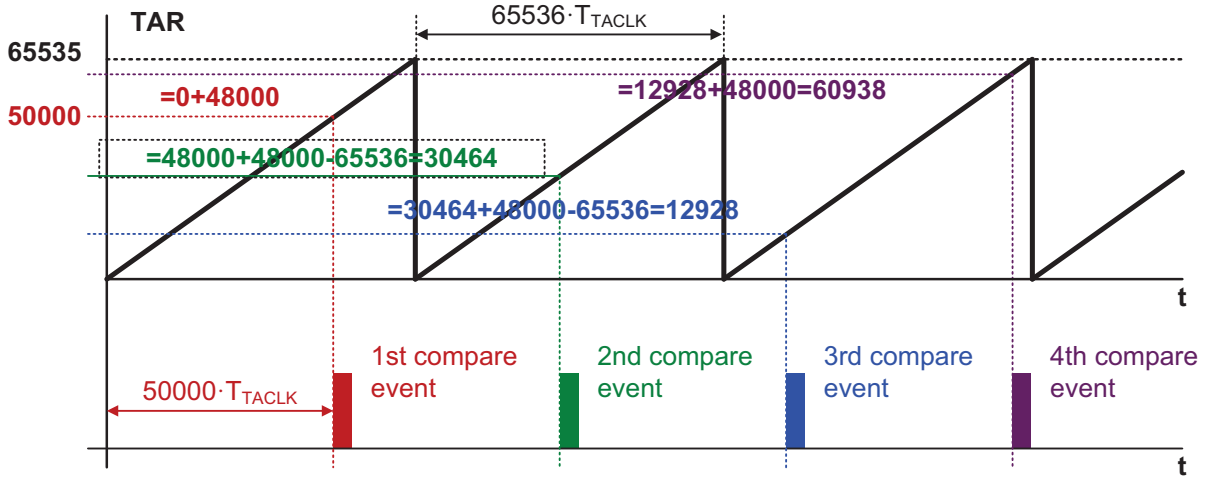


Figure 11. Continuous mode of TA operation and time interval measurements

Continuous mode is convenient for measurements of constant or variable time intervals. Then the new number for comparison can be obtained as a sum of the current TAR and measured time interval (it of

course must be smaller than 65535). The comparison event can be detected through the continuous monitoring of TAR and comparing it with the new number. This procedure can be done manually or automatically utilizing the compare function of the timer TA.

Up mode

In the up counting mode TAR counts from 0 to a 16-bit number written in special register TACCR0 (Figure 12). After that TAR instantaneously returns to 0. Then the full period of the timer (in T_{TACLK}) is equal to the content of TACCR0. It is possible to change the period of TA operation through modifying TACCR0.

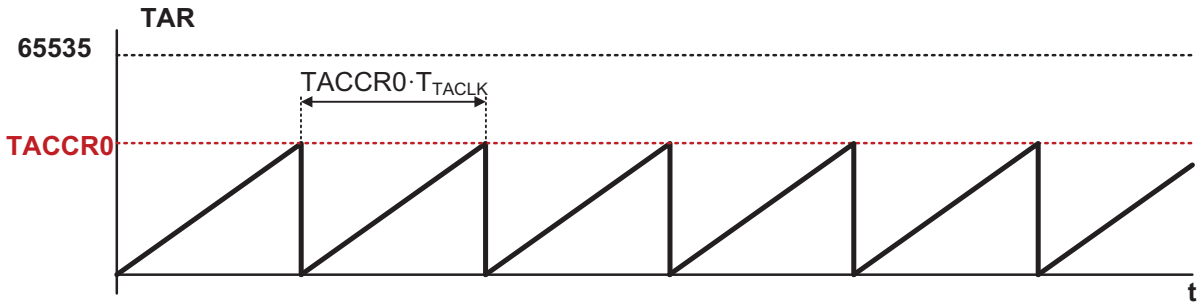


Figure 12. Up mode of TA operation

Up-Down mode

In the up-down mode TAR counts up from 0 to a 16-bit number written in TACCR0 and then counts down back from TACCR0 to 0. As it is seen in Figure 13 the full period of the timer (in T_{TACLK}) is equal to the doubled content of TACCR0, but it is still possible to change the period of TA operation through modifying TACCR0.

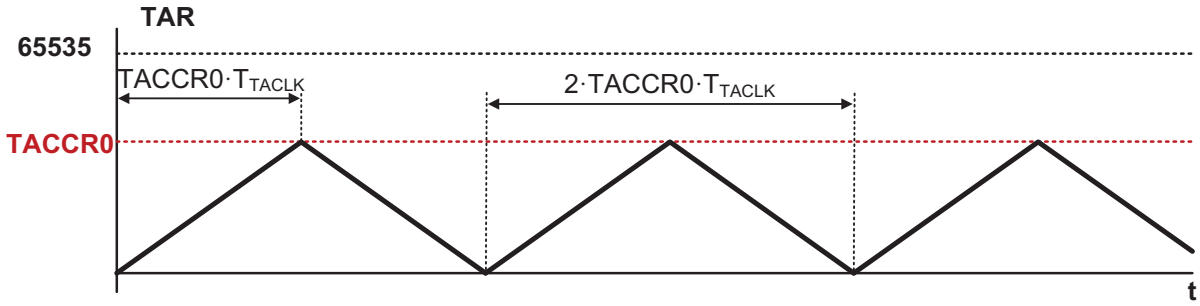


Figure 13. Up-down mode of TA operation

Choice of counting mode for Timer A

Counting modes of Timer A, as well as other options of its operation are chosen through the specific bits in the control registers of Timer A – TACTL (Figure 14).

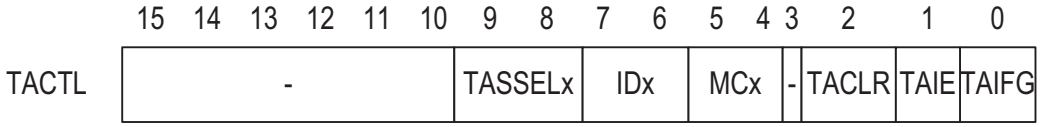


Figure 14. TA control register

Register TACTL has the following bits and their values:

TASSEL1, 0 – select clock signal for timer TA (00 – external signal TACLK, 01 – internal clock ACLK, 10 – internal clock SMCLK, 11 – external signal INCLK);

ID1, 0 – select a divider for clock (00 – no divider or divider 1, 01 – divider 2, 10 – divider 4, 11, divider 8);

MC1, 0 – select the counting mode (00 – selects no counting, 01 - up mode, 10 – continuous mode, but 11 – up-down mode);

TACLR – value 1 resets the counting register TAR; this is a dynamic bit – it is1 only one clock cycle writing and then is automatically cleared; always read as 0;

TAIE – enable bit of TA interrupts;

TAIFG – interrupt flag of TA interrupts.

It has been mentioned that the up-down mode is programmed with MC=11 that requires the following command:

```
TACTL = MC_3
```

The counting mode of Timer TA is usually programmed in the beginning of a program. Then it is reasonable to program all options of the timer at once. For instance, programming the up-down mode, resetting TAR, choice of clock frequency (SMCLK) and setting input divider (8) require the following command:

```
TACTL = TASSEL_2 + ID_3 + MC_3 + TACLR
```

It must be also noted that the up-down mode is preferable for generating PWM signals because it provides so called “dead” times for both transients (1 to 0 and 0 to 1) that is useful for controlling complimentary transistors. In this course up-down mode will be utilized for generating PWM signals.

The above examples contain C constants TASSEL_2, MC_3 and TACLR. Like in previous examples these constants in conjunction create the necessary programming code. They, as well as the addresses of control register TACTL (0x0160) are defined in header file for the particular microcontroller (like “msp430g2231.h”).

Compare(/capture) modules of the timer TA

Timer TA itself does not provide PWM. Pulse mode signals are generated by compare modules associated with the timer that are discussed below.

Basic information

Compare event is equality state of the counting register TAR and special compare data register TACCRx (for instance TACCR1). As soon as the compare event has happen its flag CCIFG in the compare control register TACCTLx (for example TACCTL1) is turned on. In addition the output signal of the compare module is toggled in the way defined by dedicated bits of the compare control register.

Each compare module X can be tuned and controlled through compare control register TACCTLx that is presented in Figure 15.

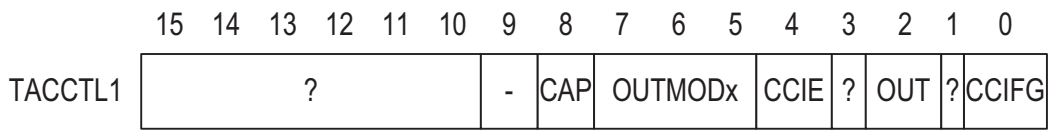


Figure 15. Control register of compare module

Register TACCTLx has the following bits and their values that regard the compare function:

CAP – this bit must be 0 to activate compare function;

OUTMOD2, 1, 0 – defines rules of generation of the output signal of the module; Value 0 programs direct control of the output from bit OUT, value 1 – set mode, 2 – toggle/reset mode, 3 – set/reset mode, 4 – toggle mode, 5 – reset mode, 6 – toggle/set mode and 7 – reset/set mode (Figure 16);

CCIE – enable bit for compare interrupts;

CCIFG – flag of compare interrupts;

OUT – this bit defines state of the output signal in the output mode0

Output signals

The output signals of compare modules are generated on the dedicated pins of microcontroller. Information about the particular pins can be found in microcontroller's datasheet. For example, in MSP430G2231 Timer A has 2 compare modules (0 and 1) and two outputs, but there are up to 5 possible connections of these outputs that are given in Table 4.

