آموزش کاربردی

AVR

با مثال های
جامع و متنوع

مؤلفین:

- کوروش خلج منفرد
- پیمان حق گویی
- ناصر حکیمی

آموزش کاربردی AVR با مثال های جامع و متنوع

کوروش خلج منفرد ▫ پیمان حق گویی ▫ ناصر حکیمی

Practical Learning

AVR

- خودآموزی کامل برای علاقه مندان به میکروکنترلر و علم رباتیک.
- آموزش کاربردی قسمت های مختلف میکروکنترلر با پوشش کامل مطالب در قالب مثال هایی ساده و روان.
- کار با کدویژن و پروتنوس بصورت تصویری و مرحله به مرحله.
- همراه با سی دی : شامل نرم افزار های کدویژن و پروتنوس به همراه آموزش نصب، فایل کامل پروژه های کتاب، معرفی قطعات پرکاربرد به همراه دیتاشیت آن ها .
- روند کلی هر فصل : شروع با چند مثال ساده، کامل تر شدن مبحث در ادامه و اتمام بحث به صورت "یک گام فراتر".

9786001026256


بسم الله الرحمن الرحيم

آموزش کاربردی میکروکنترلر AVR

مولفین:
کوروش خلج منفرد
پیمان حقگویی
ناصر حکیمی

سازمان انتشارات جهاد دانشگاهی
۱۳۹۳

سرشناسه	: خلج‌منفرد، کوروش، ۱۳۷۱-
عنوان و نام پدیدآور	: آموزش کاربردی میکروکنترلر AVR/ مولفین کورش خلج منفرد، پیمان حقگویی، ناصر حکیمی
مشخصات نشر	: تهران: سازمان انتشارات جهاد دانشگاهی، ۱۳۹۳.
مشخصات ظاهری	: ۴۴۰ص.
شابک	: ۹۷۸-۶۰۰-۱۰۲-۶۲۵-۶
وضعیت فهرست نویسی	: فیپا
موضوع	: میکروکنترلر ا.وی آر.اتمل
موضوع	: کنترل کننده‌های برنامه‌پذیر
موضوع	: سیستم‌های کنترل رقمی
شناسه افزوده	: حقگویی، پیمان، ۱۳۷۱-
شناسه افزوده	: حکیمی، ناصر، ۱۳۷۲-
شناسه افزوده	: سازمان انتشارات جهاد دانشگاهی
رده‌بندی کنگره	: ۱۳۹۳ ۸خ۹ک/TJ۲۲۳
رده‌بندی دیویی	: ۶۲۹/۸۹۵
شماره کتابشناسی ملی	: ۳۵۶۷۱۴۱

آموزش کاربردی میکروکنترلر AVR

مولفین: کوروش خلج منفرد- پیمان حقگویی- ناصر حکیمی

ناشر: سازمان انتشارات جهاد دانشگاهی

چاپ: اول - ۱۳۹۳

شمارگان: ۱۰۰۰ نسخه

قیمت: ۱۸۰۰۰ تومان

شابک: ۶-۶۲۵-۱۰۲-۶۰۰-۹۷۸

این اثر، مشمول قانون حمایت مؤلفان و مصنفان و هنرمندان مصوب ۱۳۴۸ است، هر کسی تمام یا قسمتی از این اثر را بدون اجازه مؤلف (ناشر) نشر یا پخش یا عرضه نماید مورد پیگرد قانونی قرار خواهد گرفت.

سازمان انتشارات

نشانی سازمان انتشارات: تهران- خیابان انقلاب اسلامی- خیابان فخررازی- خیابان شهدای ژاندارمری- پلاک ۷۲ ، تلفن: ۶۶۹۵۲۹۴۸

مرکز پخش:

خیابان انقلاب - بین فلسطین و چهارراه ولیعصر - جنب مؤسسه نمایشگاه‌های فرهنگی ایران- تلفن: ۶- ۶۶۴۸۷۶۲۵

پایگاه اطلاع‌رسانی: ***www.isba.ir*** **پست الکترونیکی:** ***info@isba.ir*** **فروشگاه اینترنتی:** ***www.Jdbookfair.com***

پیشگفتار

دکتر محمد عشقی (عضو هیئت علمی دانشگاه شهیدبهشتی)

اختراع میکروپروسسورها، آغاز عصرانقلاب دیجیتال بشر است.

پیدایش فیزیک نیمه‌هادی‌ها- مباحث الکترونیک- اختراع خازن و مقاومت و سلف، اختراع دیود و ترانزیستور و دیگر مدارات الکترونیکی البته باعث بوجود آمدن و خلق بسیاری وسائل برقی مانند رادیو – تلویزیون- رادار- موتور برقی و . . . شد؛ ولی انقلاب دیجیتال و عصر دیجیتال آغاز نشد مگر پس از اختراع میکروها: میکروپروسسورها و میکروکنترلرها. سخت‌افزارهایی که بتوانند یک برنامه نرم‌افزاری- دستورالعمل‌های از پیش تعریف شده و ذخیره شده در حافظه - را خط به خط بخواند و اجرا نمایند.

کتاب حاضر، "آموزش کاربردی میکروکنترلرهای AVR"، هم در توضیح بخش سخت‌افزاری و هم در بخش نرم‌افزاری میکروها بسیارکاربردی و عملی نوشته شده است. این کتاب اگر نگوییم که می‌تواند بصورت خودآموز استفاده شود حداقل می‌توان گفت که کتاب بسیار خودکفایی است و با کمترین آموزش می‌توان بطور کاربردی میکروکنترلر AVR و وسایل جانبی آن را، با دنبال کردن مثالها و مدارهایش، آموخت. کاربردی بودن، عملیاتی بودن، پرمثال وخودآموز بودن، در کنارسادگی و روانی نوشتار از مشخصه‌های برجسته این کتاب است.

نویسندگان جوان کتاب، آقایان کوروش خلج منفرد- پیمان حقگویی و ناصر حکیمی، از دانشجویان با استعداد و پرکار دانشکده‌ی ما هستند، که همزمان با گذراندن دروس خود در دانشکده، در فعالیتهای علمی و عملی در پروژه‌های خارج از دانشکده نیز شرکت داشته اند و از این رهگذر تجربیات عملی فراوان در استفاده از میکروکنترلر AVR کسب نموده‌اند سپس با زحمات شبانه‌روزی و صرف وقت بسیار، این کتاب را نیز حاضر و در اختیار جامعه دانشجویی و مهندسی ایران قرار داده‌اند. این سعی وکوشش و پژوهش و زایش سزد عبرت دیگر جوانان دانشجوی ما گردد که اکثراً وقت و عمری ۴ تا ۵ ساله را در دانشکده‌های مهندسی ایران می‌گذرانند و در نهایت به یک مدرک کارشناسی بسنده می‌کنند.

امید است که کاربران این میکرو و این کتاب بتوانند حداکثر استفاده را از آن ببرند و در صورت رضایت، باارسال یک پست الکترونیکی به نویسندگان کتاب از زحمات آنها قدردانی نمایند تا رسم نویسندگی، بخصوص در حوزه کتابهای مرتبط با الکترونیک و فناوری، در این مرز و بوم تشویق و نهادینه شود.

آرزوی توفیق، پیشرفت، موفقیت این نویسندگان جوان و مهندسان آینده کشورم را دارم.

محمد عشقی

دانشیار دانشکده مهندسی برق و کامپیوتر

دانشگاه شهید بهشتی

تابستان ۱۳۹۳

مقدمه‌ی مولفین

به نام آن که جان را فکرت آموخت　　چراغ دل به نور جان برافروخت

ز فضلش هر دو عالم گشت روشن　　ز فیضش خاک آدم گشت گلشن

خدا را شاکریم که به ما توفیق داد تا کتاب حاضر را به سرانجام برسانیم. کتابی که پیش رو دارید شامل ۱۵ فصل به همراه ۱۰پروژه می‌باشد که تا حد امکان سعی بر آن بوده که مطالب با زبانی کاملاً ساده و بی‌تکلف و به دور از گزافه‌گویی بیان شود چراکه ساده‌گویی بس نکته‌ی مهم و قابل توجهی است. همانطور که در مورد موضوعات مختلف کتاب‌های بسیاری وجود دارد، کتاب‌های متعددی نیز در مورد میکروکنترلرهای AVR نوشته شده است که هر کدام با توجه به نحوه‌ی نگارش، گفتمان و نوع مباحث، مخاطبین خاص خود را یافته است. در کتاب پیش‌رو سعی و هدف، آن بوده که کلیه‌ی مباحث به صورت کاملاً کاربردی و در قالب مثال قدم به قدم بیان شود. به‌طوری‌که مباحث هر فصل به صورت کامل و جامع مطرح شوند و هیچ قسمتی بدون توضیح نماند که این موضوع در فهرست کتاب نیز قابل مشاهده است. امتیاز دیگر این کتاب آن است که حتی کسانی که هیچ‌گونه آشنایی با میکروکنترلر هم ندارند، بتوانند به راحتی و به صورت کاربردی کار با آن را بیاموزند.

امیدواریم این کتاب، بتواند کمکی اگرچه بسیار کوچک به علاقه‌مندان میکروکنترلر داشته باشد.

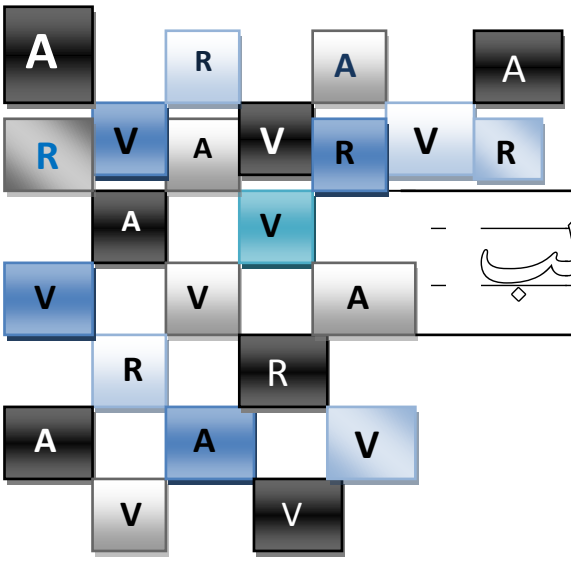
در پایان به رسم قدردانی، تشکر می‌کنیم از جناب دکتر محمد عشقی بابت پیش‌گفتار ایشان که مایه‌ی فخر و مباهات ماست. همچنین از ریاست محترم استعدادهای درخشان دانشگاه شهیدبهشتی جناب دکتر صدوق بابت همکاری ایشان و از دوست عزیزمان، جناب آقای علی ساریخانی که طرح جلد را پیش‌کش نگاهتان نمودند کمال تشکر را داریم. همچنین تشکر می‌کنیم از مدیریت محترم سازمان انتشارات جهاد دانشگاهی و کارکنان زحمتکش این سازمان که از هیچگونه تلاشی در جهت بهبود کیفیت کتاب دریغ ننمودند و لازم می‌داریم که از تمامی

دانشجویان و اساتید دانشکده مهندسی برق و کامپیوتر دانشگاه شهید بهشتی به ویژه مهندس احمدی و مهندس علیپور قدردانی کنیم.

با توجه به اینکه هیچ تالیفی خالی از اشکال نیست لذا از همه اساتید، دانشجویان و صاحب‌نظران استدعا داریم عنایت فرمایند و اشکالات این کتاب را از طریق ایمیل به آدرس زیر ارسال فرمایند.

pnk.avr@gmail.com

سربلند باشید
کوروش خلج منفرد
پیمان حقگویی
ناصر حکیمی
تابستان ۱۳۹۳



فهرست مطالب

فصل سوم: ورودی و خروجی در AVR(PIN & PORT)

فصل چهارم: LCD های کاراکتری

فصل پنجم: وقفه‌های خارجی(External Interrupts)

فصل ششم: واحد مبدل آنالوگ به دیجیتال (ADC)

فصل هفتم: تایمر/کانتر

فصل هشتم: مقایسه‌کننده‌ی آنالوگ

فصل نهم: ارتباط سریال

فصل دهم: ارتباط USART

فصل یازدهم: ارتباط SPI

فصل دوازدهم: ارتباط I2C

فصل سیزدهم: کنترل توان

فصل چهاردهم: کنترل موتور DC

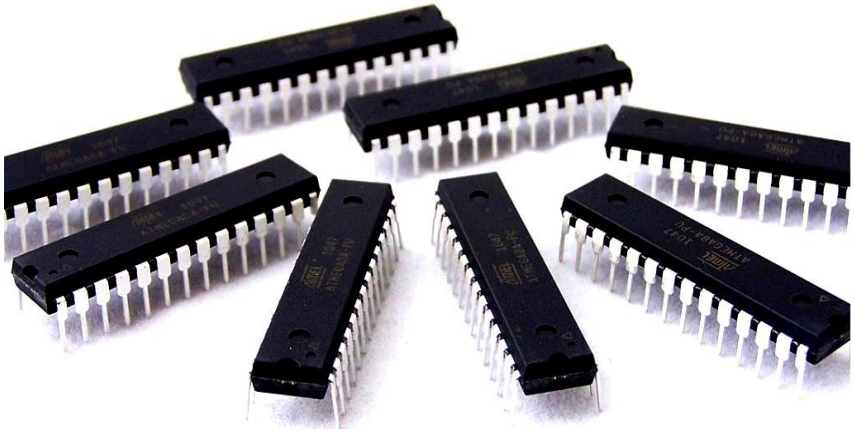
فصل پانزدهم: فیوزبیت‌ها

فصل شانزدهم: پروژه‌ها

پیوست

فصل اول

آشنایی با میکروکنترلرها



در این فصل خواهیم خواند:

۱. میکروکنترلر چیست؟
۲. تفاوت میکروکنترلر با میکروپروسسور چیست؟
۳. تفاوت پروسسور بامیکروپروسسور چیست؟
۴. با میکروکنترلر چه کارهایی می‌توان انجام داد؟
۵. بخش‌های مختلف یک میکروکنترلر
۶. تاریخچه‌ای کوتاه از انواع میکروکنترلرها
۷. ساختار داخلی میکروکنترلرهای AVR
۸. میکروکنترلرهای ۸ بیتی AVR
۹. انواع میکروکنترلرهای AVR
۱۰. حافظه در AVR
۱۱. میکروکنترلر ATmega16

آشنایی با میکروکنترلرها

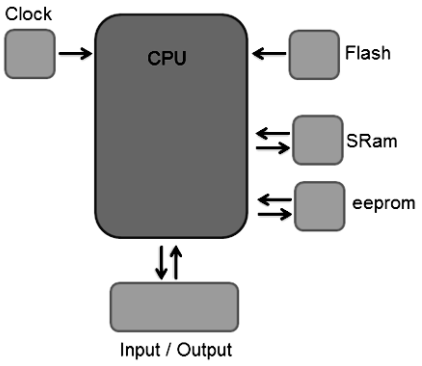
در این فصل قصد داریم با انواع میکروکنترلرها و برخی مفاهیم اساسی و کاربردی آن و همچنین با خصوصیات میکروکنترلرهای AVR آشنا شویم.

میکروکنترلر چیست؟

میکروکنترلر یک قطعه‌ی الکترونیکی است که قابل برنامه‌ریزی می‌باشد. میکروکنترلر از پردازنده، حافظه و تعداد زیادی عناصر دیجیتالی ساخته شده است که کاربر بنا به نیاز می‌تواند از طریق برنامه‌ریزی از هر کدام از آن‌ها استفاده کند. در حالت کلی، این تراشه‌ها دارای پردازشگر مرکزی(CPU)، حافظه‌ی داخلی و امکانات ارتباطی برای کنترل وسایل خارجی هستند. میکروکنترلرها امروزه تقریباً در تمامی وسایل الکترونیکی و دیجیتالی یافت می‌شوند و همچنین به وفور در دسترس می‌باشند و کار با آنها بسیار ساده می‌باشد.

تفاوت میکروکنترلر با میکروپروسسور چیست؟

یک اشتباه متداول که وجود دارد این است که گاهی افراد میکروکنترلر را با میکروپروسسور اشتباه می‌گیرند و هر دو را به یک مفهوم می‌شناسند،درحالی‌که میکروپروسسور ریزپردازنده‌ای است که فقط قادر به پردازش اطلاعات و انجام محاسبات می‌باشد و برای استفاده از آن باید در کنار آن از حافظه، کلاک، ورودی و خروجی و... استفاده کرد. ریزپردازنده در کنار دیگر قطعات الکترونیکی معنا پیدا می‌کند برای مثال اگر مغز انسان را پردازنده فرض کنیم، مغز به تنهایی قادر به انجام هیچ کاری نمی‌باشد و باید در کنار آن حواس ۵گانه‌ی انسان نیز باشد که مغز بتواند از آن‌ها ورودی بگیرد و نیز باید دست‌وپایی وجود داشته باشد که بتواند به آنها دستور بدهد پس مغز (یا پردازنده) به تنهایی قابل استفاده نیست.



میکروکنترلر از قسمت‌های مختلفی تشکیل شده است که ریزپردازنده(یا میکروپروسسور) یکی از اجزای آن می‌باشد. در حقیقت میکروکنترلر قطعه‌ای می‌باشد که به تنهایی قابل استفاده است زیرا علاوه بر پردازنده‌ی مرکزی دارای حافظه و ورودی/ خروجی نیز می‌باشد. شکل روبرو اجزای تشکیل‌دهنده‌ی یک میکروکنترلر را نشان می‌دهد.

همانطور که می‌دانید CPU همان واحد پردازنده‌ی مرکزی می‌باشد وFlash, Sram eeprom سه واحد حافظه می‌باشند و کلاک (Clock) نیز

مشخص‌کننده‌ی سرعت انجام دستورالعمل‌ها می‌باشد (که در ادامه در مورد آنها بیشتر توضیح خواهیم داد). البته برخی از میکروکنترلرها دارای حافظه‌ی eeprom یا Sram نمی‌باشند.

تفاوت پروسسور با میکروپروسسور چیست؟

تفاوت پروسسور (یا پردازنده) و میکروپروسسور (یا ریزپردازنده) در ابعاد آن‌ها می‌باشد که ابعاد میکروپروسسور بسیار کوچک‌تر از پروسسور است.

حال سوالی که مطرح می‌شود این است که آیا تنها دلیل استفاده از میکروپروسسور به جای پروسسور بخاطر کوچک بودن آن است؟ به عبارتی آیا تنها دلیل کوچک کردن ابعاد پردازنده به خاطر این است که جای کمتری را اشغال کند؟

پاسخ منفی است، یکی از اصلی‌ترین دلایلی که ابعاد یک پردازنده را کوچک می‌کند آن است که پردازنده به یک قطعه‌ی فشرده تبدیل شود.

عنصرفشرده: عنصری که ابعاد آن در مقایسه با طول موج سیگنال که آن عنصر را تحریک می‌کند بسیار کوچک باشد، یعنی ابعاد آن قابل صرف‌نظر باشد. یک مدار که از عناصر فشرده تشکیل شده‌است را مدار فشرده می‌نامیم. قوانین KVL و KCL فقط برای مدارات فشرده برقرار می‌باشند و اگر مداری فشرده نباشد باید برای حل آن از معادلات ماکسول استفاده کرد، پس هرچه فرکانس کلاک بالاتر باشد طول موج کوچک‌تر می‌شود و اگر ابعاد قطعات را کوچک نکنیم مدار از فشردگی در می‌آید، پس برای استفاده از فرکانس‌های بالا یا به عبارتی برای بالاتر بردن سرعت محاسبات بایستی ابعاد قطعات را کوچک کنیم.

با میکروکنترلر چه کارهایی می‌توان انجام داد؟

میکروکنترلرها شرایط یک کامپیوتر در ابعاد کوچک و با قدرتی کمتر را دارا می‌باشند. بیشتر این میکروکنترلرها برای کنترل وسایل مختلف و انجام برخی محاسبات استفاده می‌شوند. میکروکنترلرها دارای کاربردهای بسیار زیادی هستند که از جمله می‌توان به کنترل ربات‌ها تا استفاده در کارخانه‌ها و کارهای صنعتی و... اشاره کرد.

بخش‌های مختلف یک میکروکنترلر

CPU(Central Process Unit): واحد پردازش مرکزی

ALU(Arithmetic Logic Unit): واحد محاسبه و منطق

I/O(Input / Output): ورودی‌ها و خروجی‌ها

RAM(Random Access Memory): حافظه‌ی اصلی میکروکنترلر

ROM(Read Only Memory): حافظه‌ای که برنامه روی آن ذخیره می‌گردد.

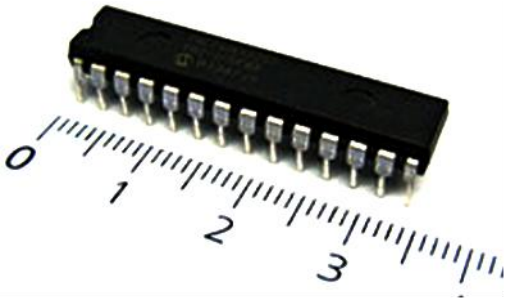
TIMER: واحد کنترل و محاسبه‌ی زمان

تاریخچه‌ای کوتاه از انواع میکروکنترلرها

اولین میکروکنترلر با نام ۸۰۵۱ در سال ۱۹۶۹ توسط شرکت INTEL وارد بازار شد. شرکت INTEL بعد از مدتی اجازه‌ی ساخت این میکروکنترلرها را به شرکت‌های بزرگی چون DALLAS، SIEMENS، PHILIPS،ATMEL و... داد و این شرکت‌ها نیز ساخت این میکرو کنترلرها را ادامه دادند. در شکل زیر نمونه‌ای از این میکروکنترلرها را مشاهده می‌کنید:

شکل ۱-۱ : میکروکنترلر ۸۰۵۱

بعد از مدتی شرکت MicroChip، میکروکنترلرهای پیشرفته‌تری با نام pic طراحی کرد. میکروکنترلرهای pic دارای مزیت‌های به مراتب بیشتری نسبت به ۸۰۵۱ هستند، از جمله:
EEPROM داخلی (Electrically Erasable Programmable Read Only Memory) که حافظه‌ای است که می‌توان توسط برنامه بر روی آن نوشت و یا نوشته‌های قبلی را از روی آن پاک کرد. A/D داخلی که تبدیل‌کننده‌ی ولتاژ آنالوگ به ولتاژ دیجیتال و قابل فهم برای میکروکنترلر می‌باشد. PWM (Pulse Width Modulation) خروجی برای تولید موج مربعی بر روی پایه‌های خروجی و نیز نویزپذیری بسیار کم در محیط‌های صنعتی. این امکانات میکروکنترلرهای pic باعث رشد و پیشرفت آنها در محیط‌های صنعتی شد به طوری که تا امروز نیز یکی از پرکاربردترین انواع میکروکنترلرها در محیط‌های صنعتی به شمار می‌رود. شکلی از بسته‌بندی DIP این میکروکنترلر را در شکل زیر مشاهده می‌کنید:



شکل ۱-۲ : میکروکنترلر PIC

شرکت ATMEL (Atmel)در سال ۲۰۰۱ میکروکنترلر جدیدی به نام AVR را به بازار معرفی کرد که دارای امکانات بیشتری نسبت به میکروکنترلرPIC بود. البته امروزه میکروکنترلرهای pic و AVR هر دو بسیار پیشرفت کرده‌اند و نمی‌توان به طور کلی قضاوت کرد که کدام میکروکنترلر بهتر است اما به صورت کلی میکروکنترلرهای pic به دلیل مدارهای محافظتی بهتر و نویزپذیری کمتر در محیط‌های صنعتی کاربرد بیشتری دارند و میکروکنترلرهای AVR بیشتر در کاربردهای دیگر از جمله ربات‌ها و مدارهای کنترلی و... استفاده‌ی بیشتری دارند ولی به طور کلی این دو خانواده از میکروکنترلرها همیشه در حال رقابت با همدیگر بوده‌اند و کمبود امکانات نسبت به هم را جبران می‌کنند. در شکل زیر یک ATmega16 که یکی از میکروکنترلرهای خانواده‌ی AVR می‌باشد را مشاهده می‌کنید:

شکل ۱-۳ : میکروکنترلر ATmega16

ساختار داخلی میکروکنترلرهای AVR

منظور از ساختار داخلی یا به عبارتی معماری داخلی نوع طراحی مدارهای داخلی و دستوراتی است که میکروکنترلر می‌تواند اجرا کند. معماری داخلی میکروکنترلرهای AVR مبتنی بر معماری RISC (Reduced instruction Set Computer) است. این نوع معماری دارای مدارها و محاسبه‌گرهایی است که توانایی اجرای عملیات‌های ساده ولی با سرعت بیشتری را دارا می‌باشد. در این معماری تعداد دستورات قابل اجرا کم و طول همه‌ی دستورات مساوی است که این باعث افزایش بسیار زیاد سرعت می‌شود، در واقع این وظیفه‌ی برنامه‌نویس است که دستوراتی که مستقیماً توسط CPU اجرا نمی‌شود را با کدنویسی بسازد با این شرایط در این نوع معماری کدنویسی کمی دشوارتر می‌شود ولی سرعت اجرا بالا می‌رود.

در مقابل معماری RISC، معماری CISK(Complex Instruction Set Computer) وجود دارد که در آن ساختار داخلی پیچیده‌تر است و CPU قادر به اجرای دستورات بیشتر و متنوع‌تری است ولی سرعت کمتری دارد.

از ویژگی‌های معماری RISC می‌توان به نکات زیر اشاره کرد:

۱)در این نوع معماری طول دستورات یکسان است لذا رمزگشایی یا decode کردن آنها بسیار ساده‌تر است.

۲)دستورات قابل اجرا کم است ولی دستورات با سرعت بسیار بالایی اجرا می‌شوند.
۳)اکثر دستورات در یک پالس اجرا می‌شوند که این ویژگی سرعت این معماری را بالا می‌برد.

میکروکنترلرهای ۸ بیتی AVR

این میکروکنترلرها ۸ بیتی هستند، یعنی از رجیسترهای ۸ بیتی ساخته شده‌اند. رجیستر نوعی حافظه است که به طور مستقیم با پردازنده در ارتباط است. رجیسترها دو نوع هستند، یک نوع رجیسترهای همه‌منظوره که قابل دسترس برنامه‌نویس هستند و یکی رجیسترهای خاص و تک‌منظوره که معمولاً در دسترس مستقیم برنامه‌نویس نیستند. برای مثال زمانی که از تایمر میکروکنترلر استفاده می‌کنیم در صورت رسیدن به زمان مورد نظر درون یکی از رجیسترها مقدار یک ریخته می‌شود و زمانی که CPU مقدار یک را درون این رجیستر ببیند دستور مورد نظری که ما برای رسیدن به آن زمان خاص نوشته‌ایم را اجرا می‌کند. سرعت رجیسترها به دلیل ارتباط مستقیم با CPU نسبت به دیگر حافظه‌ها بیشتر است. این میکروکنترلرها دارای ۳۲ رجیستر همه منظوره(به‌نام‌های R0 تا R31) می‌باشند.
میکروکنترلرهای AVR از نوع CMOS می‌باشند و دارای توان مصرفی بسیار کمی هستند (CMOS مخفف complementary metal oxide semiconductor می‌باشد).
میکروکنترلر AVR می‌تواند هر دستور را تنها در یک پالس ساعت (یک کلاک) انجام دهد یعنی اگر برای مثال در برنامه یک میلیون دستور وجود داشته باشد و فرکانس کاری میکروکنترلر بر روی یک مگاهرتز تنظیم شده باشد هر کلاک در یک میکروثانیه زده می‌شود پس هر دستور در یک میکروثانیه انجام می‌شود پس تمام یک میلیون دستور تنها در یک ثانیه انجام می‌شوند.

انواع میکروکنترلرهای AVR

میکروکنترلرهای AVR به سه دسته تقسیم بندی می‌شوند:

۱- Tiny AVR سری ATtiny:

اعضای این خانواده عبارتند از:

ATtiny10,ATtiny11,ATtiny12,ATtiny13,ATtiny15,ATtiny22,ATtiny26 و...که دارای ۱ تا ۸ کیلوبایت حافظه‌ی قابل برنامه‌نویسی می‌باشند.

انواع مختلف این خانواده دارای ۸ تا ۳۲ پایه هستند.

امکانات این خانواده از AVR نسبت به دو نوع دیگر کمی محدودتر است.

۲- AT90S (classic AVR): سری کلاسیک

اعضای این خانواده عبارتند از:

AT90S4433 ,AT90S2343 ,AT90S2323, AT90S2313 ,AT90S1200 و...

این خانواده دارای حافظه‌ی بیشتری نسبت به ATtiny است.

۳- AVR mega سری ATmega:

دارای ۴ تا ۲۵۶ کیلوبایت حافظه‌ی قابل برنامه‌نویسی می‌باشند.

امکانات بیشتری نسبت به دو مدل قبل دارند.

انواع مختلف این خانواده دارای ۲۸ تا ۱۰۰ پایه هستند.

اعضای این خانواده عبارتند از :

ATmega8،ATmega16،ATmega32،ATmega48،ATmega2560،ATmega329و... (در این کتاب با این خانواده از میکروکنترلرهای AVR کار خواهیم کرد).

۴- Xmega سری ATxmega:

دارای ۱۶ تا ۳۸۴ کیلوبایت حافظه‌ی قابل برنامه نویسی می‌باشند.

دارای انواع ۴۴ و ۶۴ و ۱۰۰ پایه هستند.

دارای ویژگی‌های گسترده ای مانند سیستم رویدادگرا

۵- AVRهای کاربرد خاص:

که دارای کاربردهای مخصوص می‌باشند که باقی خانواده‌ها ندارند، مانند کنترلر USB، کنترلر PWM،LCD پیشرفته و...

حافظه در AVR

میکروکنترلرهای AVR دارای دو نوع حافظه هستند، حافظه‌ی داده و حافظه‌ی برنامه و نیز دارای یک بخش جداگانه به نام حافظه‌ی EEPROM هستند که یک حافظه‌ی دائمی است و با خاموش و روشن شدن میکروکنترلر این حافظه از بین نمی‌رود و می‌توان توسط برنامه بر روی آن نوشت و از روی آن خواند و همچنین قابلیت ده هزار بار پر و خالی شدن را دارد.

۱.حافظه‌ی برنامه

همانطور که می‌دانید میکروکنترلرها آی‌سی‌های قابل برنامه‌ریزی هستند. برنامه‌ای که توسط برنامه‌نویس نوشته می‌شود باید درون میکروکنترلر ذخیره شود تا هنگام استفاده از آن برنامه‌ی مورد نظر را اجرا کند، حافظه‌ای که باید این برنامه درون آن ذخیره شود از نوع FLASH می‌باشد.

۲.حافظه‌ی داده

حافظه‌ی داده شامل رجیسترهای عمومی، رجیسترهای ورودی/خروجی، حافظه‌ی داخلی SRAM، حافظه‌ی داده‌ی خارجی و... می‌باشد.

رجیسترهای عمومی شامل ۳۲ عدد رجیستر می‌باشند و مستقیماً بـه واحـد محاسـبه‌گـر(ALU) متصل هستند. اگر در برنامه بخواهیم از حافظه‌ای استفاده کنیم ابتدا داده‌ی آن حافظه درون یکی از این ۳۲ رجیستر ریخته می‌شود سپس از طریق آن وارد محاسبه‌گر می‌شود.

۳.حافظه‌ی داده‌های داخلی (SRAM)

حافظه‌ی موقتی است که متغیرهایی که درون برنامه تعریف می‌کنید در آن حافظه ذخیره می‌شود. فرض کنید برنامه‌ای نوشته‌اید که میکروکنترلر ولتاژ یکی از پایه‌های ورودی را خوانده و متناسب با آن عملیاتی را انجام دهد وقتی میکروکنترلر ولتاژ پایه‌ی مورد نظر را خواند باید آن را در یک متغیر که در برنامه نوشته شده بریزد و متناسب با مقدار آن متغیر عملیاتی را انجام دهد. وقتی می‌گوییم که باید این عدد درون یک متغیر ذخیره شود پس به یک حافظه نیاز داریم که مقـدار متغیـرهـای برنامه درون آن ذخیره شوند، این حافظه همان حافظه‌ی SRAM می‌باشد. همانطور که اشاره شد زمانی که برنامه بخواهد از این متغیرها استفاده کند ابتدا مقدار این متغیرها از حافظـه‌ی SRAM به رجیسترهای عمومی منتقل می‌شود و سپس به CPU فرستاده می‌شود.

میکروکنترلر ATmega16

تا به اینجا تقریباً با انواع مدل‌های میکروکنترلر آشنا شدیم و خصوصیات آنها را تا حدودی شناختیم. در این کتاب قصد داریم که از میکروکنترلر AVR مدل ATmega16 استفاده کنیم و با خصوصیات و نحوه‌ی راه‌اندازی قسمت‌های مختلف آن آشنا شویم. در ادامه قسمتی از خصوصیات ATmega16 را مشاهده می‌کنید:

- دارای معماری داخلی RISC
- کارایی بالا و توان مصرفی کم
- دارای ۱۳۱ دستورالعمل با کارایی بالا که اکثراً تنها در یک کلاک سیکل اجرا می‌شوند.
- ۳۲ عدد رجیستر همه‌منظوره‌ی ۸ بیتی
- سرعت کلاک تا فرکانس ۱۶ مگاهرتز
- حافظه‌ی برنامه و داده‌ی غیرفرار
- ۱۶ کیلوبایت حافظه‌ی FLASH داخلی قابل برنامه‌ریزی
- حافظه‌ی FLASH پایدار با قابلیت ۱۰۰۰۰ بار نوشتن و پاک کردن
- ۱۰۲۴ بایت حافظه‌ی داخلی SRAM
- ۵۱۲ بایت حافظه‌ی EEPROM داخلی
- پایداری حافظه‌ی EEPROM با قابلیت ۱۰۰۰۰۰ بار نوشتن و پاک کردن.
- قفل برنامه‌ی EEPROM, FLASH

خصوصیات جانبی:

- دو تایمر/کانتر ۸ بیتی
- یک تایمر/ کانتر ۱۶ بیتی و دارای مدهایCompare و Capture.
- ۴ عدد کانال PWM.
- ۸ عددکانال مبدل آنالوگ به دیجیتال ۱۰ بیتی.
- یک مقایسه‌کننده‌ی آنالوگ داخلی.
- WATCHDOG قابل برنامه‌ریزی با اسیلاتور داخلی.
- قابلیت ارتباط با پروتکل سریال دوسیمه(TWO WIRE).
- قابلیت ارتباط سریال SPI (Serial Peripheral interface) به صورت Master یاSlave .

ارتباط USART سریال قابل برنامه‌ریزی.

خصوصیات ویژه‌ی میکروکنترلرATmega16:

-دارای اسیلاتور RC داخلی کالیبره شده.

-منابع وقفه‌ی (interrupt) داخلی و خارجی.

- توان مصرفی پایین و سرعت بالا توسط تکنولوژی COMS

این نوع میکروکنترلرها می‌توانند دارای پسوند نیز باشند برای مثال ATmega16 یا ATmega16L یا ATmega16A که این مدل‌ها دارای تفاوت‌هایی مطابق با جدول ۱-۱ می‌باشند.

نام میکروکنترلر	محدوده ولتاژ تغذیه	فرکانس قابل قبول کریستال
بدون پسوند	۵,۵-۴,۵ ولت	0 تا 16MHz
با پسوند L	۵,۵-۲,۷ ولت	0 تا 8MHz
با پسوند A	۵,۵-۲,۷ ولت	0 تا 16MHz
با پسوند V	۵,۵-۱,۸ ولت	0 تا 4MHz

جدول ۱-۱

ATmega16 دارای ۴۰ پایه می‌باشد که ترتیب آن به صورت زیر می‌باشد:

۳۲ پایه‌ی آن مربوط به پورت‌های A ,B , C وD است

۲ پایه‌ی آن مخصوص به مثبت و منفی تغذیه‌ی میکروکنترلر

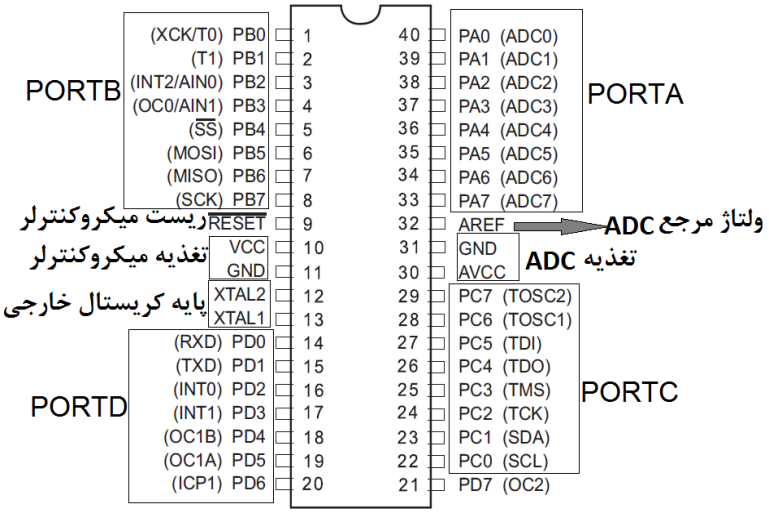
۲ پایه مخصوص به مثبت و منفی تغذیه‌ی مربوط به قسمت ADC

۱ پایه مربوط به ولتاژ مرجع در قسمت ADC

۲ پایه مربوط به کریستال خارجی

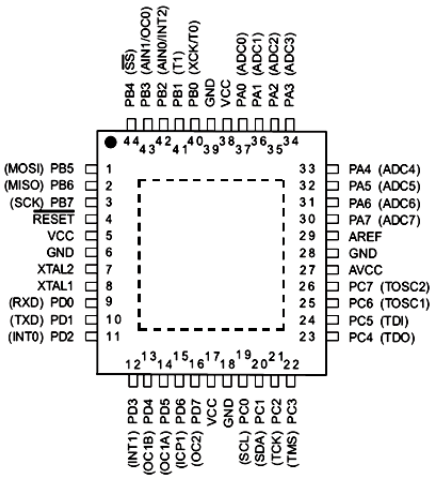
۱ پایه مربوط به Reset میکروکنترلر می‌باشد.

پایه‌های میکروکنترلر ATmega16 و نحوه‌ی قرارگیری آن‌ها را در شکل زیر مشاهده می‌کنید:



شکل ۱-۴ : پایه‌های میکروکنترلر ATmega16

روبه‌روی هر پایه نام آن ذکر شده است. اگر دقت کنید روبه‌روی هر پایه و در پرانتز نام دیگری نیز ذکر شده است، برای مثال در کنار پایه‌ی ۱۴ میکروکنترلر(پایه‌ی PD0) عبارت RXD نیز نوشته شده است که به معنای آن است که این پایه دارای نام دیگر RXD است که مربوط به قسمت ارتباط USART می‌باشد به این معنی که این پایه علاوه بر آن که پورت شماره‌ی صفر D به شمار می‌رود می‌تواند عملیات دریافت در ارتباط USART را نیز داشته باشد. با نام‌های مربوط به هر پایه در ادامه آشنا می‌شویم.به عنوان آخرین مطلب از این بخش باید بدانیم که میکروکنترلر ATmega16 علاوه بر بسته‌بندی DIP که دارای ۴۰ پایه و دارای عرض حدود ۱۳,۵ در ۵۲ میلی‌متر می‌باشد یک بسته‌بندی کوچک‌تر نیز برای مدارهای SMD در ابعاد حدود ۱۰ در ۱۲ میلی‌متر وجود دارد. این بسته‌بندی را در شکل روبرو مشاهده می‌کنید:



شکل ۱-۵ : بسته‌بندی SMD میکروکنترلر ATmega16

فصل دوم

آشنایی مقدماتی با زبان سی

در این فصل خواهیم خواند:

۱. ساده‌ترین برنامه در زبان C
۲. کاراکترهای کنترلی
۳. حساسیت C به بزرگی و کوچکی حروف
۴. جنس متغیرها در زبان C
۵. اعلان و تخصیص مقدار به متغیرها
۶. عملگرها
۷. دستورات شرطی در زبان C
۸. دستورات تکراردر زبان C
۹. نحوه‌ی ایجاد توابع در زبان C
۱۰. آرایه‌ها در زبان C
۱۱. رشته‌ها در زبان C
۱۲. معرفی کتابخانه‌های مهم کدویژن

در این بخش به صورت کاملاً گذرا با کتابخانه‌ها و فرم دستورات زبانC آشنامی‌شویم.

ساده‌ترین برنامه به زبان C

ساده‌ترین برنامه‌ای به زبان C که می‌توان نوشت مطابق زیر می‌باشد:

```
int main()
{
        .
        .
        .
}
```

نکته:

۱.عبارت int main() یا void main() در هر برنامه به زبانC باید وجود داشته باشد.

۲. شروع برنامه با { آغاز می‌گردد.

۳. خاتمه‌ی برنامه با } انجام می‌گیرد.

۴.دستورات برنامه در داخل {} نوشته می‌شوند و بعد از هر دستور بایستی از **;** استفاده کرد.

یکی از ساده‌ترین برنامه‌ها در زبان C نمایش متغیر می‌باشد. حال می‌خواهیم برنامه‌ای بنویسیم که پیغام "Hello" را چاپ‌کند:

```
#include<stdio.h>
int main()
{
printf("Hello");
}
```

کاراکترهای کنترلی

فرض‌کنید دستور "Hello Welcome" را چاپ کرده‌ایم. کد آن به صورت زیر است:

```
printf("Hello welcome");
```

در این حالت فاصله‌ها نیز در نمایش نشان داده می‌شوند.حال فرض‌کنید بخواهیم به صورت زیر و در دو خط چاپ شود:

Hello

Welcome

در این حالت باید از کاراکترهای کنترلی مانند **\n** استفاده کنیم(با دستور **\n** یک enter فاصله ایجاد می‌شود):

```
#include<stdio.h>
int main()
{
printf("Hello\nwelcome");
}
```

کاراکتر کنترلی بعدی **\t** می‌باشد که به اندازه‌ی یک تب فاصله(۸ کاراکتر)ایجاد می‌کند:

```
#include<stdio.h>
int main()
{
printf("Hello\twelcome");
}
```

که نتیجه‌ی آن به صورت زیر می‌شود:

Hellow welcome

بعنوان مثال دیگر کاراکتر کنترلی **\v** موجب انتقال کنترل به ۸ سطر بعد می‌شود:

```
#include<stdio.h>
int main()
{
printf("Hello\vwelcome");
}
```

در جدول زیر انواع کاراکترهای کنترلی را مشاهده می‌کنید:

کاراکتر	عملی که انجام می‌شود
\f	موجب انتقال کنترل به صفحه جدید می‌شود
\n	موجب انتقال کنترل به خط جدید می‌شود
\t	انتقال به ۸محل بعدی صفحه نمایش
\"	چاپ کوتیشن (")
\'	چاپ کوتیشن (')
\o	رشته تهی (Null)
\\	Back slash
\v	انتقال کنترل به ۸سطر بعدی
\N	ثابت‌های مبنای ۸(Nعدد مبنای ۸ است)
\xN	ثابت‌های مبنای ۱۶(xNعدد مبنای ۸ است)

جدول ۲-۱ : انواع کاراکترهای کنترلی

حساسیت به بزرگی و کوچکی حروف در زبان C

زبان C یک زبان حساس به بزرگی و کوچکی حروف است، یعنی به عنوان مثال زبان C برای متغیرهای xy,Xy,Yx,XY تمایزقائل می‌شود همچنین در نامگذاری متغیرها نباید از کلمات کلیدی زبان C استفاده کرد. در جدول زیر برخی از کلمات کلیدی در زبان C را مشاهده می‌کنید:

And	Sizeof	then	xor	Template
Float	False	Friend	While	continue
extern	Private	Switch	Default	Const
delete	typedef	if	this	Virtual

جدول ۲-۲ : کلمات کلیدی زبان C

جنس متغیرها در زبان C

متغیرها یا از نوع اعداد صحیح یا اعداد اعشاری یاحروف(کاراکتر)یا بیت یا... می‌باشند که بسته به نوع مورد نیاز آنها را انتخاب می‌کنیم. در جدول ۲-۳ خلاصه‌ای از متغیرهای زبان C را مشاهده می‌کنیم:

نوع داده	مقادیر	حافظه لازم
Int	32768- تا 32767	۲بایت
unsigned int	0 تا 65535	۲بایت
long int	2147483648- تا 2147483647	۴بایت
unsigned long int	0 تا 4294967295	۴بایت
Char	یک کاراکتر	۱بایت
signed char	128- تا 127	۱بایت
Float	1.2e-38 تا 3.4e38	۴بایت
Double	2.2e-308 تا 1.8e308	۸بایت

جدول ۲ - ۳ : جنس متغیرهای زبان C

اعلان و تخصیص مقدار به متغیرها

قبل از آنکه در برنامه به متغیرها مقداری تخصیص‌داده شود و از آنها استفاده گردد بایستی آنها را در برنامه اعلان نمود, برای مثال:

```
float x;
x=2.3;
```

با استفاده از عملگر = می‌توان به متغیرها مقدار اولیه تخصیص داد:

```
int y;
y=123;
```

برای نمایش داده‌های از نوع Char در حافظه کامپیوتر از جدول ASCII استفاده می‌شـود.جدول اسکی به هریک از ۲۵۶ کاراکتر یک عدد منحصر به فرد بین ۰ تا ۲۵۵ تخصیص‌می‌دهد:

```
Char c;
C='a';
C=97;
```

در جدول زیر کدهای اسکی را مشاهده می‌کنید:

ASCII	Hex	Symbol
0	0	NUL
1	1	SOH
2	2	STX
3	3	ETX
4	4	EOT
5	5	ENQ
6	6	ACK
7	7	BEL
8	8	BS
9	9	TAB
10	A	LF
11	B	VT
12	C	FF
13	D	CR
14	E	SO
15	F	SI

ASCII	Hex	Symbol
16	10	DLE
17	11	DC1
18	12	DC2
19	13	DC3
20	14	DC4
21	15	NAK
22	16	SYN
23	17	ETB
24	18	CAN
25	19	EM
26	1A	SUB
27	1B	ESC
28	1C	FS
29	1D	GS
30	1E	RS
31	1F	US

ASCII	Hex	Symbol
32	20	(space)
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/

ASCII	Hex	Symbol
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

ASCII	Hex	Symbol
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O

ASCII	Hex	Symbol
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

ASCII	Hex	Symbol
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o

ASCII	Hex	Symbol
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	□

جدول ۲ - ۴ : کدهای اسکی

در جدول ۲-۵ کاراکترهای فرمت در دستور printf() را مشاهده می‌کنید:

کاراکتر	نوع اطلاعاتی که باید به خروجی رود
%c	یک کارکتر
%d	اعداد صحیح دهدهی مثبت و منفی
%i	اعداد صحیح دهدهی مثبت و منفی
%e	نمایش علمی عدد همراه با حرف e
%E	نمایش علمی عدد همراه با حرف E
%f	عدد اعشاری ممیز شناور
%g	عدد اعشاری ممیز شناور
%G	عدد اعشاری ممیز شناور
%o	اعداد مبنای ۸ مثبت
%s	رشته‌ای از کاراکترها
%u	اعداد صحیح بدون علامت(مثبت)
%x	اعداد مبنای ۱۶ مثبت با حروف کوچک
%X	اعداد مبنای ۱۶ مثبت با حروف بزرگ
%p	اشاره‌گر
%n	موجب شمارش کاراکترهای خروجی تا قبل از این کاراکتر می‌شود
%%	علامت %

جدول ۲ - ۵ : کاراکترهای فرمت دستور Printf()

عملگرها

انواع عملگرها در زبان C عبارتند از: عملگرهای محاسباتی، عملگرهای رابطه‌ای، عملگر مقایسه‌ای و منطقی و عملگرهای بیتی که در جداول ۲-۶ تا ۲-۱۱ آنها را به همراه تقدم عملگرها در هر گروه مشاهده می‌کنید.