Table 4. Connections of outputs of compare modules of Timer A in MSP320G2231

Output signal	Pin (in 14-pin DIP)	Port
Out 0	3	P1.1
Out 1	4	P1.2
Out 0	7	P1.5
Out 1	8	P1.6
Out 1	13	P2.6

From the datasheets and from Table 4 is seen that the output of compare modules have pins shared with other devices (first of the all with digital input/output ports). The choice of connected device is done through the bits of registers PxSEL. Value 1 disconnects the pin from the port and connects to a peripheral device, for instance, to a compare module. If, for example, it is necessary to utilize pin 3 as output of compare module 0 and pin 13 as output of compare module 1 then the following lines have to be included in the program:

```
P1SEL = BIT1
P2SEL = BIT6
```

Here BIT1 and BIT6 are C constants that create the necessary code for SEL registers. These constants as well as the address corresponding to P1SEL (0x0026) and P2SEL (0x002E) are defined in header file for the particular microcontroller (for instance "msp430g2231.h").

Output modes

The dedicated output signals of the capture modules appear on the microcontroller pin associated with the compare module.

There are two kinds of output patterns: output modes with one compare event and output modes with two compare events. In “set”, “toggle” and “reset” output modes the changes of the corresponding output signal occurs when the counting register TAR become equal to one of compare registers. For example in “reset” mode the output is cleared each time when TAR=TACCRx

In “toggle/reset”, “set/reset”, “toggle/set” and “reset/set” mode switching pattern consists of two events. The first event occurs when TAR become equal to one of compare registers except TACCR0, but the second one – when TAR is equal to TACCR0. For example, in “toggle/reset” mode of compare module 1 its output is inverted at TAR=TACCR1, but cleared – at TAR=TACCR0. Obviously these output modes cannot be used in compare module 0.

“Toggle/reset” output mode is the most suitable for occasions when width of pulses must be proportional to the content of TACCR – i.e. in the case of PWM.

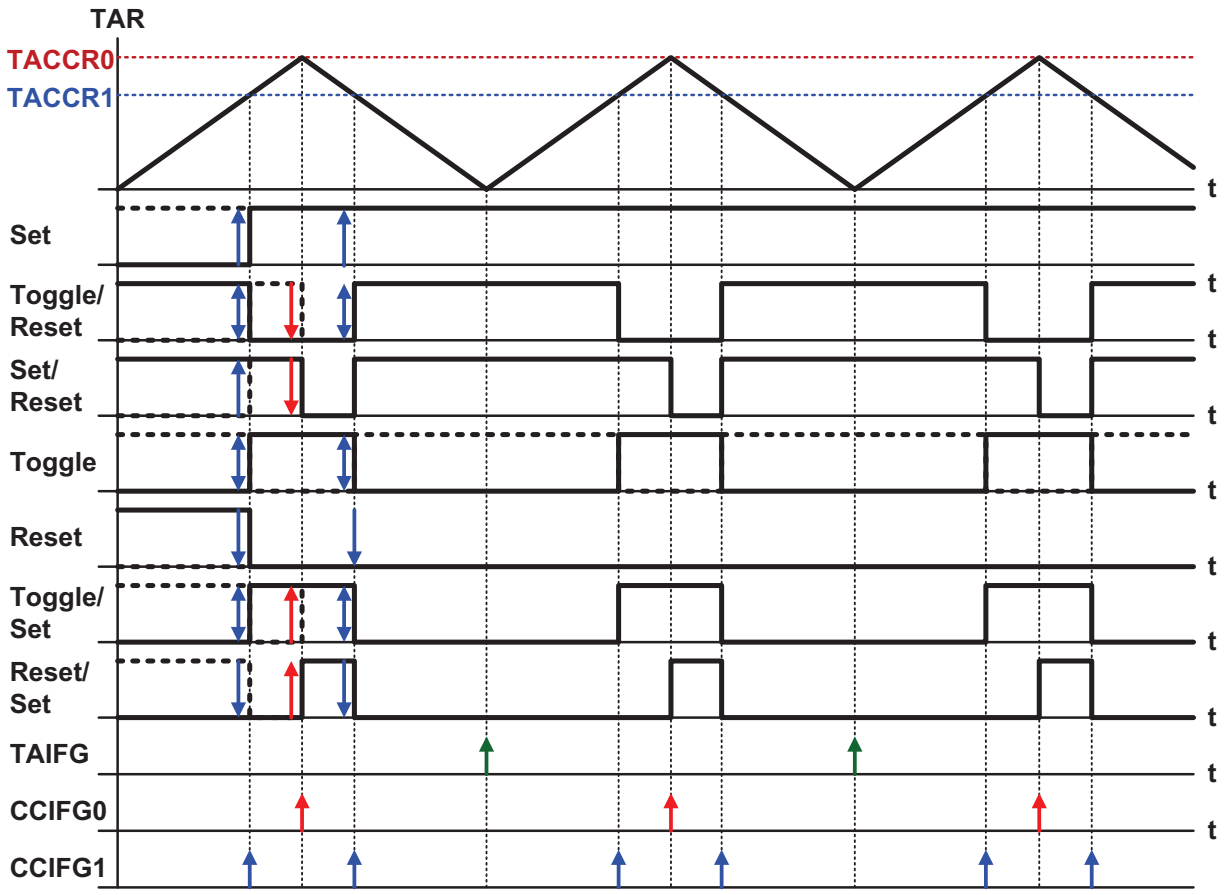


Figure 16. Output modes of compare module of timer TA in up-down counting mode

Example of PWM programming

Form the above mentioned it can be concluded that the complete example of PWM programming consists of several steps that regard not only the timer itself but also other peripheral devices. The following steps of PWM programming can be emphasized:

1. Programming the basic clock system to provide necessary SMCLK or ACLK. In the given course ACLK is not used but MCLK=SMCLK=DCOCLK and f_{SMCLK} =16MHz or T_{SMCLK} =62.5ns.
2. Programming of TA itself. For PWN up-down mode is optimal. In the given example (Program 5) timer’s clock is set to SMCLK, but timer’s period is $T_{TA}=200T_{TACLK}$=12.5µs that corresponds to 80kHz.

3. Programming of a compare module in proper output mode. For example, let's program module 1 in toggle/reset output mode for "directly proportional" PWM. Let's also program the duty cycle of 75% that at the given parameters of the timer TA is achieved with compare value 75.

4. Programming of the dedicated microcontroller's pin. In the given example the generated PWM signal appears at P2.6 (Pin 13).

Program 5. An example of PWM programming

```
BCSCTL1 = XT2OFF + RSEL3 + RSEL2 + RSEL1 + RSEL0;  //Maximal DCO range
DCOCTL = DCO2 + DCO1 + DCO0;                       //Max DCO frequency 16MHz
                                                   //Default divider is 1
TACTL = TASSEL_2 + MC_3 + TACLR;                   //TACLK=SMCLK, up-down mode
TACCR0 = 100;                                      //T_TA=2*100*62.5ns=12.5us
TACCTL1 = OUTMOD_2;                                //Programs toggle/reset mode
TACCR1 = 75;                                       //D=75%
P1SEL = BIT6;                                      //PWM at Pin 8 P1.6
P1DIR = BIT6;                                      //P1.6 - output
```

ADC10 – 10-bit Analog to Digital Converter

Basic information and control registers of ADC10

Analog to Digital converter (ADC10) installed in MSP430G2231 microcontrollers is a 10-bit successive approximation digitally controlled circuit. It is capable of measuring analog signals in single/multiple channel and single/multiple conversion modes. This peripheral module is controlled through a set of registers. In this course multiple channels and multiple conversions is not necessary and all the attention is focused on single channel single conversion. Bellow the control registers of ADC10 are discussed from the point of view of this mode.

Registers ADC10DTC0, ADC10DTC1 and ADC10SA defines data stream for ADC10 to RAM in the case of multiple conversion modes. In single channel single conversion mode they are not used.

Register ADC10MEM contains the conversion result. There are 2 formats of the result: binary format and 2's complement. In the case of binary format (that will be used in later examples) 6MSBs of this registers are 0 while 10LSBs represents the result of conversion – a number from 0 to 1023.

One more register ADC10AE enables (at bit value 1) pins of microcontroller for analog input and disables.