عملگر	نام عمل	مثال
-	تفریق یکانی	x-y
+	جمع	x+y
*	ضرب	x * y
/	تقسیم	x / y
%	باقیمانده‌ی تقسیم	x %y
--	کاهش یک واحدی	x - - یا –x
++	افزایش یک واحدی	X + + یا + + x

جدول ۲- ۶ : عملگرهای محاسباتی

بالاترین تقدم	++ --
	منهای یکانی
	* / %
پایین‌ترین تقدم	+ -

جدول ۲- ۷ : تقدم عملگرهای محاسباتی

عملگر	نام عمل	مثال
>	بزرگتر	x>y
>=	بزرگتر یا مساوی	x>=y
<	کوچکتر	x<y
<=	کوچکتر یا مساوی	x<=y
==	متساوی	x==y
!=	نامساوی	x!=y

جدول ۲- ۸ : عملگرهای رابطه‌ای

عملگر	نام عمل	مثال
!	نقیض(not)	!x
&&	و(and)	x>y && a<b
||	یا(or)	x>y || a<b

جدول ۲- ۹ : عملگرهای منطقی به ترتیب تقدم

عملگرهای ترکیبی:

عملگر	نام	مثال	معادل
+=	انتساب جمع	x+=y	x=x+y
-=	انتساب تفریق	x-=y	x=x-y
*=	انتساب ضرب	x*=y	x=x*y
/=	انتساب تقسیم	x/=y	x=x/y
%=	انتساب باقیمانده	x%=y	x=x%y

جدول ۲-۱۰: عملگرهای ترکیبی

عملگر	نام عمل
&	و (and)
|	یا (or)
^	یای انحصاری (xor)
~	نقیض (not)
>>	شیفت به راست (Srl)
<<	شیفت به چپ (Sll)

جدول ۲-۱۱: عملگرهای بیتی

مثالی از عملکرد عملگرهای بیتی:

X	Y	x&y	x|y	x^y	~x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

جدول ۲-۱۲

مثالی از عملکرد عملگرهای شیفت:

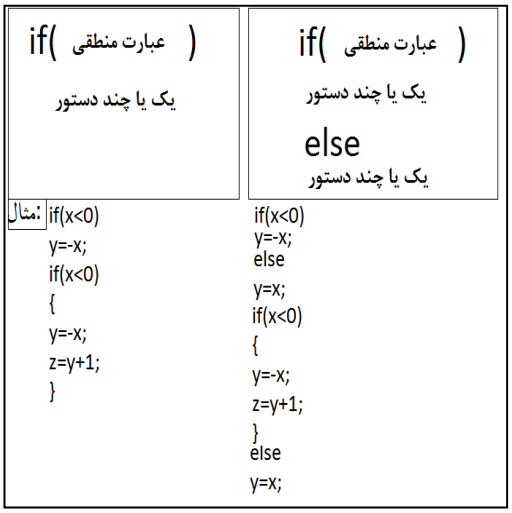
مقدار دهدهی x	مقدار دودویی x	Unsigned char x;
7	00000111	X=7;
14	00001110	x=x<<1
112	01110000	x=x<<3
192	11000000	x=x<<2
96	01100000	x=x>>1
24	00011000	x=x>>2

جدول ۲-۱۳

دستورات شرطی در زبان C

دســـــتور شـــــرطی if و if-else:

پیکره‌بندی این دستور مطابق شـکل روبـرو است:



۱-عبارات منطقی حتماً باید در داخل پرانتز باشد.

۲-در حالتی که بخش‌های if یا else بیش از یک دستور باشند بایـد بـین } و { قـرار گیرند.

مثال: در کد زیر نحوه‌ی محاسبه‌ی قدرمطلق را مشاهده می‌کنید:

```
#include<stdio.h>
void main()
{
float x;//float تعریف متغیر از نوع
printf("Enter a number"); // Enter a number چاپ عبارت
با دستورscanf یک عدد از جنس اعداد اعشاری دریافت می‌گردد و در متغیر x ذخیره
scanf("%f",&x);// می‌گردد
if(x<0)//x<0 بررسی عبارت منطقی
printf("%f",-x);// –x چاپ متغیر
else // x<0 بررسی حالت غیر از شرط
printf("%f",x);// x چاپ متغیر
}
```

دستور شرطی switch-case: برای تصمیم‌گیری‌های چندگانه بر اساس مقادیر مختلف یک عبارت می‌باشد

```
switch( متغیر از نوع صحیح یا کاراکتر )
{
case عدد صحیح یا کاراکتر :
دستور(ات)
break;
case عدد صحیح یا کاراکتر :
دستور(ات)
break;
.....
default:
دستور(ات)
}
```

متغیر یا مقدار صحیح با مقادیر موجود در caseها مقایسه می‌شود اگر با مقدار یکی از caseها مطابقت داشت، دستورات داخل آن اجرا می‌شود. در صورت عدم تطابق با هیچ یک از caseها دستورات بخش default اجرا می‌شود. پیکره‌بندی این دستور به صورت شکل روبرو می‌باشد.

نکته:

۱- آوردن بخش default اختیاری است.

۲- نیازی به قراردادن {} ما بین دستورات caseها نیست.

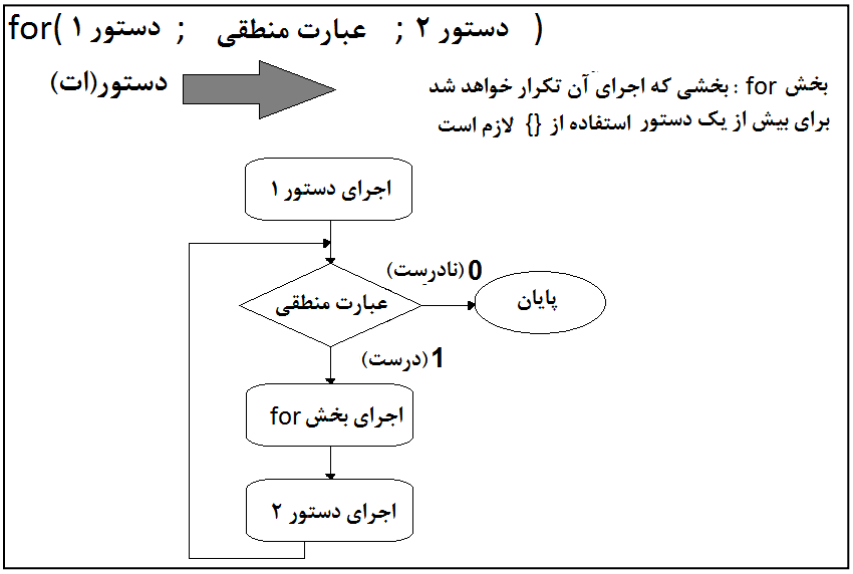
۳- نوشتن break در انتهای دستورات case الزامی است.

۴- دستور break موجب خروج از دستور switch-case می‌شود بعنوان نمونه به کد زیر دقت کنید:

```
x=1;
switch(x)
{
case 1:
  printf("*");
 break;
case 2:
 printf("**");
 break;
case 3:
  printf("***");
  break;
default:
  printf("Error");
}
```

دستورات تکرار(حلقه‌ها) در زبان C

دستور for: شکل کلی این دستور را در شکل زیر مشاهده می‌کنید:

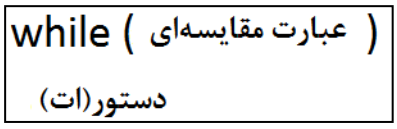


شکل ۲-۳ : پیکره‌بندی دستور**for**

مثال: برنامه‌ای بنویسید که ۱۰۰عدد را از کاربر بگیرد و جمع بزند:

```
#include<stdio.h>
void main()
{
int i;
float sum;
float x;
sum=0;
for(i=0;i<100;i++)
{
scanf("%f",&x);
sum+=x;
}
printf("The Result =%f",sum);
}
```

دستور ()while: پیکره‌بندی این دستور به شکل زیر است:



شکل ۲-۴ : پیکره‌بندی دستور ()while

عملکرد این دستور بدین صورت است که اگر عبارت مقایسه‌ای درست باشد دستور(ات) اجرامی‌شود و اگر درست نباشد، از حلقه خارج می‌شود و در صورتی که تعداد دستورات بیش از یک دستور باشد باید از {} استفاده کرد.

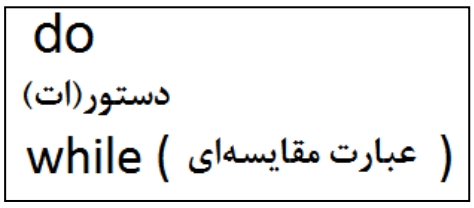
نکته(تفاوت مهم for و while): با استفاده از دستور while می‌توان حلقه‌هایی نوشت که در آنها تعداد تکرار از قبل مشخص نیست برخلاف for که تعداد تکرار از قبل معمولاً مشخص‌است.

مثال:چاپ ۰۱۲۳ بر روی نمایشگر:

```
i=0;
while (i<4)
{
   printf("%d",i);
   i++;
}
```

دستور ()do-while: پیکره‌بندی این دستور را درتصویر روبرو مشاهده می‌کنید:

این دستور عملکردی کاملاً مشابه while دارد با این تفاوت که در while عبارت مقایسه‌ای در ابتدا و قبل از اجرای دستورات چک می‌شود اما در do-while این کار در انتها و بعد از اجرای دستورات انجام می‌شود(بخش حلقه لااقل یکبار اجرا می‌شود).



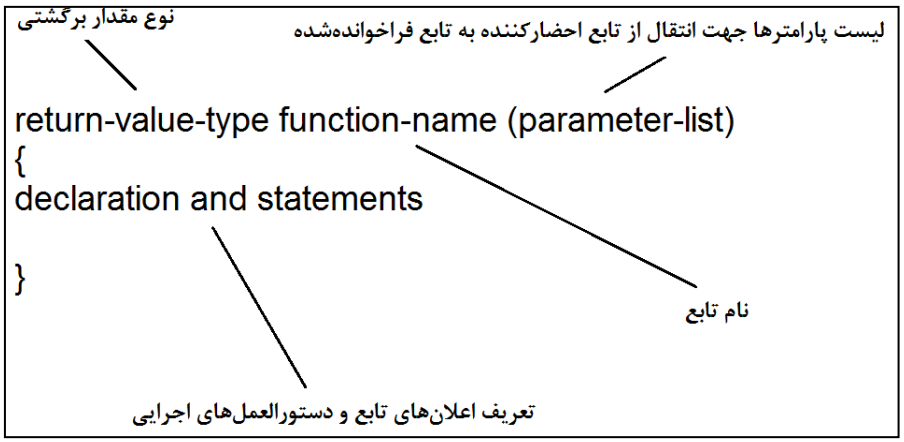
شکل ۲-۵ : پیکره‌بندی دستور()do while

دستور break: این دستور موجب خروج از حلقه‌ی تکرار می‌شود. همانند کد مثال زیر:

```
#include <stdio.h>
void main()
{
int t;
for (t=0;t<100;t++)
{
      printf("%d",t);
      if(t==5)
            break;
}
}//          ***در این برنامه اعداد صحیح ۰ تا ۵ چاپ می‌گردد و با برابر شدن متغیرt=5 برنامه از حلقه خارج می‌شود
```

نحوه‌ی ایجاد توابع در زبان C

استفاده از توابع به برنامه‌نویس این امکان را می‌دهد که برنامه‌های خود را به صورت قطعه قطعه بنویسد. شکل کلی توابع را در شکل زیر مشاهده می‌کنید:



شکل ۲-۶ : نحوه‌ی ایجادتوابع در زبانC

نکته‌ی۱: اسامی پارامترها و آرگومانهای یک تابع می‌تواند همنام باشد.

نکته‌ی۲: وقتی در تابعی, تابع دیگر احضارمی‌گردد بایستی تعریف تابع احضارشونده قبل از تعریف تابع احضارکننده در برنامه ظاهرگردد.

نکته‌ی ۳: اگر بخواهیم در برنامه‌ها ابتدا تابع main ظاهرگردد، بایستیprototype یعنی پیش‌نمونه‌ی تابع که شامل نام تابع، نوع مقدار برگشتی تابع، تعداد پارامترهایی که تابع انتظار دریافت آنها را دارد و انواع پارامترها و ترتیب قرارگرفتن این پارامترها را به اطلاع کامپایلر برساند.

مثالی از نکته‌ی ۳ (محاسبه‌ی فاکتوریل):

```
#include <stdio.h>
#include <conio.h>

long int factorial(int);//function prototype
int main()
{
int n;
printf("Enter a positive integer");
scanf("%d",&n);
printf("%ld\n",factorial(n));
return 0;
}
long int factorial(int n)
{
long int fact =1;
if (n>1)
for(int i=2;i<=n;++i)
fact*=i;
return(fact);
}
```

نکته‌ی ۴: درصورتی‌که تابع مقداری برنگرداند، نوع مقدار برگشتی تابع را void اعلان می‌کنیم و درصورتی‌که تابع مقداری را دریافت نکند، به جای parameter از void یا () استفاده می‌گردد.

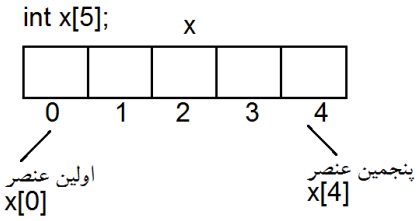
مثالی از توابعی که مقداری را برنمی‌گردانند (تابع void):

```
#include <stdio.h>
#include <conio.h>
void maximum(int,int);
int main()
{
int x,y;
clrscr()
scanf("%d%d",&x,&y);
maximum(x,y);
return 0;
}
void maximum(int x,int y)
{
int z;
z=(x>=y) ? x:y ;
printf("max value \n %d",z);
return ;
}
```

آرایه‌ها در زبان C

آرایه یک فضای پیوسته از حافظه‌ی اصلی کامپیوتر می‌باشدکه می‌تواند چندین مقدار را در خود جای دهد. توجه به ۳ نکته در باب آرایه مهم می‌باشد: ۱-کلیه عناصر آرایه از یک نوع می‌باشند. ۲-عناصر آرایه بوسیله‌ی اندیس آنها مشخص می‌شود. ۳-در زبان C اندیس آرایه از صفر شروع می‌شود.

مثالی از یک آرایه‌ی یک بعدی را مطابق شکل زیر مشاهده می‌کنید:



شکل ۲-۷ : آرایه‌ی یک‌بعدی

تخصیص مقادیر به عناصر آرایه‌ی یک بعدی:

```
int x[5]={4,2,5,17,30}
```

x

4	2	5	17	30
0	1	2	3	4

شکل ۲-۸ : مثالی از تخصیص مقادیر به عناصر آرایه‌ی یک بعدی

مثالی از یک‌آرایه‌ی دو بعدی:

	ستون 0	ستون 1	ستون 2	ستون 3
سطر 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
سطر 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
سطر 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

شکل ۲-۹ : آرایه دو بعدی

تخصیص مقادیر به عناصر آرایه‌ی دوبعدی:

```
int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

	ستون 0	ستون 1	ستون 2	ستون 3
سطر 0	1	2	3	4
سطر 1	5	6	7	8
سطر 2	9	10	11	12

شکل ۲-۱۰ : **مثالی از تخصیص مقادیر به عناصر آرایه‌ی یک بعدی**

نکات:

نکته‌ی۱: به نحوه‌ی پرشدن آرایه‌ها در شکل زیر توجه کنید:

```
int a[3][4]={ {1},{2,3},{4,5,6}};
```

	ستون 0	ستون 1	ستون 2	ستون 3
سطر 0	1	0	0	0
سطر 1	2	3	0	0
سطر 2	4	5	6	0

شکل ۲-۱۱ : نحوه‌ی پر شدن خانه‌های آرایه

نکته‌ی۲: به نحوه‌ی پرشدن آرایه‌های شکل زیر توجه‌کنید:

```
int a[3][4]={1,2,3,4,5};
```

	ستون 0	ستون 1	ستون 2	ستون 3
سطر 0	1	2	3	4
سطر 1	5	0	0	0
سطر 2	0	0	0	0

شکل ۲-۱۲ : : نحوه‌ی پر شدن خانه‌های آرایه

نکته‌ی۳: در یک آرایه‌ی دو بعدی هر سطر در حقیقت آرایه‌ی یک‌اندیسی می‌باشد. در اعلان آرایه‌های دو اندیسی ذکر تعداد ستون‌ها لازم است.

رشته‌ها در زبان C

در زبان C رشته‌ها به صورت آرایه‌ای از کاراکترها پیاده می‌شوند. رشته یا string عبارتست از دنباله‌ای از کاراکترها که بین " " قرارداده می‌شوند و در کامپیوتر انتهای هر رشته به \0 ختم می‌شود.

```
charst[20]="Pnk"
```

0	1	2	3
P	n	k	\0

شکل ۲-۱۳ : نمایشی از یک رشته

همچنین می‌توان بدون تعیین‌کردن طول رشته، آن را مقداردهی‌کرد. مطابق شکل روبرو:

```
char name[]="sara";
```

s
a
r
a
\0

شکل ۲-۱۴

نکته: برای استفاده از توابع مربوط به رشته‌ها حتماً از کتابخانه‌ی string.h بایستی استفاده کرد.

معرفی کتابخانه‌های مهم کدویژن

کتابخانه‌های زیادی در کدویژن وجود دارد که می‌توان از آنها نیز استفاده کرد، اما آنچه معرفی می‌گردد مهم‌ترین کتابخانه‌های مورد استفاده در زبان C و کاربردی میکروکنترلرها در کدویژن می‌باشند.

۱)کتابخانه‌ی stdio.h: این کتابخانه دارای توابع مهمی است که کار با ورودی و خروجی‌ها را آسان می‌کند. البته مهم‌ترین کاربرد این کتابخانه برای کار آسان با ارتباط سریال و LCD می‌باشد. مهم‌ترین توابع این کتابخانه را در جدول ۲-۱۴ مشاهده‌می‌کنید:

تابع	عملکرد
getchar(void)	بازگرداندن کاراکتر دریافتی از Uart (تا زمانی که کاراکتر جدید نگیرد مقداری را برنمی‌گرداند)
putchar(char s)	ارسال کاراکتر s از طریق Uart
puts(char*str)	ارسال رشته str واقع در حافظه sram از طریق Uart
putsf(char flash*str)	ارسال رشته str واقع در حافظه flashاز طریق Uart

جدول ۲ - ۱۴ : برخی از مهم‌ترین توابع کتابخانه‌ی Stdio.h

۲)کتابخانه‌ی delay.h: از این کتابخانه برای ایجاد تاخیر در برنامه‌ها استفاده می‌کنیم. در جدول ۲-۱۵ دو دستور مهم این کتابخانه را می‌بینید:

تابع	عملکرد
delay_ms(unsigned int x)	ایجاد تاخیری به اندازه‌ی x میلی‌ثانیه
delay_us(unsigned int x)	ایجاد تاخیری به اندازه‌ی x میکروثانیه

جدول ۲ - ۱۵ : برخی از مهم‌ترین توابع کتابخانه‌ی delay.h

۳)کتابخانه‌ی stdlib.h: برخی از مهم‌ترین توابع این کتابخانه را در جدول ۲-۱۶ مشاهده می‌کنید:

تابع	عملکرد
itoa(intx,char*str)	عدد x را به رشته‌ی str تبدیل‌می‌کند
atoi(char*str)	رشته‌ی str را به عدد صحیح x تبدیل‌می‌کند
atof(char*str)	رشته‌ی str را به عدد اعشاری تبدیل‌می‌کند
rand(void)	تولید یک عدد تصادفی بین ۰ تا۳۲۷۶۷
srand(void)	تولیدکننده‌ی عدد تصادفی را مقداردهی می‌کند

جدول ۲ - ۱۶ : برخی از مهم‌ترین توابع کتابخانه‌ی Stdlib.h

۴)کتابخانه‌ی string.h: این کتابخانه دارای توابعی است که کار بارشته‌ها را راحت‌تر می‌کند. مهم‌ترین توابع این کتابخانه مطابق جدول ۲-۱۷:

تابع	عملکرد
char*stract(char*str1,char*str2)	اتصال رشته‌ی str2 به انتهای رشته‌ی str1
strcmp(char*str1,char*str2)	دو رشته را بر مبنای حروف الفبا مقایسه‌کرده و اگر str1>str2 باشد مقداری منفی و اگر str2>str1 مقداری مثبت و اگر مساوی باشند صفر را برمی‌گرداند.
strpos(char str1,char c)	اندیس اولین خانه‌ی وقوع کاراکتر c را در رشته‌ی str1 برمی‌گرداند در غیر این‌صورت منفی یک را برمی‌گرداند
strcpy(char*str1,char*str2)	رشته‌ی str1 را در رشته‌ی str2 کپی‌می‌کند
strlen(char*str)	طول رشته‌ی str را مشخص‌می‌کند

جدول ۲ - ۱۷ : برخی از مهم‌ترین توابع کتابخانه‌ی String.h

۵)کتابخانه‌ی math.h: این کتابخانه دارای مهم‌ترین توابع ریاضی است کـه اعمـال ریاضـی را برای ما راحت‌ترمی‌کند. در جدول ۲-۱۸ مهم‌ترین توابع آن را مشاهده می‌کنید:

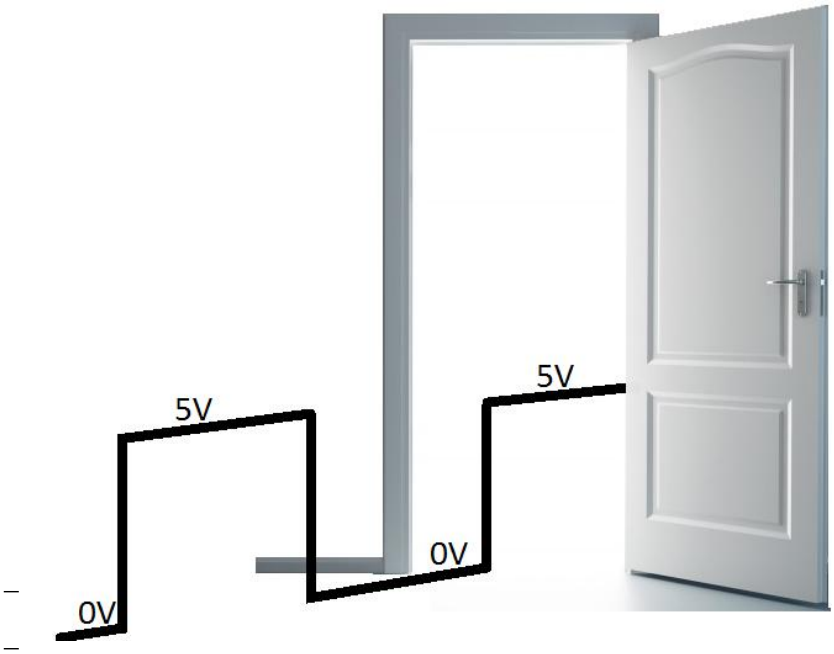
تابع	عملکرد
abs(int x)	محاسبه‌ی قدرمطلق x
acos(double arg)	محاسبه آرک‌کسینوس arg
asin(double arg)	محاسبه آرک‌سسینوس arg
atan(double arg)	محاسبه آرک‌تانژانتarg
atan2(double x,double y)	X را بر y تقسیم و از نتیجه آرک‌تانژانت می‌گیرد
Complex	شامل ۳ تابعimag() وreal() وconj() می‌باشـد که به ترتیب قسمت موهومی ، حقیقـی و مـزدوج یک عدد مختلط را حساب می‌کند.
ceil(double x)	محاسبه کوچکترین عدد صحیح بزرگتر یا مسـاوی با x
cos(double arg)	محاسبه کسینوس arg
cosh(double arg)	محاسبه کسینوس‌هایپربولیک arg
exp(double arg)	محاسبهe به توان arg
fabs(double x)	محاسبه قدرمطلق اعداد اعشاری
floor(double x)	محاسبه بزرگترین عدد صحیح بزرگتر یا مساوی با x
fmode(double x,double y)	محاسبه باقیمانده‌یx/y
ldexp(double x,int y)	محاسبه x*(2^(y))
hypot(double x,double y)	X وy را بعنوان اضلاع مثلث قائم‌الزاویه دریافـت و وتر آن‌را محاسبه‌می‌کند
log(double x)	گرفتن ln(x)
log10(double x)	گرفتن log(x)
pow(double x,double y)	محاسبهx^y
sin(dublearg)	محاسبه سینوس arg
sinh(dublearg)	محاسبه سینوس‌هایپربولیک arg
sqrt(duble x)	محاسبه جذر x
tan(dublearg)	محاسبه تانژانت arg
tanh(dublearg)	محاسبه تانژانت‌هایپربولیک arg

جدول ۲- ۱۸ : برخی از مهم‌ترین توابع کتابخانه‌ی math.h

کتابخانه‌های دیگری در کدویژن وجود دارد ولی آنچه معرفی شد مهم‌ترین و کاربردی‌ترین کتابخانه‌ها و دستورات در کار با میکروکنترلرهای AVR می‌باشند.

فصل سوم

ورودی و خروجی در AVR



در این فصل خواهیم خواند:

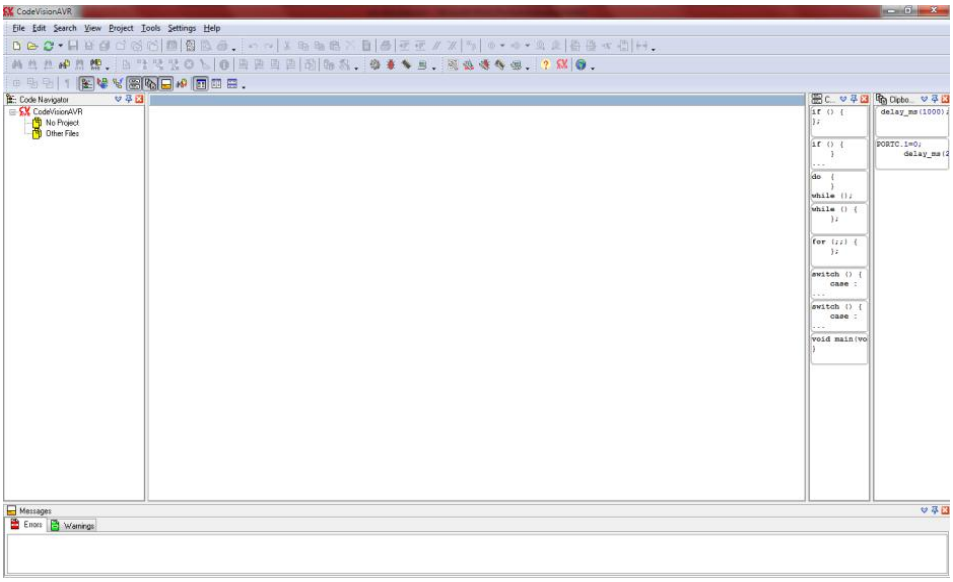
۱. نحوه‌ی ایجاد پروژه در کدویژن
۲. تنظیمات اولیه میکروکنترلر در پروژه‌ها
۳. تنظیمات PIN و PORT در کدویزارد
۴. نحوه ذخیره کردن پروژه ایجاد شده در کدویژن
۵. کدنویسی اولیه در محیط کدویژن
۶. آشنایی اولیه با محیط پروتئوس
۷. یک گام فراتر
۷-۱. رجیستر DDRx
۷-۲. رجیستر PORTx
۷-۳. رجیستر PINx

ورودی و خروجی (PIN & PORT)

در این بخش قصد داریم نحوه‌ی ایجاد یک پروژه در کدویژن، کار با بخش ورودی‌ها و خروجی‌های میکروکنترلر و تنظیمات مربوط به کدویزارد در بخش PIN & PORT آشنا شویم سپس قدم به قدم سطح خود را بالاتر برده و در بخش یک گام فراتر بر این مبحث مسلط شویم، فقط باید به این نکته توجه کنیم که برای آنکه در یک مبحث بتوانیم به خوبی بر مفاهیم و کاربردها مسلط شویم نیازمند تمرین بسیار هستیم. در این کتاب سعی بر این بوده که مفاهیم با مثال‌هایی ساده(معمولاً خاموش و روشن کردن LED) بیان گردد که این مثال‌ها قابل تعمیم به کارهای پیچیده‌تر و LED ها نیز نماد از دستورات و کارهای مورد نظر هستند.

نحوه‌ی ایجاد یک پروژه در کدویژن

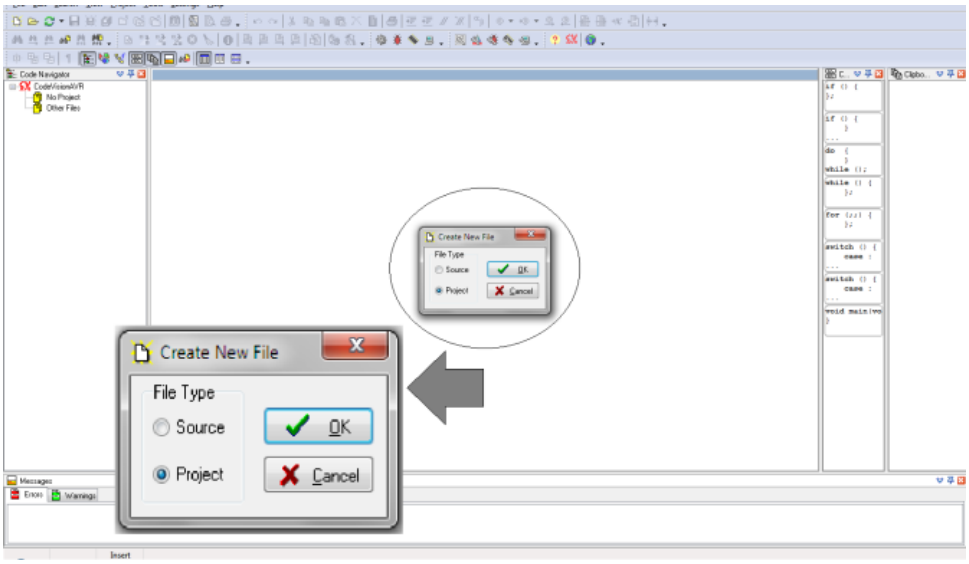
پس از بازکردن نرم‌افزار CODEVISION با صفحه‌ای مطابق شکل زیر روبرو می‌شویم که این صفحه محیط نرم‌افزار را نشان می‌دهد:



شکل ۳-۱ : محیط نرم‌افزار کدویژن

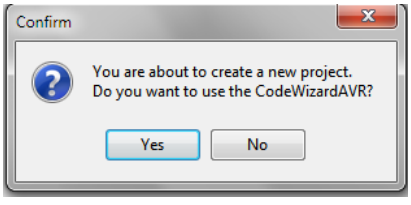
در صفحه‌ای که باز می‌شود برای ایجاد یک پروژه‌ی جدید ابتـدا از منـوی File گزینـه‌ی New را انتخاب می‌کنیم(می‌توانیم از کلید میانبر Ctrl+N نیـز اسـتفاده کنـیم). بـا انتخـاب ایـن گزینـه پنجره‌ای به نام Create New File باز می‌شود، در این پنجره دو گزینه‌ی Source و Project را مشاهده می‌کنیم که اگر گزینه‌ی Source را انتخاب کنیم در کدنویسی باید تمـامی تنظیمـات مربوط به رجیسترهای میکروکنترلر را خودمان و با نوشتن کدهـای مربوطـه انجـام دهـیم ولـی از آنجایی که این کار برای کسانی که کار با میکروکنترلر را تازه شروع کرده‌اند کمی دشوار اسـت، راه دوم انتخاب گزینه‌ی Project است که با انتخاب این گزینه می‌توانیم از بخش کدویزارد نـرم‌افـزار کدویژن استفاده‌کنیم که به صورت طبقه‌بندی شده می‌توان تمامی تنظیمات میکروکنترلر را انجـام داد. در ادامه مراحل تشکیل پروژه با کدویزارد را مشاهده می‌کنیم:

۱)گزینه Project را انتخاب کرده و سپس گزینه‌ی OK را می‌زنیم:



شکل ۳-۲ : ایجاد یک پروژه‌ی جدید

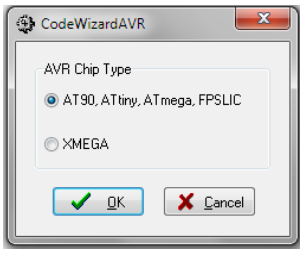
با این کار پنجره‌ای باز می‌شود که مطابق شکل۳-۳ از کاربر مـی‌پرسـد کـه آیـا بـرای تنظیمـات مربـوط بـه میکروکنترلـر و ایجـاد پـروژه تمایـل‌داریـد از



شکل ۳-۳ : انتخاب کدویزارد

CodeWizard استفاده‌کنید؟ و چون در این بخش قصد آموزش کدویزارد را داریم گزینه‌ی Yes را انتخاب می‌کنیم.

سپس نرم‌افزار از کاربر می‌پرسد که نوع قطعه‌ی شما (AVRChipType) AT90,ATtiny,ATmega است یا Xmega؟ قطعه‌ای که در این کتاب با آن کار می‌کنیم از خانواده‌ی ATmega است، پس گزینه‌ی اول را انتخاب می‌کنیم.

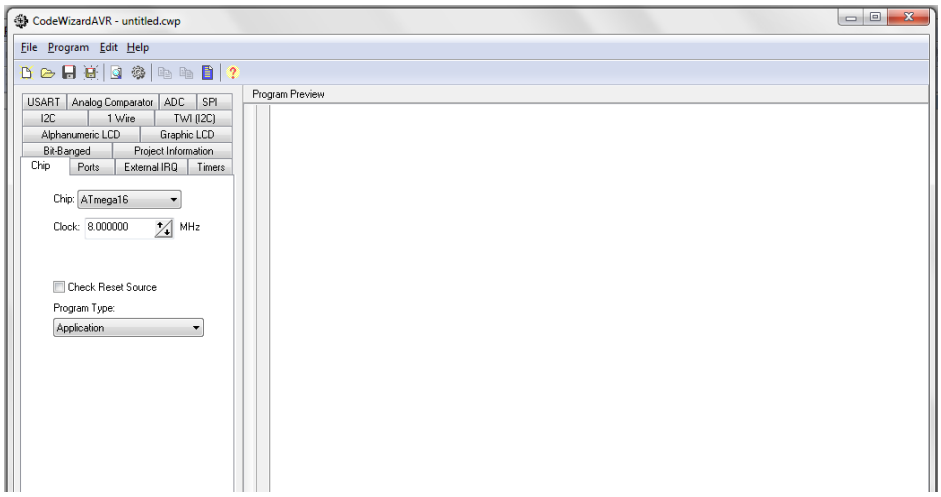


شکل ۳-۴ : انتخاب نوع میکروکنترلر

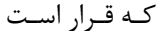
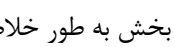
تا به اینجا تنظیمات اولیه‌ی ایجاد یک پروژه در کدویژن با استفاده از کدویزارد را انجام دادیم. در ادامه با مابقی تنظیمات لازم برای ایجاد یک پروژه آشنا می‌شویم.

تنظیمات اولیه‌ی میکروکنترلر در پروژه‌ها

با زدن گزینه‌ی OKپنجره‌ای مانند شکل زیر باز می‌شود:



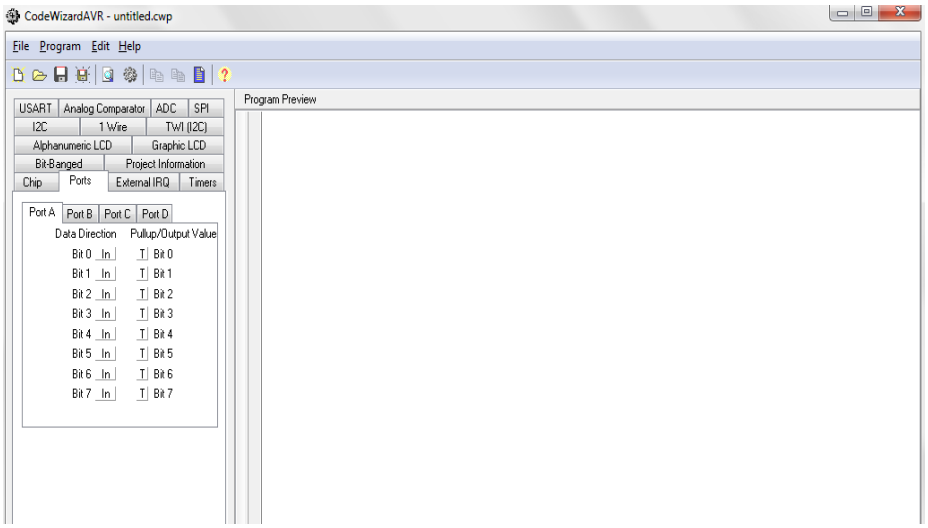
شکل ۳-۵ : محیط کدویزارد

حال از قسمت Chip ( )  نوع قطعه‌ای که قرار است با آن کارکنیم را انتخاب می‌کنیم، که در این کتاب هدف یادگیری نحوه‌ی کار با میکروکنترلرهای ATmega16 است. پس از انتخاب نوع قطعه(Chip) باید فرکانس‌کاری میکروکنترلر یا Clock() را انتخاب کنیم. در این بخش به طور خلاصه در مورد فرکانس‌کاری صحبت می‌شود و کار با فرکانس‌های مختلف، در بخش تایمر بررسی می‌گردد. هر چه فرکانس‌کاری میکروکنترلر بیشتر باشد، سرعت پردازش نیز بیشتر است و عملیات‌های واحدهای مختلف سریعتر انجام می‌گردد. میکروکنترلر بدون استفاده از کریستال خارجی توانایی تولید فرکانس تا ۸ مگاهرتز را دارا می‌باشد(استفاده از کریستال خارجی در بخش تایمر توضیح داده می‌شود). حال سوالی که ممکن است مطرح شود این است که وقتی می‌گوییم هر چه فرکانس بالاتر باشد سرعت افزایش می‌یابد، پس چرا همیشه ۸مگاهرتز را انتخاب نکنیم و چرا گزینه‌هایی برای سایر فرکانس‌ها مانند ۱ مگاهرتز وجود دارد؟ پاسخ این است که گاهی اوقات نیازمند تولید فرکانس خاصی در برخی کاربردها می‌باشیم(همانند تولید شکل موج با فرکانسی خاص) و میکروکنترلر باید با توجه به مقسم‌های فرکانس و فرکانس‌کاری خود، فرکانس موردنظر را تولیدکند برای مثال اگر فرکانس کاری میکرو را بر روی ۸ مگاهرتز تنظیم کنیم در بخش تایمر می‌توانیم با فرکانس‌های ۸ مگاهرتز، ۱ مگاهرتز، ۱۲۵ کیلوهرتز و ... کار کنیم ولی اگر فرکانس کاری میکرو را بر روی ۴ مگاهرتز تنظیم کنیم در بخش تایمر می‌توانیم با فرکانس‌های ۴ مگاهرتز، ۵۰۰ کیلوهرتز، ۶۲,۵ کیلوهرتز و . . . کار کنیم. در این بخش ما تنظیمات خود را با فرکانس ۸مگاهرتز انجام می‌دهیم.

تنظیمات PIN & PORT در کدویزارد

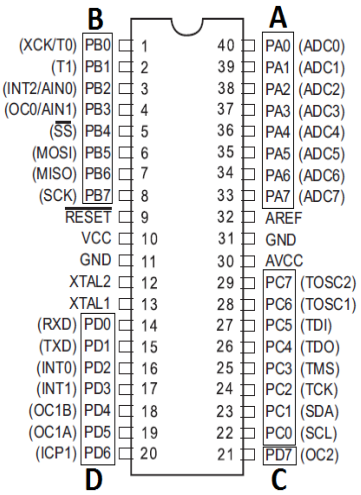
میکروکنترلرهای ATmega16 دارای ۴ بخش ورودی و خروجی می‌باشند؛ پورت A(PORTA) ، پورت B (PORTB) ، پورت C (PORTC) و پورت D (PORTD) که هر پورت دارای ۸ پایه می‌باشد، برای مثال پورتA از ۸پایه‌ی PORTA.0 تا PORTA.7 تشکیل شده است، این پایه‌ها می‌توانند به صورت ورودی و یا خروجی تنظیم شوند که -ورودی یا خروجی بودن این پایه‌ها باید در ابتدای برنامه مشخص شود. اگر پایه‌ای به صورت ورودی انتخاب شد میکروکنترلر دیگر قادر به تولید ولتاژ توسط آن پایه نیست و فقط می‌تواند توسط این پایه ولتاژهای ورودی را بخواند. علاوه بر ورودی و خروجی بودن پایه‌ها مقدار پیش‌فرض آنها نیز اهمیّت دارد، برای مثال اگر پایه‌ی اول

پورت B (PORTB.0) به عنوان خروجی انتخاب شد، این پایه می‌تواند دارای مقدار اولیه‌ی صفر و یا یک باشد یعنی به محض روشن شدن میکروکنترلر ولتاژ خروجی این پایه صفر یا ۵ ولت باشد که این مقدار پیش‌فرض نیز در ابتدای برنامه مشخص می‌شود.اکنون می‌خواهیم با نحوه‌ی تنظیمات ورودی و خروجی میکروکنترلرها آشنا شویم. از تب‌های موجود، تب Ports را انتخاب می‌کنیم. پنجره‌ای مانند شکل زیر باز می‌شود:



شکل ۳-۶ : تنظیمات ورودی وخروجی

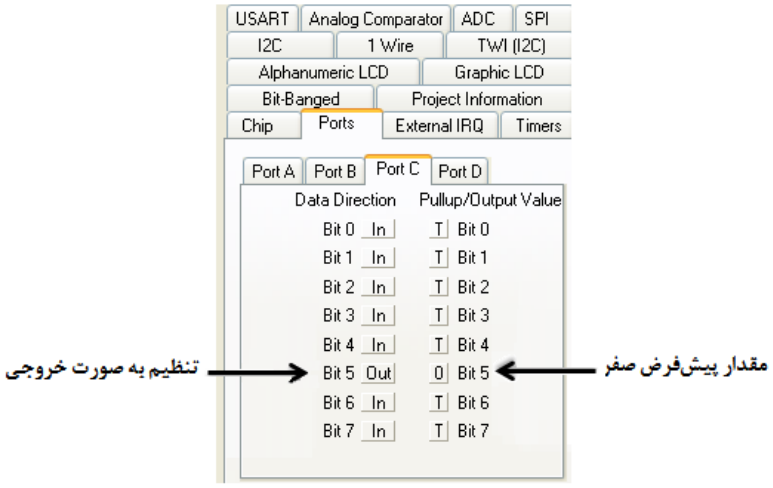
همانطورکه مشاهده می‌کنید این تب از ۴ بخش PORTA,PORTB,PORTC, PORTD تشکیل شده است که می‌توانیم هر یک از آنها را به صورت ورودی یا خروجی تنظیم کنیم.



شکل ۳-۷ : درگاه‌های میکروکنترلر

قصد داریم در اولین مثال این کتاب، پایه‌ای دلخواه از میکروکنترلر (مثلاً پایه‌ی PC5) را به صورت خروجی تنظیم کنیم و یک LED به آن وصل‌کرده و میکروکنترلر LED را به صورت یک ثانیه یکبار خاموش و روشن کند. (خاموش بودن پایه معادل صفر و روشن بودن آن معادل یک است. اگر پایه‌ای یک باشد به معنای آن است که دارای ولتاژی برابر ۵ ولت است (در صورتی که تغذیه‌ی میکروکنترلر برابر ۵ ولت باشد)).

پس در تب Ports وارد PORTC می‌شویم و PC5 که بیت شماره۵ پورت C است را به صورت OUT(خروجی) تنظیم می‌کنیم و مقدار آن را به صورت پیش‌فرض بر روی صفر ولت تنظیم می‌کنیم:

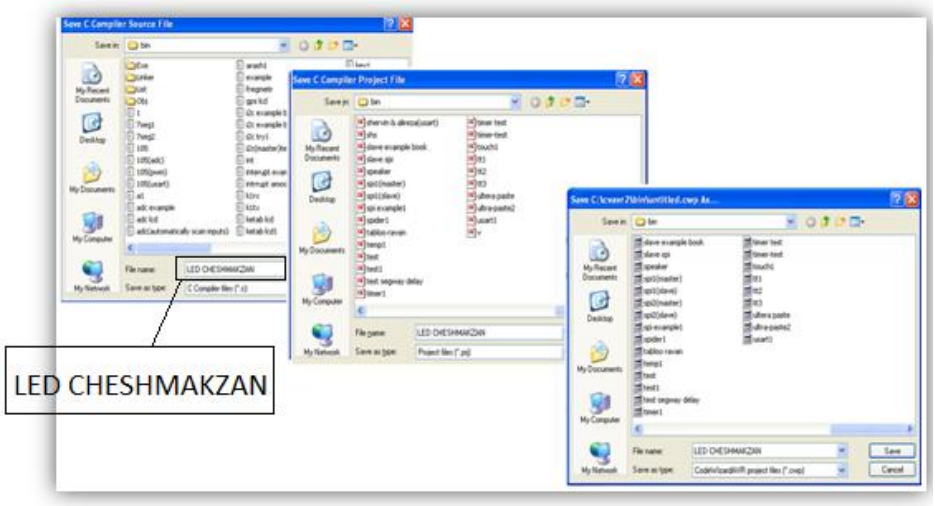


شکل ۳-۸ : تنظیم پایه‌ی PC5

نحوه‌ی ذخیره کردن پروژه‌ی ایجاد شده در کدویژن

کار تنظیمات اولیه در این مثال به اتمام رسید. برای ذخیره و ایجاد کد مربوط به این تنظیمات از قسمت Program گزینه‌ی Generate, Save and Exit(Generate, Save and Exit را انتخاب می‌کنیم. با این کار پنجره‌ای مطابق شکل ۳-۹ باز می‌شود، سپس فایل خود را با نام دلخواه ذخیره می‌کنیم. عملیات ذخیره‌سازی سه بار به سه فرمت مختلف انجام می‌شود(.C /.Prg/.Cwp)بهتر است

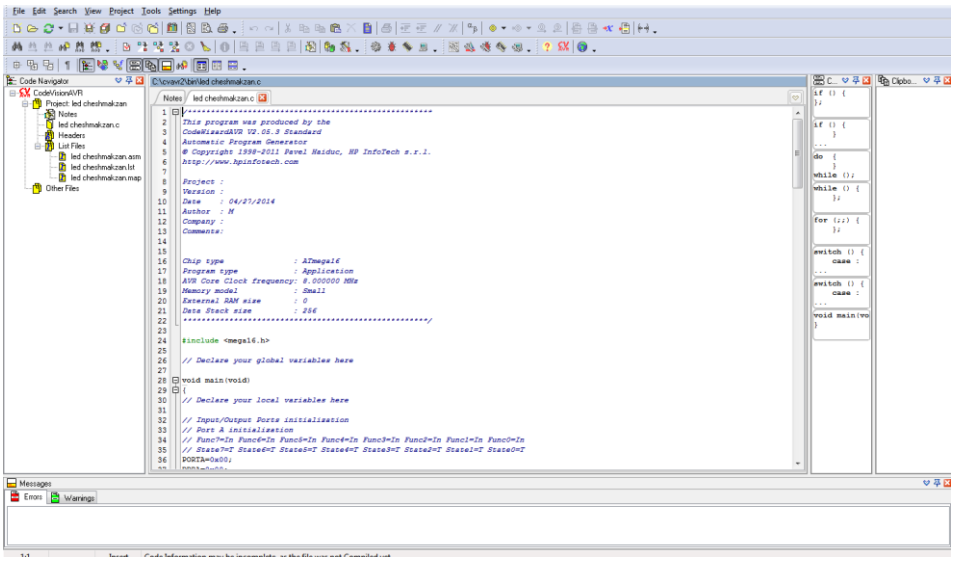
نام ۳ فرمتی که ذخیره می‌کنید یکسان باشد، در این پروژه هر ۳فرمت را با نام LED CHESHMAKZAN ذخیره کرده‌ایم.



شکل ۳-۹ : ذخیره‌ی تنظیمات

کدنویسی اولیه در محیط کدویژن

بعد از ۳ بار ذخیره کردن صفحه‌ای مانند شکل زیر باز می‌شود:



شکل ۳-۱۰

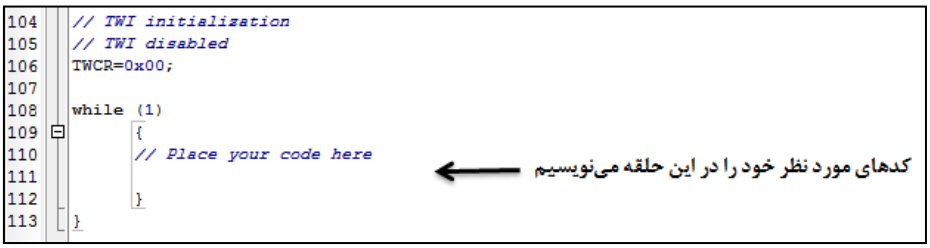
در این صفحه کدهایی مشاهده می‌شود که این کدها مربوط به تنظیمات اولیه‌ی میکروکنترلر است (تنظیمات مربوط به قسمت‌های مختلف مانند وروردی و خروجی‌ها و..) اگر به جای کار با کدویزارد گزینه‌ی New Source را انتخاب می‌کردیم، باید این کدها را خودمان می‌نوشتیم ولی چون در تنظیمات خود از کدویزارد استفاده کردیم تنظیمات را به صورت تصویری انجام دادیم. ما با این کدهای تولید شده در حال حاضر کاری نداریم فقط در بخش کتابخانه‌ها، کتابخانه‌ی تاخیر یا delay.h را اضافه می‌کنیم چراکه می‌خواهیم LED را یک‌ثانیه یکبار خاموش و روشن کنیم و به تاخیرهای یک ثانیه‌ای نیاز داریم. مطابق شکل زیر با تایپ #include<delay.h> کتابخانه را اضافه می‌کنیم:

```
15
16  Chip type               : ATmega16
17  Program type            : Application
18  AVR Core Clock frequency: 8.000000 MHz
19  Memory model            : Small
20  External RAM size       : 0
21  Data Stack size         : 256
22  *****************************************************/
23
24  #include <mega16.h>
25  #include <delay.h>
26
27  // Declare your global variables here
28
29  void main(void)
30  {
```

حال به نکات مربوط به کد توجه کنید:

در انتهای کدهای تولید شده یک حلقه‌ی while(1) مشاهده می‌شود. همانطورکه می‌دانید حلقه‌ی while() تا زمانی‌که عبارت درون پرانتز آن درست باشد اجرا می‌شود و زمانی‌که درون آن عدد ۱ نوشته شود، به این معناست که این عبارت همواره صحیح است و این حلقه تا بی‌نهایت تکرار می‌شود. در برنامه‌نویسی معمولاً علاقه‌مند هستیم که دستورات به صورت پیوسته تکرار شوند(برای مثال اگر برنامه‌ی مربوط به یک دماسنج را نوشته باشیم می‌خواهیم که دما به طور مداوم اندازه‌گیری شود یا اگر دستور چشمک‌زدن یک LED را نوشته باشیم می‌خواهیم تا زمانی‌که

میکروکنترلر روشن است LED خاموش و روشن شود) پس می‌توانیم دستوراتی که می‌خواهیم مدام مورد بررسی قرار بگیرند را درون حلقه‌ی while(1) بنویسیم:



شکل ۳-۱۱: حلقه‌ی بینهایت-محل نوشتن دستورات

اکنون می‌خواهیم برنامه‌ای بنویسیم که LED متصل به پایه‌ی PC5 یک‌ثانیه روشن و یک‌ثانیه خاموش باشد. همانطورکه می‌دانید میکروکنترلر یک قطعه‌ی دیجیتال است و قطعات منطقی یا دیجیتال فقط دارای دوحالت در خروجی خود هستند(یک و یا صفر). یک به معنای آن است که در خروجی ولتاژ داریم و صفر به معنای آن است که در خروجی ولتاژ نداریم. ولتاژ خروجی بستگی به تغذیه‌ی میکروکنترلر دارد، یعنی اگر تغذیه‌ی میکروکنترلر را به ولتاژ ۵ولت متصل‌کرده باشیم یک بودن پایه به معنای آن است که بر روی خروجی ولتاژ ۵ولت قرار دارد(ولتاژ تغذیه‌ی میکروکنترلرها می‌تواند در مدل‌های مختلف متفاوت باشد. برای مثال در مدل معمولی میکروکنترلرهای ATmega ولتاژ تغذیه باید بین ۴,۵ تا ۵ ولت باشد و نمی‌تواند مقادیر بیشتر یا کمتر باشد ولی برای مثال در مدل ATmega16L می‌تواند تغذیه‌ی میکروکنترلر ولتاژی بین ۲,۷ تا ۵ ولت باشد آنگاه عبارت یک بودن خروجی دیگر به معنای ۵ ولت بودن نیست و عددی متناسب با ولتاژ تغذیه‌ی میکروکنترلر است. چون در این کتاب ولتاژ تغذیه را برابر ۵ ولت در نظر گرفته‌ایم، یک بودن خروجی به معنای آن است که ولتاژ خروجی ۵ولت است). برای یک شدن پایه‌ی PC5 از دستور PORTC.5=1 و برای صفر شدن آن از دستور PORTC.5=0 استفاده می‌کنیم و برای ایجاد تاخیرهای یک‌ثانیه‌ای بین آنها از دستور delay_ms(1000) استفاده می‌کنیم که به مقدار ۱۰۰۰ میلی‌ثانیه یا ۱ثانیه تاخیر ایجاد می‌کند. همچنین دستور دیگر این تابع که در کتابخانه‌ی delay.h موجود می‌باشد، دستور delay_us() است که به اندازه میکروثانیه می‌تواند تاخیر ایجاد

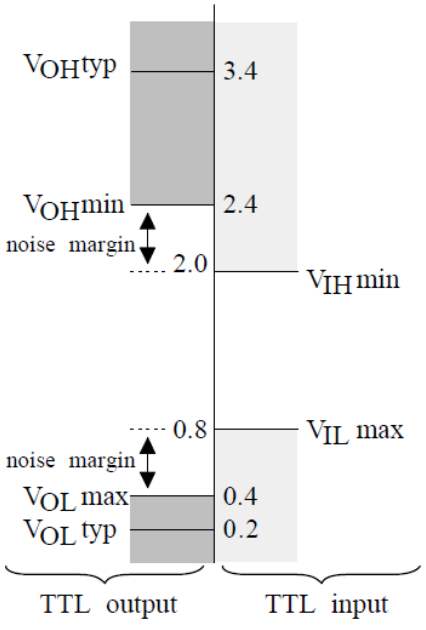
کند. برای مثال delay_us(834) به مقدار ۸۳۴ میکروثانیه در برنامه تاخیر ایجاد می‌کند. اکنون کد نوشته شده را داخل حلقه‌ی بی‌نهایت مشاهده می‌کنیم:

```
143
144     while (1)
145           {
146           PORTC.5=1;
147           delay_ms(1000);
148           PORTC.5=0;
149           delay_ms(1000);
150           }
```

حال لازم است دو نکته را مورد توجه قرار دهیم:

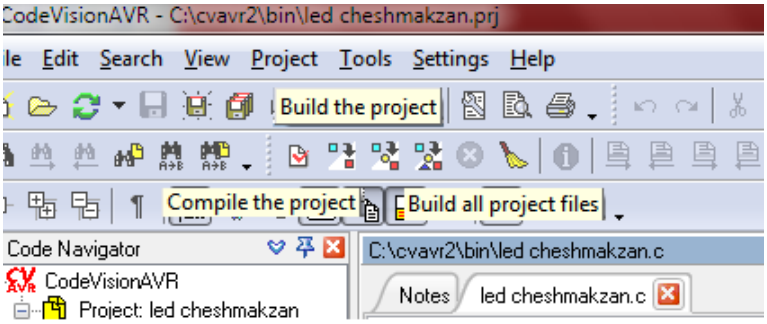
نکته‌ی۱: در بخش کدنویسی نرم‌افزار کدویژن همیشه نام‌ها را با حروف بزرگ و دستورها را با حروف کوچک می‌نویسیم. برای مثال PORTA.0 نام یک ریجستر است(دستور نیست)، پس برای مقداردهی به آن باید از حروف بزرگ استفاده کنیم. مانند PINB.0=1 یا PORTA.0=1 ولی برای نوشتن دستور تاخیر، آن را با حروف کوچک می‌نویسیم delay_ms(1000).

نکته‌ی۲: میکروکنترلر با سطح منطقی TTL کار می‌کند و در این سطح منطقی ولتاژهای ورودی پایه‌ها اگر بین ۰ تا ۰٫۴ولت باشند صفر منطقی حساب می‌شوند و اگر ۲٫۴ ولت به بالا باشند ۱ منطقی به حساب می‌آیند. به شکل روبرو دقت کنید:



شکل ۳-۱۲: سطوح منطقی TTL

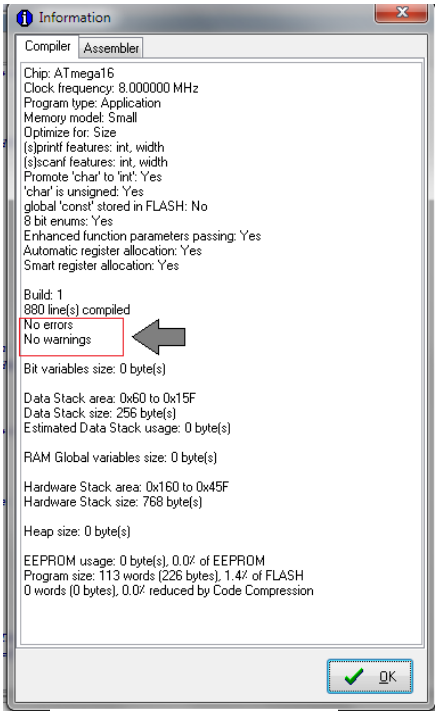
بعد از تمام شدن کدنویسی باید برنامه را کامپایل(Compile) کنیم. برای این‌کار مطابق شکل گزینه‌ی build and compile project را از نوار بالا انتخاب می‌کنیم( ):



شکل ۳-۱۳ : کامپایل کردن برنامه

گزینه‌ی Compile the Project: این گزینه برنامه را بررسی‌می‌کند که خطا و هشداری در آن وجود نداشته باشد و عاری از ایراد باشد.

گزینه‌ی Build the project: این گزینه برنامه‌ی نوشته‌شده را در قسمتی که ذخیره کردیم دوباره ذخیره می‌کند.



شکل ۳-۱۴

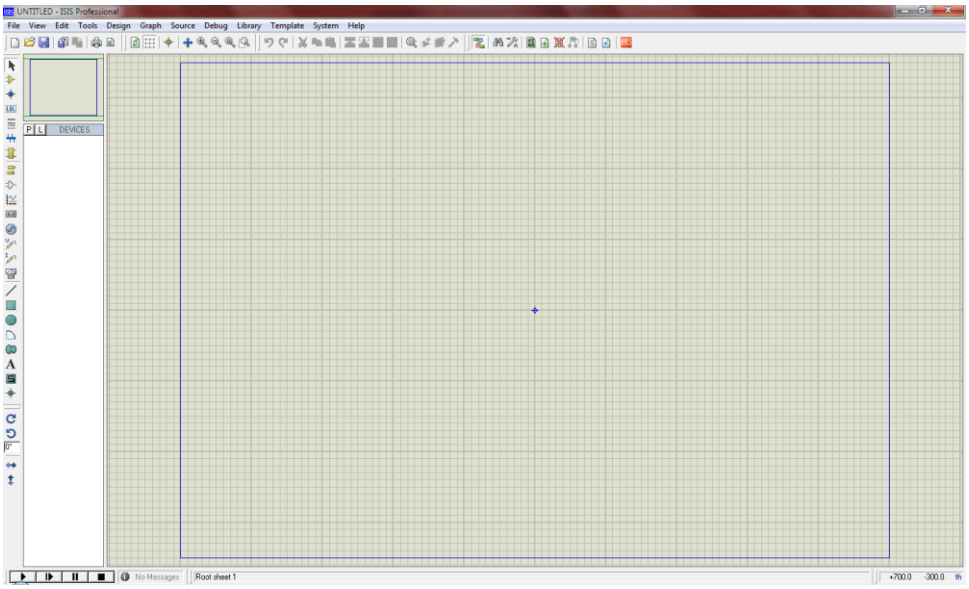
گزینه‌ی Build all project files: این گزینه هم کار گزینه‌ی Compile the Project the و هم کار گزینه‌ی Build the project را انجام می‌دهد. پس بهترین گزینه‌ی در دیباگ کردن(debug) پروژه‌ها همین گزینه می‌باشد.

بعد از انتخاب گزینه‌ی build and compile project با پنجره‌ی روبرو مواجه می‌شویم که این پنجره به ما نشان می‌دهد برنامه درست کامپایل شده و اینکه چه تعداد error(خطا) و warning(هشدار) داریم که همانطورکه در شکل قبل مشاهده می‌کنید، در این برنامه هیچ خطا و

هشداری وجود ندارد.گزینه‌ی Ok را انتخاب می‌کنیم و بدین ترتیب کار کدنویسی این پروژه تمام می‌شود.

آشنایی اولیه با محیط پروتئوس

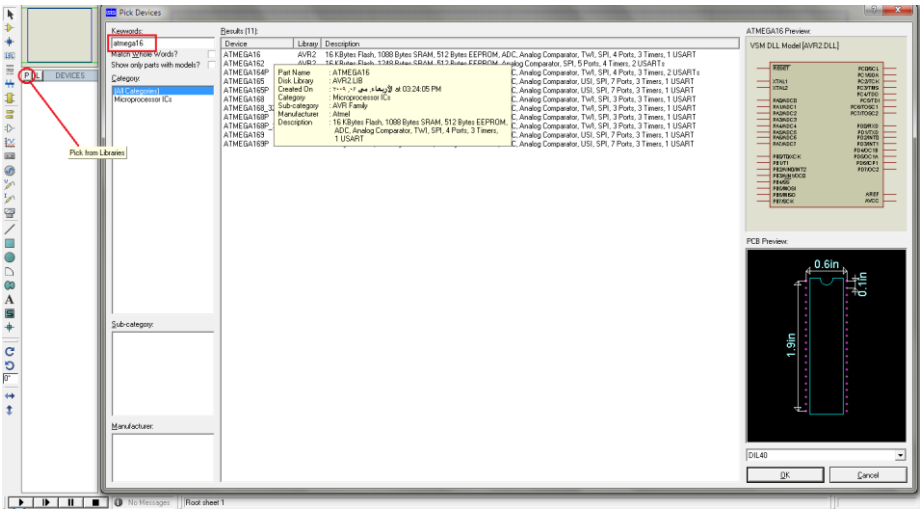
در این بخش می‌خواهیم کد نوشته‌شده را بر روی میکروکنترلر بریزیم و نتیجه را به صورت شبیه‌سازی مشاهده کنیم. برای این کار می‌توانیم از نرم‌افزار پروتئوس PROTEUS(ISIS) استفاده‌کنیم. وقتی این برنامه را باز می‌کنیم با محیط زیر روبرو می‌شویم:



شکل ۳-۱۵ : محیط نرم‌افزار پروتئوس

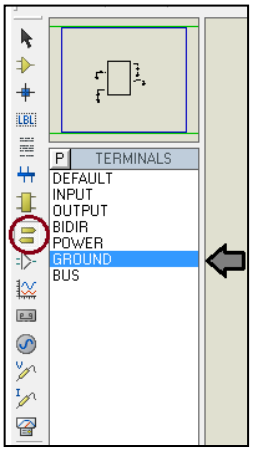
در نرم افزار PROTEUS(ISIS) از قسمت Pick from libraries که به اختصار P نوشته‌شده‌است، قطعات مورد نظرمان را انتخاب می‌کنیم. در این پروژه به یک عدد ATmega16 و یک عددLED نیاز داریم، پس ابتدا روی P کلیک می‌کنیم و در پنجره‌ای که باز می‌شود اسامی

قطعات مورد نظرمان را به ترتیب atmega16,LED-Red می‌نویسیم و بر روی آن‌ها دبل کلیک می‌کنیم:



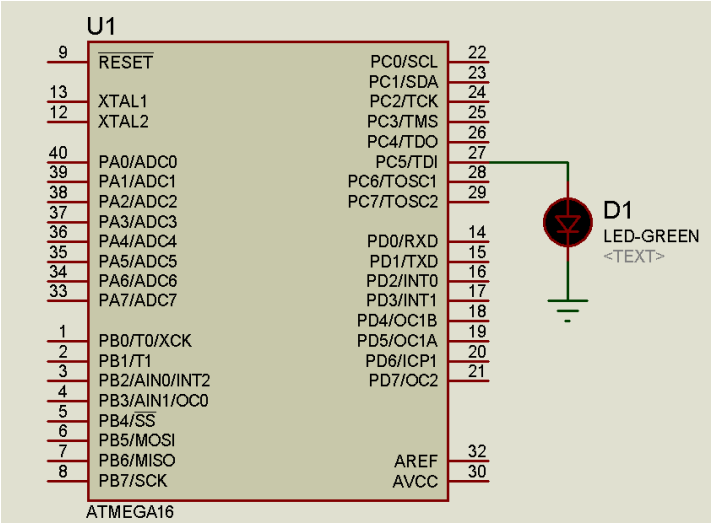
شکل ۳-۱۶ : انتخاب قطعه

قطعاتی که انتخاب‌کردیم را در صفحه‌ی شبیه‌سازی قرار می‌دهیم. در شبیه‌سازی علاوه بر دو قطعه‌ی LED و ATmega16 به یک زمین(GND) هم نیز نیاز داریم زیرا که ولتاژ ۵ولت یا به عبارت بهتر سطح منطقی یک توسط میکروکنترلر به پایه‌ی مثبت LED متصل می‌شود و باید پایه‌ی منفی LED را به سطح منطقی صفر متصل کنیم تا LED روشن شود. بنابراین مطابق شکل صفحه بعد قطعات را به هم متصل می‌کنیم. در محیط شبیه‌سازی پروتئوس نیازی به تغذیه‌ی میکروکنترلر نیست و خود شبیه‌ساز به صورت پیش‌فرض ولتاژ ۵ ولت را به عنوان تغذیه‌ی میکروکنترلر در نظر می‌گیرد. زمین یا همان صفر منطقی را مطابق شکل روبرو انتخاب می‌کنیم:



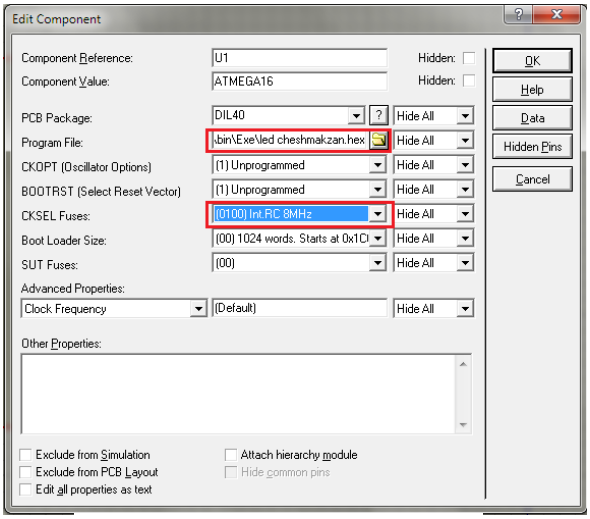
شکل ۳-۱۷ : انتخاب زمین

مدار بسته شده به صورت شکل زیر می‌باشد:



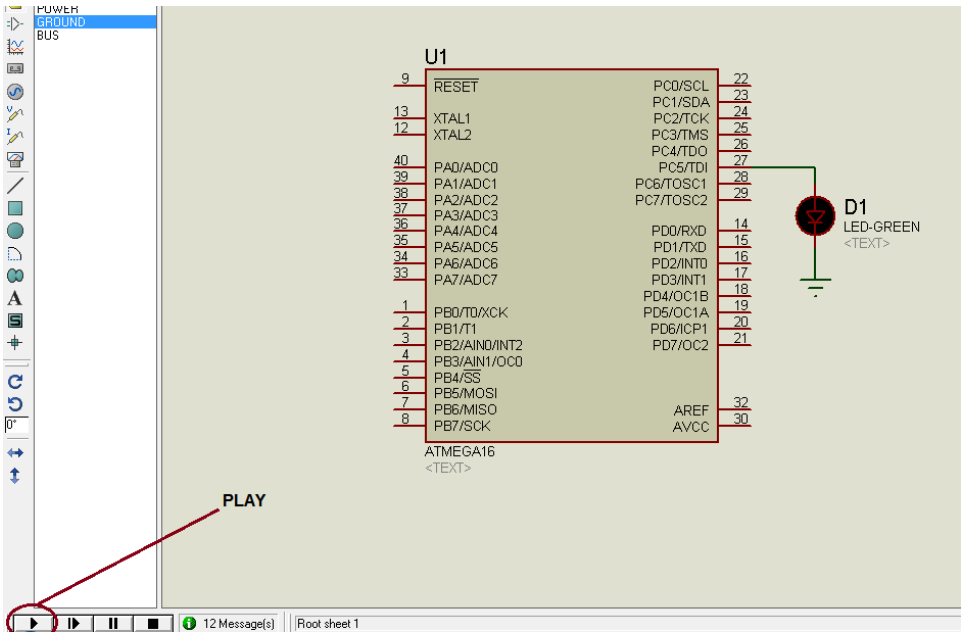
می‌خواهیم برنامه‌ی نوشته شده را بر روی این میکروکنترلر بریزیم. برای این‌کار روی میکروکنترلر دبل‌کلیک می‌کنیم که پنجره‌ای مطابق شکل ۳-۱۹ باز می‌شود. در قسمت ۱ برنامه‌ی نوشته شده را با پسوند HEX در جایی که ذخیره کرده‌ایم Load می‌کنیم.[اگر کدویژن را در درایو c نصب کرده باشید فایل Hex را می‌توانید از مسیر:

Exe → bin → cvavr2 → c انتخاب کنید] و در قسمت ۲ هم فرکانس کاری میکروکنترلر را بر روی ۸ مگاهرتز می‌گذاریم (همان مقدار فرکانس کاری که در کدویزارد تنظیم کردیم):



شکل ۳-۱۹: انتخاب فایل برنامه‌نویسی‌شده hex

اکنون آماده‌ایم نتیجه‌ی شبیه‌سازی را مشاهده کنیم. برای این‌کار باید برنامه را از قسمت پایین سمت چپ اجرا (Play) کنیم. این کار را انجام داده و نتیجه را مشاهده می‌کنیم (که LED یک ثانیه روشن و یک ثانیه خاموش است):



شکل ۳-۲۰-شبیه‌سازی مدار بسته‌شده

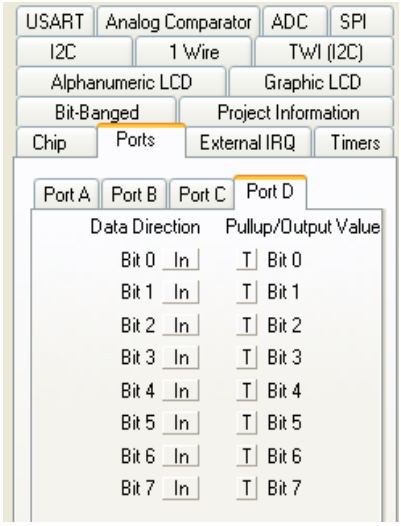
با انجام این مثال کمی با نحوه‌ی ایجاد پروژه آشنا شدیم. در ادامه ۳ مثال دیگر وجود دارد که شما را به تسلط نسبی در بخش PIN & PORT می‌رساند.

مثال۲: برنامه‌ای بنویسید که اگر پایه‌ی PA2 برابر صفر (صفر ولت) شد پایه‌ی PB5 روشن شود (یک شود) و اگر پایه‌ی PD4 برابر صفر شد پایه‌ی PC1 دوثانیه یکبار صفر و یک بشود:

در این مثال دوباره باید یک پروژه جدید ایجاد کنیم و مطابق مراحل گفته شده در مثال قبل این کار را انجام دهیم و پس از آن به سراغ تنظیمات کدویزارد می‌رویم. مطابق مثال قبل اول از همه باید نوع قطعه (Chip)را انتخاب کنیم. قطعه‌ای که با آن کار می‌کنیم همان ATmega16

می‌باشد، بعد از انتخاب قطعه، فرکانس کاری را روی ۸ مگاهرتز تنظیم می‌کنیم. پس از انجام تنظیمات مقدماتی به سراغ تنظیمات ورودی خروجی‌ها (Pin_Port) می‌رویم. در بخش Ports ابتدا در PORTA بیت ۲ را به صورت In (ورودی) تنظیم می‌کنیم و حالت آن را پیش‌فرض به صورت P(Pull-up) می‌گذاریم که به این معناست که این پایه همیشه یک (۵ ولت) را دریافت می‌کند مگر آنکه آن را زمین کنیم. حالت دیگر این تنظیمات حالت T(3 States) یا حالت(High-Impedance) می‌باشد یعنی اینکه این پایه پذیرای ۳ حالت می‌باشد، اگر به این پایه ۵ ولت بدهیم برابر یک می‌شود، اگر آن را زمین کنیم برابر صفر می‌شود و اگر آن را به هیچ ولتاژی وصل نکنیم (یعنی به صورت مدار باز باشد) می‌تواند در هر لحظه ولتاژ متفاوتی بنابر نویز موجود در هوا داشته باشد. در این بخش با Pull-up مقاومتی، و نحوه‌ی بستن آن در پروتئوس آشنا می‌شویم.

در ادامه‌ی تنظیمات در تب مربوط به PORTB بیت ۵ را به صورت Out (خروجی) تنظیم می‌کنیم و مقدار اولیه‌ی آن را صفر می‌گذاریم. بعد از این در تب مربوط به PORTC بیت ۱ را به صورت Out (خروجی) تنظیم می‌کنیم و مقدار اولیه‌ی این پایه را هم به صورت دلخواه تنظیم می‌کنیم که در این مثال صفر انتخاب نموده‌ایم. در آخر هم در بخش PORTD بیت ۴ را به صورت ورودی (In) می‌گذاریم و مقدار اولیه‌ی آن را T تنظیم می‌کنیم:

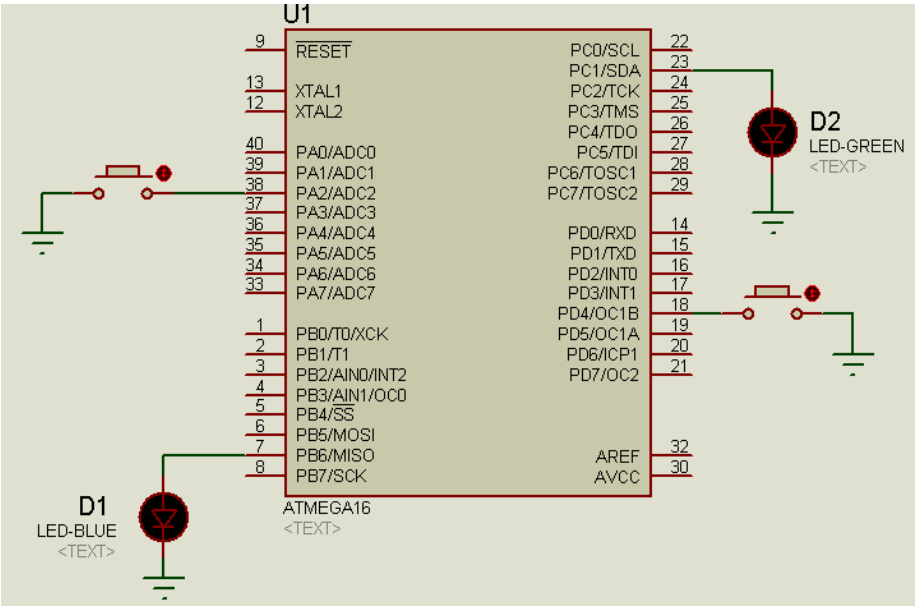


شکل ۳-۲۱ : تنظیمات درگاه D

حال باید کد را دوباره Generate کنیم و در ۳ مرحله آن را ذخیره کنیم. پس از انجام تنظیمات اولیه به سراغ بخش کدنویسی می‌رویم. فقط دوباره یادمان باشد که کتابخانه‌ی delay.h را به برنامه اضافه کنیم. در ادامه کد نوشته شده در بخش While(1) را مشاهده می‌کنیم. تنها دستور برنامه‌نویسی که نسبت به برنامه‌ی LED چشمک‌زن قبلی اضافه شده‌است، دستور PINx.x است که برای کار با ورودی می‌باشد.

```
130
131   while (1)
132         {
133         if(PINA.2==0)
134         PORTB.5=1;
135
136         if(PIND.4==1){
137         PORTC.1=0;
138         delay_ms(2000);
139         PORTC.1=1;
140         delay_ms(2000);
141         }
142
143         }
```

پس از اتمام کدنویسی، کد نوشته شده را کامپایل کرده و مشاهده می‌کنیم هیچ خطا و هشداری در برنامه‌ی نوشته شده نداریم. در شکل زیر شبیه‌سازی برنامه را در پروتئوس مشاهده می‌کنید، فقط نسبت به مثال قبل از یک کلید استفاده کرده‌ایم [(Push button) (برای پیدا کردن کلید در قسمت انتخاب قطعات بنویسید button]



مثال ۳: برنامه‌ای بنویسید که پایه‌ی PD1 به صورت ورودی و T تنظیم شود و مقاومت Pull-Up را بر روی این پایه بسته و زمانیکه این پایه ۱ باشد، LED متصل به PB3 روشن و اگر پایه PD1 صفر شد LED خاموش شود(هدف آشنایی با مقاومت Pull-Up)

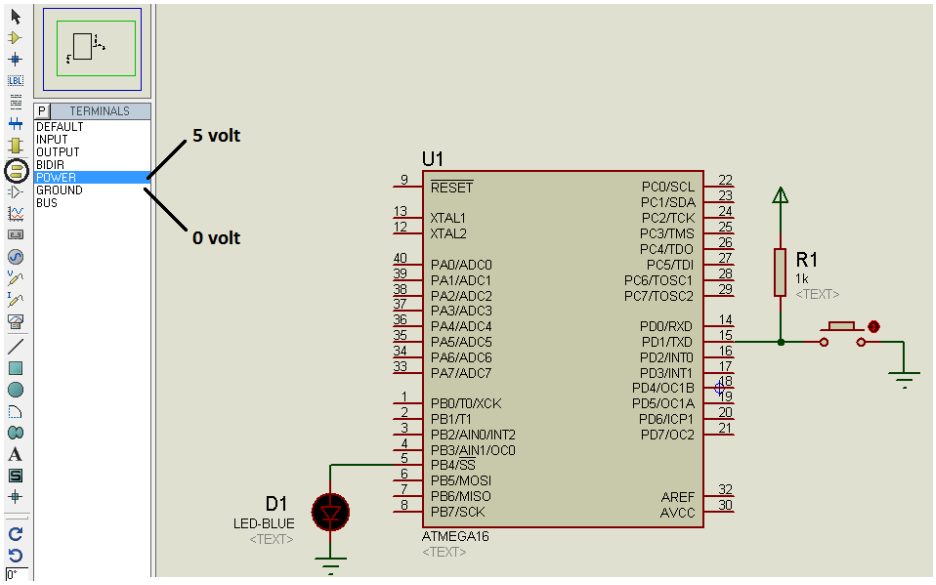
تنظیمات کدویزارد:

بیت اول PORTD به صورت In با حالت T و بیت سوم PORTB به صورت Out با مقدار اولیه‌ی صفر تنظیم می‌شود.

کد نوشته شده در حلقه‌ی بی نهایت را مشاهده می‌کنید:

```
while (1)
      {
      if(PIND.1==0)
      PORTB.3=0;
      else
      PORTB.3=1;
      }
```

حال برای شبیه‌سازی، برنامه‌ی پروتئوس را باز می‌کنیم. به یک منبع ۵ولت نیاز داریم که



شکل ۳-۲۳

می‌توان از همان جایی که زمین مدار را در مثال ۱ انتخاب کردیم منبع ۵ ولت را با نام POWER (⇧)پیدا کنیم.

نحوه‌ی کارکرد مقاومت Pull-Up به این صورت است که اگر کلید باز باشد (قطع باشد)، پایه PD1 با مقاومت به منبع متصل می‌شود و چون جریانی از این مقاومت عبور نمی‌کند پس ولتاژ پایه‌ی PD1 بدون افت ولتاژ برابر ۵ ولت می‌باشد و زمانی‌که کلید را وصل می‌کنیم پایه‌ی PD1 دو انتخاب دارد: ۱- اینکه با مقاومت به منبع (۵ ولت) برود ۲- با سیم(اتصال کوتاه) به زمین برود، که طبق اصل اتصال‌کوتاه(انتخاب مسیری با مقاومت کمتر) ولتاژ این پایه برابر ولتاژ زمین (صفر ولت) می‌شود. پس به این ترتیب ولتاژ PD1 در حالت عادی برابر ۵ ولت است و اگر کلید را فشار دهیم ولتاژ این پایه برابر صفر ولت می‌شود(دقت کنید که در این حالت ولتاژ این پایه یا ۵ ولت و یا صفر ولت است و دیگر سه حالت نیست و امکان ندارد بنا به نویز موجود در هوا تغییر کند.)

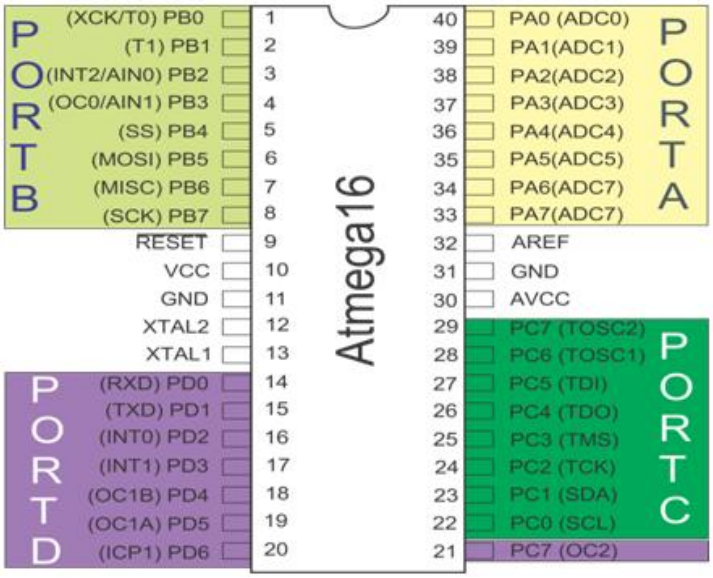
یک‌گام‌فراتر PIN & PORT

در این بخش می‌خواهیم کمی بیشتر از واحد ورودی و خروجی‌ها در میکروکنترلر بدانیم و نحوه‌ی کار مستقیم با رجیسترهای ورودی و خروجی را یاد بگیریم، در ادامه خواهید دید که کار با این رجیسترها بسیار ساده می‌باشد.

پیش از هر چیز باید بگوییم که رجیستر چیست؟ رجیسترها نوعی حافظه هستند که به طور مستقیم با بخش پردازشگر میکروکنترلر در ارتباط هستند. در میکروکنترلرهای AVR هر رجیستر یک بایت یا ۸ بیت است. یکی از ویژگی‌های رجیسترها این است که به خاطر ارتباط نزدیک با پردازنده، سرعت بسیار بالاتری نسبت به سایر خانه‌های حافظه دارند. حال به زبان ساده‌تر اگر بخواهیم رجیسترهای ورودی و خروجی که در این بخش مورد نیاز است را بشناسیم کافیست که از طریق یک مثال قیاسی آن را یاد بگیریم.

فرض‌کنید میکروکنترلرهای ATmega16 همانند یک قلعه هستند و هر قلعه‌ای دارای چندین دروازه است. در این قلعه اسامی دروازه‌ها A,B,C,D هستند و از طریق این ۴ ورودی می‌توان رفت

و آمدها را کنترل و بررسی‌کرد. فرض‌کنید در این قلعه افراد به سه بخش غنی و فقیر و بی‌هویت تقسیم می‌شوند، افراد ثروتمند را معادل ولتاژ ۵ولت، افراد کم بضاعت را معادل ولتاژ صفر ولت و افراد بی‌هویت هم کسانی هستند که می‌توانند خود را در هر لحظه به صورت غنی یا فقیر نشان‌دهند و به نوعی در میکروکنترلرها معادل نویز هستند که ما آن را هیچی درنظر می‌گیریم(مشخص نیستند که صفر هستند یا یک و در لحظه متغیرند). حال اول از همه این دروازه‌ها را مطابق شکل زیر مشاهده می‌کنید:



شکل ۳-۲۴

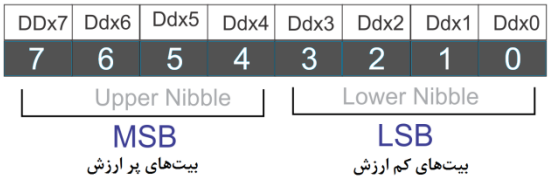
رجیستر DDRx

از هر دروازه‌ای افراد وارد و یا خارج می‌شوند، خارج‌شدن را در میکروکنترلر معادل ۱ و واردشدن را معادل صفر در نظر بگیرید. در میکروکنترلر رجیستری به نام DDRx وجود دارد که حالت ورود یا خروج را مشخص می‌کند و جای x هم هر کدام از ۴ تا درگاه A,B,C,D قرار می‌گیرند، یعنی اینکه افراد (سیگنال‌ها) از این درگاه‌ها خارج یا وارد می‌شوند. اگر هرکدام از این ۸ بیت مورد نظر

رجیستر DDRx صفر باشند، یعنی این درگاه به صورت ورودی و هر کدام ۱ باشند یعنی به صورت خروجی می‌باشد، در دو شکل بعد این رجیستر را مشاهده می‌کنید:

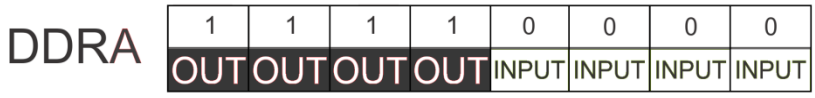
DDRx0	DDRx1	DDRx2	DDRx3	DDRx4	DDRx5	DDRx6	DDRx7

شکل ۳- ۲۵ : رجیستر DDRx



شکل ۳-۲۶

برای مثال اگر این رجیستر برای درگاه A به شکل زیر باشد می‌بینید که کدام پایه‌ها ورودی و کدام پایه‌ها خروجی می‌باشند:



شکل ۳-۲۷

رجیستر PORTx

حال فرض‌کنید افرادی که از قلعه می‌خواهند خارج شوند تنها افراد ثروتمند هستند یا افراد کم بضاعت(۵ولت یا صفر ولت) و افراد بی‌هویت هم هیچگاه از قلعه خارج نمی‌شوند. این حرف به این معناست که میکروکنترلر همیشه خروجی پایه‌هایش یا به صورت صفر ولت می‌باشد یا ۵ ولت و حالت سومی وجود ندارد. برای تعیین این حالت گفته شده در میکروکنترلر رجیستری به نام PORTx وجود دارد که جای x می‌تواند هر کدام از ۴ درگاه A,B,C,D قرار بگیرند. این رجیستر را در شکل زیر می‌بینید:

PORTx0	PORTx1	PORTx2	PORTx3	PORTx4	PORTx5	PORTx6	PORTx7

شکل ۳-۲۸: رجیستر PORTx

صفر شدن هرکدام از بیت‌های این رجیستر به معنای صفر ولت خروجی و یک شدن به معنای ۵ ولت خروجی می‌باشد.

رجیستر PINx

حال فرض‌کنید در این قلعه پیش‌فرض نگهبانان قلعه این است که یا افراد بی‌هویت وارد قلعه می‌شوند یا افراد ثروتمند که این پیش‌فرض در میکروکنترلر معادل رجیستر PINx می‌باشد که جای x هم می‌تواند هر کدام از ۴ درگاه A,B,C,D قرار بگیرد. در شکل زیر این رجیستر را مشاهده می‌کنید:

PINx0	PINx1	PINx2	PINx3	PINx4	PINx5	PINx6	PINx7

نکته: بایستی توجه کرد که در آدرس‌دهی رجیسترها در کدویژن می‌توان به ۳ صورت یعنی توسط اعداد باینری،دسیمال و هگزادسیمال این کار را انجام داد که در ادامه در مثال‌ها با نحوه‌ی انجام این کار بیشتر آشنا می‌شویم.

صفر شدن هر کدام از این ۸ بیت یعنی اینکه نگهبان این پیش‌فرض را دارد که افراد بی‌هویت در حال ورود هستند (همان حالت T(امپدانس بالا)) و یک شدن یعنی اینکه پیش‌فرض نگهبان این است که افراد ثروتمند وارد می‌شوند (همان حالت P (Pull-Up)). پیش‌فرض نگهبان‌ها به همین صورت باقی می‌ماند مگر اینکه افراد ورودی ماهیت اصلی خود را اثبات‌کنند، که این کار را اگر به یاد داشته باشید مثلاً با Pull-Up کردن یا زمین کردن انجام می‌دادیم. حال برای اینکه سردرگم نشوید هر آنچه تا الان گفتیم را خلاصه می‌کنیم.

میکروکنترلر یا می‌خواند یا می‌نویسد (read /write). نوشتن مانند خارج شدن است، یعنی اینکه یک چیزی را به محیط پیرامون اعمال می‌کند و خواندن مانند وارد شدن است، یعنی اینکه یک چیزی را از محیط پیرامون دریافت و تشخیص می‌دهد. رجیسترهای DDRx و PORTx هم

می‌توانند بخوانند و هم بنویسند و رجیستر PINx تنها می‌تواند بخواند. در جدول زیر آنچه گفته شد را در قالب رجیسترهای گفته شده می‌بینید:

Register	Type	Description	Notes
DDRA	Read/Write	Port B Data Direction Register	1 = output, 0 = input
PORTA	Read/Write	Port B Data Register	
PINA	Read only	Port B Input Register	
DDRB	Read/Write	Port B Data Direction Register	1 = output, 0 = input
PORTB	Read/Write	Port B Data Register	
PINB	Read only	Port B Input Register	
DDRC	Read/Write	Port C Data Direction Register	1 = output, 0 = input
PORTC	Read/Write	Port C Data Register	
PINC	Read only	Port C Input Register	
DDRD	Read/Write	Port D Data Direction Register	1 = output, 0 = input
PORTD	DBO to DB7	Port D Data Register	
PIND	DBO to DB7	Port D Input Register	

جدول ۳-۱

کل مباحثی که گفته شد بدین معناست که با دوخط دستور زیر در کدویژن می‌توان رجیسترها را آدرس‌دهی کرد:

PORTx=0xXX OR PORTx=0bXXXXXXXX

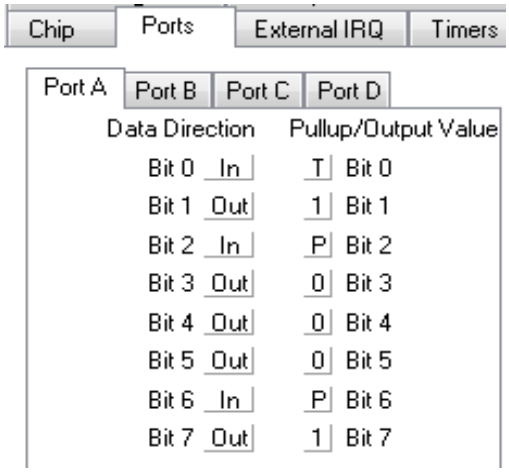
DDRx=0xXX OR DDRx=0bXXXXXXXX

که اولین دستور یعنی PORTx=0xXX بدین گونه عمل می‌کند که بیت‌هایی که در تنظیماتی که در کدویزارد انجام می‌دادیم به صورت P(Pull-Up) یا 1 هستند و در بیت‌های رجیستر به صورت ۱ می‌باشند و بیت‌هایی که در تنظیمات کدویزارد به صورت T یا 0 هستند، در بیت‌های این رجیستر به صورت صفر می‌باشند و جای x بعد از PORT هم هر کدام از ۴ درگاه A,B,C,D

قرار می‌گیرند. این را هم اضافه کنیم که 0x نماد اعداد هگزادسیمال و 0b نماد اعداد باینری است (اگر با اعداد باینری و هگزادسیمال آشنا نیستید به پیوست آخر کتاب مراجعه‌کنید).

به مثال زیر برای روشن‌تر شدن مطلب دقت‌کنید:

مثال ۴: فرض‌کنید تنظیمات رجیستر A به صورت شکل زیر در کدویزارد انجام شده است. حال می‌خواهیم که این تنظیمات را خود به صورت مستقیم و به وسیله‌ی کار با رجیستر و بدون کار با کدویزارد انجام بدهیم:



شکل ۳-۳۰: تنظیمات درگاه A

شکل رجیسترها:

Input or output	out	In	Out	Out	Out	In	Out	In
DDRA	1	0	1	1	1	0	1	0

State	1	P	0	0	0	P	1	T
PORTA	1	1	0	0	0	1	1	0

معادل هگزادسیمال جدول قبل به همراه مدل باینری آن:

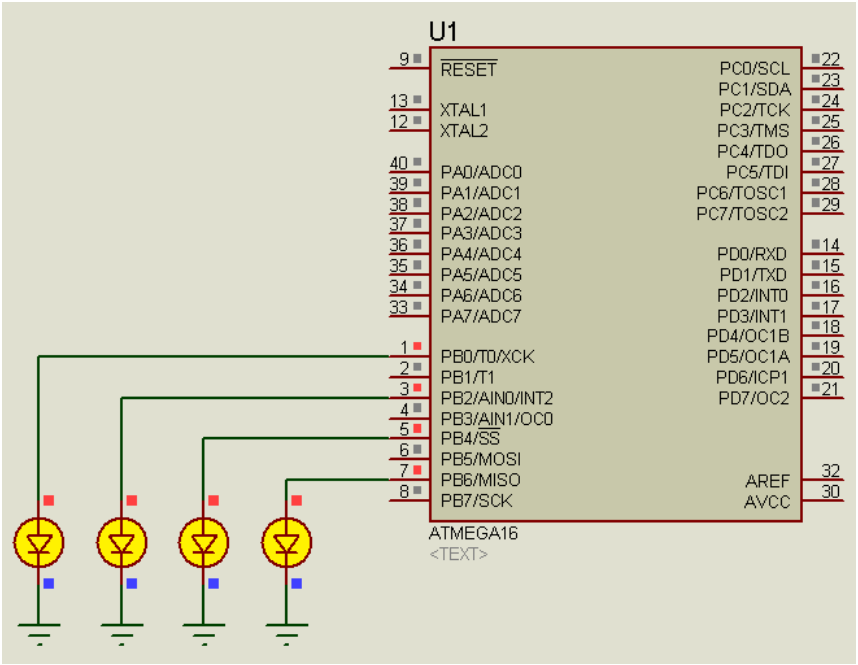
PORTA=0xC6 OR PORTx=0b11000110

DDRA=0xBA OR DDRx=0b10111010

کدی که نوشته می‌شود:

```
// Input/Output Ports initialization
// Port A initialization
// Func7=Out Func6=In Func5=Out Func4=Out Func3=Out Func2=In Func1=Out Func0=In
// State7=1 State6=P State5=0 State4=0 State3=0 State2=P State1=1 State0=T
PORTA=0xC6;
DDRA=0xBA;
```

مثال۵: می‌خواهیم با یک‌خط کد در while(1) برنامه‌ای بنویسیم که پایه‌های PORTB را به صورت شکل زیر روشن شوند(پورت‌های B0,B2,B4,B6 به صورت خروجی تنظیم و LEDهای متصل به خود را روشن کنند):

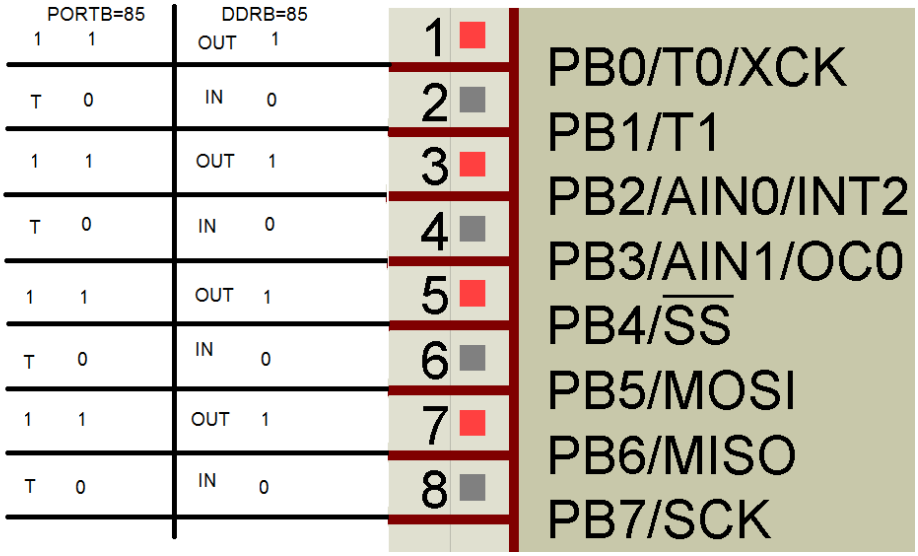


شکل ۳-۳۱

کد زیر را درون حلقه‌ی while(1) می‌نویسیم:

```
while (1)
      {
      DDRB=85;
      PORTB=85;
      }
```

در کد فوق DDRB=85 و PORTB=85 مقداردهی شده است که عدد ۸۵ در مبنای ۱۰ می‌باشد و این عدد در مبنای دو به صورت **1010101** می‌شود و مفهوم این اعداد در PORTB به صورت شکل زیر می‌باشد:



شکل ۳-۳۲

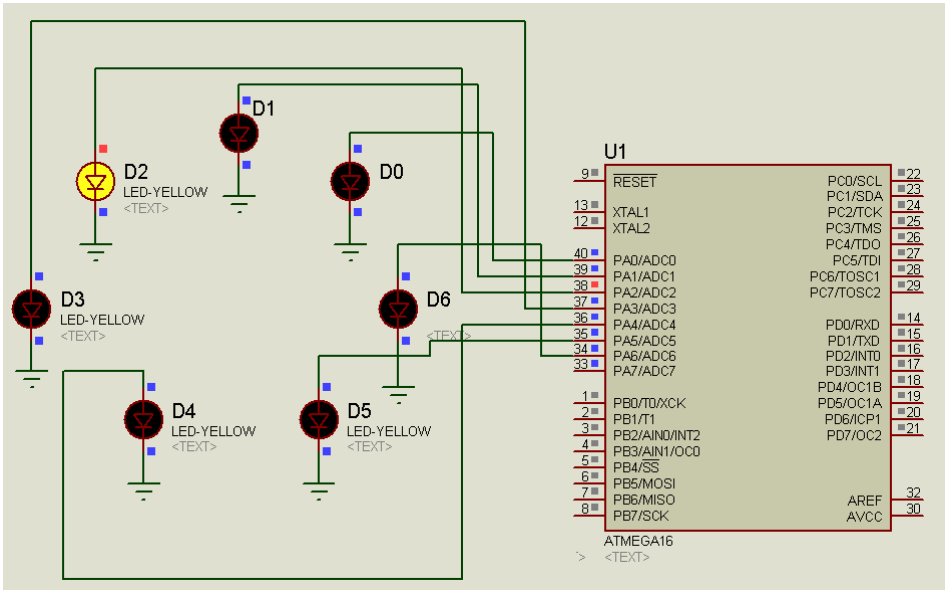
اکنون به آخرین مثال از این فصل می‌رسیم، این مثال را به دقت دنبال کنید.

مثال۶: برنامه‌ای بنویسید که ۷ عدد LED را به ترتیب و با فاصله‌ی زمانی ۵۰۰ میلی ثانیه روشن کند و با روشن شدن هر LED ، LED قبلی خاموش شود. (اگر این LED ها به شکل دایره چیده شوند جهت چرخش پادساعتگرد باشد)

همانطور که در توضیحات بالا آموختیم دستور PORTA=0 به معنی خاموش بودن تمام پایه‌های پورت A و برای مثال دستور PORTA=8 به معنی آن است که فقط چهارمین پایه از پورت A (یعنی PORTA.3) روشن باشد(زیرا عدد ۸ در مبنای ۲ برابر ۱۰۰۰ می‌باشد، یعنی فقط چهارمین پایه ۱ است). حال اگر بخواهیم ابتدا اولین پایه یک باشد، سپس دومین پایه، سپس سومین پایه و . . . باید کدی بنویسیم که ابتدا PORTA=1 باشد سپس PORTA=2 بعد PORTA=4 و . . . یعنی اگر PORTA برابر توان‌های عدد ۲ باشد یعنی در هر لحظه یک پایه از این پورت روشن می‌باشد. برای نوشتن برنامه‌ی مثال ۶ به گونه‌ی زیر عمل می‌کنیم که ابتدا PORTA را برابر عدد یک قرار می‌دهیم و سپس به وسیله‌ی یک حلقه‌ی for آن را ۶ بار در عدد ۲ ضرب می‌کنیم و زمانی که آخرین LED روشن شد، دوباره PORTA را برابر ۱ قرار می‌دهیم که دوباره اولین LED روشن شود:

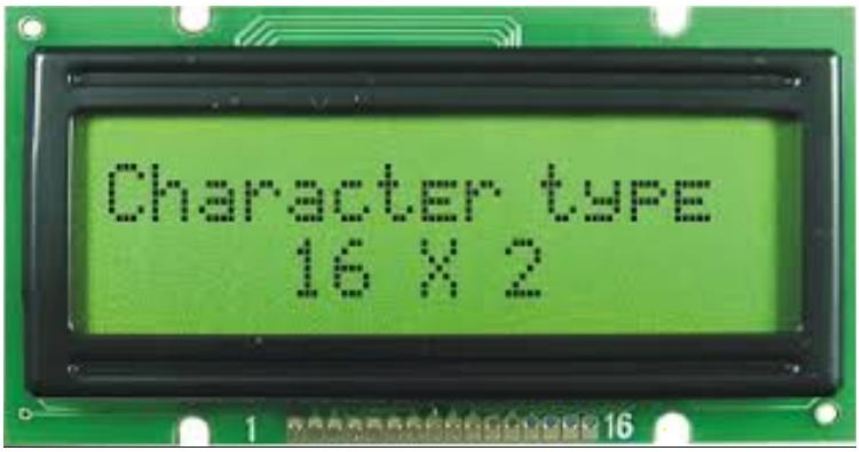
```
PORTA=1;
while (1)
      {
      for(i=0;i<6;i++){
      PORTA*=2;
      if(PORTA>64)
      PORTA=1;
      delay_ms(500);
      }
      }
```

می‌توانیم این برنامه را در پروتئوس به شکل زیر شبیه‌سازی کنیم:



فصل چهارم

ال سی دی های کاراکتری



lcd character

در این فصل خواهیم خواند:

۱. نحوه‌ی اتصال LCD به میکروکنترلرهای ATmega16 و نمایش داده بر روی آن و دستورات کاربردی
۲. تنظیمات بخش Characters/Line
۳. یک گام فراتر(ایجاد یک تصویر دلخواه بر روی ال سی دی‌های کاراکتری)

LCD کاراکتری

در این بخش قصد داریم نحوه‌ی کار با LCDهای کاراکتری ۲*۱۶ را فرا بگیریم. شاید یکی از مهیج‌ترین بخش‌های کار با میکروکنترلرهای AVR برای کسانی که تازه کار با آن را آغاز کرده‌اند بخش LCDهای کاراکتری باشد چرا که به وسیله‌ی آن می‌توانیم اطلاعات کاراکتری مورد نظر خود را بر روی یک صفحه‌ی نمایش LCD نشان دهیم(به‌عنوان یک مثال ساده ساخت دماسنجی که دمای محیط را بر روی یک LCD کاراکتری نمایش می‌دهد).

در این فصل به دو نکته توجه می‌کنیم:

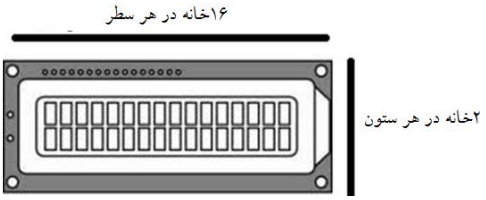
۱. LCDها به دو نوع گرافیکی و کاراکتری تقسیم می‌شوند که در این فصل با نحوه‌ی کار با LCDهای کاراکتری آشنا می‌شویم.

۲.LCDهای کاراکتری به دو صورت سریال و موازی می‌توانند به میکروکنترلرهای AVR متصل شوند که ما در این فصل، LCDها را به صورت سریال به میکروکنترلر متصل می‌کنیم که مزیت آن نسبت به بستن به صورت موازی این است که از پایه‌های کمتری از میکروکنترلر استفاده می‌شود.

LCDها از نظر اندازه متفاوت می‌باشند مثلاً LCDهای کاراکتری ۲*۱۶، ۲*۳۲، ۲*۴۰، ۴*۳۲ و ... در بازار موجود هستند که بر حسب نیاز از هر کدام از آنها استفاده می‌شود. درLCDهای کاراکتری مثلاً(۲*۱۶) عدد اول تعداد ستون‌ها و عدد دوم تعداد سطرها را نشان می‌دهد(۲سطر و ۱۶ ستون). در این فصل با LCDهای کاراکتری ۲*۱۶ آشنا می‌شویم، این مدل از LCDها از ۳۲ خانه تشکیل شده‌اند که در هر خانه‌یک کاراکتر(character) قرار می‌گیرد و این خانه‌ها به صورت ۲ سطر و ۱۶ ستون می‌باشند. به تصاویر زیر دقت کنید:

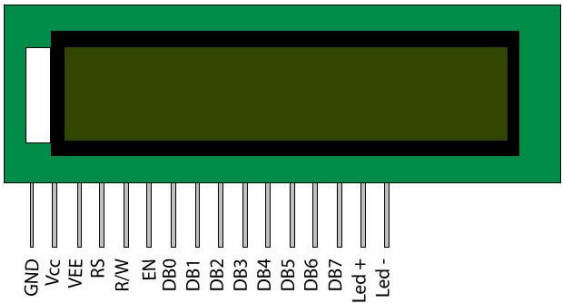
تبصره: در ادامه هر جا از واژه‌ی LCD استفاده شد منظور LCD کاراکتری ۲*۱۶ می‌باشد.

شکل ۴-۱ LCD : کاراکتری



شکل ۴-۲ LCD : کاراکتری، ۲*۱۶

در شکل زیر پایه‌های LCD را مشاهده می‌کنید که در ادامه نحوه‌ی اتصال آن به میکروکنترلرکنترلرهای ATmega16 را توضیح خواهیم داد:



شکل ۴-۳: پایه‌های LCD

در جدول زیر نحوه‌ی کارکرد هر کدام از پایه‌ها را مشاهده می‌کنید:

پایه	ورودی/خروجی	نام پایه	توضیحات
۱	هیچکدام	GND یا Vss	زمین مدار
۲	هیچکدام	Vcc یا Vdd	۵ ولت تغذیه مدار
۳	هیچکدام	VEE	ولتاژ برای نور صفحه LCD
۴	ورودی	RS	اگر RS=0 رجیستر دستور می‌آید اگر RS=1 رجیستر داده می‌آید
۵	هیچکدام	R/W	اگر صفر باشد برای نوشتن اگر یک باشد برای خواندن
۶	ورودی	E	فعال‌ساز (Enable)
۷	ورودی یا خروجی	D0	بیت ۰ داده
۸	ورودی یا خروجی	D1	بیت۱ داده
۹	ورودی یا خروجی	D2	بیت ۲ داده
۱۰	ورودی یا خروجی	D3	بیت ۳ داده
۱۱	ورودی یا خروجی	D4	بیت ۴ داده
۱۲	ورودی یا خروجی	D5	بیت ۵ داده
۱۳	ورودی یا خروجی	D6	بیت ۶ داده
۱۴	ورودی یا خروجی	D7	بیت ۷ داده
۱۵	هیچکدام	Led+	آند LED پشت صفحه LCD
۱۶	هیچکدام	Led-	کاتد LED پشت صفحه LCD

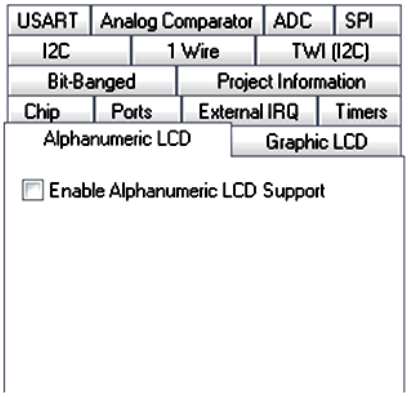
جدول ۴-۱: پایه‌های LCD

نحوه‌ی اتصال LCD به میکروکنترلرهای ATmega16 و نمایش داده بر روی آن

کار را با یک مثال آغاز می‌کنیم:

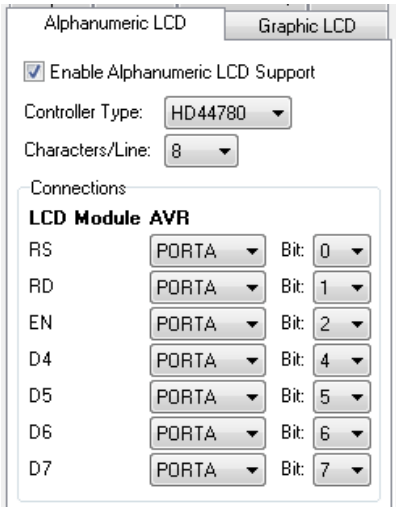
مثال۱: در اولین مثال می‌خواهیم روی LCD عبارت " hello" را نمایش دهیم:

ابتدا یک پروژه‌ی جدید ایجاد می‌کنیم و وارد کدویزارد می‌شویم، سپس به تب مربوط Alphanumeric LCD (LCD الفبایی) می‌رویم:



شکل ۴-۴ : فعال کردن کدویزارد برای LCD کاراکتری

تیک گزینه‌ی Enable Alphanumeric LCD Support را فعال می‌کنیم، پس از انتخاب گزینه، صفحه‌ای مانند شکل زیر باز می‌شود:



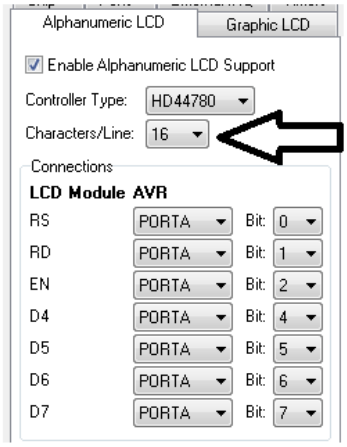
شکل ۴-۵ : تنظیمات کدویزارد

تنظیمات بخش controller Type

در بخش تنظیمات Alphanumeric LCD، گزینه‌ی controller Type مربوط به خانواده‌ای از LCDها است که می‌خواهیم با آن کار کنیم که برگه‌ی مشخصات فنی(datasheet) کامل دوخانواده‌ی HD44780 و KS0073 در CD کتاب آمده است. ما در این بخش با خانواده‌ی HD44780 کار می‌کنیم.

تنظیمات بخش Characters/Line

در بخش Characters/Line اندازه‌ی تعداد کاراکترهای LCD را مشخص می‌کنیم و چون در این بخش قصد کار با LCD های ۲*۱۶ را داریم آن‌را بر روی ۱۶ کاراکتری تنظیم می‌کنیم:



شکل ۴-۶ : تنظیم تعداد ستون‌های LCD

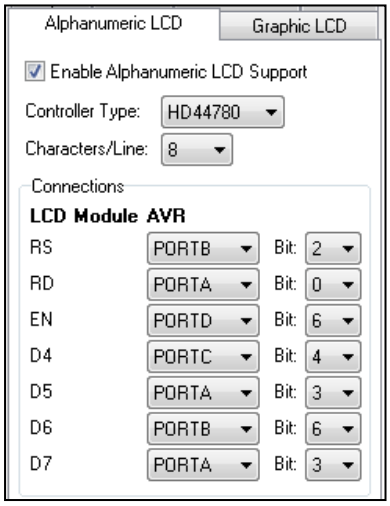
نحوه‌ی اتصال LCD به میکروکنترلر (connections)

در این بخش می‌خواهیم نحوه‌ی اتصالات LCD به پایه‌های میکروکنترلر را نشان دهیم. همانطور که در شکل بالا مشاهده می‌کنید در بخش connections از کاربر سوال می‌شود که پایه‌های RS و D4،E،RD تا D7 به کدام پایه‌های میکروکنترلر متصل شوند. برای مثال فرض کنید می‌خواهیم اتصال پایه‌ها به صورت شکل روبرو باشد .

RS_____PortB.2
RD_____PortA.0
EN_____PortD.6
D4_____PortC.4
D5_____PortA.3
D6_____PortA.6
D7_____PortB.3

شکل ۴-۷ : اتصال پایه‌های LCD در این مثال

در بخش connections نحوه‌ی اتصالات را مانند شکل زیر تنظیم می‌کنیم:



شکل ۴-۸ : تنظیم پایه‌های میکرو برای اتصال به LCD

هدف از اتصال بالا این است که نشان دهیم هیچ محدودیتی در اتصال پایه‌های LCD به میکروکنترلر وجود ندارد و کاربر به هر صورت که بخواهد می‌تواند پایه‌ها را متصل کند.

در مثال ۱ پایه‌ها را مطابق پیش فرض تنظیمات کدویزارد بر روی PORTA قرار می‌دهیم. پس از Generate کردن تنظیمات، در محیط برنامه‌نویسی به بخش While(1) می‌رویم تا کد مربوط به نوشتن عبارت"hello" بر روی LCD را بنویسیم، البته قبل از آن کتابخانه‌ی stdio.h را به برنامه اضافه می‌کنیم چرا که بعضی از توابع مربوط به کار با LCD در این کتابخانه موجود می‌باشد:

```
24  #include <mega16.h>
25
26  // Alphanumeric LCD functions
27  #include <alcd.h>
28
29  #include <stdio.h>
30  // Declare your global variables here
```

اکنون کد زیر را در حلقه‌ی While(1) می‌نویسیم:

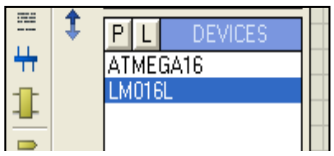
```
147 while (1)
148       {
149
150         lcd_puts("hello");
151         lcd_clear();
152
153       }
```

دستور lcd_Puts ("string") مربوط به نوشتن یک رشته بر روی LCD می‌باشد و دستور lcd_clear() صفحه نمایش LCD را پاک می‌کند.

با این کد ممکن است هنگام شبیه‌سازی مشاهده شود که عبارت نوشته شده بر روی LCD به صورت پرشی نمایش داده می‌شود زیرا در کد بالا عبارت "hello" به صورت مداوم پاک و دوباره نوشته می‌شود. برای آنکه این پرش تصویر را مشاهده نکنیم بهتر است به جای دستور lcd_clear() از دستور lcd_gotoxy(0,0)، استفاده کنیم، این دستور مشخص می‌کند که نوشتن از کدام خانه شروع شود که عدد اول شماره ستون و عدد دوم شماره سطر را مشخص می‌کند پس اگر این دستور را در ابتدای کد بنویسیم هر بار که برنامه به ابتدای حلقه‌ی while(1) می‌رسد از خانه‌ی اول شروع به نوشتن می‌کند و به تبع نوشته‌های جدید جایگزین نوشته‌های قبلی می‌شوند. با این کار دیگر پرش در نوشته دیده نمی‌شود.

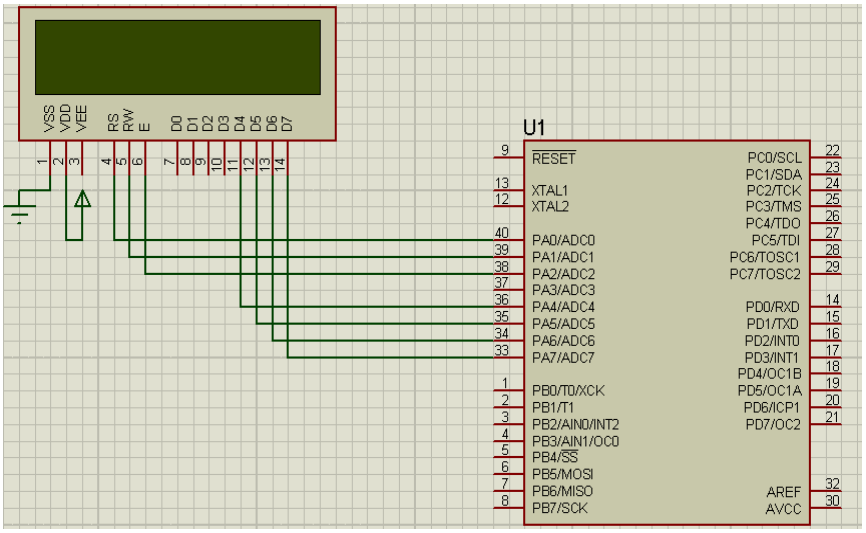
```
147  while (1)
148      {
149          lcd_gotoxy(0,0);
150          lcd_puts("hello");
151
152      }
```

بعد از اتمام کدنویسی به سراغ نرم‌افزار شبیه ساز پروتئوس می‌رویم. در پروتئوس به یک عدد ATmega16( ) و LCD( ) نیاز داریم. LCD را با نام LM016L جستجو می‌کنیم:



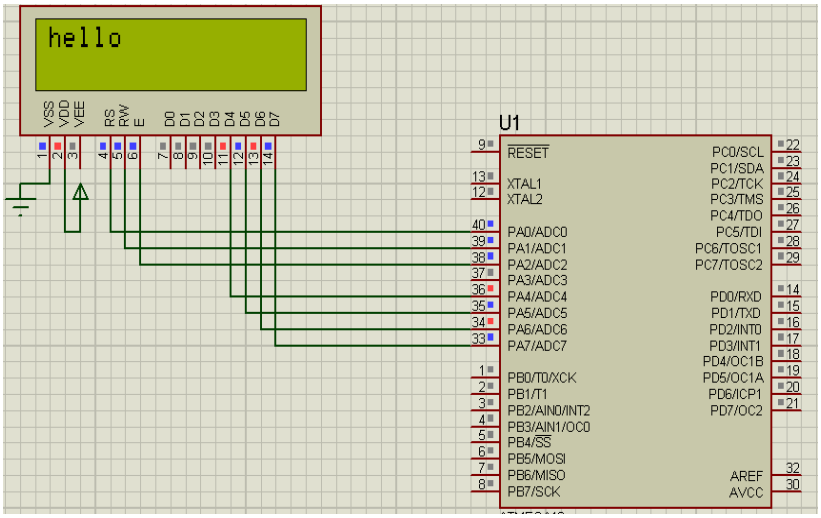
شکل ۴-۹ : انتخاب LCD در پروتئوس

LCD را مانند تنظیمات کدویزارد(مطابق شکل زیر) به میکروکنترلر متصل می‌کنیم:



شکل ۴-۱۰ : اتصال میکروکنترلر به LCD

ولتاژ پایه‌ی VEE میزان نور پس‌زمینه را مشخص می‌کند که می‌توانیم با یک تقسیم مقاومتی ولتاژ این پایه را تنظیم کنیم، اگر این پایه را به زمین متصل کنیم وضوح نوشته ها حداکثر و نور پس‌زمینه خاموش می‌شود. حال کد نوشته شده در کدویژن را در شبیه‌سازی Load می‌کنیم و نتیجه‌ی شبیه‌سازی را مطابق شکل زیر مشاهده می‌کنیم:



شکل ۴-۱۱ : نتیجه‌ی شبیه‌سازی مثال ۱

مثال۲: در این مثال می‌خواهیم یک متغیر صحیح(int)را با LCD نمایش بدهیم (برای مثال عدد ۳۲):

با تنظیمات بخش کدویزارد آشنا شده‌ایم پس وارد بخش کدنویسی در کدویژن می‌شویم:

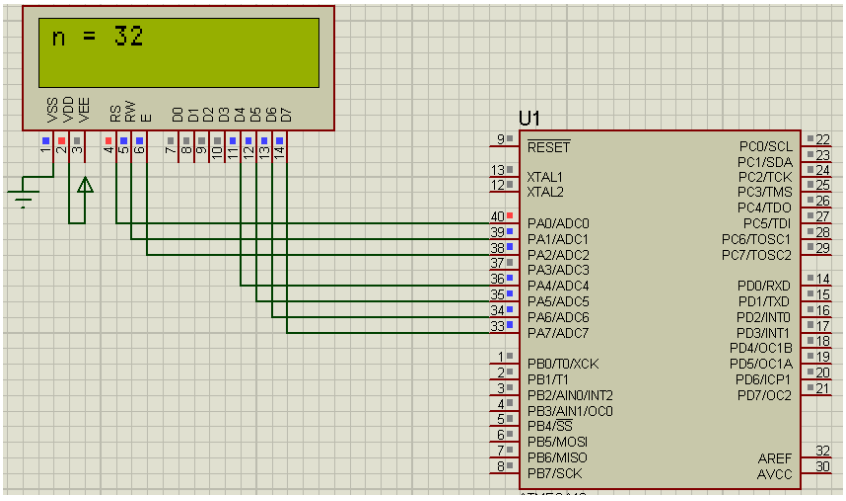
ابتدا یک متغیر صحیح (int) به نام x را که دارای مقدار اولیه‌ی۳۲ می‌باشد و نیز یک رشته‌ی ۲۰ خانه‌ای str[20] را تعریف می‌کنیم(متغیرهای خود را به صورت عمومی(public) تعریف می‌کنیم یعنی در ابتدای برنامه و بالای همه‌ی توابع آنها را تعریف می‌کنیم):

```
24  #include <mega16.h>
25
26  // Alphanumeric LCD functions
27  #include <alcd.h>
28
29  #include <stdio.h>
30  // Declare your global variables here
31  int x=32;
32  char str[20];
33  void main(void)
34  {
```

می‌خواهیم برنامه‌ای بنویسم که روی LCD در کنار حروف(char)، مقدار یک متغیر صحیح نیز نمایش داده شود. برای این کار از تابعی به نام sprintf استفاده می‌کنیم، این تابع می‌تواند عبارات و متغیرها را درون یک رشته بریزد. دستور این تابع به صورت sprintf(1,"2") می‌باشد که در قسمت (۱) نام رشته‌ی مورد نظر و در قسمت (۲) عبارتی که می‌خواهیم درون رشته ریخته شود را می‌نویسیم. اگر قصد داشتیم مقدار یک متغیر را نیز درون رشته بریزیم تابع به صورت sprint(1,"2%d",3) نوشته می‌شود(به جای %d می‌توان از %c برای ریختن یک متغیر کاراکتری(char) درون رشته نیز استفاده کرد) که به جای (۳) باید نام متغیر آورده شود. به کد زیر دقت کنید:

```
while (1)
      {
      lcd_gotoxy(0,0);
      sprintf(str,"n = %d",x);
      lcd_puts(str);
      }
```

نتیجه‌ی شبیه‌سازی کد نوشته شده در نرم‌افزار پروتئوس را در شکل زیر مشاهده می‌کنیم:



شکل ۴-۱۲ : نتیجه‌ی شبیه‌سازی مثال ۲

حال اگر بخواهیم یک عدد اعشاری را نمایش دهیم چه باید کنیم؟در مثال زیر به این سوال پاسخ می‌دهیم.

مثال ۳: می‌خواهیم بر روی نمایشگر LCD متغیری از نوع float را که دارای مقدار ۱۳۸,۹۴۷ است نمایش دهیم.

در ابتدا یک متغیر از جنس float(اعشاری) با نام y و با مقدار اولیه‌ی ۱۳۸,۹۴۷ تعریف می‌کنیم سپس دو رشته به نام‌های str[20] و strl[20] تعریف می‌کنیم، سپس کتابخانه‌ی stdlib.h را به برنامه اضافه می‌کنیم چرا که در ادامه می‌خواهیم از تابع ftoa استفاده کنیم که در این کتابخانه موجود می‌باشد.

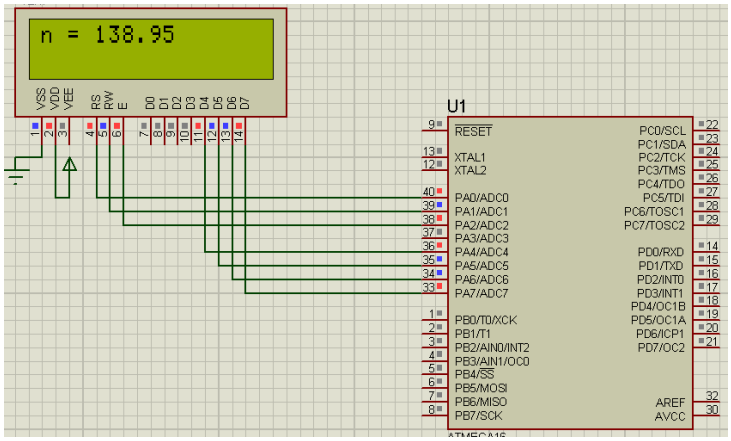
```
#include <mega16.h>

// Alphanumeric LCD functions
#include <alcd.h>

#include <stdio.h>
#include <stdlib.h>
// Declare your global variables here
float y=138.947;
char str[20],strl[20];
void main(void)
{
```

کد زیر را در حلقه‌ی While(1) می‌نویسیم:

```
while (1)
      {
      lcd_gotoxy(0,0);
      ftoa(y,2,strl);
      sprintf(str,"n = %s",strl);
      lcd_puts(str);
      }
```

با دستور ftoa می‌توان عدد اعشاری را به رشته تبدیل کرد. مطابق دستور ftoa(y,2,str1) عدد اعشاری y به رشته‌ی strl تبدیل می‌شود و عدد 2 که در دستور نوشته شده است مشخص می‌کند تا چند رقم اعشار عدد اعشاری نوشته شود، که ما تا ۲ رقم اعشار را مشخص کرده‌ایم.در ادامه نیز با دستور sprintf رشته‌ی strl به همراه عبارت"n=" در رشته‌ی str ریخته می‌شود و با دستور lcd_puts بر روی LCD نمایش داده می‌شود.

نتیجه‌ی شبیه سازی کد نوشته شده را در نرم‌افزار پروتئوس مشاهده می‌کنیم:



شکل ۴-۱۳ : مدار و نتیجه‌ی شبیه‌سازی مثال ۳

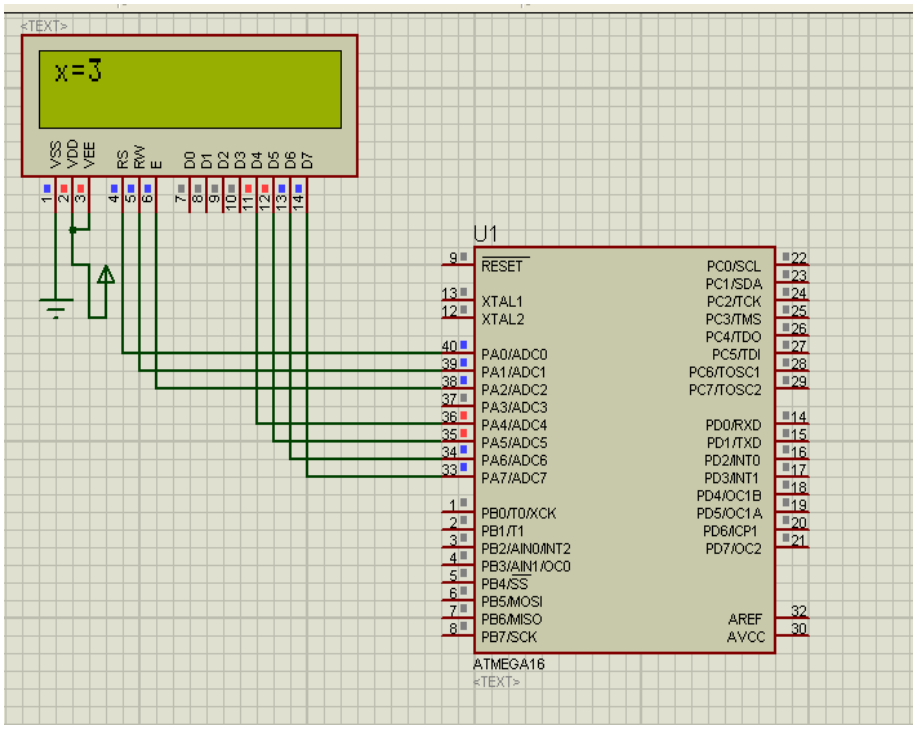
مثال ۴: در این مثال می‌خواهیم از عدد ۱ تا ۱۰ را با فاصله زمانی یک ثانیه بر روی LCD نمایش دهیم:

ابتدا دو متغیر تعریف می‌کنیم: متغیر صحیح i=0 و رشته‌ی str[20]، همچنین کتابخانه‌ی delay.h را برای ایجاد تاخیرهای ۱ثانیه‌ای با دستور delay_ms(1000) در برنامه اضافه می‌کنیم.

حال کد زیر را در حلقه‌ی while(1) می‌نویسیم:

```
while (1)
      {
      for(i=0;i<=10;i++){
      sprintf(str,"x=%d",i);
      lcd_gotoxy(0,0);
      lcd_puts(str);
      delay_ms(1000);
      }
```

نتیجه‌ی شبیه‌سازی کد نوشته شده در نرم‌افزار پروتئوس را در شکل زیر مشاهده می‌کنیم:



شکل ۴-۱۴ : مدار مثال ۴

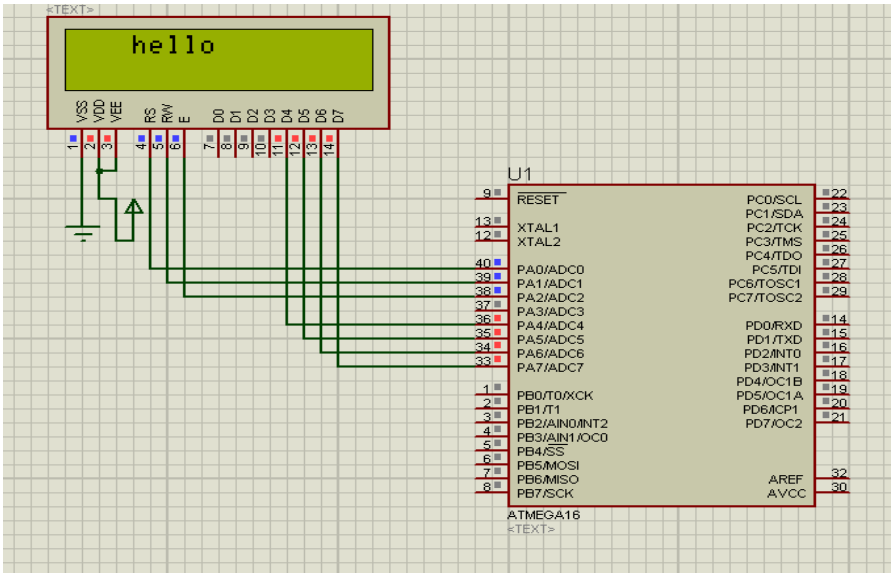
مثال۵: در این مثال قصد داریم کلمه‌ی "hello" را با شروع از خانه‌ی چهارم LCD با تاخیرهای یک ثانیه‌ای، حرف به حرف نمایش دهیم:

در این بخش به تعریف متغیری نیاز نداریم. کافیست کد روبـرو را درون حلقه‌ی while(1) بنویسیم:

```
while (1)
      {
      lcd_gotoxy(3,0);
      lcd_putchar('h');
      delay_ms(1000);
      lcd_putchar('e');
      delay_ms(1000);
      lcd_putchar('l');
      delay_ms(1000);
      lcd_putchar('l');
      delay_ms(1000);
      lcd_putchar('o');
      delay_ms(1000);
      lcd_clear();
      }
```

دستور lcd_gotoxy(x,y) مشخص می‌کند کـه کـاراکتر در کدام خانه‌ی LCD نوشته شود که در این مثال شروع نوشتن حروف(کاراکترها) از خانه‌ی چهـارم، ردیف اول می‌باشد(y=0،x=3). دستور بعدی دسـتور lcd_putchar() اسـت کـه بوسـیله‌ی آن می‌توان یک کاراکتر را نمایش داد و در این مثال با استفاده از آن حروف کلمه‌ی "hello" را یـک به یک نمایش می‌دهیم.

حال پس از اتمام کدنویسی نتیجه‌ی شبیه‌سازی را در نرم‌افزار پروتئوس مشاهده می‌کنیم:



شکل ۴-۱۵ : نتیجه‌ی شبیه سازی مثال ۵

در جدول زیر خلاصه‌ای از مهم‌ترین دستورات کار با LCDهای کاراکتری رامشاهده می‌کنید:

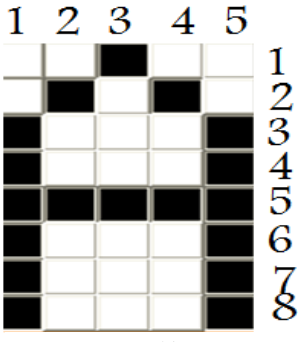
دستور	توضیحات
lcd_clear()	این دستور LCD را پاک می‌کند و مکان‌نما را بر روی خانه‌ی(۰و۰) می‌گذارد.
lcd_gotoxy(x,y)	این دستور مکان‌نما را به خانه‌ی LCD(y،x) منتقل می‌کند.
lcd_putchar(c)	این دستور کاراکتر c را در موقعیت کنونی مکان‌نما نشان می‌دهد.
lcd_puts(str)	این دستور رشته ی str را که در حافظه sram است را نشان می‌دهد.
lcd_putsf(str)	این دستور رشته‌ی str را که در حافظه flash هست را نمایش می‌دهد.
sprint(str,%...,x2)	این دستور متغیر x2 را به رشته‌ی str تبدیل می‌کند.

جدول ۴-۲ : دستورات پرکاربرد برای کار با LCD

یک گام فراتر(LCD)

یکی از کارهایی که می‌توان با LCDهای کاراکتری انجام داد نمایش کاراکترهایی دلخواه توسط کاربر است، برای مثال می‌توان یک شکل دلخواه یا یک کلمه‌ی فارسی را بر روی LCDهای کاراکتری ایجاد کرد،در ادامه به طور کامل با این ویژگی LCDهای کاراکتری آشنا می‌شویم.

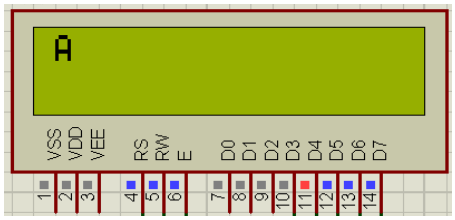
همانطور که می‌دانیم یک LCD ۲ در ۱۶ دارای ۳۲ خانه می‌باشد که هر کدام از این خانه‌ها می‌تواند یک کاراکتر(یک حرف مثل A) را نمایش دهد، هر کدام از این خانه‌ها دارای ۵ در ۸ پیکسل می‌باشند(مجموعاً ۴۰ پیکسل) و می‌توان هرکدام از این پیکسل‌ها روشن و خاموش کرد. پیکسل‌های روشن شده مربوط به کاراکتر A را در شکل روبرو مشاهده می‌کنید:



شکل ۴-۱۶

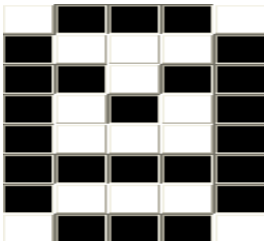
همانطور که در شکل روبرو مشاهده می‌کنید یک کاراکتر دارای ۵ پیکسل به صورت ستون و ۸ پیکسل به صورت سطر می‌باشد یعنی در کل هر کاراکتر دارای ۴۰ پیکسل می‌باشد.

کاراکتر A در صفحه‌ی LCD مانند شکل زیر نمایش داده می‌شود :



شکل ۴-۱۷ : نمایش کاراکتر A در LCD

حال اگر بخواهیم روی هر کاراکتر یک شکل دلخواه بکشیم مانند شکل زیر چه باید کنیم؟



شکل ۴-۱۸ : شکل دلخواه برای نمایش در LCD

پاسخ این است که به کمک نرم‌افزار lcd_char که در CDهمراه کتاب موجود می‌باشد به سادگی می‌توان کد هگزادسیمال مربوط به هر پیکسل برای شکل مورد نظر را تولید کرد، به این ترتیب که شما شکل مورد نظر خود را در هر کاراکتر(۵در۸ پیکسل) می‌کشید و این نرم‌افزار آرایه‌ای از اعداد هگزادسیمال مربوط به آن را تولید می‌کند. محیط این نرم‌افزار را در شکل زیر مشاهده می‌کنید:

شکل ۴-۱۹ : نرم‌افزار LCD_char

برای آنکه بیشتر با آنچه گفته شد آشنا شویم به سراغ یک مثال می‌رویم.

مثال۱: در این مثال می‌خواهیم کدی بنویسم که شکل بالا را بر روی LCD نمایش دهد:
ابتدا تنظیمات مربوط به LCD کاراکتری در کدویزارد را انجام می‌دهیم و کد راgenerate و ذخیره می‌کنیم، سپس سراغ کد نویسی می‌رویم. ابتدا کد تولید شده توسط نرم‌افزارlcd_char را در بالای تابع void کپی می‌کنیم، مطابق زیر:

```
#include <mega16.h>

// Alphanumeric LCD functions
#include <alcd.h>
#include <delay.h>

// Declare your global variables here

   flash unsigned char char0[8] = { 0xE, 0x11, 0x1B, 0x15, 0x11, 0x1F, 0x11,0xD };
```

به جای ؟ مقدار 0 را قرار دادیم(cahar?[8]~char0[8]) که می‌توان هر عدد دلخواه دیگری را نیز قرار داد، سپس برای آنکه این دستور اجرا شود باید یک تابع اضافه شود که پیکسل‌های مورد نظر را که در کد بالا موجود است را روشن کند. این تابع به صورت آماده نوشته شده است و به شکل زیر می‌باشد:

```
void define_char(unsigned char flash*pc،unsigned char char_code)
{
unsigned char i،a;
a=(char_code<<3)|0x40;
for(i=0;i<8;i++) lcd_write_byte(a++،*pc++);
}
```

کافیست این تابع را قبل از تابع Void main بنویسید. محل نوشتن تابع فوق را مطابق شکل زیر مشاهده می‌کنید:

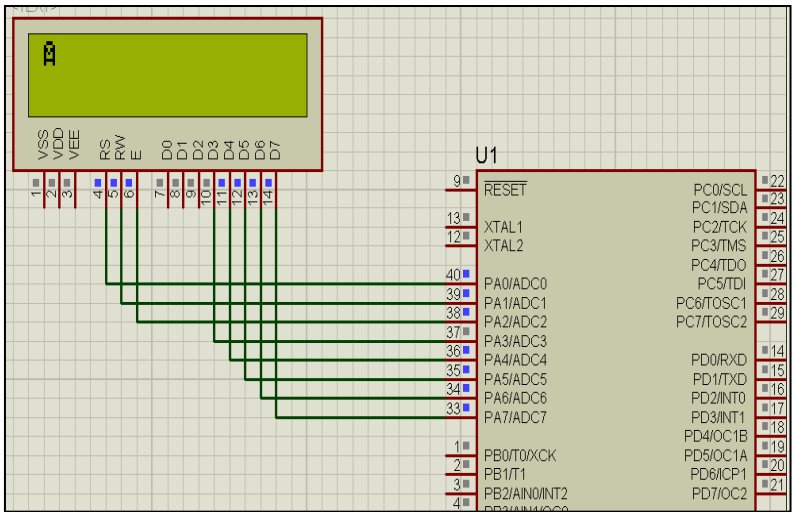
```
#include <mega16.h>
#include <alcd.h>
#include <delay.h>
flash unsigned char char0[8]={ 0xE,0x11,0x1B,0x15,0x11,0x1F,0x11,0xD };

void define_char(unsigned char flash*pc,unsigned char char_code)
{
unsigned char i,a;
a=(char_code<<3)|0x40;
for(i=0;i<8;i++) lcd_write_byte(a++,*pc++);
}

// Declare your global variables here

void main(void)
{
// Declare your local variables here
```

محل نوشتن تابع

سپس کد مربوط به فراخوانی این تابع را همانند کد زیر قبل از حلقه‌ی While(1) می‌نویسم:

```
// D7 - PORTA Bit 7
// Characters/line: 16
lcd_init(16);


define_char(char0,0);
lcd_gotoxy(0,0);
lcd_putchar(0);


while (1)
      {
```

با این کار شکل زیر را در LCD مشاهده می‌کنیم:



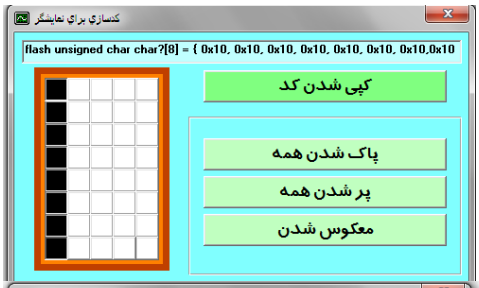
شکل ۴-۲۰ : نمایش شکل دلخواه روی LCD

مثال۲: در این مثال قصد داریم واژه‌ی "ایران" را بر روی LCD کاراکتری نمایش دهیم:

شکل ۴-۲۱ : نمایش کلمه‌ی ایران در LCD

کلمه‌ی ایران از ۵ حرف الف ،ی،ر،ا،ن تشکیل شده است، برای ایجاد کردن شکل بالا باید در هر کاراکتر هر کدام از حروف کلمه‌ی ایران را تولید کنیم، که به ترتیب صفحه‌ی بعد عمل می‌کنیم.

اول حرف "الف":



حرف "ی":

حرف"ر":

دوباره "الف":

و سرانجام "ن":

کدهای تولید شده برای هر حرف توسط نرم‌افزار LCD_char را به صورت زیر در برنامه قرار می‌دهیم:

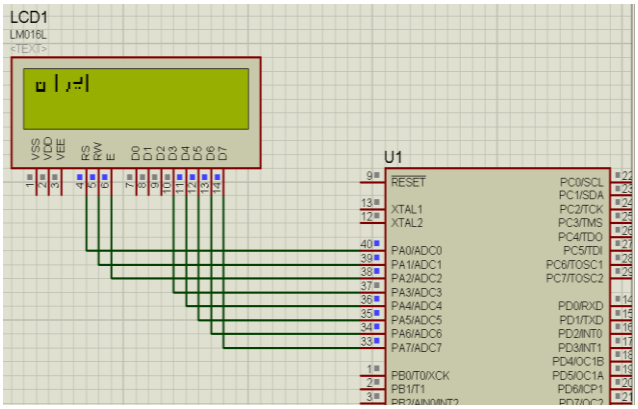
```
#include <alcd.h>
#include <delay.h>

// Declare your global variables here

   flash unsigned char char0[8] = { 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10,0x10 };///ا

   flash unsigned char char1[8] = { 0x0, 0x1, 0x1, 0x1F, 0x0, 0xA, 0x0,0x0 }; //ی

   flash unsigned char char2[8] = { 0x0, 0x0, 0x0, 0x1, 0x1, 0x1, 0x2,0x1C };///ر

  flash unsigned char char3[8] = { 0x1, 0x1, 0x1, 0x1, 0x1, 0x1, 0x1,0x2 };///ا

    flash unsigned char char4[8] = { 0x0, 0x0, 0x0, 0x15, 0x11, 0x11, 0x1F,0x0 };//ن

void define_char(unsigned char flash*pc,unsigned char char_code)
{
unsigned char i,a;
a=(char_code<<3)|0x40;
for(i=0;i<8;i++) lcd_write_byte(a++,*pc++);
}
```

سپس برای ترتیب حروف کد زیر را قبل از حلقه‌ی while(1) می‌نویسیم:

```
lcd_init(16);

define_char(char0,4);
lcd_gotoxy(4,0);
lcd_putchar(4);

define_char(char1,3);
lcd_gotoxy(3,0);
lcd_putchar(3);

define_char(char2,2);
lcd_gotoxy(2,0);
lcd_putchar(2);

define_char(char3,1);
lcd_gotoxy(1,0);
lcd_putchar(1);

define_char(char4,0);
lcd_gotoxy(0,0);
lcd_putchar(0);

delay_ms(1000);
while (1)
      {

      }
```

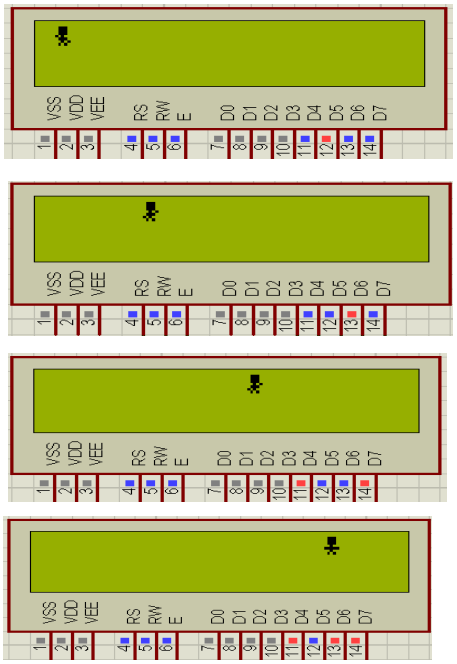
همانطور که در کد بالا مشخص است ابتدا حرف "الف" در خانه‌ی ۴ LCD، حرف "ی" در خانه‌ی ۳ ، حرف "ر" در خانه‌ی ، حرف "الف" در خانه‌ی ۱ و سرانجام حرف "ن" در خانـه‌ی صـفرLCD نمایش داده می‌شوند.

می‌توانیم این برنامه را در شبیه‌ساز پروتئوس نیز مشاهده کنیم:



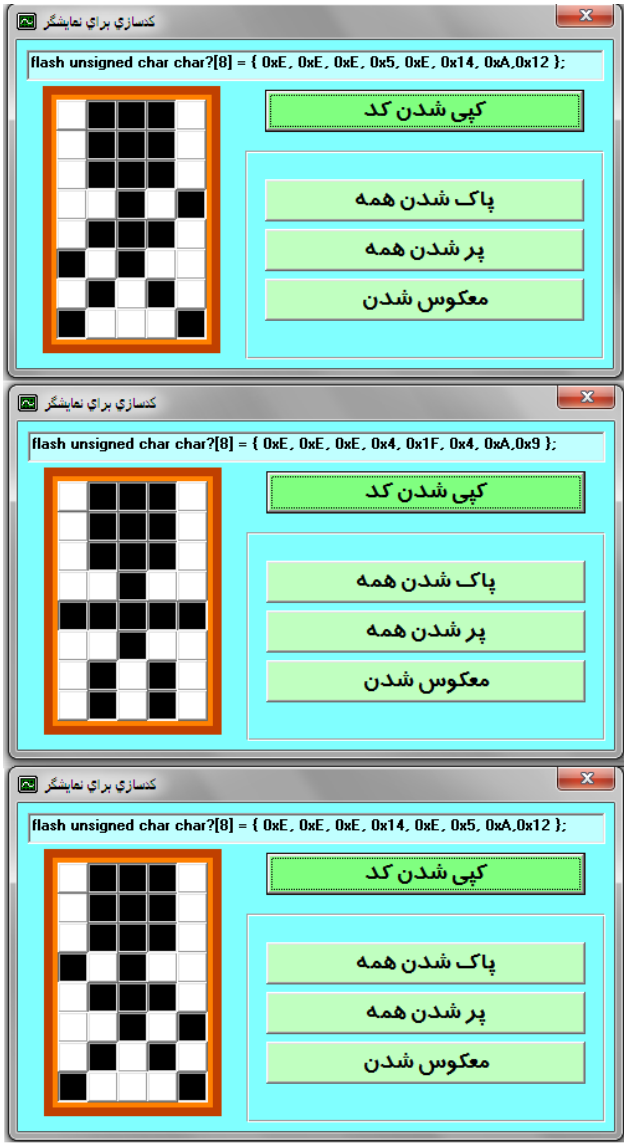
شکل ۴-۲۲ : نتیجه‌ی مثال ۲

مثال ۳: حال با استفاده از مفاهیمی که تا به اینجا آموختیم قصد داریم انیمیشنی بسازیم که یک آدمک به صورت پیوسته مطابق اشکال زیر از ابتدای صفحه‌ی LCD یعنی خانه‌ی(۰و۰) حرکت کرده و به انتهای LCD یعنی خانه‌ی(۰و۱۵) برود:



شکل ۴-۲۳ : ساخت انیمیشن روی LCD

ابتدا تنظیمات بخش LCD را با استفاده از کدویزارد انجام می‌دهیم سپس با استفاده از نرم افـزار Lcd_char اشکال آدمک را در هنگام حرکت مطابق زیر می‌کشیم و رشته‌ی تولیـد شـده توسـط نرم‌افزار Lcd_char را در برنامه‌ی خود کپی می‌کنیم:



شکل ۴-۲۴ : مراحل کشیدن آدمک

رشته‌های تولید شده توسط نرم‌افزار و تابع گفته شده برای روشن و خاموش کردن پیکسل‌های LCD را مطابق کد زیر می‌نویسیم:

```
#include <alcd.h>
#include <delay.h>

// Declare your global variables here

char j=0;
flash unsigned char char0[8] = { 0xE, 0xE, 0xE, 0x5, 0xE, 0x14, 0xA,0x12 };
flash unsigned char char1[8] = { 0xE, 0xE, 0xE, 0x4, 0x1F, 0x4, 0xA,0x9 };
flash unsigned char char2[8] = { 0xE, 0xE, 0xE, 0x14, 0xE, 0x5, 0xA,0x12 };

void define_char(unsigned char flash*pc,unsigned char char_code)
{
unsigned char i,a;
a=(char_code<<3)|0x40;
for(i=0;i<8;i++) lcd_write_byte(a++,*pc++);
}
```

کد مربوط به نمایش هر کاراکتر LCD بدین ترتیب نوشته می‌شود که در خانه‌های ۰و۳و۶و۹و۱۲و۱۵ LCD باید شکل اول آدمک قرار بگیرد، در خانه‌های ۱و۴و۷و۱۰و۱۳ شکل دوم آدمک و در خانه‌های ۲و۵و۸و۱۱و۱۴ شکل سوم آدمک. نحوه‌ی پر کردن خانه‌های LCD با توجه به دستور زیر که در مثال ۱ همین بخش گفته شد انجام می‌گیرد:

define_char(charx,X);

به جای x در charx عدد رشته‌ی تولید شده برای هر کدام از ۳ شکل آدمک قرار می‌گیرد. برای مثال برای نمایش شکل اول آدمک که رشته‌ی تولید شده توسط نرم‌افزار LCD_Char را با نام char0 در کد کپی کردیم، جای xعدد صفر را قرار می‌دهیم (برای شکل دوم char1 و برای شکل سوم char2) و جای X هم خانه‌ای از LCD را که قصد نمایش شکل(آدمک) را داریم می‌نویسیم.

حال با توجه به توضیحات گفته شده خانه‌ها را مطابق کد زیر پر می‌کنیم:

```
lcd_init(16);

define_char(char0,0);
define_char(char1,1);
define_char(char2,2);

define_char(char0,3);
define_char(char1,4);
define_char(char2,5);

define_char(char0,6);
define_char(char1,7);
define_char(char2,8);

define_char(char0,9);
define_char(char1,10);
define_char(char2,11);


define_char(char0,12);
define_char(char1,13);
define_char(char2,14);

define_char(char0,15);

while (1)
      {
```

باید در داخل حلقه‌ی while(1) دستوری نوشته شود که خانه‌های ۰ تا ۱۵ که مطابق کد فوق پر شده‌اند به ترتیب و با تاخیر ۱۰۰ میلی‌ثانیه‌ای نمایش داده شوند تا انیمیشن آدمک در حال حرکت ساخته شود. برای تحقق آنچه گفته شد مطابق زیر کد را در داخل حلقه‌ی while(1) می‌نویسیم:

```
while (1)
      {
      lcd_clear();
      lcd_gotoxy(j,0);
      lcd_putchar(j);
      lcd_gotoxy(j,0);
      j++;
      if(j==15)
      j=0;
      delay_ms(1000);
      }
```

حال می‌توانید کد فوق را در نرم‌افزار پروتئوس شبیه‌سازی کنید و نتیجه را مشاهده کنید.

در ادامه دو بخش دیگر که مربوط به کار با توابع سطح پایین در LCDها و تنظیم نور پس‌زمینه‌ی LCD می‌باشد، توضیح داده می‌شود.

برای کار کردن با توابع سطح پایین کافیست از دو دستور زیر استفاده کنیم:

```
lcd_ready();
lcd_write_data(code);
```

به جای کد داخل پرانتز به ازای هر عدد یک کار مشخص انجام می‌شود که این کارها در جدول زیر خلاصه شده‌اند:

نوع عملکرد	کد فرمان به عدد HEX
پاک کردن صفحه نمایش	1
بازگشت مکان‌نما به محل اولیه	2
شیفت به چپ مکان‌نما	4
شیفت به راست مکان‌نما	6
شیفت به راست کاراکترها	5
شیفت به چپ کاراکترها	7
خاموش شدن کاراکترها و مکان‌نما	8
کاراکترها خاموش و مکان‌نما روشن	A
کاراکترها روشن و مکان‌نما خاموش	C
کاراکترها روشن و مکان‌نما روشن	E
کاراکترها خاموش و مکان‌نما چشمکزن	F
شیفت به چپ مکان‌نما	10
شیفت به راست مکان‌نما	14
شیفت کل به چپ	18
شیفت کل به راست	1C

جدول ۴-۳ : کار با توابع سطح پایین

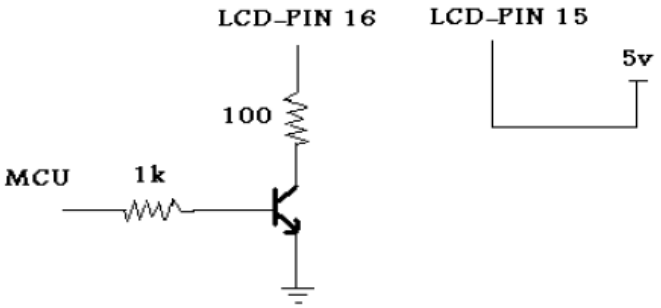
مثال ۴: فرض کنید بخواهیم کل صفحه‌ی نمایش شیفت به چپ پیدا کند:

برای این کار دستور زیرا را می نویسیم:

```
lcd_ready();
lcd_write_data(18);
```

توجه: شاید این دستورات سطح پایین در شبیه‌سازی پروتئوس درست کار نکنند ولی در واقعیت درست هستند.

تنظیم نور پس‌زمینه: نور پس‌زمینه LCD می‌تواند هم روشن و هم خاموش باشد که بستگی به نیاز پروژه‌ی مورد نظر دارد. برای تنظیم نور مورد نظر پس‌زمینه‌ی LCD می‌توانیم از مدار زیر استفاده کنیم:



شکل ۴-۲۵ : تنظیم نور پس‌زمینه‌ی LCD

که MCU(مخفف Micro-Controller) به میکروکنترلر متصل می‌شود و برای ترانزیستور هم می‌توانیم از یک ترانزیستور BC107 استفاده کنیم که بسته به تنظیم پایه‌ی میکروکنترلر(MCU) که ۰ یا ۵ ولت باشد نور پس زمینه LCD روشن یا خاموش می‌شود.

نکته‌ی کاربردی: گاهی اوقات ممکن است که LCD را به PORTC وصل کنید و مشاهده کنید که روشن نمی‌شود، برای رفع این مشکل شما باید بخشJTAG میکروکنترلر را غیرفعال کنید(در میکروکنترلرهای ATmega16 و ATmega32، JTAG بر روی پورت C قرار دارد، برای آشنایی بیشتر با JTAG به بخش فیوزبیت‌ها مراجعه کنید).

در تصاویر صفحه‌ی بعد کاراکترهایی که توسط LCDهای خانواده‌ی HD44780 قابل نمایش است به همراه کد هگزادسیمال هر کاراکتر که در برگه‌ی مشخصات فنی(datasheet) آن آمده است را مشاهده می‌کنید.

Lower 4 Bits \ Upper 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)															
xxxx0001	(2)															
xxxx0010	(3)															
xxxx0011	(4)															
xxxx0100	(5)															
xxxx0101	(6)															
xxxx0110	(7)															
xxxx0111	(8)															
xxxx1000	(1)															
xxxx1001	(2)															
xxxx1010	(3)															
xxxx1011	(4)															
xxxx1100	(5)															
xxxx1101	(6)															
xxxx1110	(7)															
xxxx1111	(8)															

فصل پنجم

وقفه‌های خارجی

در این فصل خواهیم خواند:

۱. تقسیم بندی وقفه‌ها
۲. تنظیمات کدویزارد (بخش INT0,1,2)
۳. مدهای کاری وقفه‌های خارجی
۴. یک گام فراتر

۴-۱. تنظیم رجیسترهای مربوط به وقفه (رجیسترهای Status GICR,GIFR,MCUCR,MCUCSR)

وقفه‌های خارجی

گاهی قصد داریم در برنامه با زدن یک کلید اجرای برنامه متوقف شود و یک عملیات خاص انجام شود و پس از آن دوباره برنامه ادامه پیداکند. برای مثال فرض‌کنید مشغول کارهای روزمره هستید که تلفن همراهتان زنگ می‌خورد ، شما در این لحظه در حال انجام هر کاری که هستید، آن‌را متوقف می‌کنید و به تلفن خود پاسخ می‌دهید و پس از اتمام مکالمه دوباره به ادامه‌ی کار قبل می‌پردازید. وقفه نیز دقیقاً به همین صورت است. زمانی‌که یک وقفه در برنامه ایجاد می‌شود برنامه در هر خطی که باشد آن را متوقف می‌کند و به تابع وقفه رفته و دستورات آن را انجام می‌دهد، زمانی‌که دستورات تابع وقفه تمام شد، دوباره برنامه به حالت عادی بازگشته و دستورات را از همان خطی که به وقفه رفته بود آغاز کرده و خط به خط اجرا می‌کند.

در حالت کلی وقفه‌ها به ۲دسته تقسیم می‌شوند:

۱- وقفه‌های داخلی: وقفه‌هایی که در اثر یک دستور داخلی ایجاد می‌گردند.

۲- وقفه‌های خارجی: وقفه‌هایی که در اثر یک عملیات خارجی که بر روی برخی پایه‌های میکروکنترلر اعمال می‌گردد، انجام می‌شود.

البته یک سری وقفه‌ها هم وجود دارند که به آن‌ها وقفه‌های نرم‌افزاری گفته شده و در اثر فراخوانی توابع سیستمی‌توسط برنامه رخ می‌دهد که غالباً جزئی از همان وقفه‌های داخلی میکروکنترلر حساب می‌شوند.

در جدول صفحه‌ی بعد منابع وقفه‌ی میکروکنترلرهای AVR را مشاهده می‌کنید.

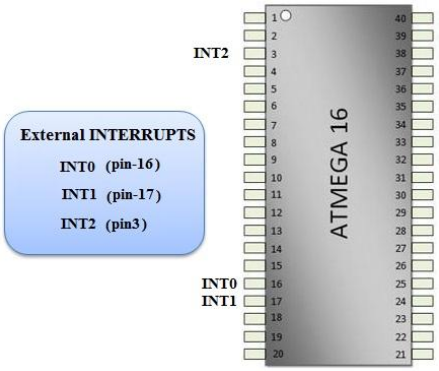
میکروکنترلرهای AVR دارای ۲۱ منبع برای وقفه می‌باشند که این منابع را در جدول زیر مشاهده می‌کنید:

Vector No.	Program Address	Source	Interrupt Definition
1	$000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	TIMER2 COMP	Timer/Counter2 Compare Match
5	$008	TIMER2 OVF	Timer/Counter2 Overflow
6	$00A	TIMER1 CAPT	Timer/Counter1 Capture Event
7	$00C	TIMER1 COMPA	Timer/Counter1 Compare Match A
8	$00E	TIMER1 COMPB	Timer/Counter1 Compare Match B
9	$010	TIMER1 OVF	Timer/Counter1 Overflow
10	$012	TIMER0 OVF	Timer/Counter0 Overflow
11	$014	SPI, STC	Serial Transfer Complete
12	$016	USART, RXC	USART, Rx Complete
13	$018	USART, UDRE	USART Data Register Empty
14	$01A	USART, TXC	USART, Tx Complete
15	$01C	ADC	ADC Conversion Complete
16	$01E	EE_RDY	EEPROM Ready
17	$020	ANA_COMP	Analog Comparator
18	$022	TWI	Two-wire Serial Interface
19	$024	INT2	External Interrupt Request 2
20	$026	TIMER0 COMP	Timer/Counter0 Compare Match
21	$028	SPM_RDY	Store Program Memory Ready

جدول ۵-۱: ۲۱ منبع وقفه‌ی AVR

در این فصل هدف این است که تنها با وقفه‌های خارجی آشنا شویم. میکروکنترلر ATmega16 به طور کلی دارای ۳پایه است که به عنوان وقفه‌های خارجی(ExternalInterrupts) عمل

می‌کنند و با نام‌های INT0,INT1,INT2 شناخته مـی‌شـوند. در شـکل زیـر پایـه‌هـای وقفـه‌ی خارجی در ATmega16 نشان داده شده است:

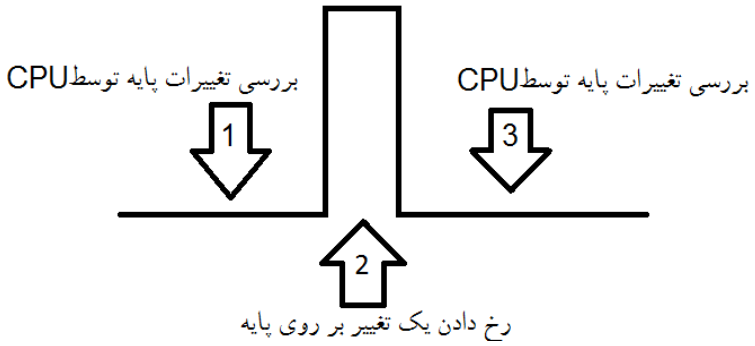


شکل ۵-۱: پایه‌های وقفه‌های خارجی در ATmega16

به طور کلی CPU برای بررسی وقفه‌های داخلی و خارجی از ۲ روش استفاده می‌کند:

۱-روش سرکشی یا Polling

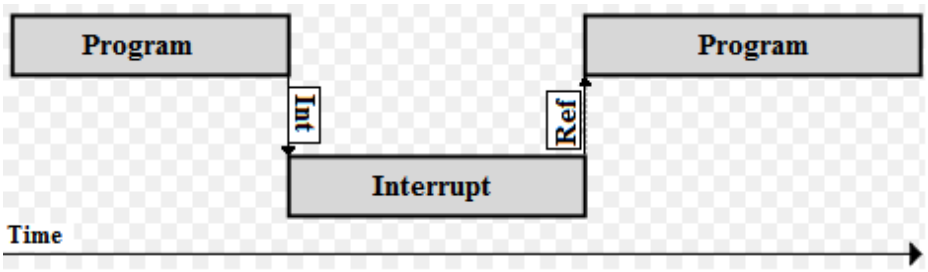
برای اطلاع CPU از پایه‌های میکروکنترلر یکـی از روش‌هـا سرکشـی یـا Polling اسـت کـه در خطوط برنامه توسط کدهای نوشته شده تغییرات پایه‌ها مورد بررسی قرار گیرد و اگـر تغییـرات بـه صورت مداوم صورت بگیرد، بایستی CPU به صورت مداوم وضعیت پایه‌ها را بررسـی کنـد کـه در این صورت بخش قابل توجهی از زمان CPU بخاطر این بررسی‌ها اشغال می‌گردد. همچنـین اگـر تغییرات بر روی پایه‌های میکروکنترلر بسیار سریع باشد ممکن است این تغییرات را CPU متوجـه نشود همانند شکل زیر:



همانطور که در شکل مشخص است در لحظات ۱ و ۳ تغییرات پایه توسط CPU مورد بررسی قرار گرفته ولی تغییر در لحظه‌ی ۲ رخ داده است و CPU این تغییر را متوجه نشده است.

۲-روش وقفه یا Interrupt

با توجه به توضیحات بالا می‌توان به اهمیت تعبیه پایه‌هایی به صورت وقفه‌ی خارجی در میکروکنترلرها پی برد که باعث می‌شود زمان CPU برای بررسی مداوم یک پایه اشغال نگردد و همچنین تغییرات سریع توسط پایه‌هایی از میکروکنترلر که به صورت پایه‌ی وقفه‌ی خارجی می‌باشند باتوجه به سخت افزارهای داخلی بررسی شود و هیچ تغییری بر روی پایه نادیده گرفته نشود. در روش وقفه ،CPU بدون در نظر گرفتن رویداد به انجام سایر اعمال مشغول می‌شود و با وقوع اتفاق موردنظر CPU انجام خط جاری برنامه را متوقف کرده و به خط مربوط به تابع وقفه پرش کرده و پس از اجرای خطوط مورد نظر وقفه که به آن سرویس روال وقفه یا Interrupt Service routine می‌گویند به خطی که از آن پرش کرده بازمی‌گردد:



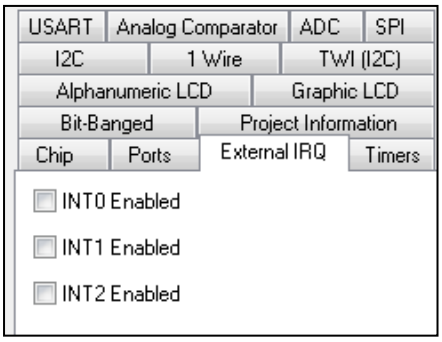
شکل ۵- ۲:نحوه‌ی کارکرد وقفه در برنامه

همانطور که گفته شد در میکروکنترلر ATmega16 تنها ۳ پایه به صورت وقفه‌ی خارجی می‌باشند ولی در میکروکنترلرهای سری Xmega تمامی پایه‌های میکروکنترلر این ویژگی را دارا می‌باشند.

با این توضیحات به سراغ چند مثال می‌رویم و در طی این چند مثال با نحوه‌ی تنظیمات وقفه‌ها در کدویزارد و انواع مدهای کاری وقفه‌های خارجی آشنا می‌شویم.

مثال۱: می‌خواهیم برنامه‌ای بنویسیم که زمانیکه پایه‌ی INT0 صفر شد(پایه شماره ۱۶ در ATmega16 زمین شد) ، کاراکتر K بر روی LCD نمایش داده‌شود:

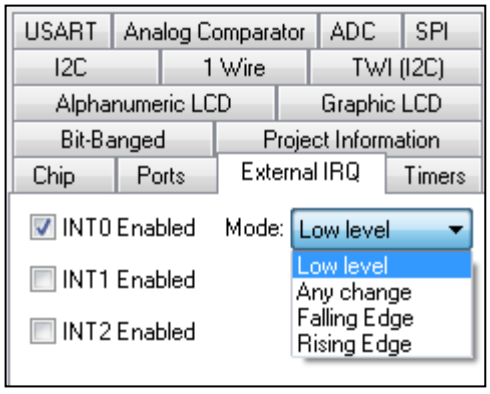
تنظیمات وقفه‌ی خارجی کدویزارد: برای تنظیمات وقفه‌های خارجی ابتدا به تب External IRQ(وقفه‌های خارجی) می‌رویم:



شکل ۵-۳: تنظیمات وقفه‌ی خارجی

بخش INT0 Enabled

درتب External IRQ در اولین صفحه، ۳ گزینه‌ی INT0 Enabled و INT1 Enabled و INT2 Enabled را مشاهده می‌کنیم که اگر تیک هر کدام را بزنیم آن وقفه فعال‌می‌شود. در این مثال می‌خواهیم از وقفه‌ی خارجیINT0 استفاده کنیم پس تیک آن را می‌زنیم تا فعال گردد:

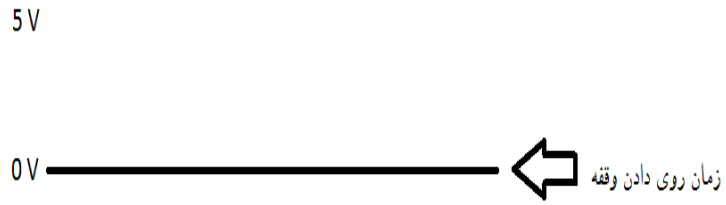


شکل ۵-۴: استفاده از وقفه‌ی خارجی INT0

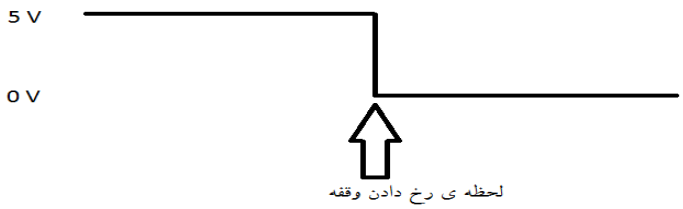
مدهای کاری وقفه‌های خارجی(Mode)

در بخش Mode، ۴حالت کاری داریم که به‌طور خلاصه هر کدام از این حالت‌ها را توضیح می‌دهیم و با انجام مثال‌های بعدی دقیق‌تر با این مدها آشنا می‌شویم:

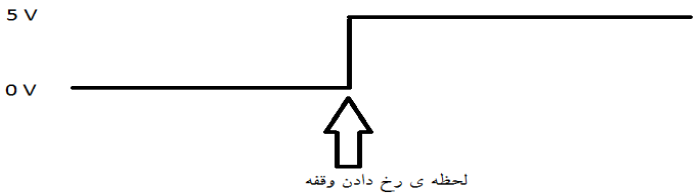
۱- **مد Low level**: در این مد وقفه‌ی مورد نظر زمانی صورت می‌گیرد که پایه‌ی Interrupt(اینتراپت یا وقفه) خارجی در حالت پایین یا صفر قرار داشته باشد(پایه‌ی INT0 زمین(صفر) شود):



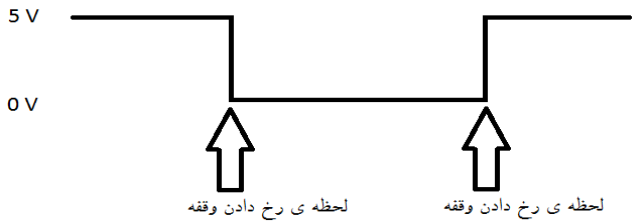
۲- **مد Falling Edge(لبه‌ی پایین رونده)**: در این مد وقفه‌ی موردنظر زمانی ایجاد می‌گردد که پایه‌ی Interrupt از حالت منطقی سطح بالا به سطح پایین تغییر پیداکند، یعنی لحظه‌ای که این پایه از یک به صفر تغییر کند:



۳- **مد Rising Edge(لبه‌ی بالا رونده)**: در این مد وقفه‌ی موردنظر زمانی ایجاد می‌گردد که پایه‌ی Interrupt از حالت منطقی سطح پایین به سطح بالا تغییر پیداکند، یعنی لحظه‌ای که این پایه از صفر به یک تغییرکند:



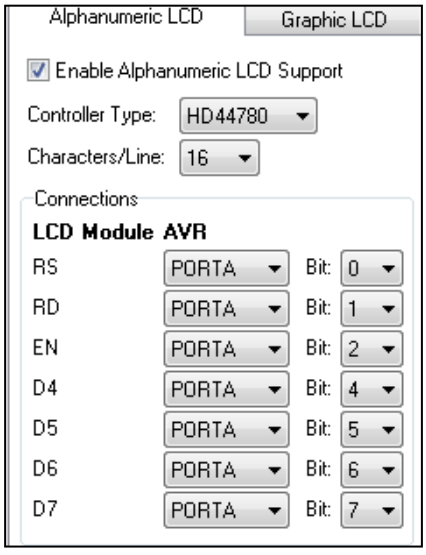
۴- مد **Any Change**(**هر گونه تغییر**): در این مد وقفه‌ی موردنظر زمانی ایجاد می‌گردد که پایه‌ی Interrupt از حالت منطقی سطح پایین به سطح بالا یا از حالت منطقی سطح بالا به سطح پایین تغییر حالت پیداکند، یعنی لحظه‌ای که این پایه از یک به صفر یا از صفر به یک تغییرکند که در اصطلاح می‌گویند مد Any Change(حساس به لبه) می‌باشد:



به این نکته باید توجه داشت که تنها دو پایه‌ی INT0,INT1 در میکروکنترلر را می‌توان در ۴ حالت نام برده در بالا تنظیم کرد و پایه‌ی INT2 در تنظیمات خود تنها دارای دو حالت لبه‌ی بالا رونده و لبه‌ی پایین رونده است.

حال به سراغ مثالی که در ابتدا گفته‌شد می‌رویم، در این مثال می‌خواهیم با صفر شدن پایه‌ی Interrupt (که در این مثال پایه‌ی وقفه‌ی خارجی موردنظر، پایه‌ی INT0 است) روی LCD حرفK نوشته شود، پس تنظیمات آن‌را بر روی مد Low level تنظیم می‌کنیم.

تنظیمات LCD را مانند شکل روبرو انجام می‌دهیم:



شکل ۵-۵: تنظیمات LCD

پس از Generate کردن کد، وارد قسمت برنامه‌نویسی می‌شویم. همانطور که در شکل زیر مشخص است با فعال‌کردن وقفه‌ی خارجی INT0 کدویزارد یک تابع با نام نوشته شده در زیر را تولید می‌کند:

interrupt [EXT_INT0] void ext_int0_isr(void)

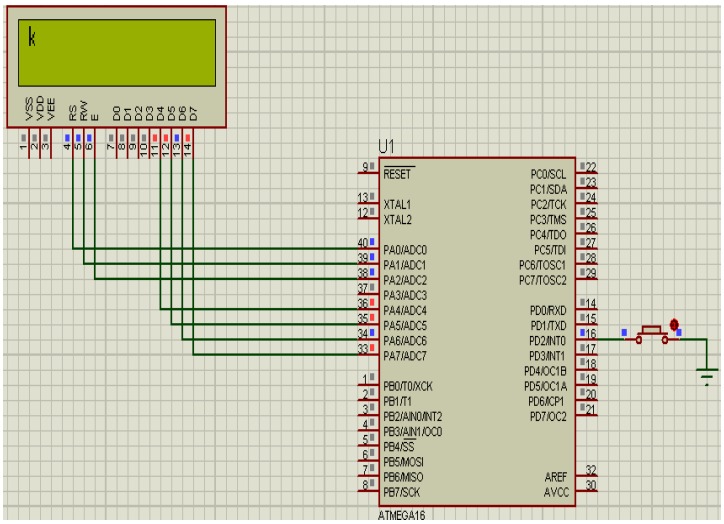
```
#include <stdio.h>
// Alphanumeric LCD functions
#include <alcd.h>
int i=0;
char str[20];
// External Interrupt 0 service routine
interrupt [EXT_INT0] void ext_int0_isr(void)
{



}

// Declare your global variables here

void main(void)
```

به ازای صفر شدن پایه‌ی INT0 برنامه در هر خطی که باشد آن خط را رها کرده و وارد تابع وقفه‌ی خارجی می‌شود. حال کافیست درون تابع وقفه‌ی خارجی پایه‌ی INT0 دستور چاپ کاراکتر "K"را بر روی LCD بنویسیم:

```
interrupt [EXT_INT0] void ext_int0_isr(void)
{
lcd_puts("k");
}

// Declare your global variables here
void main(void)
{
// Declare your local variables here
```

در این مثال در حلقه‌ی While(1) نیازی به کدنویسی نیست.

می‌توانیم این مدار را در پروتئوس نیز شبیه‌سازی کنیم:

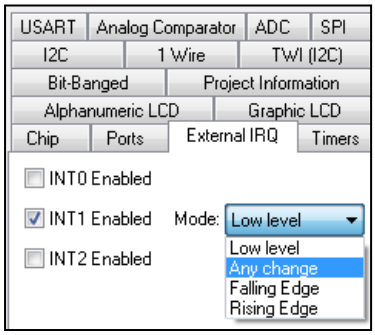


شکل ۵-۶: شبیه‌سازی در پروتئوس

در ادامه با انواع دیگر Modeهای کاری، در قالب مثال آشنا می‌شویم.

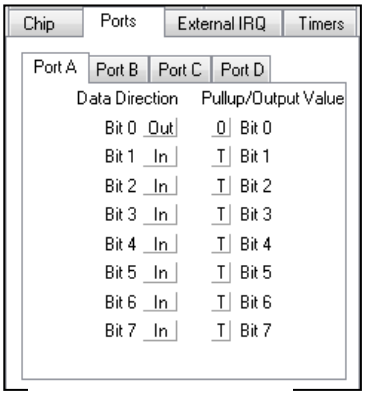
مثال۲: می‌خواهیم برنامه‌ای بنویسیم که با هر تغییری در پایه‌یINT1(پایه شماره ۱۷ میکروکنترلر)، LED که به پایه‌ی A0 میکروکنترلر متصل است روشن‌شود:

برای این کار وقفه‌ی خارجی پایه‌ی INT1 را فعال کرده و مد کاری آن‌را روی مدChange-Any می‌گذاریم، مطابق شکل زیر:



شکل ۵-۷: فعال‌کردن INT1 در مد کاری Any Change

همچنین پایه‌ی A0 را نیز به صورت خروجی و با مقدار اولیه‌ی صفر قرار می‌دهیم:

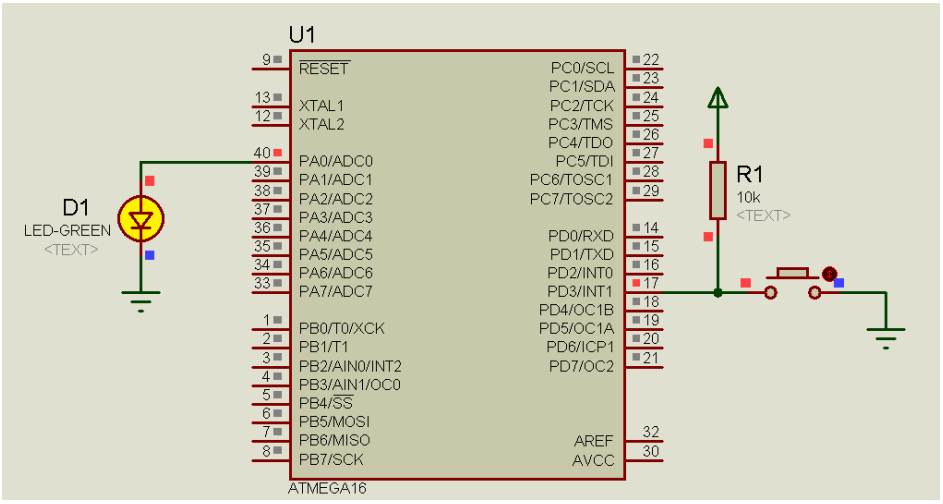


شکل ۵-۸: تنظیمات درگاه A

پروژه را ذخیره‌کرده و به سراغ کدنویسی در کدویژن می‌رویم:در تابع ایجادشده برای وقفه‌ی خارجی پایه‌ی INT1 ،دستور روشن شدن پایه A0 را برای ۱ثانیه و سپس خاموش شدن آن‌را نوشته‌ایم.

```
#include <mega16.h>
#include <delay.h>
// External Interrupt 1 service routine
interrupt [EXT_INT1] void ext_int1_isr(void)
{
PORTA.0=1;
delay_ms(1000);
PORTA.0=0;
}

// Declare your global variables here

void main(void)
{
```

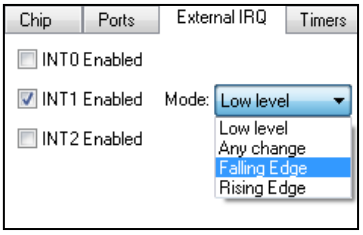
شبیه‌سازی این مدار در پروتئوس:



شکل ۵-۹: شبیه‌سازی در پروتئوس

مثال ۳: در این مثال می‌خواهیم همان دستور بالا یعنی روشن شدن LED متصل به پایه‌ی A0 را به ازای لبه‌ی پایین‌رونده(Failing Edge) بنویسیم:

ابتـدا تنظـیم وقفـه‌ی خـارجی پایـه‌ی INT1 را انجـام می‌دهیم:

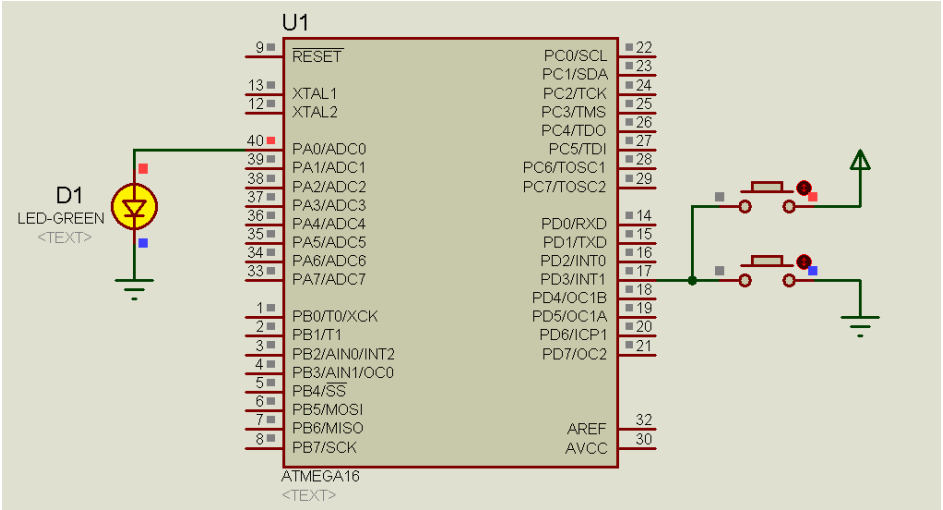


پایه‌ی A0 را نیز مانند مثال قبل به صورت خروجی تنظیم می‌کنیم.

درون تابع وقفه کد روبرو را می‌نویسیم:

```
#include <mega16.h>
#include <delay.h>

// External Interrupt 1 service routine
interrupt [EXT_INT1] void ext_int1_isr(void)
{
PORTA.0=1;
delay_ms(1000);
PORTA.0=0;

}
```

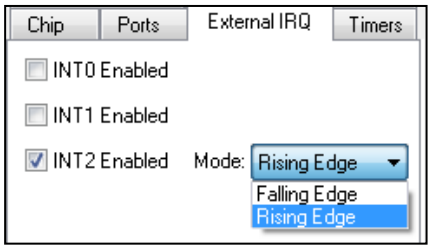
شبیه‌سازی این مدار در نرم‌افزار پروتئوس:



شکل ۵-۱۰: شبیه‌سازی در پروتئوس

همانطورکه در شکل فوق مشاهده می‌کنید، پایه‌ی INT1 بـه صـورت Pull-Up بـه منبـع ۵ولـت متصل شده است و با زدن کلید،ولتاژ پایه از ۵ولت به صفر رفته و وقفه رخ می‌دهد.

مثال ۴: می‌خواهیم برنامه‌ای بنویسـیم کـه بـه ازای لبـه‌ی بالارونـده(Rising Edge) در وقفـه‌ی شماره۲ (INT2) ولتاژ پایه PA0 معکوس شود:



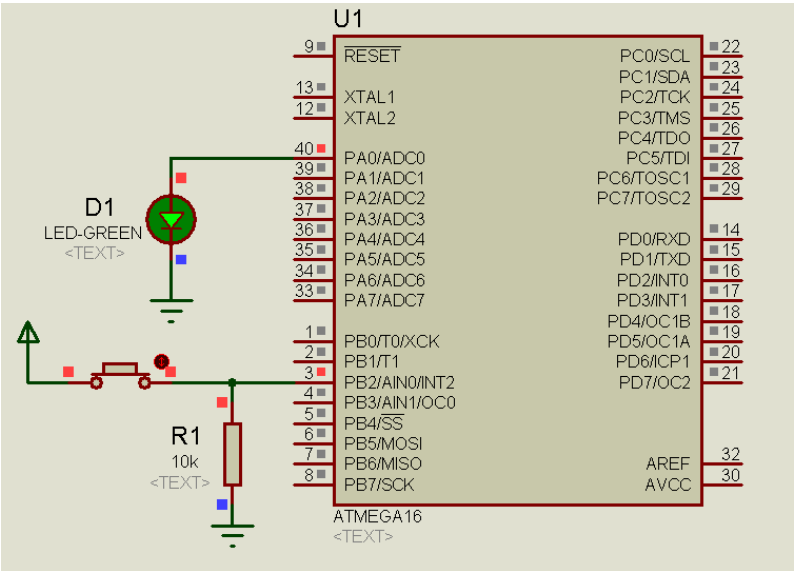
تنظیمات کدویزارد را انجام می‌دهیم:

تنظیمات پایه A0، همانند دو مثال قبل صورت می‌گیرد.

کد روبرو را درون تابع وقفه می‌نویسیم:

```
#include <mega16.h>

// External Interrupt 2 service routine
interrupt [EXT_INT2] void ext_int2_isr(void)
{
PORTA.0=~PORTA.0;
}
```

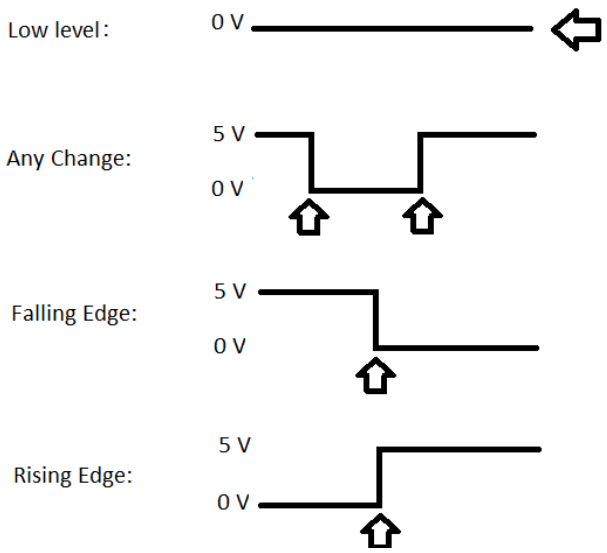
شبیه‌سازی این مدار در پروتئوس:



شکل ۵-۱۱ : شبیه‌سازی در پروتئوس

همانطور که در شکل قبل مشاهده می‌کنید، پایه‌ی INT2 به صورت Pull-Down به زمین ولت متصل شده‌است و با زدن کلید، لبه به یک(۵ولت) رفته و وقفه رخ‌می‌دهد.

هر آنچه در این ۴مثال گفته شد (یعنی زمان رخ‌دادن وقفه‌ها) به صورت خلاصه در شکل زیر نشان داده شده است. زمان رخ‌دادن وقفه‌ها با فلش در شکل مشخص شده‌است:



شکل ۵-۱۲: خلاصه‌ی ۴مثال گفته‌شده

یک گام فراتر

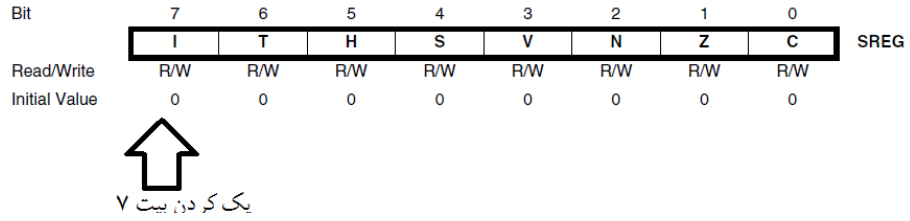
تنظیم رجیسترهای مربوط به وقفه

اکنون قصد داریم کمی بیشتر با رجیسترهای مرتبط با وقفه‌های خارجی آشنا شویم.

۱-رجیستر وضعیت (Status register)

CPU در حالت عملکرد عادی خود و به صورت پیش فرض به وقفه‌ها توجهی ندارد بنابراین بایستی با یک نمودن بیت هفتم رجیستر وضعیت ایجاد وقفه را به اطلاع CPU رساند. با رویداد هر وقفه خارجی این بیت پاک می‌شود و در نتیجه تمام وقفه‌های دیگر غیرفعال می‌شوند، در این

حالت نرم‌افزار می‌تواند با نوشتن یک بر روی ایـن بیـت آن را مجـدداً فعـال کـرده و باعـث ایجـاد وقفه‌های تو در تو گردد.



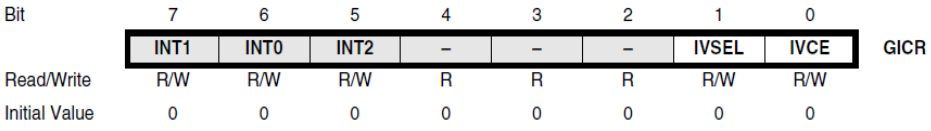
شکل ۵-۱۳ : رجیستر وضعیت یا SREG

از لحاظ کدنویسی برای فعال کردن وقفه‌ی سراسری و همچنین غیر فعال کردن وقفه از دو دستور اسمبلی زیر استفاده می‌شود:

```
#asm("sei")    فعال کردن وقفه سراسری
#asm("cli")    غیرفعال کردن وقفه سراسری
```

۲-رجیستر GICR (General Interrupt Control Register)

برای فعال کردن هر یک از وقفه‌های مورد نظر INT0,INT1,INT2 به جای بیـت مربـوط بـه آن که در شکل زیر مشخص است، ۱ قرار می‌دهیم با انجام این کار وقفه‌ی خـارجی مـورد نظـر فعـال شده و به عنوان یک پایه وقفه‌ی سخت‌افزاری به کار می‌رود:



شکل ۵-۱۴ : رجیستر GICR

به عنوان مثال با نوشتن رجیستر به صورت زیر وقفه‌ی خارجی ۲ را فعال‌سازی می‌کنیم:

7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0
R/W	R/W	R/W	R	R	R	R/W	R/W
0	0	0	0	0	0	0	0

0b00100000

شکل ۵-۱۵ : مثالی از نحوه تنظیم رجیستر GICR

۳-رجیستر GIFR (General Interrupt Flag Register):

اگر یک وقفه‌ی خارجی را بوسیله‌ی رجیستر GICR فعال کرده باشیم و وقفه‌ی همگانی یا Status register را نیز فعال کرده باشیم در هنگام تحریک وقفه‌ی خارجی مورد نظر از طریق پایه‌ی بیرونی، پرچم(Flag) متناظر با آن رجیستر فعال شده یا به عبارت ساده‌تر بیت متناظر با آن وقفه در رجیستر یک می‌گردد و درخواست اجرای وقفه را می‌دهد و برنامه به تابع وقفه‌ی موردنظر پرش می‌کند و بعد از اجرای آن به طور اتوماتیک این پرچم (همان بیت مورد نظر در رجیستر GIFR) پاک می‌گردد.

Bit	7	6	5	4	3	2	1	0	
	INTF1	INTF0	INTF2	–	–	–	–	–	GIFR
Read/Write	R/W	R/W	R/W	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

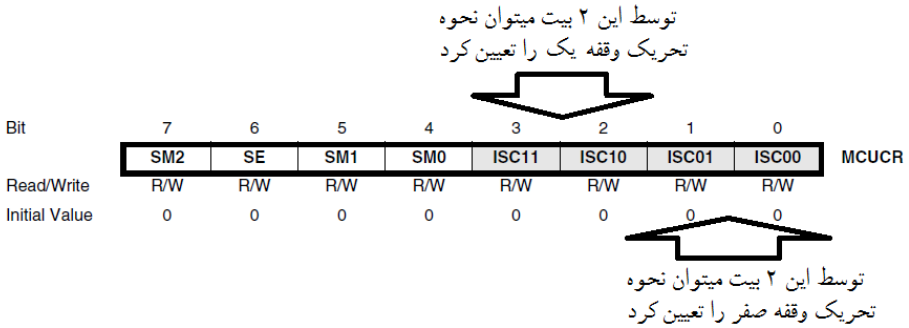
شکل ۵-۱۶ : رجیستر GIFR

به عنوان مثال فرض کنید وقفه‌ی خارجی INT0 را فعال کرده‌ایم(به نحوه‌ی وقوع وقفه فعلاً کاری نداریم یعنی اینکه لبه بالا رونده ،پایین رونده ،... باشد)، زمانیکه پایه مورد نظر وقفه حالت موردنظر وقوع وقفه را پیش آورد(برای مثال این پایه به صورت لبه بالارونده تنظیم شده بود و یک تغییر از صفر به یک بر روی پایه ایجاد شد) بیت ششم رجیستر GIFR(INTF0) پرچم(Flag) وقوع وقفه را با یک کردن خود اعلام می‌دارد.

نکته: از این رجیستر بیشتر در روش Polling استفاده می‌شود بدین صورت که با یک دستور شرطی منتظر می‌مانیم تا پرچم مرتبط با وقفه‌ی خارجی مورد نظر برای انجام تابع وقفه‌ی مورد نظر تحریک گردد:

۴-رجیستر MCUCR (MCU Control Register)

این رجیستر نحوه‌ی تریگر شدن هر یک از وقفه‌های خارجی صفر و ۱(INT1،INT0) را تعیین می‌کند. منظور از تریگر شدن یعنی اینکه فعال شدن وقفه به کدام یک از ۴ حالت Low , Any Change, Rising Edge و Failing Edge صورت گیرد.



شکل ۵-۱۷ : رجیستر MCUCR

جدول تریگر شدن وقفه‌ی صفر و یک:

ISC01	ISC00	نحوه تریگر شدن وقفه‌ی صفر
0	0	Low
0	1	Any Change
1	0	Rising Edge
1	1	Failing Edge

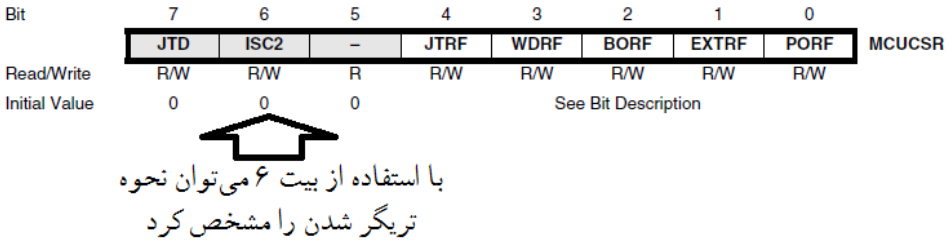
جدول ۵-۲: نحوه تریگر شدن وقفه‌ی صفر

ISC11	ISC10	نحوه تریگر شدن وقفه‌ی یک
0	0	Low
0	1	Any Change
1	0	Rising Edge
1	1	Failing Edge

جدول ۵-۳: نحوه تریگر شدن وقفه‌ی یک

۵-رجیستر MCUCSR (MCU Control and Status Register)

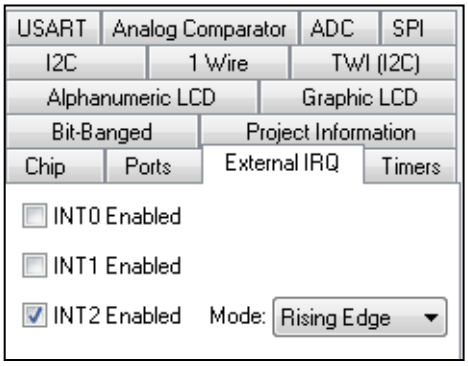
نحوه‌ی تریگر کردن وقفه‌ی شماره ۲(INT2) با استفاده از بیت ۶ رجیستر MCUCSR تعیین می‌شود. همانطور که قبلاً هم گفته شد وقفه‌ی خارجی ۲ برخلاف وقفه‌ی صفر و یک تنها در دو حالت لبه‌ی بالارونده و پایین رونده قابل پیکربندی است. نوشتن صفر در بیت ISC2 باعث تریگر شدن به صورت لبه‌ی پایین رونده و نوشتن یک باعث تریگر شدن به صورت لبه‌ی بالارونده می‌شود:



شکل ۵-۱۸ : رجیستر MCUCSR

مثال ۵: در مثال آخر از این فصل می‌خواهیم نحوه‌ی پر شدن رجیسترهای مربوط به وقفه‌ی خارجی INT2 را زمانیکه به صورت لبه‌ی بالارونده تنظیم شده است را توسط کد تولید شده‌ی کدویزارد مشاهده کنیم:

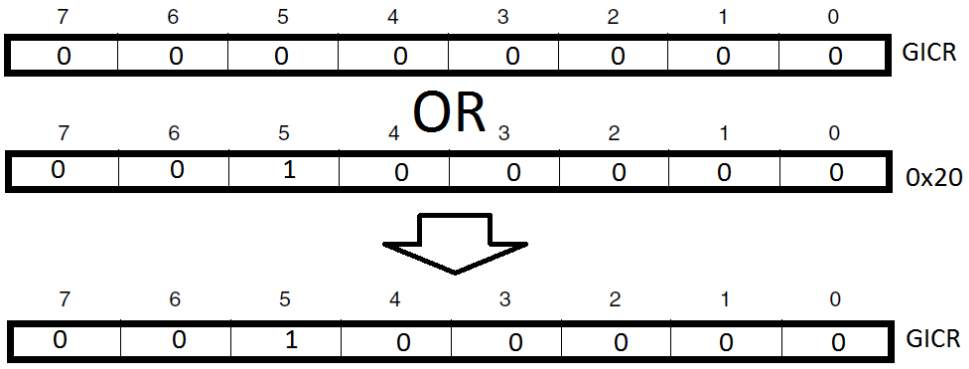
در کدویزارد تنظیمات روبرو را انجام می‌دهیم:



حال در کد تولید شده‌ی کدویزارد در بخش رجیسترهای وقفه‌های خارجی داریم:

```
// External Interrupt(s) initialization
// INT0: Off
// INT1: Off
// INT2: On
// INT2 Mode: Rising Edge
GICR|=0x20;
MCUCR=0x00;
MCUCSR=0x40;
GIFR=0x20;
```

همانطور که در بالا مشاهده می‌کنیم رجیستر GICR که در ابتدا همه‌ی بیت‌های آن صفر است با 0x20 یای منطقی(OR) شده است و حاصل به صورت زیر می‌شود:



شکل ۵-۱۹: نحوه پر شدن رجیستر GICR

و همانطور که در بالا توضیح داده شده با ۱ شدن بیت ۵ ، بیت INT2 یک می‌شود و بدین ترتیب این وقفه فعال می‌گردد.

همانطور که در کد تولید شده مشاهده می‌شود رجیستر MCUCR برابر صفر است و همانطور که توضیح داده شد در این رجیستر نحوه‌ی تریگر شدن هر یک از وقفه‌های خارجی صفر و ۱(INT0,INT1) را تعیین می‌کند و چون در اینجا از این دو وقفه استفاده‌ای نشده به صورت 0x00 باقی مانده است.

رجیستر MCUCSR نیز به صورت 0x40 تنظیم شده است که در مبنای باینری به صورت زیر می‌باشد:

7	6	5	4	3	2	1	0	
0	1	0	0	0	0	0	0	MCUCSR

شکل ۵-۲۰ : نحوه پر شدن رجیستر MCUCSR

که همانطور که قبلا گفته شد بیت ISC2 یک شده و به معنای آن است که وقفه‌ی خارجی INT2 به صورت لبه‌ی بالارونده تریگر شده است.

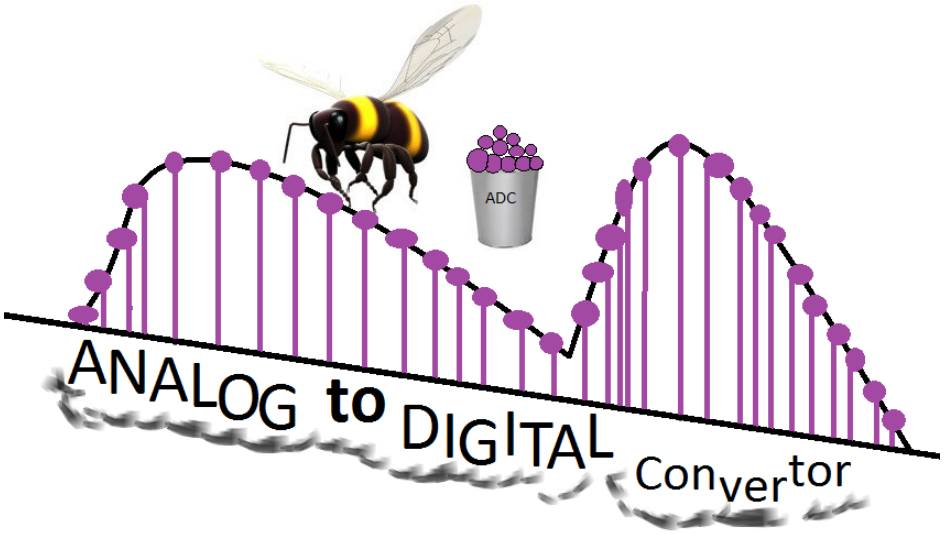
و در آخر GIFR نیز به صورت0x20می‌باشد که در مبنای باینری به صورت زیر می‌باشد و همانطور که در بالاتر توضیح داده شد بیت مربوط به INTF2 یک شده است و پرچم(Flag) متناظر با آن رجیستر فعال شده(یعنی وقفه‌ی INT2) و درخواست اجرای وقفه را می‌دهد و برنامه به تابع وقفه‌ی موردنظر پرش می‌کند.

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	GIFR

شکل ۵-۲۱ : نحوه پر شدن رجیستر GIFR

فصل ششم

مبدل آنالوگ به دیجیتال



در این فصل خواهیم خواند:

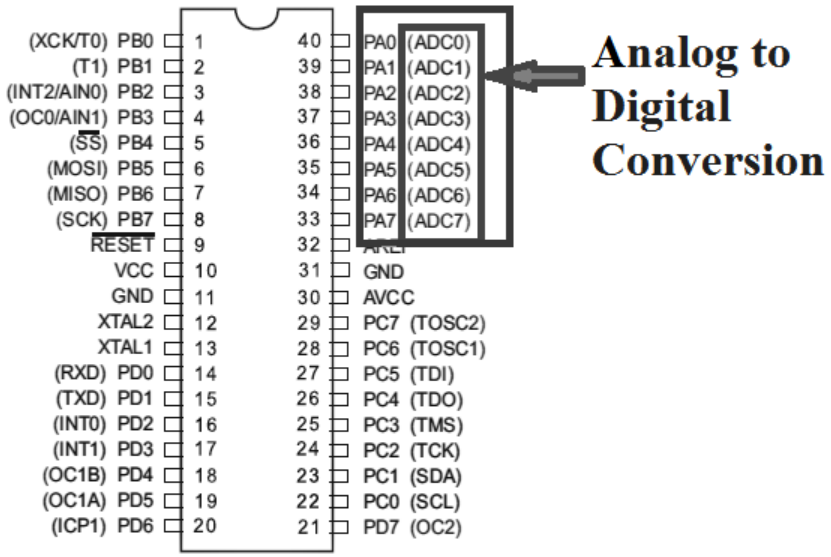
۱. آشنایی کلی با واحد ADC
۲. تابع کار با ADC در کدویژن
۳. انواع مرجع‌های مقایسه‌ی واحد ADC
۴. یک گام فراتر
   ۴-۱. واحد حذف نویز
   ۴-۲. رجیستر ADCSRA
   ۴-۳. رجیستر ADCMUX

واحد ADC

واحد ADC یکی از مهمترین بخش‌های میکروکنترلر است. بگذارید با یک مثال ساده شروع‌کنیم، فرض‌کنید شما دو پیمانه‌ی ۱ و ۵ لیتری دارید و می‌خواهید با آنها مقداری آب بردارید. در این حالت شما دارای محدودیت هستید زیرا فقط می‌توانید به اندازه‌ی ۱ و یا ۵ لیتر آب بردارید و مقدارهای بین ۱ و ۵ لیتر را نمی‌توانید به طور دقیق بردارید. پس شما برای رفع این محدودیت به یک پیمانه‌ی کوچکتر احتیاج دارید. میکروکنترلرها نیز آی‌سی‌های منطقی یا دیجیتال هستند که به صورت صفر و یک کار می‌کنند. برای مثال اگر یک را ۵ولت در نظر بگیریم، میکروکنترلر فقط می‌تواند در پایه‌های خروجی صفر یا ۵ولت داشته باشد و نیز فقط می‌تواند از پایه‌های ورودی ولتاژهای صفر و ۵ولت را بخواند.

واحد ADC (Analog to Digital conversion) این محدودیت را برطرف کرده و به کاربر اجازه می‌دهد هر ولتاژی بین صفر تا ۵ ولت را توسط پایه‌های مربوط به ADC بخواند.

پایه‌هایی از میکروکنترلر که این توانایی را دارند پایه‌های A0 تا A7 می‌باشند که کاربرد دوم این پایه‌ها بنا به معماری داخلی آنها تبدیل ولتاژ آنالوگ به دیجیتال (ADC) است. واحد ADC هر سطح ولتاژ را به یک عدد تبدیل می‌کند. برای مثال به ولتاژ ۳,۷ ولت عدد ۷۵۷ را اختصاص می‌دهد که در ادامه نحوه‌ی آنرا توضیح خواهیم داد. در شکل زیر پایه‌های ADC در میکروکنترلر ATmega16 را مشاهده می‌کنید:



شکل ۶-۱ : پایه‌های ADC میکروکنترلر ATmega16

در شکل فوق پایه‌های ۳۳ تا ۴۰ (پورت A) پایه‌های نمونه‌برداری واحد ADC می‌باشند و پایه‌های ۳۰ و ۳۱ (AVCC و GND) نیز پایه‌های مربوط به تغذیه‌ی این واحد می‌باشند که باید آنها را به ترتیب به مثبت و منفی تغذیه متصل کرد.

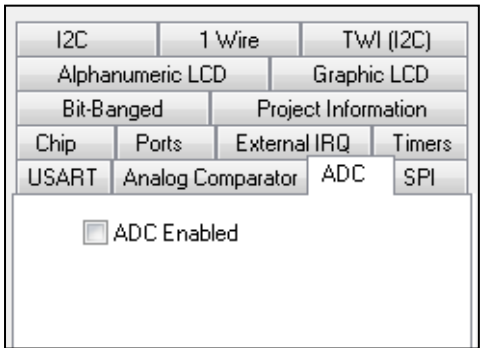
کاربردهای این واحد را در قالب چند مثال بسیار ساده توضیح می‌دهیم:

مثال۱: می‌خواهیم زمانی‌که پایه‌ی A0 میکرو (ADC0) ولتاژی بین ۰ تا ۲,۵ولت را دریافت کرد، LED که به پایه B0 متصل‌است روشن و اگر ولتاژی بین ۲,۵ تا ۵ولت را دریافت کرد خاموش شود:

ابتدا در تنظیمات کدویزارد پایه‌ی B0 را به صورت خروجی تنظیم می‌کنیم.

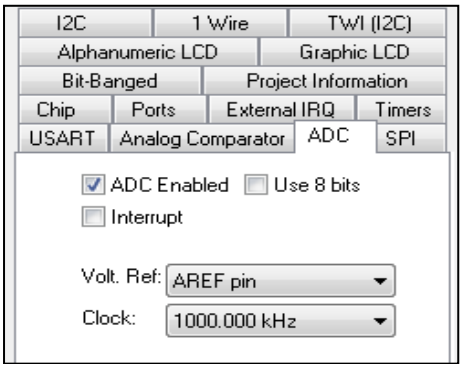
تنظیمات اولیه‌ی کدویزارد ADC

اکنون به سراغ تنظیمات کدویزارد ADC می‌رویم. در تب مربوط به واحد ADC با صفحه‌ی زیر روبرو می‌شویم:



شکل ۶-۲ : تنظیمات بخش ADC در کدویزارد

تیک گزینه‌ی ADC Enabled(ADC Enabled) را می‌زنیم تا بخش ADC فعال‌شود:



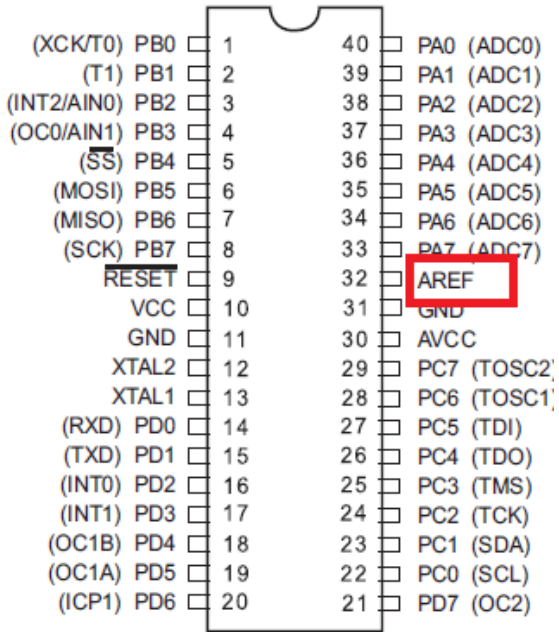
شکل ۶-۳ : فعال‌کردن ADC

زمانی‌که تیک این گزینه را بزنیم صفحه‌ای مانند شکل بالا مشاهده می‌شود.

کارکرد ADC به صورت نمونه‌برداری ۱۰ و ۸ بیتی و انتخاب مرجع مقایسه

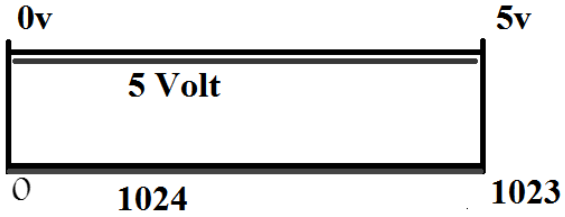
واحد ADC می‌تواند ولتاژهای آنالوگ (پیوسته) را به اعدادی دیجیتال (گسسته) تبدیل کند. این تبدیل به صورت تناسبی انجام می‌شود یعنی اگر برای مثال بازه‌ی ولتاژ بین صفر تا ۵ ولت و بازه‌ی اعداد دیجیتال بین صفر تا ۱۰۲۳ باشد میکروکنترلر ولتاژ صفر ولت را به عدد صفر و ولتاژ ۵ ولت را به عدد ۱۰۲۳ اختصاص می‌دهد باقی اعداد نیز به صورت تناسبی تخصیص می‌یابند، برای مثال به ولتاژ ۲,۵ ولت عدد دیجیتال ۵۱۲ اختصاص می‌یابد. در میکروکنترلرهای AVR بازه‌ی اعداد دیجیتال به دو صورت می‌تواند باشد به صورت ۱۰ بیتی (۰ تا ۱۰۲۳) و یا به صورت ۸ بیتی (۰ تا ۲۵۵) که این بازه به صورت پیش‌فرض ۱۰ بیتی است و اگر در بخش تنظیمات ADC در کدویزارد تیک گزینه‌ی Use 8 bits را بزنیم این بازه به صورت ۸ بیتی تنظیم می‌شود.

بازه‌ی ولتاژی که میکروکنترلر می‌تواند بخواند نیز همیشه بین ۰ تا ۵ ولت نیست و این بازه توسط ولتاژ مرجع مقایسه ADC انتخاب می‌شود؛ برای مثال اگر ولتاژ مرجع را برابر ۳ ولت در نظر بگیریم بازه‌ی ولتاژ قابل خواندن توسط میکرو نیز به صورت صفر تا ۳ ولت می‌شود. این ولتاژ مرجع را می‌توانیم از بخش Volt Ref(Voltage Reference) انتخاب‌کنیم که این گزینه به صورت پیش‌فرض بر روی AREF Pin(ADC Reference Pin یا به عبارتی پایه‌ی مرجع ADC) قرار دارد که AREF Pin پایه‌ی شماره‌ی ۳۲ در ATmega16 می‌باشد؛ ولتاژی که به این پایه بدهیم به‌عنوان مرجع مقایسه قرار می‌گیرد.



شکل ۶-۴ : پایه AREF میکروکنترلر

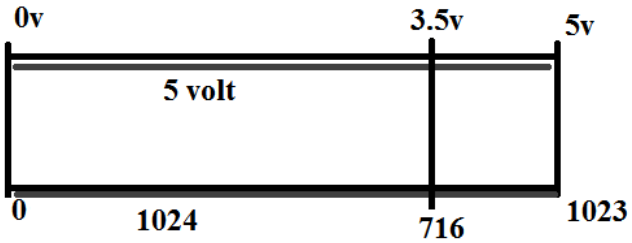
برای روشن‌تر شدن موضوع فرض‌کنید شما بـه پایـه‌ی AREF ولتـاژ ۵ ولـت را داده‌ایـد و تیـک گزینه‌ی Use 8 bits را هم نزده‌اید، در این حالت ADC به صورت ۱۰بیتی تنظـیم شـده اسـت (یعنی به هر ولتاژ بین ۰ تا ۵ولت عددی بین ۰ تا ۱۰۲۳ را اختصاص می‌دهد). به شکل زیـر نگـاه کنید:



شکل ۶-۵ : اختصاص یک عدد دیجیتال به ولتاژ ورودی

بدین ترتیب برای مثال ولتاژ آنالوگ ۳,۵ولت معادل عدد دیجیتال ۷۱۶ می‌باشد:

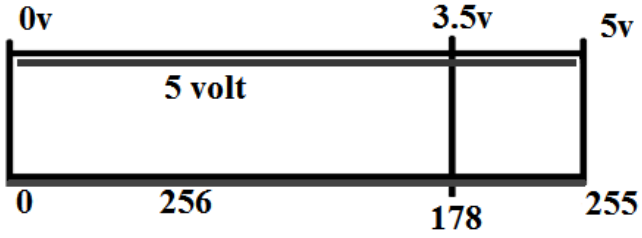
$$\frac{x}{1023} = \frac{3.5}{5} \rightarrow x = 716$$



شکل ۶-۶ : تبدیل ولتاژ آنالوگ ۳,۵ولت به مقدار دیجیتال ۱۰ بیتی

حال فرض‌کنید تیک گزینه‌ی Use 8 bits را می‌زدیم، در این صورت بازه‌ی اعداد دیجیتال بین ۰ تا ۲۵۵ شده و محاسبات به صورت زیر انجام می‌شود:

$$\frac{x}{255} = \frac{3.5}{5} \rightarrow x = 178$$



شکل ۶-۷ : تبدیل ولتاژ آنالوگ ۳,۵ولت به مقدار دیجیتال ۸ بتی

تا به اینجا با مفاهیم اولیه واحد ADC آشنا شدیم، در ادامه با تنظیمات دیگر کدویزارد در بخش ADC آشنا می‌شویم.

تنظیمات بخش Clock

یکی از مواردی که می‌تواند در واحد ADC مهم باشد سرعت نمونه‌برداری ولتاژ پایه‌ها است یعنی اگر ولتاژ ورودی تغییر کرد، میکروکنترلر در چه زمانی این تغییر را احساس می‌کند. سرعت نمونه برداری متناسب با فرکانس کلاک (Clock) میکروکنترلر است. کلاک می‌تواند با استفاده از مقسم‌های کلاک با فرکانس‌های f/2, f/4, f/8 , f/16 , f/32 , f/64 , f/128 عمل نمونه‌برداری را انجام‌دهد که f هم مربوط به فرکانس‌کاری میکروکنترلر است که ما به صورت ۸مگاهرتز تنظیم کرده‌ایم(در اینجا گزینه به صورت پیش‌فرض بر روی ۱ مگاهرتز(۱۰۰۰کیلوهرتز) تنظیم شده است). همانطور که گفتیم فرکانس کلاک ADC در حقیقت سرعت نمونه‌برداری ولتاژ پایه‌ها را مشخص می‌کند و زمانی‌که بر روی ۱ مگا هرتز تنظیم شده است، ۱میلیون بار در ثانیه از ولتاژ پایه‌های ADC نمونه‌برداری می‌شود و هرگاه که ولتاژ این پایه تغییر کرد میکروکنترلر یک میکروثانیه‌ی بعد متوجه آن می‌شود.

اگر بخواهیم تنظیم فرکانس نمونه‌برداری را بدون استفاده از کدویزارد و توسط رجیسترهای مربوطه انجام دهیم می‌توانیم با استفاده از ۳بیت ADPS2,ADPS1,ADPS0 که در رجیستر ADCSRA وجود دارد فرکانس‌های نمونه‌برداری f/2,f/4,f/8,f/16,f/32,f/64,f/128 را در واحد ADC ایجاد کنیم.

شکل رجیستر ADCSRA (ADC Control and Status Register A) و بیت‌های ADPS2,ADPS1,ADPS0 (ADC Prescaler) را مشاهده می‌کنید:

7	6	5	4	3	2	1	0	
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
0	0	0	0	0	0	0	0	

بیت های تقسیم فرکانس

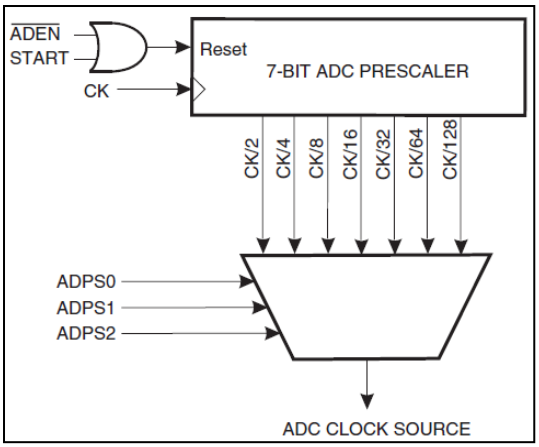
شکل ۶-۸ : شکل رجیستر ADCSRA

نحوه‌ی تقسیم فرکانس در این رجیستر به صورت جدول ۶-۱ می‌باشد:

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

جدول ۶-۱ : تقسیم فرکانس

مدار تقسیم‌کننده‌ی فرکانس که در واحد ADC وجود دارد را می‌توانید در شکل زیر مشاهده کنید:



شکل ۶-۹ : تقسیم‌کننده‌ی فرکانس

توضیحات بیشتر کار با رجیسترهای ADC در بخش یک‌گام فراتر آمده است.

تابع کار با ADC(read_ADC()) در کدویژن

با این دانسته‌هایی که تا به اینجا خواندیم اکنون این سوال که چگونه می‌توان از ولتاژهایی که از پایه‌های ADC نمونه‌برداری می‌کنند استفاده کنیم؟ پاسخ این است که با فعال‌کردن واحد ADC یک تابع با نام read_adc(x) ایجاد می‌گردد که ولتاژ پایه‌ی x(x می‌تواند ۰ تا ۷ باشد) را به صورت دیجیتال(اعداد گسسته) می‌خواند و کاربر می‌تواند از این اعداد استفاده‌کند:

read_adc(0) , read_adc(1) , … ,read_adc(7)

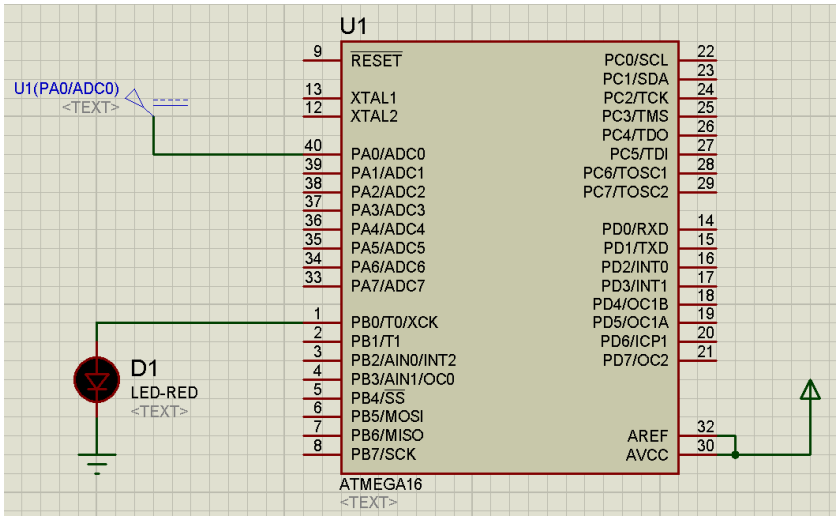
این توابع مربوط به ۸پایه‌ی ADC است که مقادیر ولتاژ آنها به صورت عدد در این توابع ذخیره می‌شود. برای مثال اگر به پایه‌ی A0 که مربوط به ADC0 است ولتاژ ۴٫۵ولت متصل کنیم و بازه‌ی اعداد دیجیتال واحد ADC هم به صورت ۱۰ بیتی(۰ تا ۱۰۲۳) تنظیم شده باشد، با نوشتن دستور(a = read_adc(0 عدد ۹۲۰ درون متغیر a ذخیره می‌شود(که a یک متغیر دلخواه است)

$(\frac{x}{1023} = \frac{4.5}{5} \rightarrow x = 920)$

در کد زیر تابع read_adc() تولید شده توسط کدویزارد را مشاهده می‌کنید :

```
// Read the AD conversion result
unsigned int read_adc(unsigned char adc_input)
{
ADMUX=adc_input | (ADC_VREF_TYPE & 0xff);
// Delay needed for the stabilization of the ADC input voltage
delay_us(10);
// Start the AD conversion
ADCSRA|=0x40;
// Wait for the AD conversion to complete
while ((ADCSRA & 0x10)==0);
ADCSRA|=0x10;
return ADCW;
}
```

اکنون دوباره به سراغ مثالی که در ابتدا گفته شد می‌رویم. فرض‌کنید پایه‌ی A0 را به ولتاژی متغیر متصل کرده‌ایم و یک LED را به پایه B0 مطابق شکل ۶-۱۰ وصل کرده‌ایم و می‌خواهیم کدی بنویسم که وقتی پایه‌ی A0(ADC0) ولتاژی بین ۰ تا ۲٫۵ولت را دریافت‌کرد LED که به پایهB0 متصل‌است روشن شود و اگر ولتاژی بین ۲٫۵ تا ۵ولت را دریافت‌کرد LED خاموش شود.

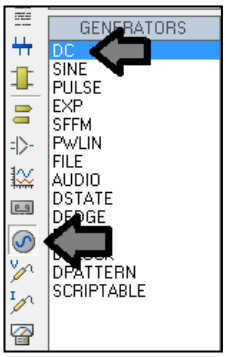


شکل ۶- ۱۰

```
while (1)
      {
      a=read_adc(0);
      if(a>512)
      PORTB.0=0;
      if(a<=512)
      PORTB.0=1;

      }
}
```

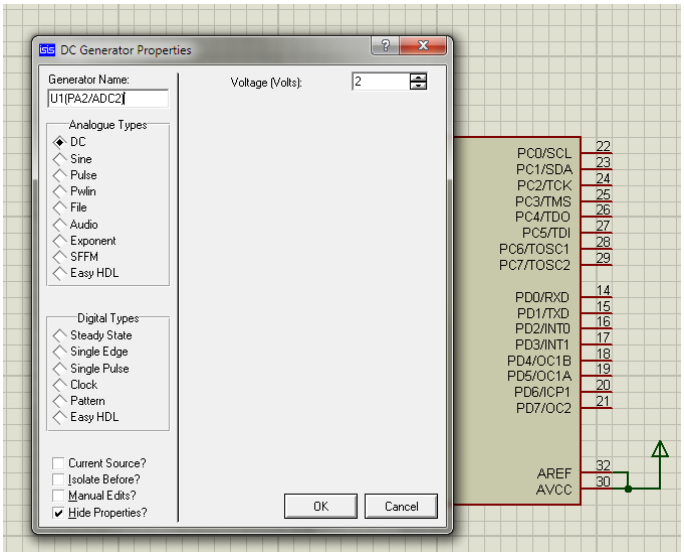
برای این کار متغیر صحیح (int) دلخواهی مانند a تعریف می‌کنیم. این متغیر را به این علت تعریف‌کردیم که مقدار read_adc(0) را درون آن ذخیره کنیم. اکنون در قسمت While(1) کد را می‌نویسیم؛ به این صورت که اگر ولتاژ پایه‌ی ADC0 بیشتر از ۲٫۵ولت بود (بزرگتر از ۵۱۲) پایه‌ی B0 خاموش شود و اگر کمتر از ۲٫۵ولت بود(کوچکتر از ۵۱۲) پایه B0 روشن شود

($\frac{x}{1023} = \frac{2.5}{5} \rightarrow x = 512$):



در بخش شبیه‌سازی پروتئوس به یک منبع ولتاژ DC احتیاج داریم که ولتاژهای مورد نظرمان را بر روی پایه‌های ADC ایجاد کنیم. این منبع DC را از بخش منابع یا GENERATORS( ) مطابق شکل روبرو پیدا می‌کنیم:

با دبل‌کلیک روی این منبع می‌توانیم مقدار آن را تنظیم کنیم. حال مقدار آن را بر روی ۲ولت تنظیم می‌کنیم(که کمتر از ۲٫۵ولت می‌باشد):

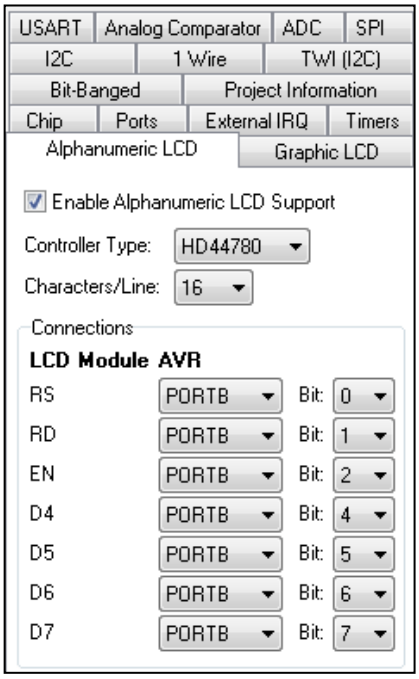


شکل ۶-۱۱ : تنظیم ولتاژ منبع DC

اگر برنامه را اجرا کنیم مشاهده می‌کنیم که LED روشن می‌شود.
حال منبع DC را بر روی ولتاژ ۳ولت تنظیم می‌کنیم(که بیشتر از ۲,۵ولت می‌باشد)
این بار اگر برنامه را اجرا کنیم مشاهده می‌کنیم که LED خاموش می‌شود.

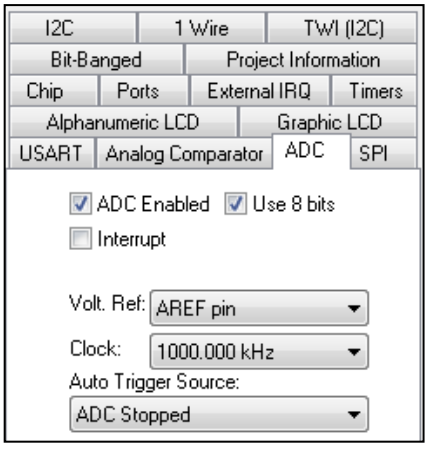
مثال۲: می‌خواهیم برنامه‌ای بنویسم که ولتاژ پایه‌ی A2(ADC2) را به صورت یک عدد ۸بیتی(۰ تا ۲۵۵) بر روی LCD متصل به پایه‌های PORTB نمایش دهد(واحد ADC هم به صورت ۸بیتی تنظیم شده است).

ابتدا تنظیمات کدویزارد مربوط به LCD را انجام می‌دهیم:



شکل ۶-۱۲ : تنظیماتLCD

سپس تنظیمات مربوط به ADC را انجام می‌دهیم:



شکل ۶-۱۳ : تنظیماتADC

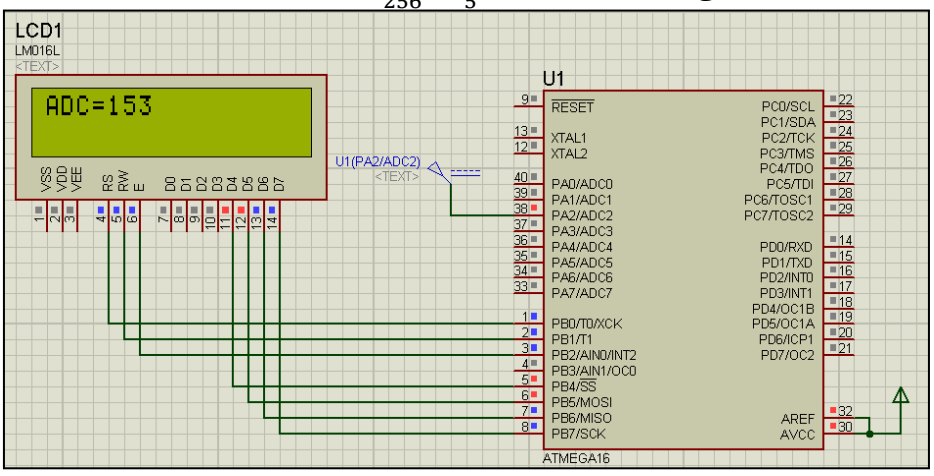
در بخش کدنویسی دوباره یک متغیر صحیح دلخواه به نام k تعریف می‌کنیم تا داده‌ی read_adc(2) را داخل آن بریزیم:

```
#include <alcd.h>
#include <stdio.h>

#define ADC_VREF_TYPE 0x20


int k=0;
char c[20];

// Read the 8 most significant bits
// of the AD conversion result
unsigned char read_adc(unsigned char adc_input)
```

کد نمایش عدد ADC بر روی LCD در داخل حلقه‌ی While(1) را به صورت زیر می‌نویسیم:

```
while (1)
      {
       k=read_adc(2);
       lcd_gotoxy(0,0);
       sprintf(c,"ADC=%d",k);
       lcd_puts(c);
      }
```

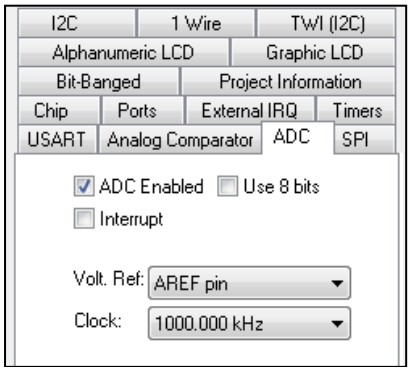
اگر این مدار را در پروتئوس شبیه‌سازی کنیم در زمانی‌که ولتاژ ۳ولت به پایه‌ی A2 متصل است نتیجه مانند شکل زیر می‌شود ($\frac{x}{256} = \frac{3}{5} \rightarrow x = 153$):



شکل ۶-۱۴ : نتیجه‌ی شبیه‌سازی مثال ۲

مثال۳: برنامه‌ای بنویسید که ولتاژ دو پایه‌ی ADC4 و ADC5 مقایسه شود و هر زمان ولتاژ پایه‌ی A5 از A4 بیشتر بود LED متصل به پایه‌ی D0 روشن‌شود و هر زمان ولتاژ پایه‌ی A4 از A5 بیشتر بود LED متصل به پایه D1 روشن شود و اگر مساوی بودند هر دو LED همزمان روشن شوند(ADC به صورت ۱۰بیتی باشد):

ابتدا پایه‌های PORTD.0 و PORTD.1 را به صورت خروجی تنظیم می‌کنیم سپس تنظیمات واحد ADC را به صورت زیر انجام می‌دهیم:

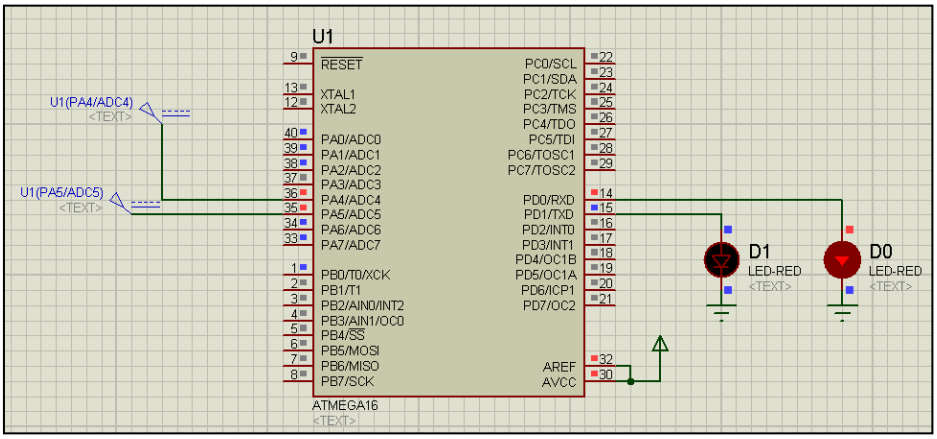


شکل ۶-۱۲

در بخش کدنویسی ابتدا دو متغیر به نام‌های a1,a2 تعریف می‌کنیم تا مقادیر read_adc(4) و read_adc(5) را درون آنها ذخیره کنیم و بعد از آن کد زیر را در حلقه‌ی While(1) می‌نویسیم:

```
while (1)
      {
      a1=read_adc(4);
      a2=read_adc(5);
      if(a2>a1){
      PORTD.0=1;
      PORTD.1=0;
      }

      if(a2<a1){
      PORTD.0=0;
      PORTD.1=1;
      }


      if(a2==a1){
      PORTD.0=1;
      PORTD.1=1;
      }
```

در بخش شبیه‌سازی پروتئوس ابتدا ولتاژ A4 را ۲ولت وA5 را ۴ولت می‌دهیم (یعنی A5>A4) که مشاهده می‌کنیم LED صفر روشن می‌شود:

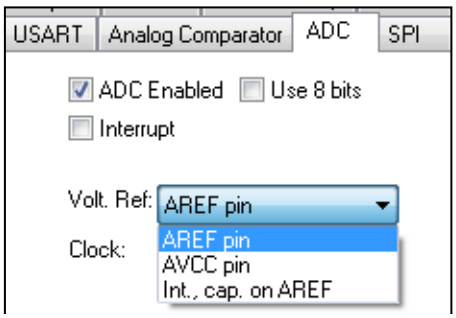


شکل ۶-۱۳ : مدار مثال ۳

حال اگر ولتاژ پایه‌ی A4 را بزرگتر از ولتاژ پایه‌ی A5 قرار دهیم مشاهده می‌کنیم که LED شماره یک روشن می‌شود و اگر ولتاژ پایه‌ها را برابر قرار دهیم هر دو LED روشن می‌شوند.

انواع مرجع‌های مقایسه واحد ADC

حال می‌خواهیم با انواع ولتاژهای مرجع مقایسه‌ی ADC (یا همان Voltage-Reference) آشنا شویم. ابتدا با مرجع مقایسه AREF pin آشنا شدیم و دیدیم که با دادن ولتاژ دلخواه به این پایه مرجع ولتاژ را تعیین می‌کردیم. دو حالت دیگر نیز برای انتخاب ولتاژ مرجع وجود دارد که در شکل زیر مشاهده می‌کنید:



شکل ۶-۱۴ : مرجع ولتاژ در واحد ADC

۱) **AVCC Pin**: این پایه که پایه‌ی شماره ۳۰ در میکروکنترلر ATmega16 می‌باشد می‌تواند به عنوان مرجع مقایسه انتخاب شود که در این حالت ولتاژ Vcc(۵ولت) به عنوان مرجع ADC

قرار می‌گیرد. کاربرد اصلی این پایه در واحد ADC میکرو برای حذف نویز می‌باشد که در ادامه توضیح خواهیم داد.

۲) **int,cap .on AREF**: این مرجع مقایسه که مخفف Internal 2.56v Voltage Reference Whit external Capacitor at AREF pin می‌باشد، عبارت است از یک ولتاژ مرجع داخلی با مقدار ۲٫۵۶ولت.

یکی از مزیت‌های مرجع داخلی با مقدار ۲٫۵۶ ولت این است که اگر ADC را به صورت ۸ بیتی انتخاب کنیم(یعنی اعداد دیجیتال متناسب با ولتاژ بین ۰ تا ۲۵۶ باشند) و ولتاژ مرجع را برابر مرجع داخلی انتخاب کنیم(یعنی ۲٫۵۶ ولت)، عدد دیجیتال حاصل از خواندن ولتاژ صد برابر مقدار واقعی ولتاژ می‌شود:

$$\frac{\text{مقدار } ADC}{256} = \frac{\text{مقدار ولتاژ}}{2.56} \rightarrow \text{مقدار } ADC = 100 * \text{مقدار ولتاژ}$$

در پروژه‌ی دماسنج انتهای کتاب از این مرجع مقایسه استفاده شده است.

نکته: درصورتی‌که کاربر از یک منبع ولتاژ ثابت متصل به پایه‌ی AREF استفاده‌کند، قادر به استفاده از ولتاژهای مرجع دیگر نخواهد بود. اگر ولتاژ خارجی به پایه‌ی AREF اعمال نگردد، کاربر می‌تواند در داخل برنامه بین ولتاژهای مرجع AVCC و ۲٫۵۶ولت داخلی سوییچ نماید.

تا به اینجا به صورت نسبی با مبحث ADC آشنا شدیم. برای آنکه با این واحد و مابقی تنظیمات کدویزارد مربوط به آن بیشتر آشنا شویم به بخش یک گام فراتر می‌رویم.

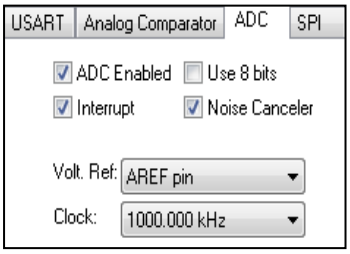
یک‌گام فراتر(ADC)

مطابق یک‌گام فراترهای سابق در این بخش هم سعی می‌کنیم کمی بیشتر با این واحد در میکروکنترلر آشنا شویم.

واحد حذف نویز(Noise Canceler)

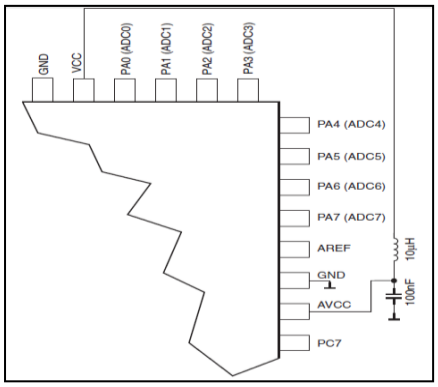
یکی از مزیت‌های واحد ADC توانایی حذف نویز ناشی از CPU می‌باشد بدین ترتیب که در اثر فرکانس بالا(۸مگاهرتز) امکان بوجود آمدن نویز وجود دارد و گاهی ما وسایلی را با میکرو مرتبط می‌سازیم که به شدت به نویز حساس هستند و این نویز CPU برای ما مخاطره‌آمیز است و

عملکرد دستگاه‌های ما را مختل می‌سازد. برای کاهش این نویز باید واحد Noise Canceler را فعال‌کنیم که در شکل زیر نحوه‌ی فعال‌کردن آن را مشاهده می‌کنید:



شکل ۶-۱۵ : فعال کردن Noise canceler

البته راه دیگری هم برای حذف نویز وجود دارد که یک مدار خارجی مطابق شکل زیر ایجاد کنیم تا نویز روی پایه‌ها که ناشی از CPU است به حداقل برسد. این مدار یک فیلتر پایین‌گذر LC است که نویزهایی را که فرکانس بالایی دارند فیلتر می‌کند):



شکل ۶-۱۹: مداری برای کاهش نویز

در قسمت آخر از این بخش می‌خواهیم با وقفه‌ی ADC آشنا شویم. در بخش کدویزارد با زدن تیک گزینه‌ی interrupt وقفه‌ی واحد ADC فعال می‌شود و کدویزارد تابعی برای وقفه به صورت زیر ایجاد می‌کند:

```
// ADC interrupt service routine
interrupt [ADC_INT] void adc_isr(void)
{
unsigned int adc_data;
// Read the AD conversion result
adc_data=ADCW;
// Place your code here

}
```

درون این تابع می‌توان کد مورد نظر خود را وارد کرد. این وقفه در ADC به این صورت می‌باشد که در هر نمونه‌برداری برنامه وارد وقفه شده و دستورات درون آن را اجرا می‌کند. ضمناً برای آنکه

حالات شروع نمونه‌برداری را ایجاد کنیم باید با رجیسترهای ADCSRA و ADMUX کارکنیم که در ادامه با نحوه‌ی کار با این رجیسترها بیشتر آشنا می‌شویم.

رجیستر ADCSRA(ADC Control and Status Register A)

این رجیستر مربوط به تنظیمات داخلی ADC میکروکنترلر بوده و شکل این رجیستر به صورت زیر می‌باشد:

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

Bit	Description
ADEN	با یک شدن: فعال کردن واحد ADC
ADSC	با یک شدن: آغاز تبدیل عدد آنالوگ به دیجیتال در واحد ADC
ADATE	با یک شدن: فعال شدن مد Auto Trigger
ADIF	با یک شدن: وارد تابع وقفه می شود
ADIE	با یک شدن: فعال شدن وقفه ی اینتراپت
ADPS2	عامل تقسیم فرکانس CPU برای تولید فرکانس ADC
ADPS1	
ADPS0	

شکل ۶-۲۰ : رجیستر ADCSRA

رجیستر ADCMUX (ADC MULTIPLEXER Selection)

این رجیستر مربوط به انتخاب پایه‌ای از ADC می‌باشد که عمل نمونه‌برداری را انجام می‌دهد(عبارت MUX مخفف MULTIPLEXER است که وظیفه‌ی انتخاب یک مسیر از چند مسیر موجود را دارد). تصویر زیر بیت‌های مربوط به این رجیستر را نشان می‌دهد:

7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

شکل ۶-۲۱ : رجیستر ADCMUX

با بیت‌های ۶ و۷ ، مرجع‌های مقایسه ADC را مشخص می‌کنیم. مطابق جدول زیر:

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

جدول ۶-۲

بیت ۵ مشخص می‌کند که نتیجه‌ی نمونه‌برداری به صورت از چپ به راست در رجیسترهای ADCH و ADCL قرار بگیرند. بیت‌های ۰ تا ۴ هم کانال ورودی ADC و بهره‌ی کانال‌ها را مشخص‌می‌کند، مطابق جدول زیر:

MUX4..0	Single Ended Input	Positive Differential Input	Negative Differential Input	Gain
00000	ADC0	N/A		
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000	N/A	ADC0	ADC0	10x
01001		ADC1	ADC0	10x
01010		ADC0	ADC0	200x
01011		ADC1	ADC0	200x
01100		ADC2	ADC2	10x
01101		ADC3	ADC2	10x
01110		ADC2	ADC2	200x
01111		ADC3	ADC2	200x
10000		ADC0	ADC1	1x
10001		ADC1	ADC1	1x
10010		ADC2	ADC1	1x
10011		ADC3	ADC1	1x
10100		ADC4	ADC1	1x
10101		ADC5	ADC1	1x
10110		ADC6	ADC1	1x
10111		ADC7	ADC1	1x
11000		ADC0	ADC2	1x
11001		ADC1	ADC2	1x
11010		ADC2	ADC2	1x
11011		ADC3	ADC2	1x
11100		ADC4	ADC2	1x

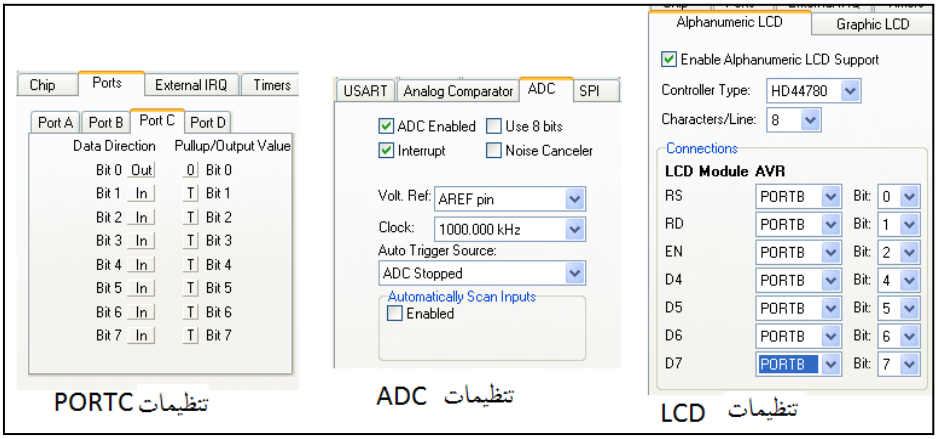
جدول ۶-۳

حال برای روشن‌تر شدن موضوع و آشنایی با مفاهیم کاربردی به مثال زیر توجه کنید:

مثال ۴: می‌خواهیم برنامه‌ای بنویسیم که در آن وقفه‌ی ADC فعال باشد و ولتاژ پایه ۳ مربوط به ADC را نمونه‌برداری‌کند، اگر این ولتاژ از ۰٫۴۹ولت بیشتر باشد LED متصل به پایه‌ی C0

روشن‌شود و همچنین عدد دیجیتال ولتاژ بر روی LCD متصل به PORTB نمـایش داده شـود(ADC به صورت ۱۰بیتی تنظیم می‌شود و مرجع مقایسه AREF به ولتاژ ۵ولت متصل شده است و ولتاژ ۰,۴۹ولت در این حالت معادل عدد ۱۰۰می‌شود)

تنظیمات کدویزارد:



شکل ۶-۲۲ : تنظیمات مثال ۴

حال کد زیر را درون وقفه‌ی ایجاد شده می‌نویسیم:

```
// ADC interrupt service routine
interrupt [ADC_INT] void adc_isr(void)
{
unsigned int adc_data;
char p[10];
// Read the AD conversion result

adc_data=ADCW;