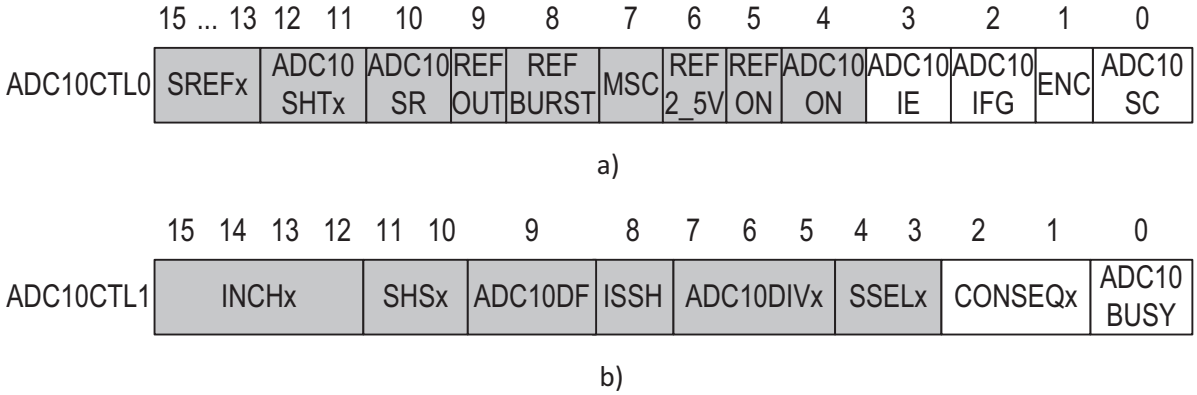


Figure 17. ADC10 control registers

Figure 17-a and Figure 17-b represent control bits of ADC10. Some of these bits are modifiable only if control bit ENC is reset (emphasized with grey background). The control bits that will be modified in the further examples are explained here:

SREF0, 1, 2 – these bits choose a reference for the converter; the values actual in the course are: 000 – for AVCC and AGND, as well as 001 – for internal reference 2.5v/1.5V to AGND;

REF2_5V – must be set for internal reference 1.5V and reset for internal reference 1.5V;

REFON – must be 1 if the internal reference is used

ADC10ON – at 1 turns on ADC10

ENC – enables conversion and disables control bit programming;

ADC10SC – starts conversion; this bit is usually set with ENC;

INCH0, 1, 2, 3 – these bits choose a channel for single channel conversion or the last channel for multiple channel conversion;

SHS0 and 1 – choose the signal for conversion start; must be 00 if ADC10SC is the start bit;

ADC10DF – data format; must be zero for binary format;

CONSEQ0 and 1 – choose conversion sequence; must be 00 for the single channel single conversion mode;

ADC10BUSY – controlled by ADC10; if 1 – conversion is active; if 0 – no conversion.

Programming of ADC10

Simplified programming of ADC10 may be limited to a three kinds of operation: 1) setup ADC; 2) starting a conversion; 3) checking if the conversion is complete.

The setup of ADC10 requires programming of ADC10CTL0 and ADC10CTL1 (usually at disabled ENC). During the setup reference voltage, conversion mode and channel must always be chosen. Analog circuit for the chosen channel must also be activated. Then, for example, 1.5V reference, single channel/conversion of A4 requires the following commands:

```
ADC10CTL0 &= ~ENC;
ADC10CTL0 = SREF_1 + REFON + ADC10ON;
ADC10CTL1 = INCH_4;
ADC10AE0 |= BIT4;
```

But those with AVCC reference – the following:

```
ADC10CTL0 &= ~ENC
ADC10CTL0 = ADC10ON;
ADC10CTL1 = INCH_4;
ADC10AE0 |= BIT4;
```

Conversion starts after ADC10SC bit is set (that is usually done together with ENC). Then the command is:

```
ADC10CTL0 |= ENC + ADC10SC;
```

When the conversion ends ADC10BUSY=0. This rule can be utilized to detect the end of conversion, and then to process new data and start new conversion. Waiting state then can be provided, for example with command:

```
while (ADC10CTL1 & ADC10BUSY);
```

in the above examples ENC, SREF_1, REFON, ADC10ON, INCH_4 are C constants, but ADC10CTL0, ADC10CTL1, ADC10AE system variables (related to the particular addresses 0x01B0, 0x01B2 and 0x004A) defined in header file for the particular microcontroller (for instance “msp430g2231.h”).

Example of Timer A and ADC10 programming

Let’s discuss more one complicated example of common utilization of the Timer A and ADC10 – controlling the duty cycle of PWM at P1.6 by a potentiometer attached to P1.4. Since PWM frequency is the same the initial programming of Timer A also remains as in Program 5. Programming ADC10 with VREF=AVCC has also been discussed previously. The working cycle of the program consists of start conversion command “ADC10CTL0|=ENC+ADC10SC;”, waiting state ”while (ADC10CTL1&ADC10BUSY);”, some calculations “x=5+ADC10MEM*63/716;” and PMW defining command “TACCR1=x;”. The calculation are necessary because ADC10 provides data from the range 0…1023, but TACCR1 must be from 0…100. Minimal and maximal values of the duty cycle are also undesirable. That is why coefficients are chosen so that D=5…95.

Program 6. PWM controlled by ADC10

```
#include "msp430g2231.h"
#define DelayCode (50000)                    //Delay Defining Number
//*****************************************************************************
unsigned int x;

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                              //Stop Watchdog Timer
  BCSCTL1 = RSEL3 + RSEL2 + RSEL1 + RSEL0;               //Maximal DCO range
  DCOCTL = DCO2 + DCO1 + DCO0;                           //Max DCO frequency 16MHz
                                                         //Default divider is 1
  TACCR0 = 100;                                          //T_TA=2*100*62.5ns=12.5us
  TACTL = TASSEL_2 + MC_3 + TACLR;                       //TACLK=SMCLK, up-down mode
  TACCR1 = 5;                                            //D=5%
  TACCTL1 = OUTMOD_2;                                    //Sets toggle/reset mode
  P1DIR = BIT6 + BIT2;                                   //P1.6 and P1.2 - outputs
  P1SEL = BIT6 + BIT2;                                   //PWM at Pin 8,4 P1.6, P1.2

  ADC10CTL0 &= ~ENC;                                     //Disable ADC10
  ADC10CTL0 = ADC10ON;                                   //Vref=AVCC, ADC10 - ON
  ADC10CTL1 = INCH_4;                                    //Measure channel A4
  ADC10AE0 |= BIT4;                                      //Enable analog for A4

  for (;;)
  {
    ADC10CTL0 |= ENC + ADC10SC;                          //Start new conversion
    while (ADC10CTL1 & ADC10BUSY);                       //Wait until done
    x = 5 + ADC10MEM*63/716;                             //Calculate new duty-cycle
    TACCR1 = x;                                          //Set new duty-cycle
  }
}
//*****************************************************************************
```

Summary on peripheral devices for automatic control

Development of a simple control loop with one feedback based on a microcontroller requires utilization of at least two chains of devices. The first chain measures process variable thus providing the feedback, but the second one generates a regulation signal. Some elements of these chains are located in the microcontroller. For the proposed object of regulation (boost converter) the process variable is voltage that through a voltage divider and isolated amplifier is passed to MSP430. Then the useful feedback device is ADC10. On the other hand the converter is regulated by means of a PWM signal driving its transistor. This brings forward Timer A as regulation device.

Timer A is not exactly a PWM generator, but is capable of generating such signals. Timer A is programmable through bits of its control register TACTL. Once programmed Timer A then can operate standalone without further intervention. PWM signals are generated by compare modules of Timer A. The compare event is defined by register TACCRx but the output signal by OUTMOD0…2 bits of compare control register TACCTLx. Output pins associated with compare module has to be attached to it through control register PxSEL of the corresponding port. Like timer itself the compare and output modules operate autonomously and require reprogramming only if some parameter of their operation (for instance – duty cycle) has to be changed.

Like Timer A, the analog to digital converter can operate stand alone and requires mostly initial programming.

Part IV: Building a Simplified Control Loop with MSP430

Interrupts

General information about MCU operation modes

In the above mentioned examples CPU is active (executing commands and making calculations) all the time when the MCU is on. This enables algorithmic approach to firmware design. However, if the number of peripheral modules and events to be processed is big then such kind of firmware may become bulky and unsafe. Besides that, this approach to CPU utilization leads to a huge consumption of energy during MCU operation (because CPU, buses, and peripherals are all the time on).

An alternative approach to hardware and software design assumes that CPU the most of the time stays in idle or sleep mode – do not execute any program. CPU awakes from the sleep mode only after some hardware event "x" (port signal changes, ADC conversion done, TA overflow etc.). Then a special bit named interrupt flag "xIFG" is set and the corresponding software routine starts. As soon as the subroutine is complete MCU and CPU returns to the sleep mode again.

Such approach shown also in Figure 18 is called "interrupt" based software. Then the main function of the software is reaction to interrupt events.

It is important to understand that each event calls its own program that processes the corresponding data.

Interrupt flags are hardware related and always set after the corresponding event even if the interrupts are not used.

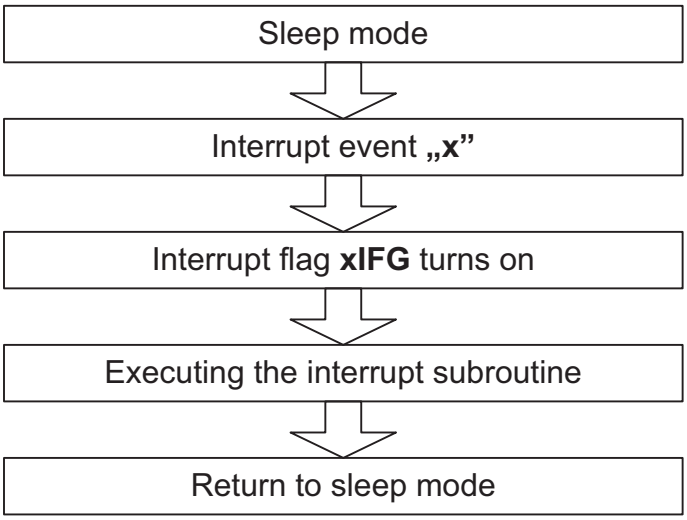


Figure 18. Interrupt service stream

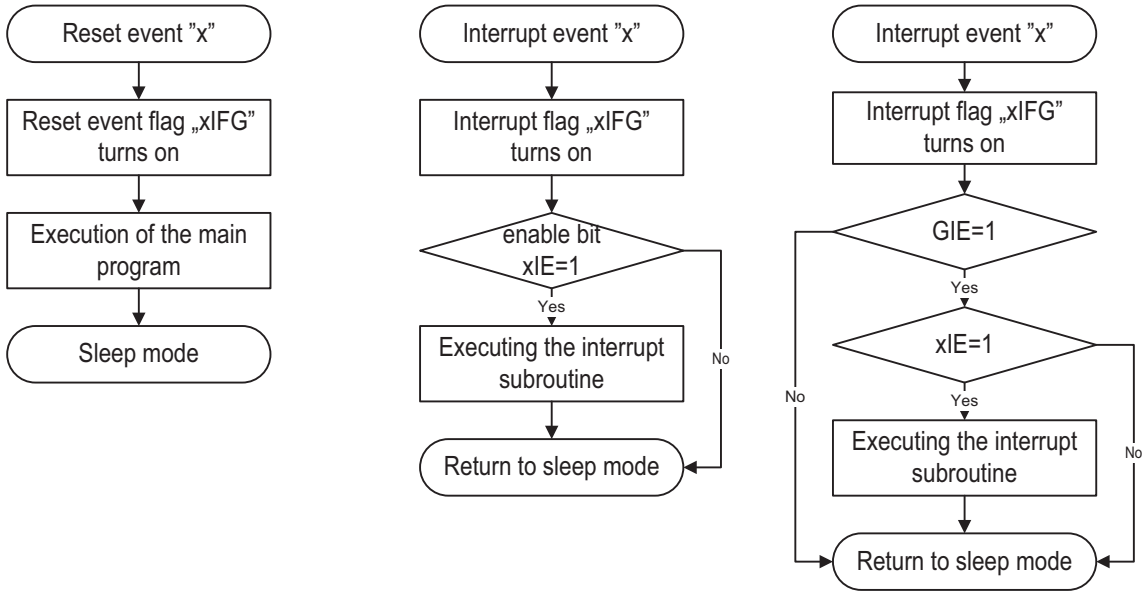
Types of interrupts

There are three types of interrupts: 1) reset interrupt; 2) non-maskable interrupts; 3) maskable interrupts. The listed interrupt types differ by their disable capability.

Reset interrupt shown in Figure 19-a cannot be disabled at all. The corresponding reset interrupt service routine, called also "main program", is always executed after reset event (power-up, reset signal, watchdog overflow etc). The only way to avoid reset interrupt is disabling of the corresponding hardware, for example, stopping the watchdog (with WDTCTL=WDTPW+WDTHOLD).

Each of non-maskable interrupts shown in Figure 19-b can be disabled by the corresponding interrupt enable bit xIE. If the bit is set then the interrupt is enabled. Then after the event "x" interrupt flag is set and the corresponding routine is executed. Otherwise (if xIE is reset), xIFG after the event is set but the subroutine is not executed.

Maskable interrupts are enabled not only with their individual bit xIE, but also with common interrupts enable bit GIE. For this reason it is possible to prohibit all maskable interrupts by clearing GIE. Group of maskable interrupts is the most advanced group. These interrupts are used to control the external hardware and acquire data from sensors.



a) reset interrupt b) non-maskable interrupts c) maskable interrupts

Figure 19. Types of interrupts

Programming maskable interrupts in C

As it is seen from the above mentioned maskable interrupts are activated through setting the individual interrupt enable bit xIE and common interrupt enable but GIE. Bit xIE is usually located in the control registers of the peripheral devices where the interrupt event happens. Bit GIE is located in status register (SR) of CPU and is set by dedicated command together with entering the sleep mode. Also peripheral device must be configured in proper way to generate the interrupt event. At last interrupt service routine must be developed and entered in the correct way.

Let's discuss programming of interrupts in C on the example of Program 6. At first it is necessary to find out what are interrupt options. Program 6 provides control of PWM from a potentiometer attached to MCU. Possible interrupt events then are: 1) ADC10 conversion end; 2) start/end/middle of the switching period; 3) edge transients of the switching pulse. The most obvious event is ADC10 conversion end, but this event happens asynchronously to TA operation that is not good for updating PWM parameters. One more type of event – edge transients of the switching pulse – is not optimal because the data calculated in the event (duty cycle) defines the event itself. Therefore, the most convenient interrupt even is start/end or middle of the modulation period. In the up-down mode the middle of modulation period can be exactly defined in the point where TAR=TACCR0. So, the formal interrupt event for new program is equality of TAR register and compare register TACCR0.

The interrupt flag for this event is bit CCIFG in register TACCTL0, while the enable bit is CCIE in the same register.

The interrupt event is Timer A related. For this reason the timer must be programmed in op-down mode. One more significant issue in the given application is the time delay between interrupts. It must be longer than ADC cycle. If the timer is programmed exactly as in Program 6 then delay between interrupts is 12.5μs. At the same time if ADC operates with 5MHz internal ADC clock (as in example) then conversion sample rate is about 200ksps that corresponds to the conversion time 5μs. Therefore both ADC and TA may be programmed as in Program 6.

From the above considerations is seen that visible modifications of Program 6 are not very significant. They are: 1) setting individual interrupt enable bit CCIE in register TACCTL0 (blue colour); 2) setting the global interrupt enable bit GIE and entering the sleep mode at the end of main program (green); 3) defining the interrupt service routine (red); 4) in order to provide the first interrupt with valid data some initial number is now written to ADC conversion register ADC10MEM (violet).

Program 7. PWM controlled by ADC10 with interrupts

```
#include "msp430g2231.h"
//****************************** Main program ***********************************
void main( void )
{
  WDTCTL = WDTPW + WDTHOLD;                 //Stop watchdog timer
  BCSCTL1 = RSEL3 + RSEL2 + RSEL1 + RSEL0;  //Maximal DCO range
  DCOCTL = DCO2 + DCO1 + DCO0;              //Max DCO frequency 16MHz
                                            //Default divider is 1
  TACCR0 = 100;                             //TTA=2*100*62.5ns=12.5us
  TACCTL0 = CCIE;                           //Enable int. from TACCR0
  TACTL = TASSEL_2 + MC_3 + TACLR;          //TACLK=SMCLK,up-down m.
  TACCR1 = 5;                               //D=5%
  TACCTL1 = OUTMOD_2;                       //Sets toggle/reset mode
  P1DIR = BIT6 + BIT2;                      //P1.6, P1.2 - outputs
  P1SEL = BIT6 + BIT2;                      //PWM @ Pin 8,4 P1.6,P1.2

  ADC10CTL0 &= ~ENC;                        //Disable ADC10
  ADC10CTL0 = ADC10ON;                      //Vref=AVCC, ADC10 - ON
  ADC10CTL1 = INCH_4;                       //Measure channel A4
  ADC10AE0 |= BIT4;                         //Enable analog for A4
  ADC10MEM = 0;                             //Initila ADC data

  _BIS_SR(LPM0_bits + GIE);                 // Enter LPM0 w/ interrupt
}
//******************* Timer A0 interrupt service routine ***********************
#pragma vector=TIMERA0_VECTOR
__interrupt void My_Timer_A (void)
{
  unsigned int x;
  x = 5 + ADC10MEM*63/716;                  //Calculate new duty-cycle
  TACCR1 = x;                               //Set new duty-cycle
  ADC10CTL0 |= ENC + ADC10SC;               //Start new conversion
}
```

The most of the commands previously listed in the endless main cycle now moved to the interrupt subroutine. However, there are some differences. Waiting state for ADC10 conversion cycle completion "while (ADC10CTL1&ADC10BUSY)" is not now necessary because the interrupt occurs always after the conversion cycle is complete. On the other hand new conversion must be initiated after the results of previous one have been processed. For this reason command "ADC10CTL0|=ENC+ADC10SC" is placed at the end of interrupt service routine.

As it is seen Program 7 actually visually does not differ a lot from Program 6. However, practical differences are significant. With new software CPU is in the sleep mode most of the time. Only once per 12.5μs the described interrupt service routine is launched, data of previous AD conversion are processed, new duty cycle is determined and validated, as well as new conversion started.