// Place your code here
lcd_gotoxy(0,0);
if(adc_data>100){
sprintf(p,"%d",adc_data);
lcd_puts(p);
delay_ms(1000);
PORTC.0=1;
ADCSRA=ADCSRA| 0x40;
   }
}
```

ایجاد حالت نمونه‌برداری مجدد

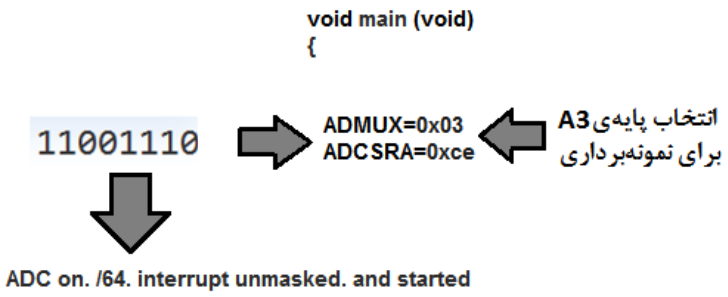
درون وقفه‌ی ADC مقدار نمونـه بـرداری شـده در داخـل متغیـر صـحیح ADC_data ریخته می‌شود، همچنین با دستور ADCSRA=ADCSRA|0x40 یک حالت آغاز نمونه- بـرداری را

ایجاد می‌کنیم و در نهایت دستور ADCSRA=ADCSRA|0x40 بیت‌های ADCSRA را با عدد هگزادسیمال 0x40، OR می‌کند. مطابق شکل زیر:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
				OR				
0x40	0	1	0	0	0	0	0	0

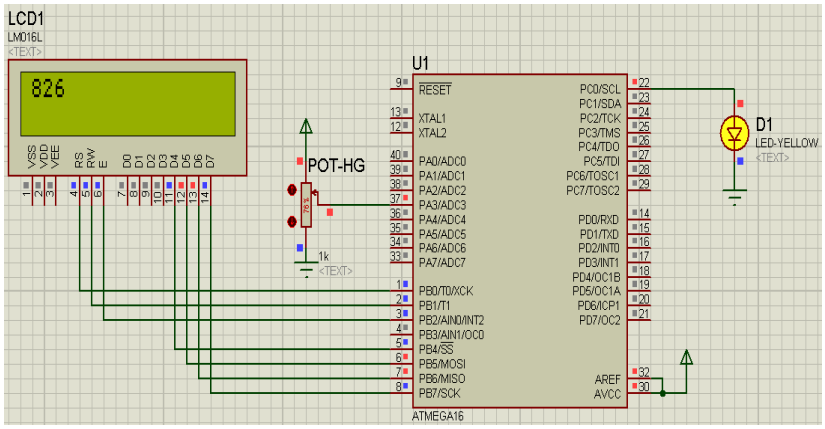
شکل ۶-۲۳ :آغاز حالت نمونه‌برداری

بعد از نوشتن دستورات مورد نظر درون تابع وقفه باید در داخل تابع void هم رجیسترهای ADCMUX و ADCSRA را تنظیم کنیم:



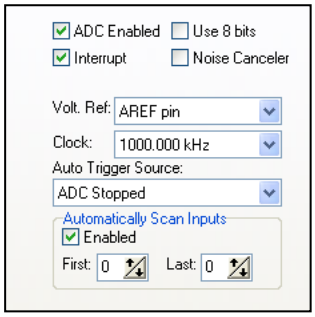
شکل ۶-۲۴ : انتخاب پایه A3 برای نمونه‌برداری و ایجاد حالت نمونه‌برداری جدید

برای آنکه متوجه شوید رجیستر ADCSRA چگونه تنظیم شده است به جدول رجیسترها که در بالا گفته شد مراجعه نمائید.شبیه‌سازی این مثال در پروتئوس:



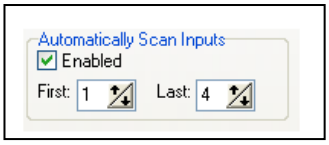
در بخش شبیه سازی مطابق شکل فوق با تغییر مقاومت متغیر که بین ۰ تا ۵ولت تغییر می‌کند مقدار ولتاژ نیز بر روی LCD تغییرکرده و تا زمانی‌که ولتاژ بزرگتر از ۰,۴۹ولت باشد(عدد دیجیتال ۱۰۰) LED روشن باقی می‌ماند.

بخش تنظیمات Automatically Scan Inputs واحد ADC



کاربرد دیگر بخش وقفه‌ها نمونه‌برداری خودکار هم زمان پایه‌های ADC می‌باشد که مطابق شکل زیر با زدن تیک گزینه‌ی Enable این بخش فعال می‌شود (مطابق شکل روبرو):

در این بخش دو گزینه‌ی First و Last مشخص می‌کنند که نمونه برداری از پایه First تا پایه‌ی Last انجام گیرد. مطابق شکل زیر با انجام این تنظیمات نمونه برداری از پایه‌ی ۱ تا ۴ ADC به صورت هم‌زمان انجام می‌گیرد.



با ایجاد این تنظیمات تابعی به شکل زیر ایجاد می‌گردد که مقادیر نمونه برداری را در داخل آرایه‌ای قرار می‌دهد که کاربر می‌تواند از آن استفاده کند:

```
#include <delay.h>

#define FIRST_ADC_INPUT 1
#define LAST_ADC_INPUT 4
unsigned int adc_data[LAST_ADC_INPUT-FIRST_ADC_INPUT+1];
#define ADC_VREF_TYPE 0x00
```

آرایه‌ای که برای استفاده در اختیار کاربر قرار می‌گیرد

```
// ADC interrupt service routine
// with auto input scanning
interrupt [ADC_INT] void adc_isr(void)
{
static unsigned char input_index=0;
// Read the AD conversion result
adc_data[input_index]=ADCW;
// Select next ADC input
if (++input_index > (LAST_ADC_INPUT-FIRST_ADC_INPUT))
   input_index=0;
ADMUX=(FIRST_ADC_INPUT | (ADC_VREF_TYPE & 0xff))+input_index;
// Delay needed for the stabilization of the ADC input voltage
delay_us(10);
// Start the AD conversion
ADCSRA|=0x40;
}
```

تابع مربوط به وقفه‌ی نمونه‌برداری هم زمان پایه‌ها (که در اینجا پایه‌های ۱ تا ۴ نمونه برداری می‌شوند)

آرایه‌ای که در اختیار کاربر قرار می‌گیرد به صورت زیر می‌باشد:

ADC_data[LAST_ADC_INPUT-FIRST_ADC_INPUT+1]

اعداد نمونه‌برداری شده در داخل این آرایه قرار می‌گیرد و کاربر می‌تواند از آنها استفاده‌کند.

فصل هفتم

تایمر / کانتر

در این فصل خواهیم خواند:

۱. تایمر / کانتر صفر
۲. مد CTC در تایمر صفر
۳. وقفه‌ی Compare Match
۴. تایمر / کانتر یک
۵. وقفه‌ی Compare Match در تایمر یک
۶. مد CTC در تایمر یک
۷. بررسی حالت تولید موج در مد CTC
۸. مد CTC Top=ICRIA
۹. فرکانس شکل موج در حالت CTC
۱۰. وقفه‌های تایمر یک
۱۱. وقفه‌ی Input Capture
۱۲. به روزرسانی مدهای تایمر
۱۳. تایمر کانتر دو
۱۴. مدهای PWM
۱۵. مد Fast PWM و Phase Correct PWM
۱۶. محاسبه فرکانس Phase Correct PWM
۱۷. مدهای PWM در تایمر کانتر یک
۱۸. Phase Correct PWMو فرکانس آن
۱۹. مد Fast PWM و فرکانس آن در تایمر یک
۲۰. به‌روز رسانی در مدهای مختلف PWM و تفاوت مدهای PWM
۲۱. PWM کاربردی و در ایور L298
۲۲. واحد سگ نگهبان یا WatchDog

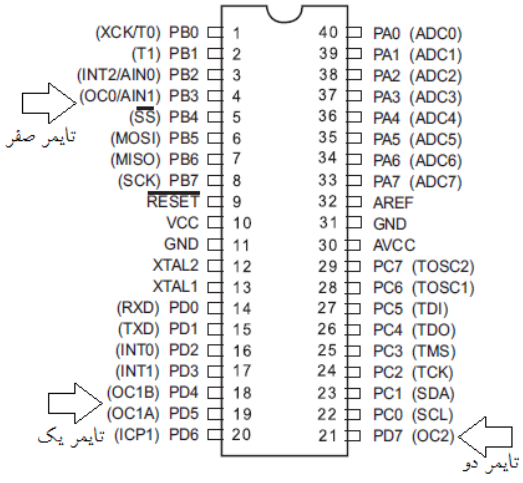
تایمر / کانتر (Timer / Counter)

تایمر/کانتر یکی از مهمترین بخش‌های هر میکروکنترلر است که مهم‌ترین وظیفه‌ی آن شمارش زمان و تولید شکل موج است.

فرض کنید باید در طول روز ده‌ها کار را در زمان‌های مختلف انجام بدهید، برای آنکه تمرکزتان فقط متوجه کاری که باید انجام می‌دهید باشد، کار محاسبه‌ی زمان و یادآوری کارهایتان سر زمان مقرر را به ساعت خود محول می‌کنید که در زمان‌های مربوط به هر کار شما را مطلع سازد، در این صورت دیگر شما هیچ کاری با زمان و نحوه‌ی شمارش آن ندارید و تمام این کارها را ساعت شما انجام می‌دهد.

در میکروکنترلرها نیز وظیفه‌ی محاسبه‌ی زمان و یادآوری زمان اجرای بعضی توابع بر عهده‌ی واحدTimer / Counter است(البته تایمر/ کانتر وظیفه‌های دیگری هم غیر از شمارش و تولید موج دارد مانند محاسبه‌ی فرکانس امواج، نمونه‌برداری از سیگنال ورودی و... که در ادامه به طور کامل توضیح داده خواهد شد). در میکروکنترلرهای AVR فرکانس‌کاری واحد تایمر/کانتر می‌تواند جدا از فرکانس‌کاری CPU تنظیم شود.

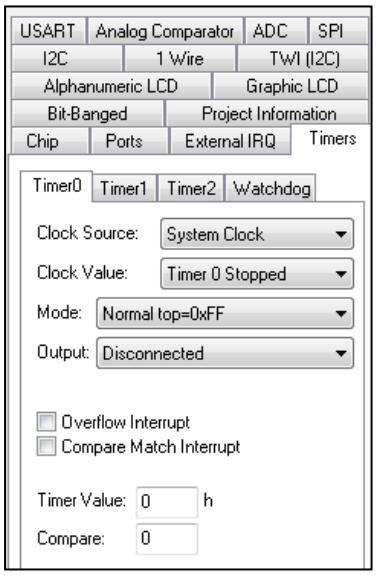
اگر بخواهیم به‌عنوان ساده‌ترین مثال یکی از کارهایی که توسط واحد تایمر میکروکنترلر به‌راحتی قابل اجرا می‌باشد را معرفی کنیم این است که کد نوشته شده بر روی میکرو به صورت پی‌درپی پس از یک زمان مشخص وارد یک تابع(وقفه) شود و دستورات درون آن‌را اجرا کند، برای مثال فرض کنید بر روی میکروکنترلر Atmega16 برنامه‌ای نوشته‌اید و می‌خواهید زمانی‌که این وسیله در حال خواندن ولتاژ از یکی از پایه‌های ADC می‌باشد به صورت همزمان و هرثانیه یک‌بار، یک LED کوچک به صورت چشمک‌زن را خاموش و روشن کند، همانطور که می‌دانید برای اینکار نمی‌توان از تابع delay استفاده کرد چراکه با این دستور کل خطوط برنامه در خط دستور تاخیر(delay) به مدت زمان دستور تاخیر(برای مثال یک ثانیه=(delay_ms(1000) متوقف می‌شود و هیچ خطی از برنامه اجرا نمی‌شود، در این حالت حتما به یک واحد نیاز داریم که جدا از خطوط اصلی برنامه گذر زمان را محاسبه کند تا تداخلی در اجرای خطوط برنامه ایجاد نشود و این



شکل ۷ – ۱ : پایه‌های تایمر در میکروکنترلر

وظیفه به عهده‌ی بخش تایمر میکروکنترلر می‌باشد.در میکروکنترلرهای سری ATmega16 ،چهار پایه مربوط به واحد Timer/Counter میکروکنترلر می‌باشند که این چهار پایه را در شکل قبل مشاهده کردیم.

برای استفاده از واحد تایمر/ کانتر در AVR و تنظیمات مربوط به آن ابتدا پنجره‌ی کدویزارد را باز می‌کنیم و بر روی تب TIMER کلیک می‌کنیم:

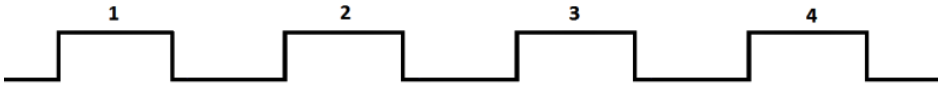


میکروکنترلرهای Atmega16 دارای ۳ واحد تایمر/کانتر می‌باشند که عبارتند از: تایمر صفر، تایمر۱ و تایمر۲، که توضیحات را ابتدا از تایمر صفر شروع می‌کنیم.

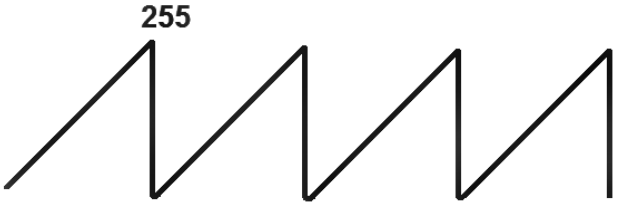
تایمر صفر (Timer 0)

تایمر صفر ساده‌ترین واحد تایمر این خانواده از میکروکنترلرها می‌باشد. این تایمر ۸ بیتی بوده و دارای چهار مد و یک خروجی می‌باشد و خروجی مربوط به این تایمر پایه‌ی PB3 یا OC0 در میکروکنترلر ATmega16 می‌باشد (که در ادامه در مورد خروجی تایمرها صحبت خواهیم کرد).

تایمر صفر یک تایمر ۸ بیتی است، یعنی می‌تواند از ۰ تا ۲۵۵ را شمارش کند و سرعت این شمارش نیز به فرکانس آن بستگی دارد. برای مثال فرض کنید که فرکانس تایمر را بر روی یک مگاهرتز تنظیم کرده‌ایم، همانطور که می‌دانید فرکانس با معکوس دوره‌ی تناوب برابر است ($T = \frac{1}{f}$) و زمانی‌که فرکانس برابر 1MHz باشد دوره‌ی تناوب برابر $1\mu s = \frac{1}{1MHz}$ می‌شود و این بدین معناست که موج مربعی تولید شده هر یک میکروثانیه صفر و یک می‌شود که هربار یک شدن این موج مربعی را یک کلاک می نامند:



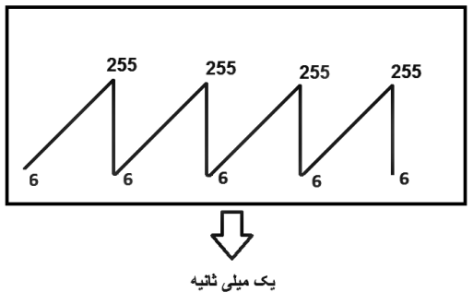
کلاک معیاری برای انجام هرگونه عملیاتی است به‌طوری که CPU هر دستورالعمل را در یک کلاک انجام می‌دهد پس زمانیکه فرکانس تایمر را برابر یک مگاهرتز قرار می‌دهیم هر کلاک در یک میکروثانیه زده می‌شود، بنابراین هر شماره در یک میکروثانیه شمارش می‌شود، پس زمانیکه می‌گوییم تایمر صفر یک شمارنده‌ی ۸ بیتی است و از ۰ تا ۲۵۵ را می‌تواند شمارش کند هر شماره را در یک کلاک انجام می‌دهد پس در این فرکانس از ۰ تا ۲۵۵ را در ۲۵۵ میکروثانیه می‌شمارد.
این شمارش در یک حافظه‌ی ۸ بیتی انجام می‌شود (یعنی این حافظه می‌تواند عددی بین ۰ تا ۲۵۵ را درون خود ذخیره کند) به‌طوری که مقدار اولیه‌ی این حافظه صفر است و با کلاک اول عدد یک درون این حافظه ذخیره می‌شود و با کلاک بعدی عدد دو و این عمل تا ۲۵۵ کلاک ادامه دارد که با کلاک آخر مقدار این حافظه ۲۵۵ می‌شود و با کلاک بعدی مقدار این حافظه دوباره صفر شده و شمارش از اول شروع می‌شود به این اتفاق که مقدار شمارش به حداکثر خود می‌رسد و از حداکثر به صفر بازمی‌گردد، سرریز (Over flow) می‌گویند و همچنین این حافظه‌های موقت را رجیستر یا ثبات می‌نامند.
نام این رجیستر که مقدار شمارش تایمر را در هر لحظه درون خود نگه می‌دارد TCNTn است (که n شماره‌ی تایمر است) که در تایمر صفر نام این رجیستر TCNT0 است.
همانطور که گفته شد تایمر صفر یک تایمر ۸ بیتی است و حداکثر می‌تواند تا ۲۵۵ را شمارش کند و این شمارش درون یک رجیستر (حافظه‌ی موقت) به نام TCNT0 ذخیره می‌شود، شکل زیر را در نظر بگیرید که این رجیستر از مقدار صفر شروع می‌کند و تا ۲۵۵ افزایش می‌یابد و سپس سرریز اتفاق افتاده و رجیستر دوباره صفر می‌شود، این روند به صورت متناوب تکرار می‌شود:



شکل ۷-۲: شمارش تایمر

یکی از خصوصیات تایمر/ کانتر این است که لحظه‌ای که سرریز اتفاق می‌افتد را خبر می‌دهد و با استفاده از این ویژگی ما می‌توانیم با فعال کردن وقفه‌ی مربوط به سرریز کاری کنیم که هر زمان سرریز اتفاق افتاد برنامه وارد وقفه بشود و دستورات درون آن را اجرا کند.
در فرکانس 1MHz هر سرریز در ۲۵۶ میکروثانیه اتفاق می‌افتد(زیرا مقدار رجیستر در ۲۵۵ کلاک از صفر تا ۲۵۵ افزایش می‌یابد و با زدن کلاک بعدی (کلاک ۲۵۶ ام) سرریز اتفاق می‌افتد).
در محاسبه‌ی بعضی زمان‌های خاص ممکن است کار با عددی مانند ۲۵۶ کمی دشوار باشد، برای مثال اگر بخواهیم زمان یک میلی‌ثانیه را تولید کنیم باید تقریبا ۳,۹ بار سرریز رخ‌دهد ($256\mu s\times$

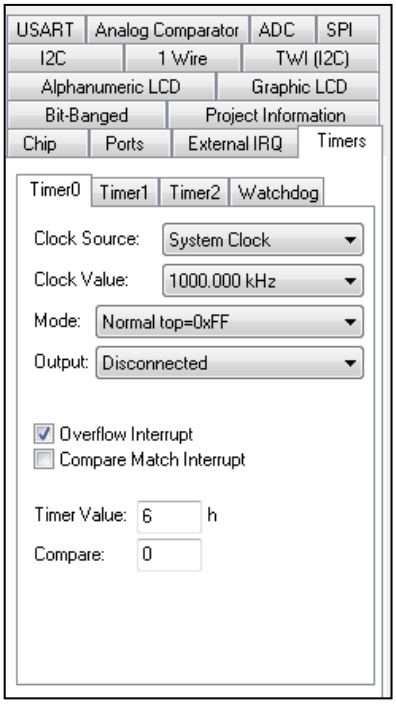
3.90625 = 1ms)، ولی اگر این عدد به جای ۲۵۶، عدد ۲۵۰ باشد با چهار بار سرریز (چهار بار شمارش از ۰ تا ۲۵۰) زمان یک میلی‌ثانیه تولید می‌شود. برای آنکه تایمر در هر سرریز به جای ۲۵۶ کلاک، ۲۵۰ کلاک بزند می‌توانیم مقدار اولیه‌ی رجیستر TCNT0 را به جای صفر عدد ۶ انتخاب کنیم:



شکل ۷ - ۳

مثال ۱: حال برنامه‌ی ساده‌ای می‌نویسیم که میکروکنترلر توسط واحد تایمر صفر هر ثانیه یک‌بار، یک LED را روشن و خاموش کند:

برای شروع بر روی تب TIMER کلیک کرده و مطابق شکل وارد تایمر صفر می‌شویم:



شکل ۷ - ۴: تنظیمات مثال ۱

اولین قسمت که Clock source نام دارد مشخص می‌کند که منبع کلاک از کجا تامین می‌شود. منبع کلاک می‌تواند توسط خود میکرو یا توسط یک منبع بیرونی تامین شود (یعنی یک موج مربعی به‌عنوان کلاک به یکی از پایه‌های میکرو وارد شود و میکرو هربار یک یا صفر شدن این منبع خارجی را به‌عنوان یک کلاک در نظر بگیرد). در این مثال منبع کلاک را در حالت کلاک داخلی (System Clock) قرار می‌دهیم(دو حالت مربوط به منبع بیرونی در قسمت تایمر یک توضیح داده خواهد شد).

در قسمت clock value فرکانس تایمر را انتخاب می‌کنیم که در این مثال فرکانس یک مگاهرتز انتخاب شده است.

در قسمت Mode چهار حالت وجود دارد که دو حالت مربوط به PWM و تولید موج مربعی بر روی پایه‌های خروجی است(که در آخر مبحث تایمرها مفصل توضیح داده می‌شود) و دو حالت دیگر به‌عنوان شمارش‌کننده و محاسبه‌ی زمان به کار می‌روند(البته این دو حالت کاربردهای دیگری هم دارند که در ادامه بیشتر با آن آشنا می‌شویم).

ابتدا در مد Normal کار می‌کنیم و در ادامه مد CTC را بررسی می‌کنیم. مقداری که به‌عنوان top در مد نرمال نوشته شده است یعنی (top = 0xFF) به این معناست که حداکثر شمارش این مد تایمر تا عدد ۲۵۵ است و بعد از آن سرریز اتفاق می‌افتد و دوباره شمارش را از صفر شروع می‌کند.

در قسمت Output با هربار سرریز، می‌تواند مقدار پایه‌ی خروجی تغییر کند که در این مثال چون هدف محاسبه‌ی زمان است و نیازی به خروجی نداریم آن را روی حالت Disconnected قرار می‌دهیم.

با زدن گزینه‌ی Overflow Interrupt بعد از هر سرریز، برنامه به وقفه‌ی تایمر می‌رود و دستورات درون آن را اجرا می‌کند.

در قسمت Time Value می‌توانیم مقدار اولیه‌ی تایمر را مشخص کنیم، نکته‌ای که باید به آن توجه کنیم این است که این عدد در مبنای ۱۶(هگزا دسیمال) است و عددی که در آن می‌نویسیم باید به هگزادسیمال باشد، در این مثال این مقدار را برابر ۶ قرار می‌دهیم که رجیستر TCNT0 از مقدار اولیه‌ی ۶ شروع شود (عدد ۶ در مبنای ۱۶ همان ۶ است). اکنون تنظیمات قسمت تایمر در مثال۱ به پایان رسیده است و در بخش تنظیمات واحد ورودی-خروجی میکروکنترلر، پایه‌ی A0 را به صورت خروجی تنظیم می‌کنیم که آن را به پایه‌ی LED متصل کنیم.

اکنون کد را generate می‌کنیم. کد مربوط به تنظیمات رجیسترهای تایمر صفر در قسمتی از کد که توسط کدویزارد با خطوط آبی رنگ(به صورت کامنت) تولید شده است، قرار دارد:

```
34  // Timer/Counter 0 initialization
35  // Clock source: System Clock
36  // Clock value: Timer 0 Stopped
37  // Mode: Normal top=0xFF
38  // OC0 output: Disconnected
39  TCCR0=0x00;
40  TCNT0=0x06;
41  OCR0=0x00;
```

به کد TCNT0=0x06 توجه کنید، این کد به دلیل اینکه مقدارTime Value را عدد ۶ انتخاب کرده‌ایم تولید شده است، اگر مقدار Time Value را خالی می‌گذاشتیم کد TCNT0=0x00 تولید می‌شد که می‌توانستیم آن را تغییر بدهیم و خودمان دستور TCNT0=0x06 را بنویسیم.

```
// Timer 0 overflow interrupt service routine
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
// Reinitialize Timer 0 value
TCNT0=0x06;
// Place your code here

}
```

چون وقفه‌ی تایمر را فعال کردیم کد مربوط به وقفه‌ی تایمر ایجاد می‌شود:

می‌دانیم تایمر هر ۲۵۰ میکروثانیه یک‌بار وارد وقفه می‌شود، حال اگر بخواهیم زمان یک میلی‌ثانیه را حساب کنیم می‌توانیم هربار که برنامه وارد وقفه شد یک واحد به یک متغیر مانند a اضافه کنیم؛ در اینصورت هر زمان مقدار این متغیر به عدد ۴ رسید زمان یک میلی‌ثانیه گذشته است.برای مثال کد روبرو را در نظر بگیرید:

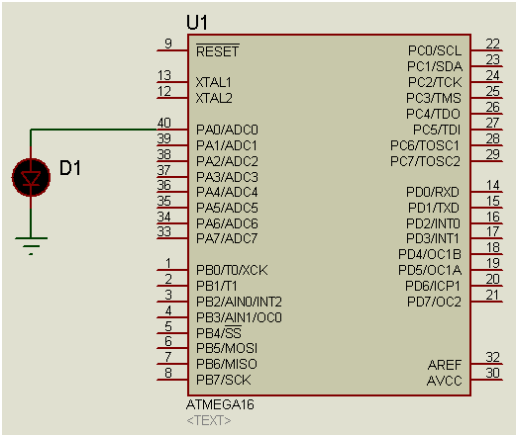
```
int a=0,time=0;
// Timer 0 overflow interrupt service routine
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
 a++;
 if(a==4){
 time++;
 a=0;
 }

TCNT0=0x06;
}
```

در این کد هربار که برنامه وارد وقفه شود یک واحد به متغیر a اضافه می‌شود و هر زمان که متغیر a به عدد ۴ برسد، یک میلی‌ثانیه زمان گذشته است پس یک متغیر به نام time با مقدار اولیه‌ی صفر تعریف می‌کنیم و هربار که مقدار متغیر a به عدد ۴ رسید یک واحد به time اضافه می‌کنیم(که مقدار time در حقیقت زمان گذشته شده بر حسب میلی‌ثانیه است). در انتهای کد که در وقفه می‌نویسیم دوباره مقدار اولیه‌ی Timer را عدد ۶ (TCNT0=0x06) قرار می‌دهیم (چون اگر این کار را نکنیم بعد از سرریز دوباره مقدار آن به صورت پیش فرض صفر می‌شود). حال اگر بخواهیم هر یک ثانیه یک‌بار LED روشن و خاموش شود می‌توانیم کد روبرو را به آن اضافه کنیم:

```
int a=0,time=0;
// Timer 0 overflow interrupt service routine
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
 a++;
 if(a==4){
 time++;
 a=0;
 }

 if(time==1000) PORTA.0=1;
 if(time==2000) {
 PORTA.0=0;
 time=0;
 }

TCNT0=0x06;
}
```

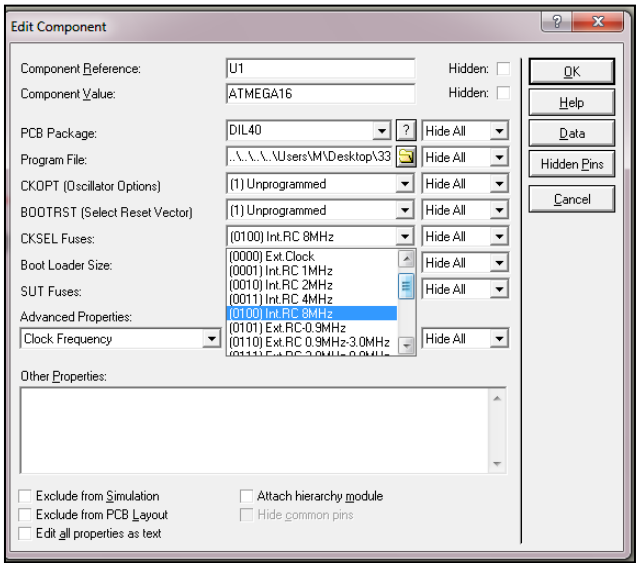
این کد هر هزار میلی‌ثانیه یک‌بار PORTA.0 را صفر و یک می‌کند.

حال اگر بخواهیم اجرای این برنامه در شبیه‌ساز را نیز مشاهده کنیم برنامه‌ی پروتئوس را باز می‌کنیم و نحوه‌ی عملکرد این کد را مشاهده می‌کنیم:



شکل ۷ - ۵

باز هم به این نکته توجه کنید که فرکانس تایمر می‌تواند با فرکانس میکروکنترلر متفاوت باشد و در تنظیمات میکروکنترلر فرکانس‌کاری CPU را انتخاب می‌کنیم نه فرکانس تایمر (که در این مثال ۸ مگاهرتز بود).



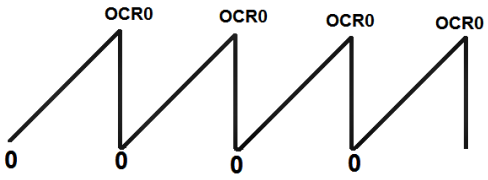
شکل ۷ - ۶: تنظیم فرکانس میکروکنترلر

اگر برنامه را اجرا کنیم می‌بینیم که چراغ چشمک‌زن هر یک ثانیه خاموش و روشن می‌شود.

تا به اینجا با تایمر صفرآشنا شدیم و توانستیم زمان‌های مختلف را با آن ایجاد کنیم و در زمان‌های دلخواه با فعال کردن وقفه‌ی تایمر کدهای دلخواهی را به اجرا درآوریم.

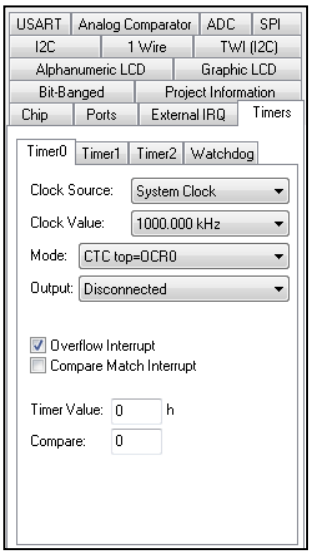
مد CTC در تایمر صفر

مد CTC (Clear Timer on Compare Match)، مد مربوط به مقایسه می‌باشد به این معنی که در این مد مقدار تایمر (یا به طور دقیق‌تر مقدار رجیستر TCNT0) به طور دائم با یک رجیستر به نام OCR0 مقایسه می‌شود و به محض برابری، مقدار TCNT0 صفر شده و دوباره شروع به افزایش می‌کند یا به عبارت دیگر در این مد مقدار تایمر فقط از صفر تا OCR0 افزایش می‌یابد. به یاد داریم که در مد Normal مقدار رجیستر تایمر از صفر (یا هر مقداری که در برنامه بنویسیم) شروع می‌شود و زمانی‌که به ۲۵۵ برسد سرریز رخ می‌دهد ولی در این مد زمانی‌که به مقدار OCR0 برسد مقدار تایمر سرریز می‌شود و به صفر بازمی‌گردد:



شکل ۷ – ۷ : سرریز تایمر هنگام رسیدن به OCR0

در مد Normal برای محاسبه‌ی بعضی زمان‌ها که احتیاج داشتیم تایمر ۲۵۰ بار کلاک بزند مقدار اولیه‌ی رجیستر TCNT0 را به جای صفر برابر عدد ۶ قرار می‌دادیم ولی در مد CTC تایمر از صفر شروع می‌شود و تا مقدار OCR0 افزایش می‌یابد که OCR0 را می‌توانیم در برنامه مقداردهی کنیم.فقط به یک نکته که باید توجه کنیم این است که در صورت فعال کردن وقفه‌ی سرریز در این مد مقدار TCNT0 از صفر تا OCR0 افزایش می‌یابد ولی زمانیکه به OCR0 رسید وقفه رخ نمی‌دهد و زمانی‌که از OCR0 رد شود و به صفر بازگردد وقفه رخ می‌دهد، پس برای تولید ۲۵۰ کلاک مقدار OCR0 را برابر ۲۴۹ قرار می‌دهیم.



مثال ۲: حال برنامه‌ی مثال ۱ را این بار در مد CTC می‌نویسیم:

با تنظیمات شکل قبل کد زیرتولید می‌شود:

```
// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: 1000.000 kHz
// Mode: CTC top=OCR0
// OC0 output: Disconnected
TCCR0=0x0A;
TCNT0=0x00;
OCR0=0x00;
```

برای آنکه زمان یک ثانیه تولید شود احتیاج به گذشت ۴۰۰۰ بار زمان ۲۵۰ میکروثانیه است پس در فرکانس 1MHz لازم است که تایمر به جای ۲۵۶ کلاک فقط ۲۵۰ کلاک برای هر وقفه بزند، برای اینکه تایمر ۲۵۰ بار کلاک بزند، مقدار OCR0 را برابر ۲۴۹ قرار می‌دهیم که در مبنای ۱۶ (هگزادسیمال) عدد ۲۴۹ معادل F9 است، پس مقدار رجیستر OCR0 را برابر عدد F9 قرار می‌دهیم. برای انجام این کار در کد دستور OCR0=0xF9 را می‌نویسیم و یا در گزینه‌ی Compare در تنظیمات کدویزارد این مقدار را می‌نویسیم(نحوه‌ی تبدیل عدد در مبنای ۱۰ به هگزادسیمال در بخش ضمیمه آموزش اعداد هگز و باینری توضیح داده شده است):

```

// Timer/Counter 0 initialization
// Clock source: T0 pin Falling Edge
// Mode: Normal top=0xFF
// OC0 output: Disconnected
TCCR0=0x0A;
TCNT0=0x00;
OCR0=0xF9;

```

کد درون وقفه هم دقیقاً مانند کد قسمت قبل است با این تفاوت که دیگر نیازی به تکرار دستور OCR0=0xF9 در پایان کد وقفه نیست چون مقدار OCR0 یک‌بار برابر عدد ۲۴۹ قرار داده شده است و با سرریز شدن تایمر مقدار آن تغییر نمی‌کند:

```
// Timer 0 overflow interrupt service routine
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{

a++;

if (a==4){
time++;
a=0;
}


     if (time == 1000)
          PORTA.0=1;
            if (time == 2000){
          PORTA.0=0;
          time=0;
          }

}
```

سوالی که ممکن است پیش بیاید این است که برای این برنامه کدام حالت بهتر است مد Normal یا CTC؟

پاسخ CTC است چون زمانیکه از CTC استفاده می‌کنیم یک‌بار OCR0 را مقداردهی می‌کنیم و مقدار تایمر از صفر شروع می‌شود و تا مقدار OCR0 افزایش می‌یابد ولی در حالت Normal تایمر بعد از رسیدن به ۲۵۵ سرریز می‌شود و به صفر باز می‌گردد و سپس وارد کد شده و دستور TCNT0=0x05 را اجرا می‌کند و مقدار TCNT0 را از صفر به عدد ۶ تغییر می‌دهد. یک زمان بسیار اندک طول می‌کشد که تایمر خطوط برنامه را اجرا کند و رجیستر TCNT0 را از صفر به ۶ تغییردهد، در برنامه‌های بسیار حساس که به دقت بسیار بالایی احتیاج دارند این مد ممکن است کمی ایجاد خطا کند ولی در برنامه‌هایی که به همچین دقت زیادی احتیاج ندارند فرقی نمی‌کند که از کدام مد Normal یا CTC استفاده شود.

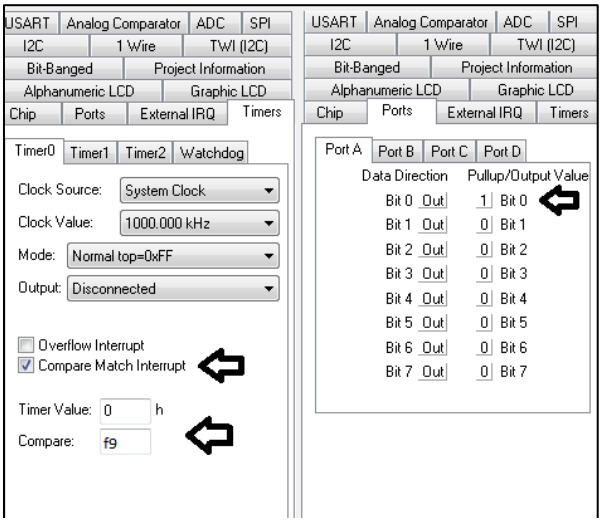
وقفه‌ی Compare Match

یکی دیگر از وقفه‌های تایمر مربوط به وقفه‌ی Compare Match می‌باشد که این وقفه زمانی رخ می‌دهد که مقدار رجیستر OCR0 با مقدار رجیستر TCNT0 برابر شود.

مثال ۳: برای مثال یک برنامه‌ی ساده می‌نویسیم که هفت LED را یکی یکی و با فاصله‌ی یک ثانیه روشن کند و هر LED که روشن شد LED قبلی خاموش شود:

برای این‌کار از وقفه‌ی Compare Match استفاده می‌کنیم به‌طوری که هر زمان مقدار TCNT0 با مقدار OCR0 برابر شد، برنامه وارد وقفه شود و به‌ترتیب LEDها را روشن کند. برای آنکه زمان یک ثانیه پدیدآید مقدار OCR0 را برابر ۲۴۹ قرار می‌دهیم و با هربار وارد وقفه شدن یک واحد به متغیری مانند a اضافه می‌کنیم و زمانی‌که مقدار این متغیر به عدد ۴۰۰۰ رسید زمان یک ثانیه تولید شده است لذا می‌توانیم LEDها را یکی یکی روشن کنیم.

تنظیمات مربوط به کدویژن مانند شکل زیر انجام می‌شود:



شکل ۷ - ۸ : تنظیمات مثال ۳

برای مقداردهی به متغیر OCR0 هم می‌توانستیم مانند مثال پیش در برنامه دستور OCR0=0xF9 را بنویسیم و هم در همین صفحه در قسمتCompare آن را مقداردهی کنیم، PORTA را هم به صورت خروجی انتخاب می‌کنیم که پایه‌های LED به آن‌ها متصل شوند، فقط PORTA.0 را با مقدار اولیه‌ی یک انتخاب می‌کنیم که در شروع برنامه اولین LED روشن باشد.

کد را به صورت روبرو می‌نویسیم:

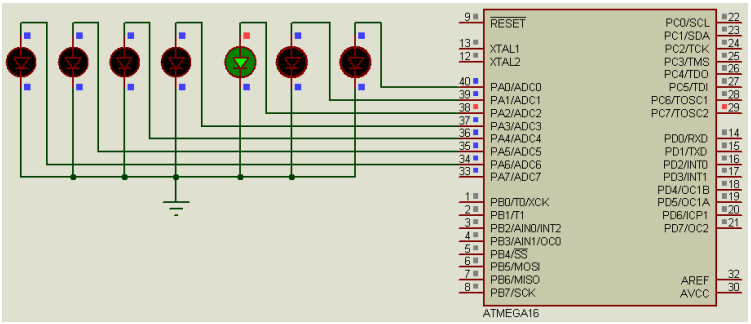
```
int a=0,b=0;
// Timer 0 output compare interrupt service routine
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
a++;
if(a==4000){
 PORTA*=2;
 a=0;
 b++;}
if(b==7) {
 PORTA=1;
 b=0;}

}
```

در برنامه‌ی فوق هر زمان که متغیر a به مقدار ۴۰۰۰ رسید PORTA در عدد ۲ ضرب می‌شود (PORTA*=2 معادل است با PORTA = PORTA *2) اگر از فصل PIN و PORT به یاد داشته باشید زمانی‌که می‌گوییم PORTA=2 به این معناست که:

PORTA = 00000010 (عدد ۲ در مبنای دو به این صورت می‌شود) و دومین خروجی یعنی PORTA.1 برابر یک و دیگر خروجی‌ها صفر هستند و زمانیکه این عدد را در ۲ ضرب می‌کنیم به

۴ تبدیل می‌شود، یعنی PORTA = 00000100 که به این معناست که PORTA.2=1 و باقی خروجی‌ها برابر صفر هستند، پس با ضرب در ۲ کردن این عدد انگار ولتاژ، یکی یکی به پایه‌های کناری منتقل می‌شود، متغیر b هم هر دفعه یک واحد به مقدارش اضافه می‌شود و زمانی‌که برابر ۷ شد یعنی اینکه هر۷ LED روشن شده‌اند و دوباره باید این برنامه از اول، LED شماره یک را روشن کند(اگرمقدار اولیه‌ی اولین خروجی (یعنی PORTA.0) را برابر ۱ قرار نمی‌دادیم و همه‌ی خروجی‌ها را برابر صفر قرار می‌دادیم (یعنی PORTA=0) با ضرب در دو کردن PORTA مقدار آن دوباره برابر صفر می‌شد و برنامه به این شکل اجرا نمی‌شد). برای تمرین می‌توانید همین برنامه را به صورت عادی و بدون ضرب در دو کردن و با گذاشتن شرط‌های متعدد بنویسید.

در شکل زیر مدار بسته شده در پروتئوس را مشاهده می‌کنیم:

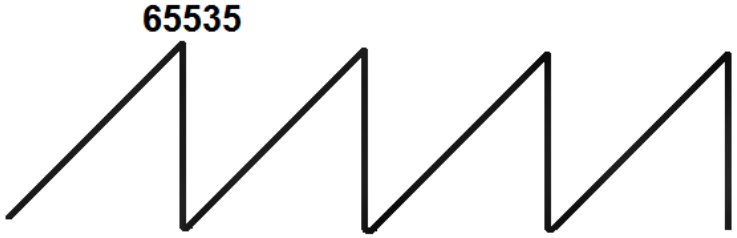


شکل ۷-۹ : مدار مثال ۳

تایمر / کانتر ۱

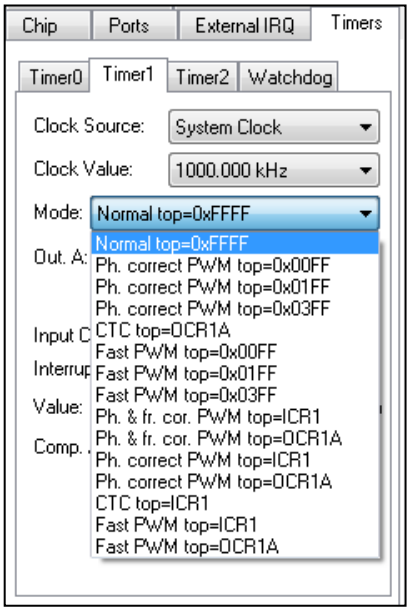
تایمر ۱ دقت بسیار بالاتری نسبت به دو تایمر دیگردارد، چراکه این تایمر ۱۶ بیتی است. پایه‌های مربوط به خروجی این تایمر پایه‌های PD4 و PD5 هستند که پایه‌یPD4 مربوط به خروجی OCR1B و پایه‌یPD5 مربوط به خروجی OCR1A می‌باشد.

تایمر صفر ۸ بیتی بود و می‌توانست از ۰ تا ۲۵۵ را شمارش کند ولی تایمر یک ۱۶ بیتی است و می‌تواند از ۰ تا ۶۵۵۳۵ را شمارش کند.



شکل ۷ - ۱۰ : شمارش تایمر یک (از صفر تا ۶۵۵۳۵)

اگر وارد تب TIMER1 شویم و نگاهی به مدهای این تایمر بیاندازیم، مشاهده می‌کنیم که این تایمر دارای یک مد Normal می‌باشد که ۱۶ بیتی است و می‌تواند تا ۶۵۵۳۵ را شمارش کند و نیز دارای چندین مد مربوط به تولید موج مربعی PWM و دو مد مربوط به CTC می‌باشد.



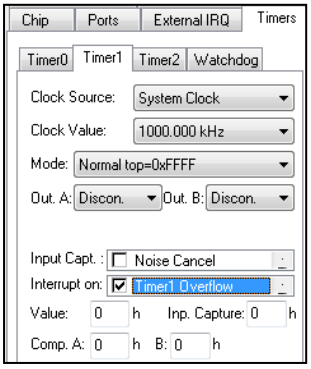
مد نرمال (Normal)

مقدار حداکثر مد نرمال(Normal)، 0xFFFF است یعنی این تایمر از صفر شروع به شمردن می‌کند و تا ۶۵۵۳۵ ادامه می‌دهد و سپس سرریز رخ می‌دهد.

همانطور که می‌دانید میکروکنترلرهای AVR ۸ بیتی هستند پس برای نمایش یک عدد ۱۶ بیتی که مربوط به تایمر یک است نمی‌توان از یک حافظه‌ی ۸ بیتی استفاده کرد به همین دلیل برای ذخیره‌ی عدد شمارش شده باید از دو عدد حافظه‌ی موقت (رجیستر) استفاده کرد که دو رجیستر TCNT1H و TCNT1L وظیفه‌ی نگهداری مقدار تایمر را بر عهده دارند به این گونه که یک عدد ۱۶ بیتی به دو قسمت ۸ بیت پرارزش و ۸ بیت کم‌ارزش تقسیم می‌شود و ۸ بیت پرارزش درون رجیستر TCNT1Hو ۸ بیت کم‌ارزش درون رجیستر TCNT1L ذخیره می‌شود و بعد از هر سرریز هر دو رجیستر برابر صفر می‌شوند(H انتهای TCNT1H به معنای High یا پرارزش و L انتهای TCNT1L به معنای Low یا کم ارزش می‌باشد). برای درک بهتر اصطلاح ارقام کم‌ارزش و پرارزش عدد ۲۹۴ را در نظر بگیرید، در این عدد رقم ۲(صدگان) پرارزش‌ترین رقم و رقم ۴(یکان) کم‌ارزش‌ترین رقم است و در عدد مبنای دو ۱۱۰۱۱۰۰۰،چهار بیت ۱۱۰۱ چهار بیت پرارزش و چهار بیت ۱۰۰۰ چهار بیت کم‌ارزش هستند.

مثال ۴: فرض کنید مثال مربوط به روشن و خاموش شدن یک LED هر یک ثانیه یک‌بار را در این تایمر بنویسیم:

ابتدا مدNormal را انتخاب می‌کنیم و مطابق شکل تیک مربوط به فعال شدن وقفه در هنگام سرریز را می‌زنیم(Timer 1 Overflow):



شکل ۷- ۱۱: تنظیمات مثال ۴

سپس PORTA.0 را به صورت خروجی تنظیم می‌کنیم و برنامه را Generate می‌کنیم. توسط کدویزارد کد زیر تولید می‌شود:

```
// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: 1000.000 kHz
// Mode: Normal top=0xFFFF
// OC1A output: Discon.
// OC1B output: Discon.
// Noise Canceler: Off
// Input Capture on Falling Edge
// Timer1 Overflow Interrupt: On
// Input Capture Interrupt: Off
// Compare A Match Interrupt: Off
// Compare B Match Interrupt: Off
TCCR1A=0x00;
TCCR1B=0x02;
TCNT1H=0x00;
TCNT1L=0x00;
ICR1H=0x00;
ICR1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
```

در مد نرمال مربوط به تایمر صفر برای سادگی در محاسبه‌ی بعضی از زمان‌ها در فرکانس یک مگاهرتز ترجیح بر این بود که این تایمر به جای ۲۵۵ کلاک، ۲۵۰ کلاک بزند و مقدار اولیه‌ی TCNT0 را به جای صفر عدد ۶ قرار می‌دادیم، در این تایمر نیز برای سادگی در محاسبه‌ی بعضی زمان‌ها در فرکانس یک مگاهرتز می‌توانیم مقدار اولیه را برابر ۱۵۵۳۶ قرار دهیم که (۵۰۰۰۰ =۱+ ۱۵۵۳۶- ۶۵۵۳۵ که ۱+ به دلیل آن است که سرریز زمانی رخ می‌دهد که مقدار تایمر از حداکثر

عبور کرده باشد) تایمر بعد از ۵۰۰۰۰ کلاک وارد وقفه شود که در فرکانس 1MHz زمـان رخـداد هر سرریز ۵۰ میلی‌ثانیه می‌شود که با ۲۰بار سرریز زمان یک ثانیـه تولیـد مـی‌شـود(عدد ۱۵۵۳۶ به‌عنوان یک مثال برای راحتی محاسبه بیان شد و این عدد هر عدد دیگری هم می‌توانست باشد). حال برای نوشتن برنامه عدد ۱۵۵۳۶ را به مبنای هگزادسیمال تبدیل می‌کنـیم کـه برابـر 3CB0 می‌شود پس باید این مقدار را در رجیستر TCNT1 بنویسیم و چون این عـدد ۱۶بیتـی اسـت ۸ بیت پرارزش آن را در TCNT1H و ۸ بیت کم‌ارزش آن را در TCNT1Lذخیره می‌کنیم(۸ بیت پر ارزش 3C و ۸ بیت کم ارزشB0 می‌باشد):

```
// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: 1000.000 kHz
// Mode: Normal top=0xFFFF
// OC1A output: Discon.
// OC1B output: Discon.
// Noise Canceler: Off
// Input Capture on Falling Edge
// Timer1 Overflow Interrupt: On
// Input Capture Interrupt: Off
// Compare A Match Interrupt: Off
// Compare B Match Interrupt: Off
TCCR1A=0x00;
TCCR1B=0x02;
TCNT1H=0x3C;
TCNT1L=0xB0;
ICR1H=0x00;
ICR1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
```

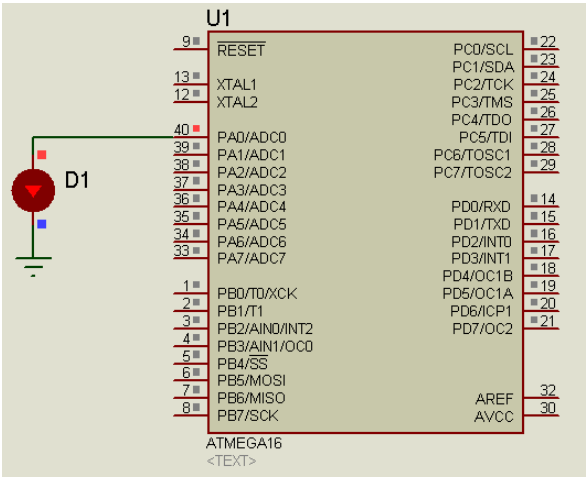
سپس در وقفه‌ی تایمر کد زیر را می‌نویسیم:

```
int a=0;
// Timer1 overflow interrupt service routine
interrupt [TIM1_OVF] void timer1_ovf_isr(void)
{
a++;
if(a==20) PORTA.0=1;
if(a==40) {
PORTA.0=0;
a=0;
}


TCNT1H=0x3C;
TCNT1L=0xB0;