C construction for interrupt programs is the following

```
#pragma vector=<VECTOR_NAME>
__interrupt void <Interrupt_Routine_Name> (void)
{
...
}
```

In this construction <VECTOR_NAME> is reserved name for the chosen interrupt event defined at the end of header file for particular microcontroller (for example in "msp430g2231.h") while <Interrupt_Routine_Name> is arbitrary chosen name for interrupt service routine (a void function).

Open Loop vs. Closed Loop

Let's present a system corresponding to Program 6 and Program 7 as a chain of transfer blocks. The chain starts from the measurement of converter's output voltage that is done by AD converter (red block). Then the measured voltage in digital form is multiplied by a coefficient (63/716) so, that the valid range (5…95) of values of the duty cycle is obtained. Later this number is written to TACCR1 register (blue block) thus controlling the duty cycle of PWM signal that, through a driver circuit (not shown) is applied to the power transistor of the converter.

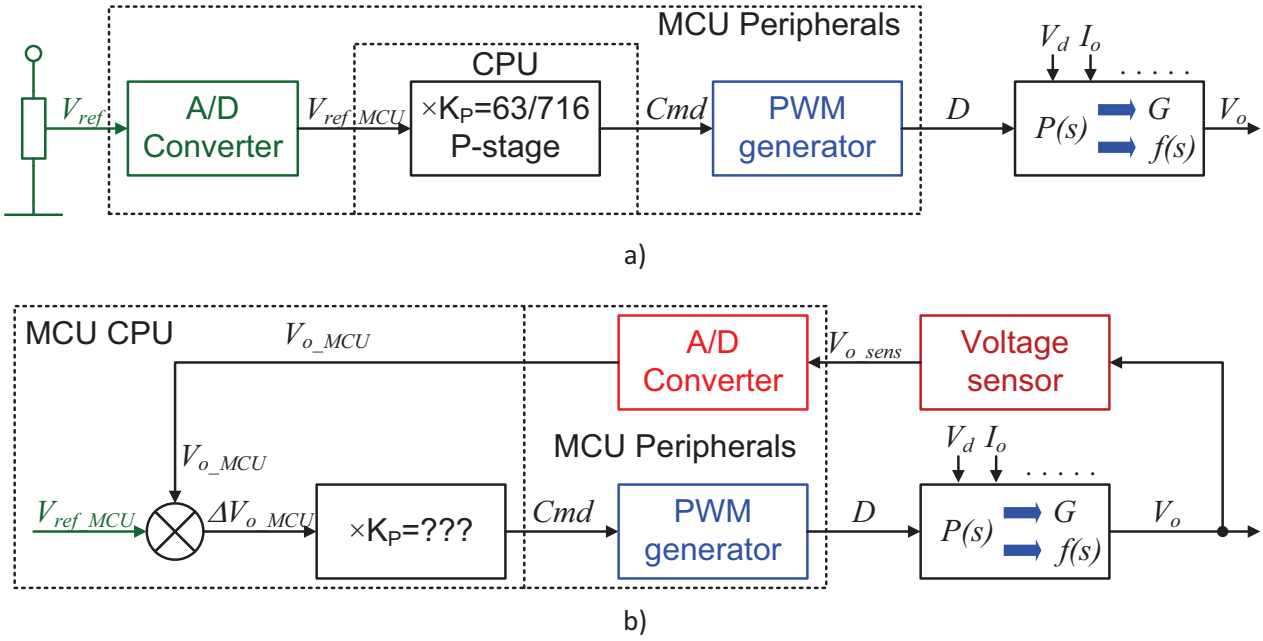


Figure 20. Open loop vs. closed loop

Program 8. Voltage stabilized with P regulator

```
#include "msp430g2231.h"
#define SetPoint (410)
#define MaxD     (80)
//***************************** Main program ***********************************
unsigned int Cmd;
unsigned int DutyCycle;
unsigned int dV;

void main( void )
{
  WDTCTL = WDTPW + WDTHOLD;                          //Stop watchdog timer
  BCSCTL1 = RSEL3 + RSEL2 + RSEL1 + RSEL0;           //Maximal DCO range
  DCOCTL = DCO2 + DCO1 + DCO0;                       //Max DCO frequency 16MHz
                                                     //Default divider is 1

  P1REN = BIT3;                                      //Pull-up at P1.3
  while (P1IN & BIT3);                               //Wait until done PB13 on

  TACCR0 = 100;                                      //TTA=2*100*62.5ns=12.5us
  TACCTL0 = CCIE;                                    //Enable int. from TACCR0
  TACTL = TASSEL_2 + MC_3 + TACLR + ID_1;            //TACLK=SMCLK/2,up-down m
  TACCR1 = 5;                                        //D=5%
  TACCTL1 = OUTMOD_2;                                //Sets toggle/reset mode
  P1DIR = BIT6 + BIT2;                               //P1.6, P1.2 - outputs
  P1SEL = BIT6 + BIT2;                               //PWM @ Pin 8,4 P1.6,P1.2

  ADC10CTL0 &= ~ENC;                                 //Disable ADC10
  ADC10CTL0 = SREF_1 + REFON + ADC10ON;              //Vref=15V, ADC10 - ON
  ADC10CTL1 = INCH_5;                                //Measure channel A4
  ADC10AE0 |= BIT5;                                  //Enable analog for A4
  ADC10MEM = 0;                                      //Initila ADC data


  _BIS_SR(LPM0_bits + GIE);                          // Enter LPM0 w/ interrupt
}
//****************** Timer A0 interrupt service routine ***********************
#pragma vector=TIMERA0_VECTOR
__interrupt void My_Timer_A (void)
{
  dV = SetPoint - ADC10MEM;                          //Find error
  Cmd = dV * 1;                                      //multiply the error
  if (Cmd>=MaxD)                                     //Restrict DutyCycle
    DutyCycle = MaxD;
  else
    DutyCycle = Cmd;
  TACCR1 = DutyCycle;
  ADC10CTL0 |= ENC + ADC10SC;                        //Start new conversion
}
```

The boost regulator produces voltage on its output in correspondence with its regulation law 1/(1-D) and depending on its operational parameters (input voltage, output current, temperature etc.). As it is seen from Figure 20-a there is no any link between regulator's output and control system. It is not possible to provide stable output in such system because the operational parameters all the time affect it.

The simplest way to establish such link – is to make the duty cycle proportional to the difference of a reference point and output voltage. At some point equilibrium of the duty cycle, reference and feedback happens. This equilibrium can be described by the following equation

$$D = K_p \times (V_{ref\_MCU} - V_{o\_MCU}) = K_p \times (V_{ref\_MCU} - V_o \times K_{fb})$$

where: K_p – is the gain of the regulator, V_{o_MCU} – feedback voltage measured by ADC10 (since in these course only the single channel/ single conversion mode has been introduced only one, more significant, voltage can be measured – it is the output voltage V_o), V_{ref_MCU} – reference voltage set once during programming, K_{fb} – gain of the feedback (the output voltage 75V produces the maximal ADC code 1023; this gives the value 13.7 [1/mV] or 73mV/1unit).

The value of gain can be estimated based on the approximate duty cycle at the definite reference. If the desired output voltage is 30V it gives internal reference 30V/0.073V=410. Let's assume that the acceptable error is 20% or 6V or 6V/0.073V≈80. At the same time 30V are obtained at the duty cycle of 50%. Therefore the initial value of the gain could be about 1.

Structure, corresponding to this equation is given in Figure 20-b. The basic modifications of the interrupt subprogram are following:

```
dV = SetPoint - ADC10MEM;                    //Find error
Cmd = dV * 1;                                //multiply the error
DutyCycle = Cmd;                             //Setup duty cycle
```

ADC10 now has to measure the feedback voltage (from channel 5). For this reason it also has to be reprogrammed:

```
ADC10CTL0 &= ~ENC;                           //Disable ADC10
ADC10CTL0 = SREF_1 + REFON + ADC10ON;        //Vref=15V, ADC10 - ON
ADC10CTL1 = INCH_5;                          //Measure channel A5
ADC10AE0 |= BIT5;                            //Enable analog for A5
```

It is also reasonable to restrict actual values of the duty cycle by 80%. The above modifications are summarized in Program 8.

Summary on Simplified Control Loops

As it can be concluded form the above listed materials realization of a simple control loop with MSP430 is not an extremely complicated, time consuming and expensive task. It can be realized as set of simple mathematical operations (subtraction, multiplication by an integer constant or by a constant that is less than 1) combined with programming of ADC10 and TA. Interrupt oriented programming provides good basis for operations with regular time step (that is important in the case of integral operations). TA interrupts are especially helpful in such case due to their close relation to generation of PWM signal.

Difficulties of realization of the simple control loops with MSP430 are mostly related to limited calculation capacity of these microcontrollers and non-ideality of compiler that often (even optimized) produce excessively big and slow executable code.

One more problem when programming PI regulator in C is related to format transformations. ADC10, TA and CPU deal with 16-bit integers while the integral sum must be typically stored in 32-bit long variables.

Appendix A: Basic theory of boost converter

Boost converter (Figure 21) is a DC to DC step-up converter. It is a switch mode converter with output voltage greater than the source (*E*) voltage which means that source current is greater than output current. At minimum the converter power part consists of inductor (*L*), transistor switch (*S*) and a diode (*D*). Usually it is equipped with output filter capacitor (*C*).

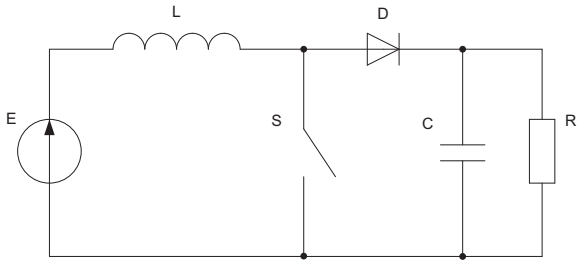


Figure 21. Basic schematic of boost converter

Basic boost converter operation can be divided in two distinct stages. During the first stage, switch *S* is turned on. Current flows from the source *E* through inductor *L* and switch *S*. Inductor current increases thus it accumulates energy. Diode *D* is not conducting. Capacitor *C* is discharging through load resistor *R*. During the second stage, switch *S* is turned off. Current flows from source *E* through inductor *L* and diode *D* to the output capacitor *C* and load resistor *R*. Capacitor voltage increases. Inductor current decreases and it acts as a voltage source. Inductor voltage adds up to the input source *E* voltage, thus producing increased output voltage.

Boost converter can operate in two modes: continuous current mode and discontinuous current mode. The difference is whether the momentary current value in the inductor *L* is always higher than 0, or for some time of the period it is 0.

Continuous current mode (CCM)

Figure 22 shows typical converter current and voltage waveforms in CCM. During the switch on state, full input voltage V_i is applied to the inductor. Inductor current changes according to equation:

$$\frac{\Delta I_L}{\Delta t} = \frac{V_i}{L}$$

where I_L is inductor current, V_i is source voltage, L is inductor inductance. At the end of switch on state, current I_L has increased to:

$$\Delta I_{Lon} = \frac{1}{L}\int_0^{DT} V_i dt = \frac{DT}{L} V_i$$

where D is switch duty cycle and T is switching period. During the switch off state, inductor current flows through the diode to the load. Inductor voltage is equal to input voltage V_i minus output voltage V_o. Inductor current can be determined according to:

$$V_i - V_o = L\frac{dI_L}{dt}$$

I_L variation during off state can be calculated according to:

$$\Delta I_{Loff} = \int_{DT}^{T} \frac{(V_i - V_o)dt}{L} = \frac{(V_i - V_o)(1-D)T}{L}$$

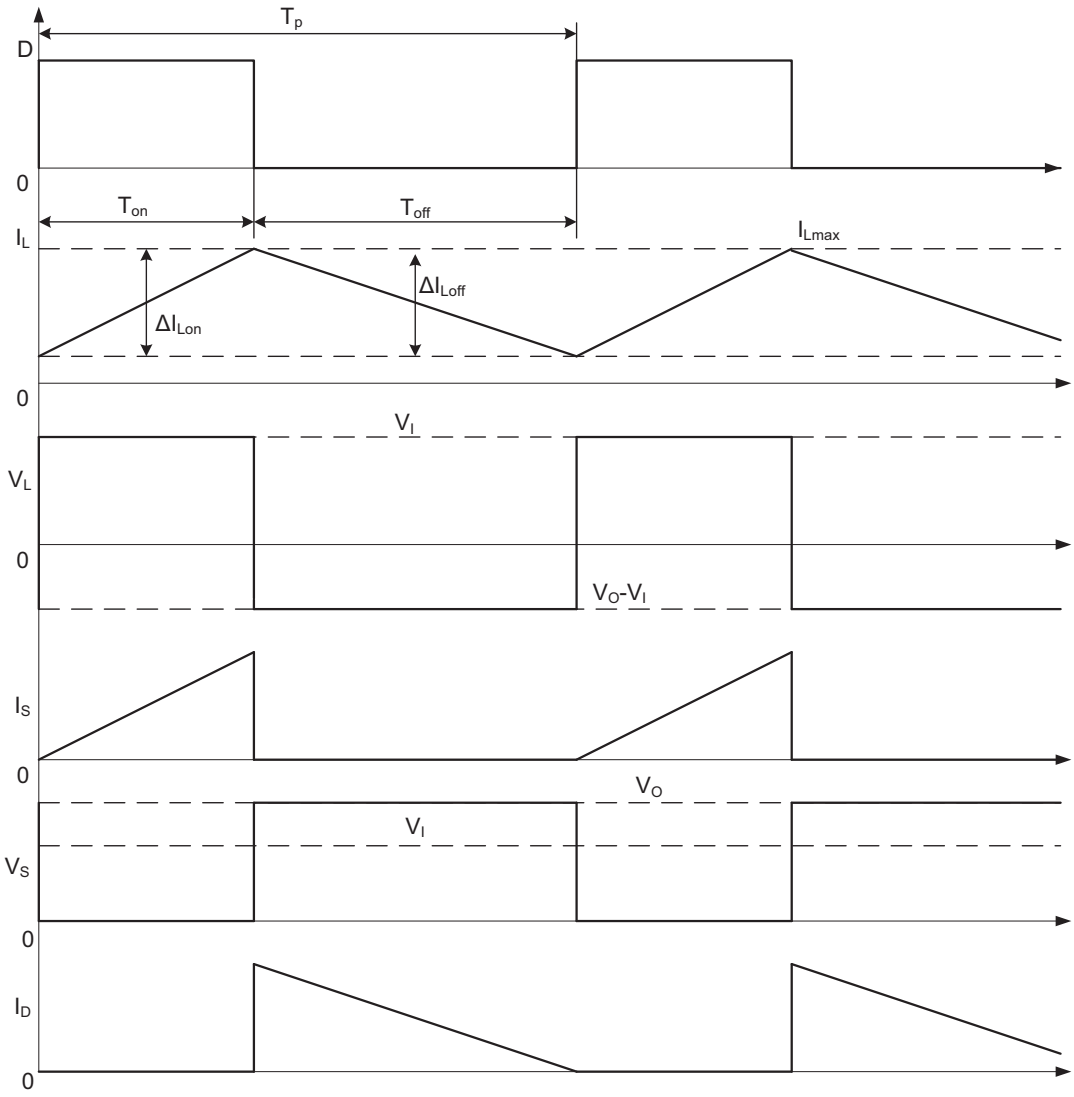


Figure 22. Continuous current mode waveforms

It is considered that the converter operates at steady state. Energy stored in each of its components has to be the same at the beginning and at the end of a commutation cycle. In particular, the energy *E* stored in the inductor is given by:

$$E = \frac{1}{2} L I_L^2$$

Inductor current has to be the same at the start and end of the commutation cycle. This means the overall change in the current is zero:

$$\Delta I_{Lon} + \Delta I_{Loff} = 0$$

Substituting ΔI_{Lon} and ΔI_{Loff} yields:

$$\Delta I_{Lon} + \Delta I_{Loff} = \frac{V_i DT}{L} + \frac{(V_i - V_o)(1-D)}{L} = 0$$

Rewriting previous equation gives:

$$\frac{V_o}{V_i} = \frac{1}{1-D}$$

This gives duty cycle to be:

$$D = 1 - \frac{V_i}{V_o}$$

It can be concluded that as the duty cycle increases from 0 to 1, output voltage increases from input voltage value to infinity. In practice, voltage cannot be boosted up more than a few times. To achieve CCM inductor value has to be larger than boundary inductance L_b:

$$L_b = \frac{(1-D)^2 DR}{2f}$$

where R is load resistance and f is switching frequency. For this type of converter the largest inductor current ripple is at 50% duty cycle, thus to calculate L_b, D = 0.5.

Discontinuous current mode (DCM)

Figure 23 shows typical converter current and voltage waveforms in DCM. In this mode during off state all inductor energy is transferred to the load, thus inductor current for a part of period is 0. This difference has strong effect on the output voltage equation which can be calculated based on the below described procedure.

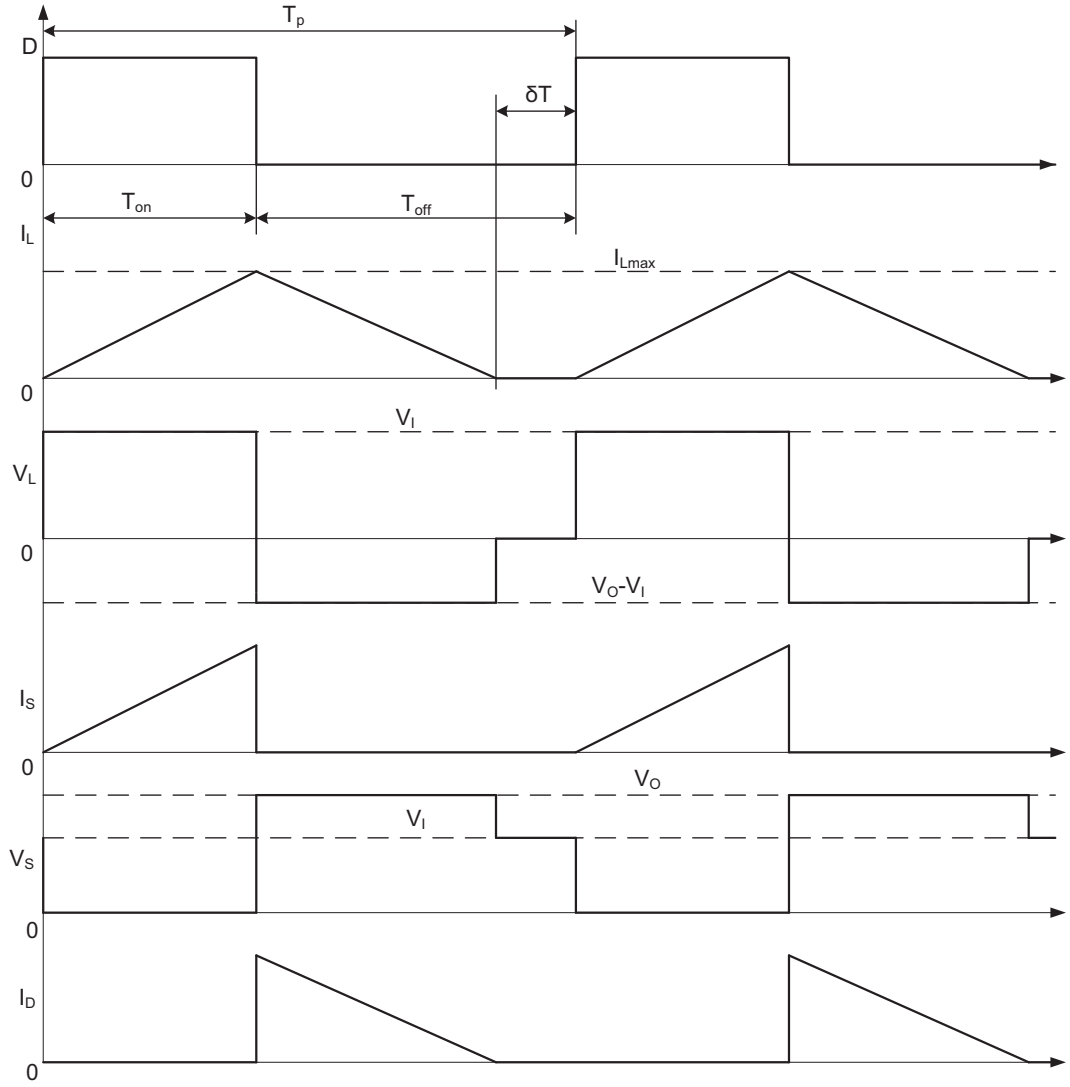


Figure 23. Discontinuous current mode waveforms

At beginning of each cycle inductor current is 0. Maximum current value I_{Lmax} is:

$$i_{L\max} = \frac{V_i DT}{L}$$

During the off period, I_L falls to 0 after time δT:

$$I_{L\max} + \frac{(V_i - V_o)\delta T}{L} = 0$$

From previous equations, δ value can be determined:

$$\delta = \frac{V_i F}{V_o - V_i}$$

The load current I_o is equal to the average diode current I_D. Diode current is equal to the inductor current during the off state. Therefore the output current can be written as:

$$I_o = \overline{I_D} = \frac{I_{L\max}}{2}\delta$$

Replacing I_{Lmax} and δ by their respective expressions yields:

$$I_o = \frac{V_i DT}{2L} * \frac{V_i D}{V_o - V_i} = \frac{V_i^2 D^2 T}{2L(V_o - V_i)}$$

Therefore, the output voltage gain can be written as follows:

$$\frac{V_o}{V_i} = 1 + \frac{V_i D^2 T}{2LI_o}$$

As can be seen, the output voltage depends on the duty cycle, inductor value, input voltage, switching frequency and output current.

Appendix B: Ziegler-Nichols tuning rules for PID controllers

Quite often PID controllers are adjusted on site, many different types of tuning rules have been proposed in literature. Using these rules, delicate and fine tuning of PID controllers can be made. PID controllers are used for different plant control. If a mathematical model of the plant can be derived, then it is possible to apply various design techniques for determining parameters of the controller that will meet the transient and steady state specifications of closed loop system. If plant model cannot be easily obtained, then it is common to use experimental approaches to the tuning of PID controllers.

The process of selecting the controller parameters to meet given performance specifications is known as controller tuning. Ziegler and Nichols suggested rules for tuning PID controllers (finding values K_p, T_i, T_d) based on experimental step responses or based on the value of K_p that results in marginal stability when only proportional control action is used. Ziegler-Nichols rules are useful when mathematical models of plants are not known, but can be used if models are known as well. Such rules suggest a set of values of K_p, T_i, and T_d, that will give a stable operation of the system. However, the resulting system may exhibit a large maximum overshoot in the step response, which is unacceptable. In such a situation, controller has to be fine tuned to achieve acceptable result. In fact, the Ziegler-Nichols tuning rules give an educated guess for the parameter values and provide a starting point for fine tuning, rather than giving the final settings for K_p, T_i, and T_d using just this method. There are two methods called Ziegler-Nichols tuning rules: the first method and the second method.

FIRST METHOD. When using this method, at first it is needed to obtain experimentally the response of the plant to a unit-step, similar as shown in Figure 24. This method can only be used if response to the step input exhibits an S shaped curve. This curve may be characterized by two constants, delay time L and time constant T. The delay time and time constant are determined by drawing a tangent line at the inflection point of the S shaped curve and determining the intersections of the tangent line with the time axis and line c(t)=K as can be seen in Figure 24. Ziegler and Nichols suggested setting the values of K_p, T_i, and T_d according to Table 5.

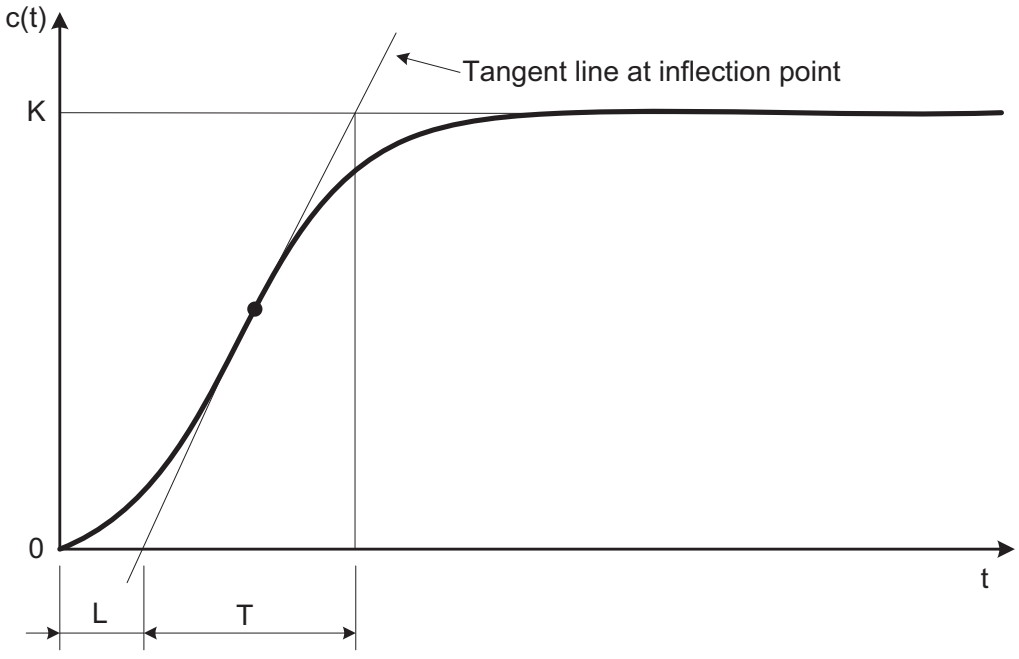


Figure 24. S shaped response curve

Table 5. Ziegler-Nichols tuning rule based on step response of plant

Type of controller	K_p	T_i	T_d
P	T/L	∞	0
PI	0.9 * T/L	L/0.3	0
PID	1.2 * T/L	2L	0.5L

SECOND METHOD. To use second method, values have to be set to $T_i=\infty$ and $T_d=0$. Using proportional control action only, increase K_p from 0 to critical value K_{cr} at which the output first exhibits sustained oscillations. If the output does not exhibit sustained oscillations for any value of K_p, then this method does not apply. K_{cr} value is the critical gain and P_{cr} is corresponding period, which is determined experimentally as seen in Figure 25. Ziegler and Nichols suggested setting the values of K_p, T_i, and T_d according to Table 6.

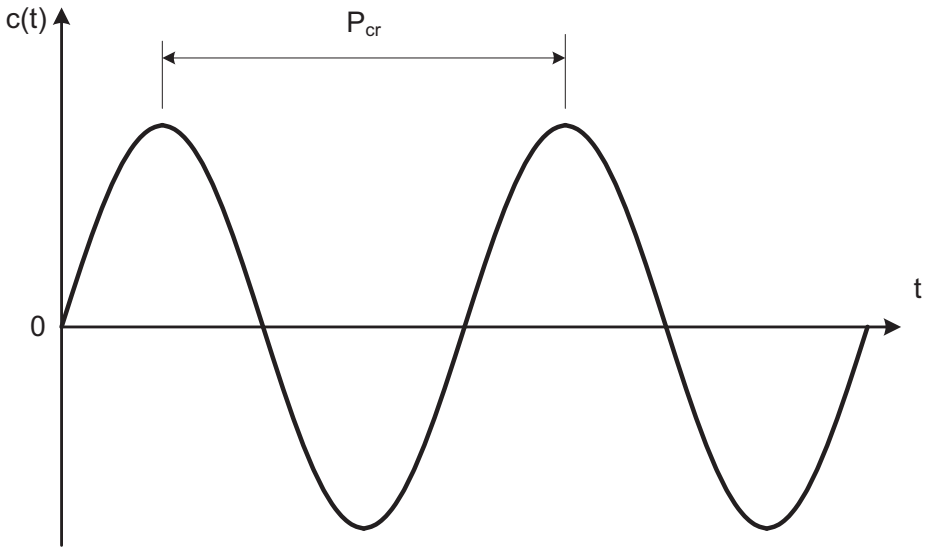
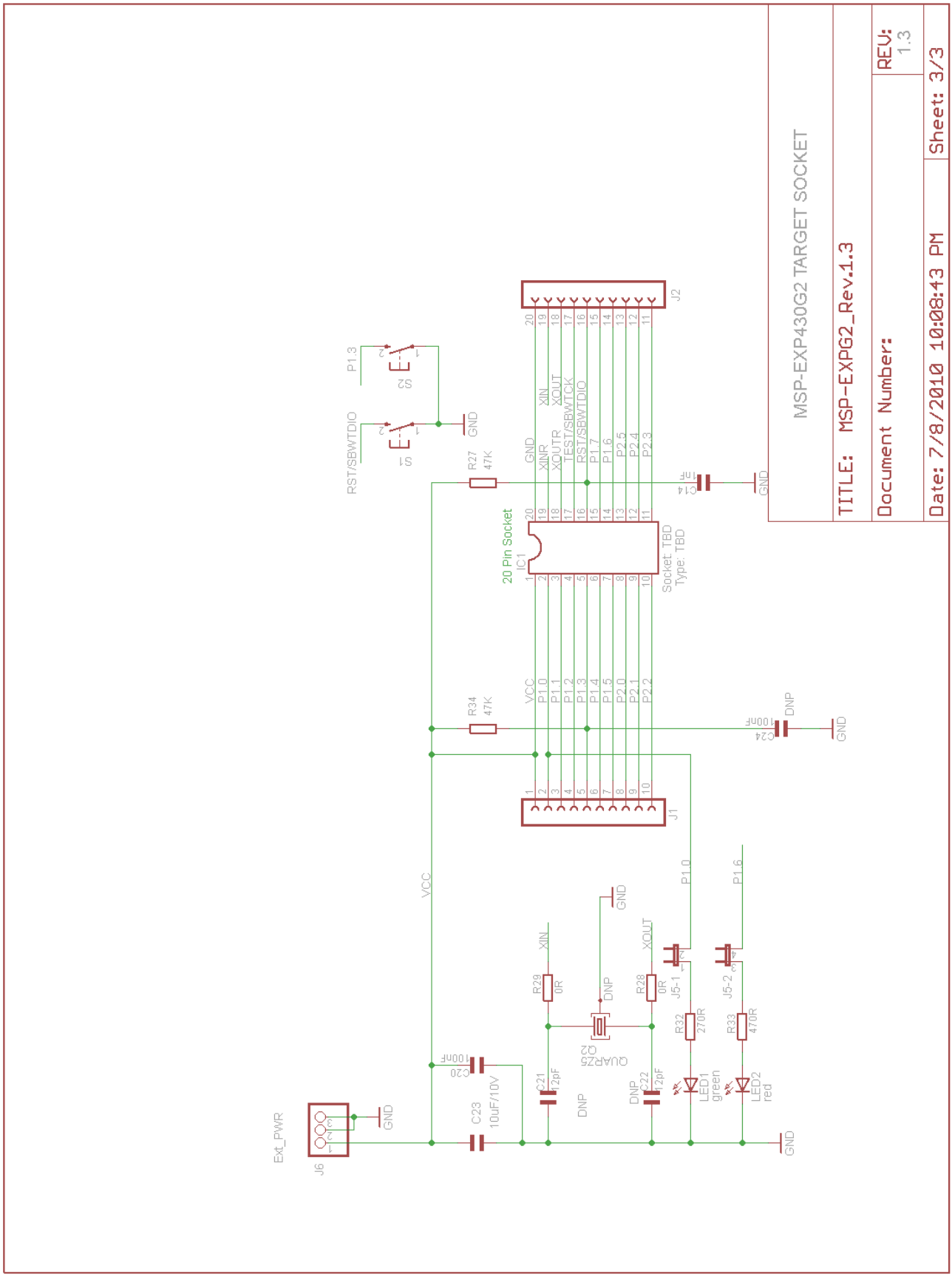


Figure 25. Sustained oscillation with period P_{cr}

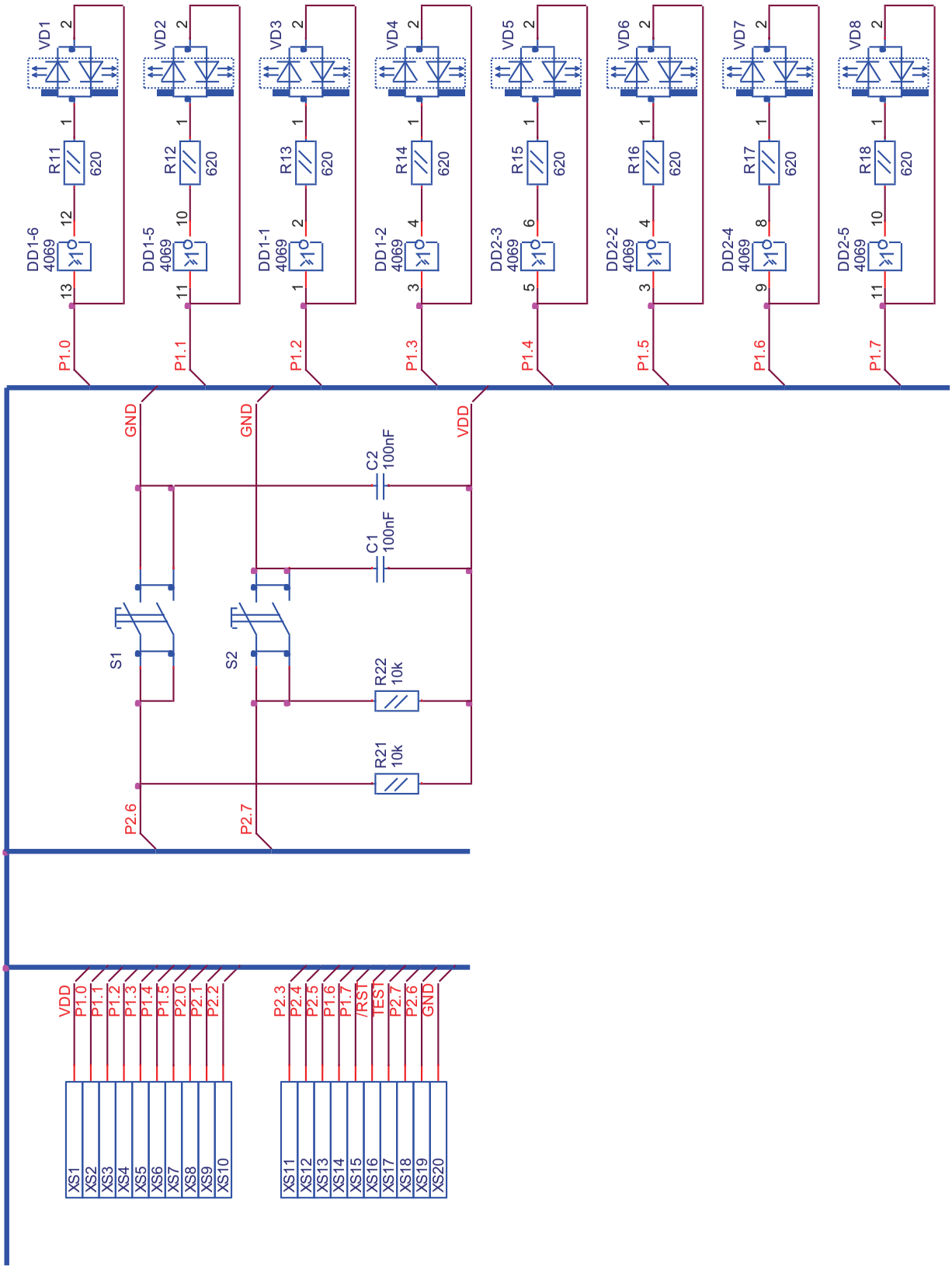
Table 6. Alternative tuning rule based on the sustained oscillations

Type of controller	K_p	T_i	T_d
P	0.5 * K_{cr}	∞	0
PI	0.45 * K_{cr}	1/1.2 * P_{cr}	0
PID	0.6 * K_{cr}	0.5 * P_{cr}	0.125 * P_{cr}

Appendix C: Schematic of LaunchPad (Target Board)



Appendix D: Schematic of LaunchPar 8LED+2PB Expansion Board



Appendix E: Schematic of LaunchPad Boost Converter Expansion Board