}
```

در کد فوق هر ۵۰ میلی‌ثانیه یک‌بار تایمر وارد وقفه می‌شود و یک واحد به مقـدار متغیـر a اضـافه می‌شود و زمانی‌که این متغیر به عدد ۲۰ رسید به این معناست که زمان یک ثانیه گذشـته اسـت و PORTA.0 را یک می‌کنیم و زمانی‌که متغیر a به عدد ۴۰ رسید یک ثانیه‌ی دیگر گذشته اسـت

و PORTA.0 را صفر می‌کنیم و در انتهای کد وقفه، دوباره مقدار اولیه‌ی تایمر را برابر 3CB0 (یا همان ۱۵۵۳۶) قرار می‌دهیم.

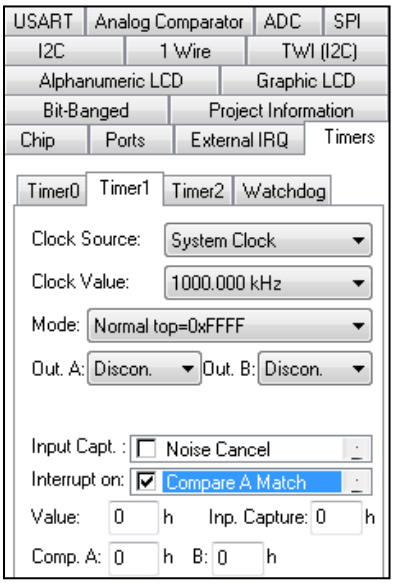
می‌توانیم مدار ساده‌ی آن را در پروتئوس ببندیم و چشمک زدن LED را مشاهده کنیم:



شکل ۷ - ۱۲

وقفه‌ی Compare Match در تایمر یک

همانطور که می‌دانید رجیستر TCNTn در تایمر یک، از مقدار صفر شروع می‌شود و تا مقدار حداکثر یعنی ۶۵۵۳۵ افزایش می‌یابد و سپس سرریز اتفاق افتاده و مجدداً شمارش از صفر آغاز می‌شود. در وقفه‌ی مربوط به Compare Match، مقدار رجیستر TCNTn دائماً با مقدار OCR1A یا OCR1B مقایسه می‌شود و در هنگام برابری وارد وقفه‌ی تایمر می‌شود.



این وقفه را می‌توانیم با زدن تیک Compare A match یا Compare B Match فعال کنیم:

اگر این وقفه را مانند شکل روبرو فعال کنیم زمانی که مقدار تایمر برابر OCR1A شد برنامه وارد وقفه می‌شود و دستورات درون آن را اجرا می‌کند، سپس مقدار تایمر افزایش می‌یابد و زمانی که به حداکثر رسید سرریز می‌شود و دوباره از صفر شروع به افزایش می‌کند و زمانی که با مقدار OCR1A برابر شد برنامه دوباره وارد وقفه می‌شود و این روند به همین شکل ادامه می‌یابد.

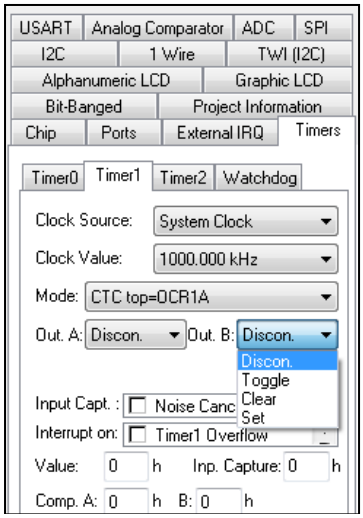
مد CTC در تایمر یک

تایمر یک دارای دو مد مقایسه(CTC) است، یک مد به صورت top=OCR1A است که حداکثر مقدار آن OCR1A می‌باشد و یک مد به صورت top=ICR1A است که حداکثر مقدار آنICR1A می‌باشد.

یک نکته‌ی بسیار مهم این است که در این مد مقدار تایمر از صفر شروع می‌شود و تا مقدار حداکثر (top) افزایش می‌یابد و سپس سرریز رخ می‌دهد و مقدار آن صفر می‌شود و مجددا افزایش می‌یابد ولی در این تایمر بر خلاف تایمر صفر زمانی‌که به top رسید، یا از آن عبور کرد با فعال کردن گزینه‌ی Timer 1 Overflowهیچ وقفه‌ای رخ نمی‌دهد مگر آنکه مقدار OCR1A برابر حداکثر (یعنی 0xFFFF یا ۶۵۵۳۵) باشد و برای آنکه در این مد و در هنگام برابری OCR1A با مقدار TCNTn وقفه رخ‌دهد باید از گزینه‌ی Compare A Match استفاده کنیم که در ادامه توضیح داده می‌شود.

بررسی حالت تولید شکل موج در مد CTC:

یکی از کاربردهای این مد تولید موج بر روی پایه‌های خروجی می‌باشد، شکل زیر انواع حالات پایه‌های خروجی تایمر یک را در این مد نشان می‌دهد:

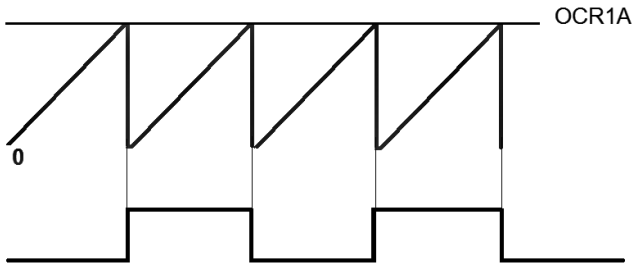


شکل ۷- ۱۳ : حالت‌های خروجی در مد CTC

در این مد هر خروجی دارای چهار حالت می‌باشد:

حالت اول (Disconnect) است که همانطور که از نامش پیداست به این معناست که بر روی خروجی هیچ سیگنالی قرار نگیرد.

حالت دوم (Toggle) به ایـن معناسـت کـه شـکل مـوج خروجـی در هربـار برابـری TCNT1 و OCR1A، Not شود (یعنی اگر صفر است یک بشود و اگر یک است صفر بشـود). بـه شـکل زیـر دقت کنید:



شکل ۷ - ۱۴ : حالت‌های خروجی در مد CTC

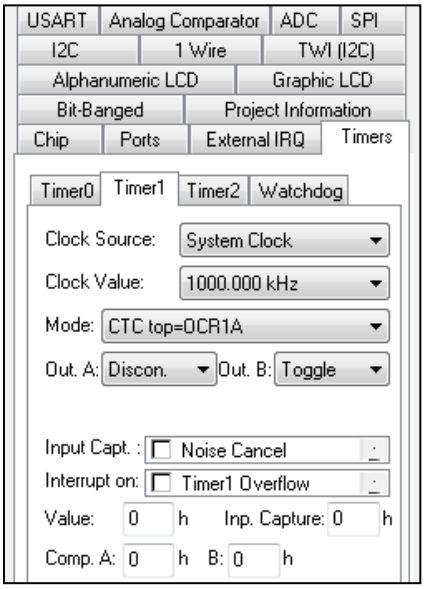
در لحظاتی که مقدار تایمر با مقدار OCR1A برابر می‌شود مقدار خروجی NOT می‌شود، کـه در اینصورت یک شکل موج مربعی ایجاد می‌شود.

مدClear به این معناست که زمانی‌که مقدار تایمر با مقدار OCR1A برابر شد خروجی برابر صـفر شود.

و مد Set به این معناست که زمانی‌که مقدار تایمر با مقدار OCR1A برابر شد خروجی برابر یـک شود.

مثال ۵: قصد داریم یک موج مربعی با دوره‌ی زمانی ۲۰۰ میکروثانیه تولید کنیم:

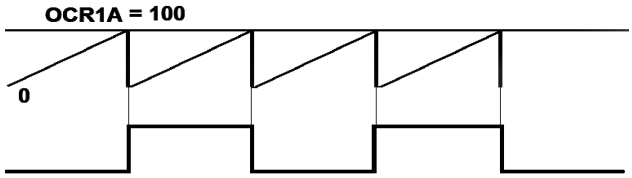
برای اینکار در تایمر یک مد CTC top=OCR1A را انتخاب می‌کنیم و خروجـی OCR1B را روی حالـت Toggle قـرار مـی‌دهـیم و خروجـی OCR1A را روی حالت Disconnect می‌گذاریم.



دوره‌ی زمـانی، ۲۰۰ میکروثانیـه اسـت یعنـی ۱۰۰ میکروثانیه یـک و ۱۰۰ میکروثانیـه صـفر و نحـوه‌ی عملکرد آن بدین‌ترتیب است که تایمر از مقدار صـفر به مدت ۱۰۰میکروثانیه طول میکشـد تـا بـه مقـدار OCR1A برسد که در ایـن زمـان مقـدار خروجـی برابر یک می‌شود و سپس مقدار تـایمر دوبـاره صـفر می‌گردد و دوباره بـه مـدت ۱۰۰ میکروثانیـه طـول میکشد تـا بـه مقـدار OCR1A برسـد و خروجـی دوباره صفر شود، پس بـرای آنکـه تـایمر از صـفر تـا OCR1A را در ۱۰۰ میکروثانیـه طـی کنـد بایـد

مطابق توضیحات زیر عمل کنیم:

چون فرکانس 1MHz است و هر کلاک در یک میکروثانیه زده می‌شود باید ۱۰۰ کلاک زده شود، پس مقدار OCR1A را برابر عدد ۱۰۰ قرار می‌دهیم:

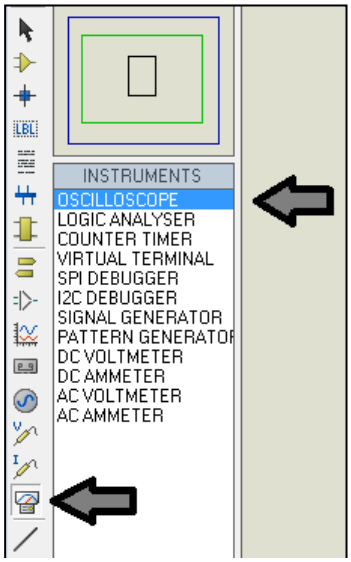


شکل ۷-۱۵

در کد تولید شده باید مقدار OCR1A را برابر عدد ۱۰۰ قرار دهیم. این عدد به هگزادسیمال برابر 0x64 می‌شود:

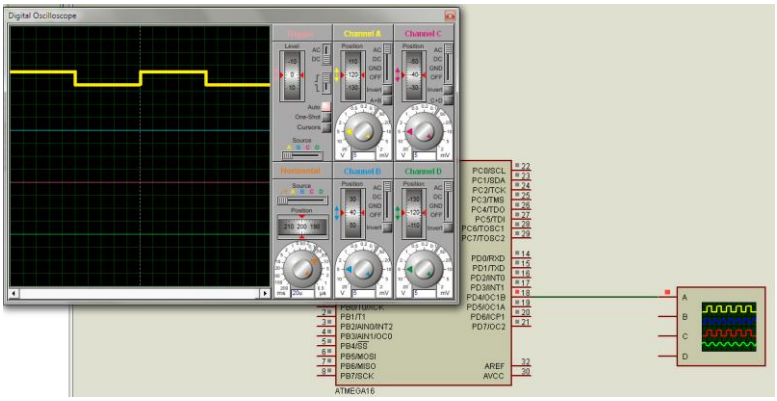
```
// Compare A Match Interrupt: Off
// Compare B Match Interrupt: Off
TCCR1A=0x10;
TCCR1B=0x0A;
TCNT1H=0x00;
TCNT1L=0x00;
ICR1H=0x00;
ICR1L=0x00;
OCR1AH=0x00;
OCR1AL=0x64;
OCR1BH=0x00;
OCR1BL=0x00;
```

این کد را در پروتئوس شبیه‌سازی می‌کنیم تا سیگنال خروجی پایه‌های مربوط به تایمر یک را مشاهده کنیم. همانطور که گفتیم پایه‌های مربوط به خروجی تایمر یک پایه‌های PD4 و PD5 هستند،که پایه‌ی PD4 مربوط به خروجی OCR1B و پایه‌ی PD5 مربوط به خروجی OCR1A می‌باشد. خروجی پایه‌ی OCR1B میکروکنترلر را به یک اسیلوسکوپ متصل کرده و شکل موج را مشاهده می‌کنیم. برای آوردن اسیلوسکوپ در پروتئوس باید مطابق شکل روبرو عمل کنیم:

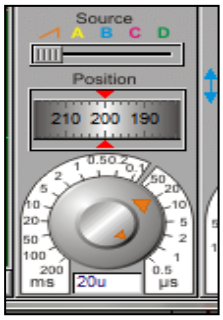


شکل ۷-۱۶: انتخاب اسیلوسکوپ در پروتئوس

که این نوع اسیلوسکوپ دارای چهار ورودی برای چهار سیگنال مختلف می‌باشد. اگر اسیلوسکوپ را به پایه‌ی خروجی مربوط به تولید موج میکرو متصل کنیم، با اجرای برنامه شکل زیر را مشاهده می‌کنیم:

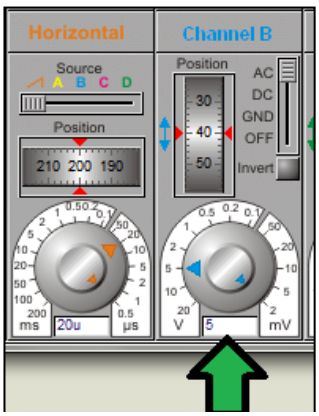


شکل ۷ – ۱۷ : اجرای برنامه‌ی مثال ۵



در شکل روبرو درجه‌ی متغیر مربوط به تقسیم بندی زمان را مشاهده می‌کنید که نشان دهنده‌ی آن است که هر خانه‌ی افقی معادل چه زمانی است و همانطور که مشاهده می‌کنید عبارت 20U نوشته شده به معنای آن است که هر خانه‌ی افقی معادل ۲۰ میکروثانیه است:

این قسمت را بر روی ۲۰ میکروثانیه قرار داده‌ایم و مشاهده کردیم که یک دوره‌ی زمانی ۱۰ خانه است که برابر می‌شود با $20\mu s * 10 = 200\mu s$ و قسمت مشخص شده در شکل شماره‌ی روبرو نیز مربوط به تقسیم بندی خانه‌های عمودی است که در شکل این قسمت بر روی ۵ ولت قرار گرفته است که به این معناست که هر خانه‌ی عمودی معادل ۵ ولت است:

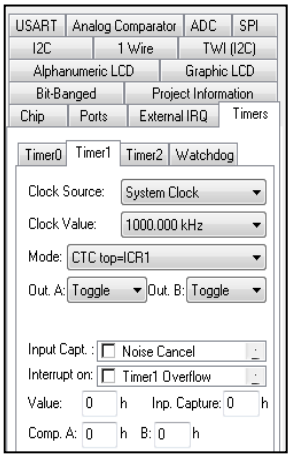


چون ولتاژ خروجی میکرو به صورت منطقی است و فقط می‌تواند صفر ولت یا ۵ ولت باشد اگر این تقسیم بندی بر روی ۵ ولت تنظیم شود زمانی‌که میکرو ولتاژ خروجی دارد شکل موج تا یک خانه‌ی عمودی بالا می‌آید.

شکل ۷ – ۱۸ : تقسیم بندی مقدار ولتاژ در اسیلوسکوپ

مد CTC top=ICR1A

مد CTC top=ICR1A نیز مربوط به مقایسه‌ی مقدار رجیستر TCNTn می‌باشد با این تفاوت که حداکثر مقدار آن می‌تواند مقدار ICR1A باشد. مزیت این مد نسبت به مد CTC top=OCR1A این است که می‌توانیم روی هر دو پایه‌ی OCR1A و OCR1B خروجی را مشاهده کنیم.

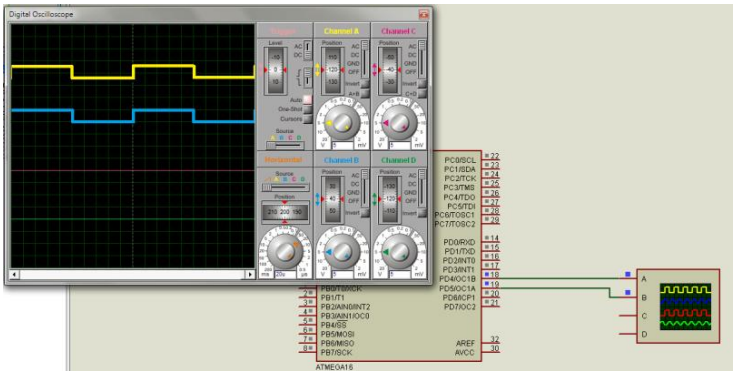


مثال ۶: برای مثال فرض کنید می‌خواهیم موج خروجی در مثال قبل را روی هر دو پایه‌ی خروجی ببینیم:

مد CTC top=ICR1A را انتخاب می‌کنیم و در کد تولید شده مقدار ICR1A را برابر 0x64 قرار می‌دهیم:

```
// Mode: CTC top=ICR1
// OC1A output: Toggle
// OC1B output: Toggle
// Noise Canceler: Off
// Input Capture on Falling Edge
// Timer1 Overflow Interrupt: Off
// Input Capture Interrupt: Off
// Compare A Match Interrupt: Off
// Compare B Match Interrupt: Off
TCCR1A=0x50;
TCCR1B=0x1A;
TCNT1H=0x00;
TCNT1L=0x00;
ICR1H=0x00;
ICR1L=0x64;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
```

این برنامه در پروتئوس به صورت شکل زیر اجرا می‌شود:



فرکانس شکل موج در حالت CTC

برای محاسبه‌ی فرکانس می‌توانیم از فرمول زیر استفاده کنیم:

$$f=\frac{f_{clk_{Io}}}{2*N*(TOP+1)}$$

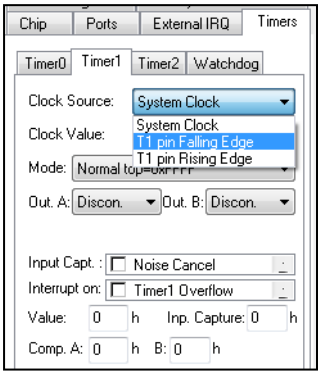
که می‌توان فرمول بالا را به صورت فرمول ساده‌تر زیر نوشت:

$$f=\frac{f_{clk-timer}}{2*(TOP+1)}$$

که صورت کسر، فرکانس تنظیم شده در تایمر و مخرج کسر دو برابر مقدار حداکثر تایمر به علاوه‌ی یک است (که مقدار Top با توجه به تایمر و حالت انتخاب شده می‌تواند برابر یکی از مقادیر OCR1Aیا ICR1A یا OCR0 یاOCR2 باشد).

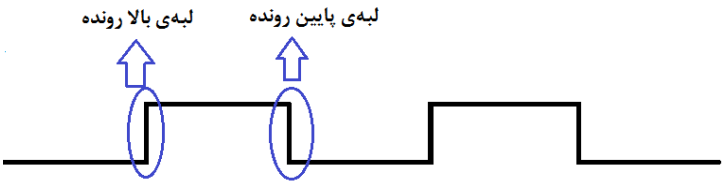
منبع کلاک

تا الان منبع کلاک را در حالت System Clock قرار می‌دادیم و از کلاک درونی خود میکرو استفاده میکردیم، اگربخواهیم از منبع بیرونی استفاده کنیم می‌توانیم در تنظیمات کدویزارد مطابق شکل زیر عمل کنیم:



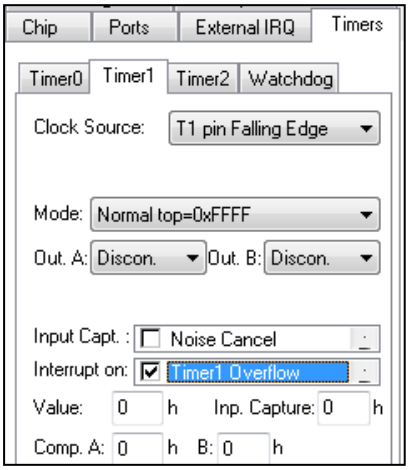
شکل ۷ – ۱۹ : منبع کلاک در تایمر ۱

دو حالت برای منبع کلاک بیرونی وجود دارد یکی لبه‌ی پایین رونده (Falling Edge) و دیگری لبه‌ی بالا رونده (Rising Edge) که در حالت اول زمانی تایمر کلاک می‌خورد که سیگنال ورودی از سطح منطقی ۱ به صفر تغییر حالت دهد و در حالت دوم زمانی‌که سیگنال ورودی از صفر به ۱ تغییر حالت دهد:



شکل ۷ – ۲۰ : لبه‌ی بالارونده و پایین رونده

مثال ۷: مثالی را بررسی می‌کنیم که تایمر یک را در حالت Falling Edge قرار می‌دهیم و کدی می‌نویسیم که بعد از ۵ کلاک، یک LED را روشن و بعد از ۵ کلاک دیگرآن را خاموش کند:



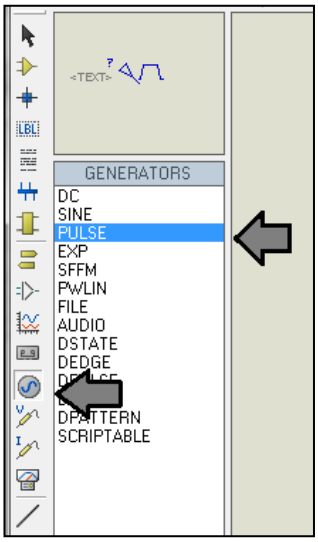
تنظمیات بخش تایمر یک را مانند شکل روبرو انجام می‌دهیم. مد را در حالتNormal قرار می‌دهیم و وقفه‌ی تایمر یک را فعال می‌کنیم، چون می‌خواهیم با گذشت ۵ کلاک برنامه وارد وقفه بشود مقدار اولیه‌ی رجیستر TCNT تایمر را برابر ۶۵۵۳۱ قرار می‌دهیم (عدد ۶۵۵۳۱ در هگزادسیمال برابر 0XFFFB می‌شود).

```
// Timer1 Overflow Interrupt: On
// Input Capture Interrupt: Off
// Compare A Match Interrupt: Off
// Compare B Match Interrupt: Off
TCCR1A=0x00;
TCCR1B=0x06;
TCNT1H=0xFF;
TCNT1L=0xFB;
ICR1H=0x00;
ICR1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
```

زمان آنکه برنامه وارد وقفه شود مشخص نیست، چون نمی‌دانیم که این ۵ کلاک در چه زمانی اتفاق می‌افتد، اگر فرکانس منبع بیرونی زیاد باشد بسیار سریع ۵ کلاک زده می‌شود و برنامه وارد وقفه می‌شود ولی اگر فرکانس این منبع کمتر باشد به‌تبع زمان انجام هر وقفه نیز بیشتر می‌شود (مثلا اگر فرکانس منبع بیرونی یک هرتز باشد هر کلاک در یک ثانیه رخ می‌دهد).

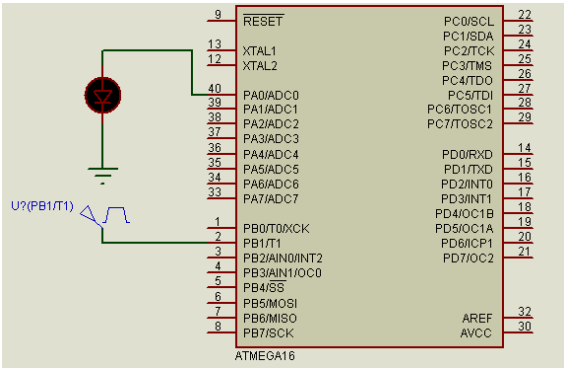
کد درون تابع وقفه نیز به شکل زیر است:

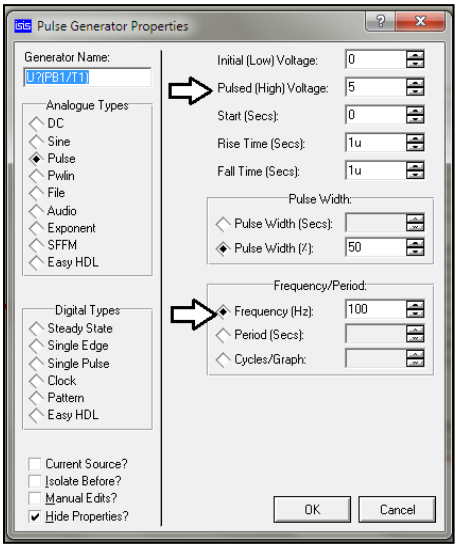
```
// Timer1 overflow interrupt service routine
interrupt [TIM1_OVF] void timer1_ovf_isr(void)
{
a++;
if(a==1)
PORTA.0=1;
if(a==2) {
PORTA.0=0;
a=0;
}
TCNT1H=0xFF;
TCNT1L=0xFB;


}
```



برای شبیه‌سازی برنامه فوق پروتئوس را باز کرده و بعد از آنکه میکرو و LED را انتخاب کردیم یک منبع پالس مربعی نیز مطابق شکل روبرو اضافه می‌کنیم:

همانطور که Clock source را روی حالت T1 PIN Falling Edge قرار دادیم منبع را به پایه‌ی T1 میکروکنترلر متصل می‌کنیم (که این پایه همان پایه‌ی PB1 است.)

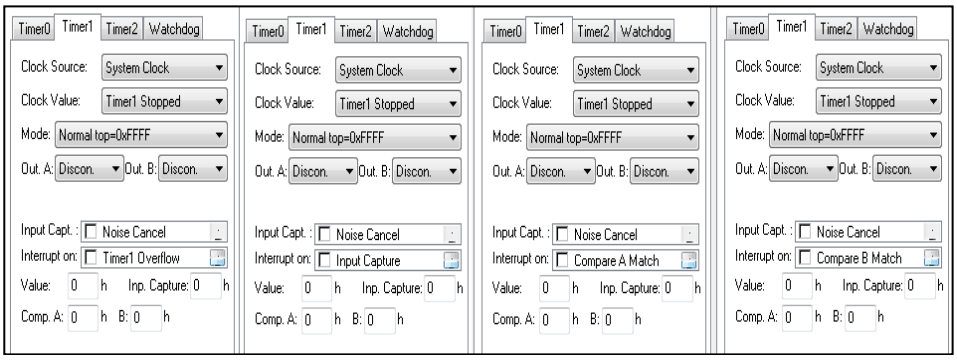




و برای تنظیمات منبع پالس بر روی آن دبل کلیک می‌کنیم. در بخش مربوط به Pulsed (High) Voltage مقدار آن را بر روی ۵ ولت قرار می‌دهیم (چون یک موج مربعی با دامنه‌ی ۵ ولت می‌خواهیم) و فرکانس هم یک عدد دلخواه انتخاب می‌کنیم، مثلاً اگر فرکانس ۱۰۰ هرتز باشد یعنی هر کلاک در یک صدم ثانیه زده می‌شود که ۵ کلاک در ۰٫۵ ثانیه یا ۵۰۰ میلی‌ثانیه زده می‌شود، پس هر ۵۰۰ میلی‌ثانیه یک بار LED چشمک می‌زند.

وقفه‌های تایمر یک

همانطور که در شکل زیر مشخص است تایمر یک دارای ۴ نوع وقفه می‌باشد:



شکل ۷ – ۲۱ : وقفه‌های تایمر ۱

اولین وقفه Timer 1 Overflow است که از قبل با آن آشنا شدیم، دومین وقفه Input Capture است که توضیحات مربوط به آن در ادامه داده می‌شود، سومین وقفه مربوط به Compare A Match است که با این نوع وقفه در تایمر صفر آشنا شدیم و در حقیقت زمانی این وقفه اتفاق می‌افتد که مقدار تایمر یک با مقدار OCR1A برابر شود و چهارمین وقفه مربوط به Compare B Match است که زمانی اتفاق می‌افتد که مقدار تایمر یک با مقدار OCR1B برابر شود.

وقفه‌ی Input Capture

این وقفه یکی از وقفه‌های مهم و پرکاربرد در این نوع میکروکنترلرها می‌باشد و زمانی رخ می‌دهد که سیگنالی به‌عنوان ورودی دریافت شود. با دریافت هر سیگنال ورودی یا به عبارت بهتر با تغییر در لبه‌ی سیگنال ورودی این وقفه اجرا می‌شود.

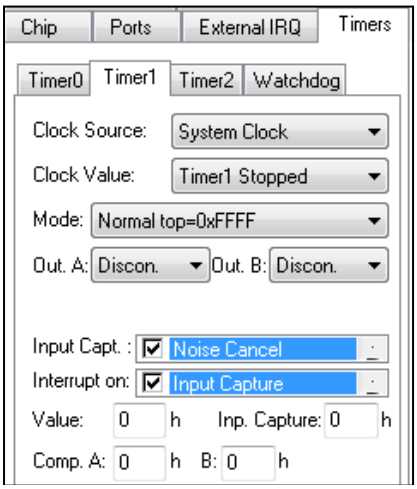
زمانی‌که از این وقفه استفاده می‌کنیم گزینه‌ی Noise Cancel را هم می‌زنیم چراکه ممکن است این ورودی توسط نویزهای موجود صفر و یک شود و برنامه آن را به‌عنوان سیگنال ورودی تشخیص دهد.

این گزینه در این حالت چهار نمونه پشت سر هم را می‌گیرد و اگر این چهار نمونه برابر بود اجازه‌ی عبور سیگنال را می‌دهد(نویز می‌تواند به صورت تصادفی پایه‌ها را برابر صفر یا یک قرار دهد و احتمال آنکه چهار بار پشت سر هم نویز یک باشد یا چهار بار پشت سر هم این نویز صفر باشد بسیار کم است و به همین دلیل است که این واحدNoise Cancel که تشکیل شده از یک فیلتر دیجیتال است چهار نمونه‌ی پشت سر هم را بررسی می‌کند و اگر هر چهار نمونه یکسان بودند اجازه‌ی عبور را می‌دهد).

پایه‌ی مربوط به ورودی Input Capture پایه‌ی PD6 در میکروکنترلر ATmega16 می‌باشد.

مثال ۸: حال در مثالی ساده برنامه‌ای می‌نویسیم که به محض دریافت سیگنال ورودی وارد وقفه شده و یک LED را روشن کند و با سیگنال بعدی آن را خاموش کند و این کار با ورود هر سیگنال ادامه یابد:

کدویزارد را باز کرده و تیک مربوط به وقفه‌ی Input Capture و همچنین تیک Noise Cancel را می‌زنیم، نیازی به شمارش تایمر یک هم نداریم پس آن را در حالت Timer 1 Stopped قرار می‌دهیم:

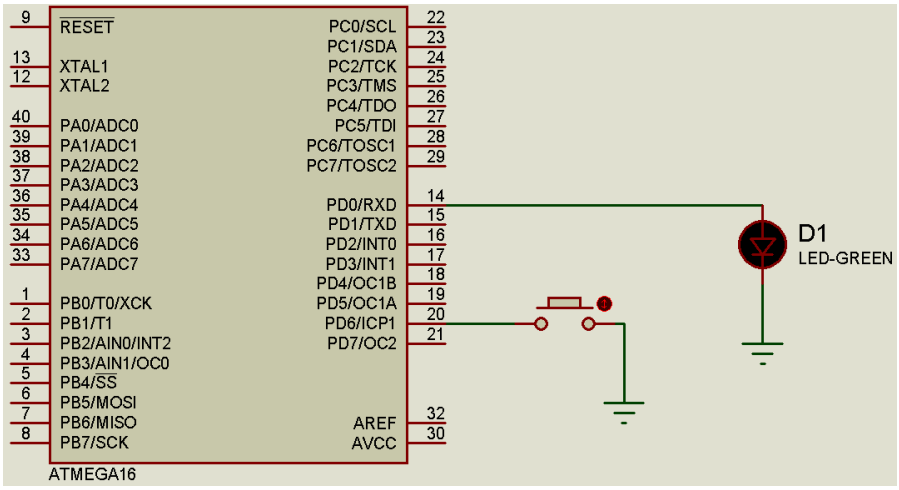


شکل ۷-۲۲

PORTD.0 را هم در حالت خروجی قرار می‌دهیم که LED را به آن وصل کنیم، سپس در تابع مربوط به وقفه یک کد ساده می‌نویسیم که هربار سیگنالی وارد می‌شود مقدار PORTD.0 برعکس شود یعنی اگر مقدار آن صفر است برابر یک شود و اگر مقدار آن یک است برابر صفر شود که برای این کار از دستور NOT (=~) استفاده می‌کنیم:

```
// Timer1 input capture interrupt service routine
interrupt [TIM1_CAPT] void timer1_capt_isr(void)
{
 PORTD.0=~PORTD.0;
}
```

حال برای شبیه‌سازی وارد محیط پروتئوس شده و مانند شکل یک ATmega16 و یک عدد LED و یک عدد کلید (BUTTON) انتخاب می‌کنیم:

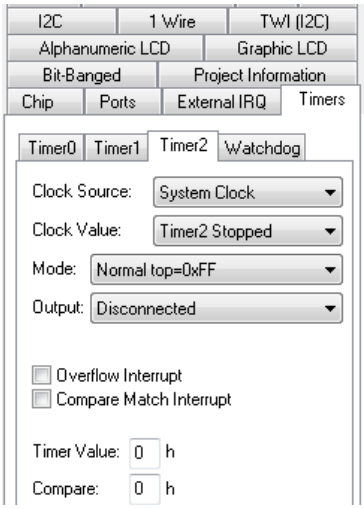


شکل ۷-۲۳

همانطور که گفتیم ورودی مربوط به Input Capture پایه‌ی PD6 می‌باشد و با هربار صفر شدن این پایه برنامه یک‌بار وارد وقفه شده و LED را روشن یا خاموش می‌کند.

این پایه می‌تواند نقش شمارنده‌ی سیگنال ورودی هم داشته باشد برای مثال می‌توانیم از این پایه به‌عنوان فرکانس‌متر یا تشخیص‌دهنده‌ی فرکانس ورودی هم استفاده کنیم که مثال مربوط به فرکانس‌متر در آخر بخش تایمر/ کانتر دو آورده شده است.

تایمر/ کانتر دو



تایمر دو نیز مانند تایمر صفر یک تایمر هشت بیتی است و مدهای آن نیز شبیه به مدهای تایمر/ کانتر صفر هستند:

یک تفاوت بارز این تایمر نسبت به تایمر صفر مربوط به بخش منبع کلاک است که این تایمر قابلیت استفاده از کریستال خارجی برای تولید کلاک را دارد با این کار عملیات کلاک‌زنی با دقت بسیار بالاتری انجام می‌شود، یعنی به جای آنکه کلاک توسط یک واحد داخلی زده شود این کار به یک کریستال خارجی واگذار می‌شود، این کار علاوه بر افزایش سرعت، افزایش دقت را نیز در بردارد زیرا می‌توان از کریستال‌های با دقت بسیار بالا استفاده کرد.

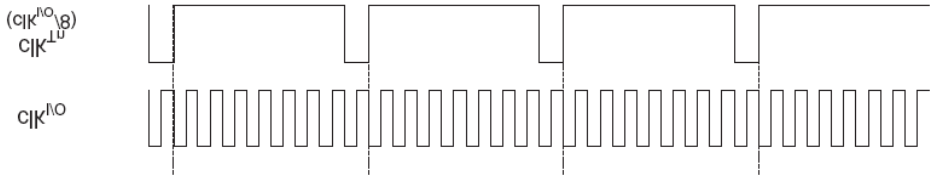
شکل زیر تصویر یک کریستال را نشان می‌دهد:

شکل ۷ - ۲۴ : کریستال

کریستال یک عنصر دوپایه است که کار آن تولید کلاک می‌باشد. مزیت دیگر استفاده از کریستال خارجی، این است که فرکانس‌های متنوع‌تری را می‌توانیم در اختیار داشته باشیم. برای مثال در تایمر صفر برای آنکه یک ثانیه را تولید کنیم مقدار اولیه‌ی و یا مقدار حداکثر تایمر را تغییر دادیم که تایمر ۲۵۰ کلاک بزند و در فرکانس یک مگاهرتز زمان ۲۵۰ میکروثانیه پدیدآید ولی اگر از یک کریستال خارجی استفاده کنیم احتیاجی به تغییر تعداد کلاک‌های تایمر نیست برای مثال یکی از کریستال‌های موجود در بازار کریستال 32.768kHz است، این کریستال با فرکانس 32.768kHz کلاک تولید می‌کند.

همانطور که گفته شد از مزیت‌های تایمر دو این است که می‌تواند فرکانس خود را از کلاک خارجی و یا از تقسیمی از فرکانس خارجی تامین کند، یعنی می‌تواند فرکانس تولیدی کلاک را بر اعدادی مانند ۱،۸، ۳۲، ۶۴، ۱۲۸، ۲۵۶و۱۰۲۴تقسیم کند و یک فرکانس جدید بسازد.

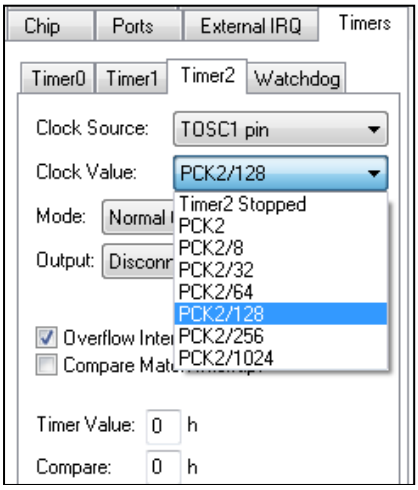
برای مثال می‌تواند هر ۸ عدد کلاک که توسط کریستال تولید می‌شود را یک کلاک محسوب کند. به شکل زیر نگاه کنید:



شکل ۷ - ۲۵ : تقسیم کلاک

برای مثال فرض کنید می‌خواهیم زمان یک ثانیه را تولید کنیم. اگر فرکانس 32.768KHz را بر 128 تقسیم کنیم فرکانس 256Hz ساخته می‌شود این فرکانس به این معناست که هر یک کلاک را در $\frac{1}{256}$ ثانیه انجام می‌دهد، پس زمانی‌که تایمر ۲۵۶ کلاک بزند زمان یک ثانیه ساخته می‌شود و دیگر نیازی به کم کردن تعداد کلاک‌ها نیست.

مثال ۹: میخواهیم همان مثال LED را که هر ثانیه یک‌بار چشمک می‌زند را با تایمر دو و در حالت کریستال خارجی بنویسیم:
کدویزارد را باز می‌کنیم و منبع کلاک را روی حالت TOSC1 pin قرار می‌دهیم.
(این حالت به این معناست که منبع کلاک توسط کریستال خارجی تامین می‌شود که روی پین TOSC1 قرار دارد(که در ATmega16 پایه‌های ۲۸ و ۲۹ است).



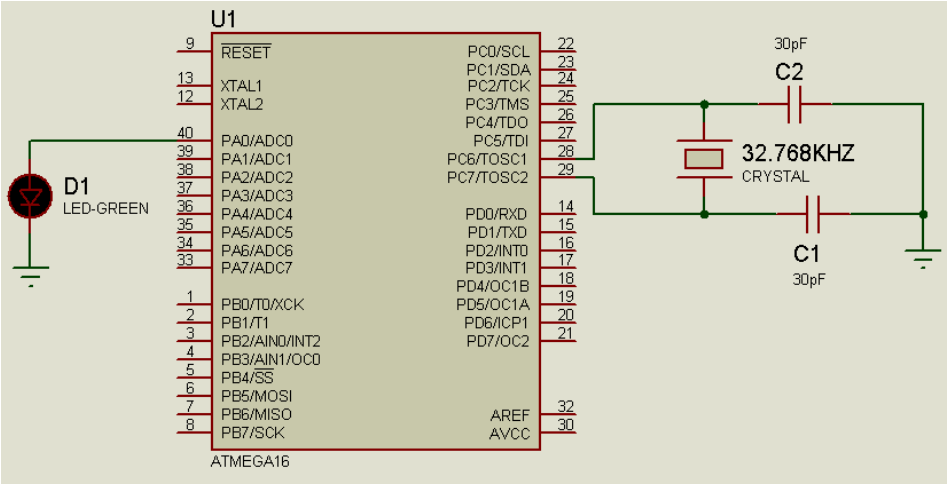
شکل ۷ - ۲۶

مد را روی PCK2 / 128 قرار می‌دهیم (که به معنی تقسیم فرکانس کریستال بر ۱۲۸ است).
تیک وقفه‌ی سرریز را می‌زنیم (Overflow interrupt) که بعد از یک ثانیه وارد وقفه شود.
سپس کد مربوطه را درون وقفه می‌نویسیم:

```
// Timer2 overflow interrupt service routine
interrupt [TIM2_OVF] void timer2_ovf_isr(void)
{

PORTA.0=~PORTA.0;

}
```

حال که برنامه نوشته شد برای شبیه‌سازی وارد پروتئوس می‌شویم و برنامه را اجرا می‌کنیم، برای انتخاب کریستال در پروتئوس باید در بخش قطعات عبارت CRYSTAL را جستجو کنیم و پس از انتخاب کریستال با دبل کلیک بر روی آن فرکانس آن را انتخاب کنیم.

دقت کنید که در عمل برای استفاده از کریستال باید هر پایه‌ی کریستال را به وسیله‌ی یک خازن به زمین وصل کنیم. (مانند شکل ۷-۲۷)



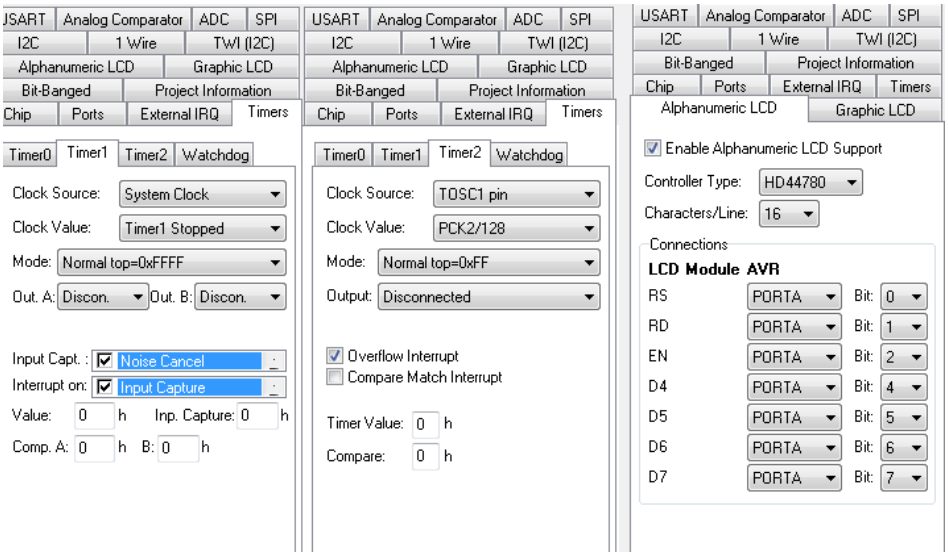
شکل ۷ – ۲۷ : نحوه‌ی اتصال کریستال

ساخت یک فرکانس‌متر موج مربعی

برای ساخت یک فرکانس‌متر باید از دو تایمر به صورت همزمان استفاده کنیم. در این مثال از تایمر دو برای محاسبه‌ی زمان و از تایمر یک برای دریافت سیگنال خارجی (Input Capture) استفاده می‌کنیم.

نحوه‌ی کار به این صورت است که واحد Input Capture در هر وقفه‌ی خود یک شماره به مقدار یک متغیر (مثلاً a) اضافه می‌کند و تایمر دو نیز زمان یک ثانیه را شمارش می‌کند، با گذشت زمان یک ثانیه برنامه وارد وقفه‌ی تایمر دو شده و مقدار متغیر a را می‌خواند که این مقدار همان فرکانس است(زیرا معنای فرکانس تعداد تکرار دوره‌ی تناوب و یا در یک موج مربعی تعداد تغییرات از صفر به یک، در یک ثانیه است و چون a با هربار تغییر، یک واحد اضافه می‌شود، برابر همان تعداد تغییرات است که اگر در لحظه‌ی رسیدن به زمان یک ثانیه مقدار آن را ببینیم، این مقدار همان فرکانس سیگنال ورودی است). سپس میکرو این فرکانس را در LCD نمایش می‌دهد و متغیر a را دوباره برای محاسبه‌ی فرکانس در لحظه‌ی بعد صفر می‌کند.

کدویزارد را باز می‌کنیم و تنظیمات مربوط به تایمر یک و تایمر دو و LCD را مانند شکل انجام می‌دهیم:

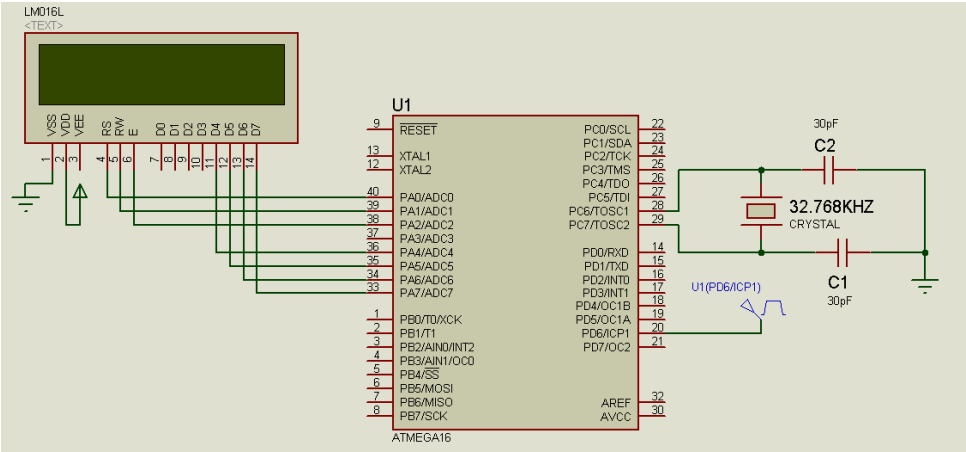


شکل ۷-۲۸

سپس کد مربوطه را در وقفه‌های تایمر یک و تایمر دو می‌نویسیم:

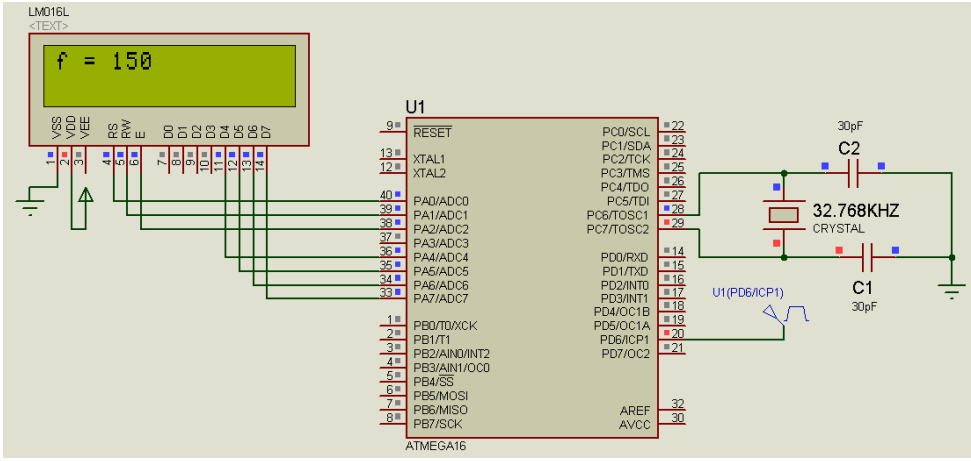
```
#include <mega16.h>

// Alphanumeric LCD functions
#include <alcd.h>
#include <stdio.h>
int a=0;
char h[20];
// Timer1 input capture interrupt service routine
interrupt [TIM1_CAPT] void timer1_capt_isr(void)
{
a++;

}

// Timer2 overflow interrupt service routine
interrupt [TIM2_OVF] void timer2_ovf_isr(void)
{

  lcd_gotoxy(0,0);
  sprintf(h,"f = %d",a);
  lcd_puts(h);
  a=0;

}
```

حال برای شبیه‌سازی مدار زیر را در پروتئوس می‌کشیم:



شکل ۷ - ۲۹

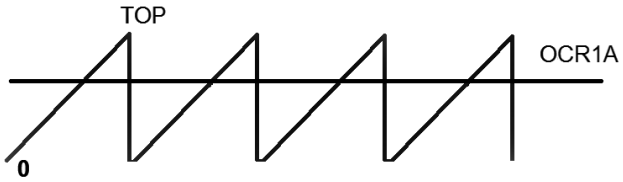
با دبل کلیک روی منبع پالس حداکثر ولتاژ آن را برابر ۵ ولت و فرکانس را روی یک عدد دلخواه قرار می‌دهیم (مثلاً ۱۵۰هرتز) که در این صورت با اجرای برنامه فرکانس روی LCD به نمایش در می‌آید:



شکل ۷ - ۳۰

به روز رسانی در مدهای Normal و CTC

فرض کنید در مد Normal وقفه‌ی Compare A Match را فعال کرده اید. در این صورت در هنگامی که مقدار تایمر(مقدار TCNT) برابر مقدار OCR1A شود تایمر وارد وقفه می‌شود.(به شکل ۷-۳۱ دقت کنید):

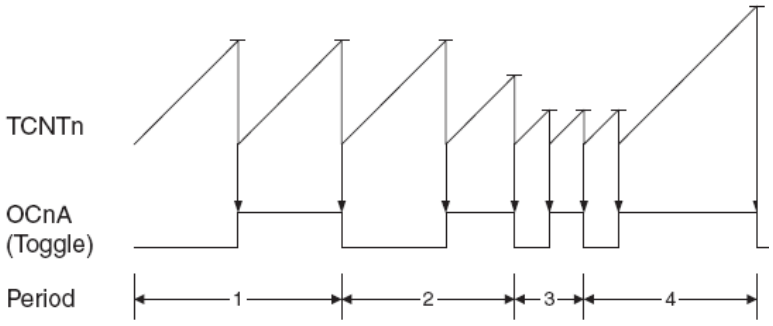


شکل ۷-۳۱

حال فرض کنید در طول برنامه مقدار OCR1A تغییر کند، آنگاه وقفه‌ی تایمر باید در محل جدید برابری OCR1A و TCNT اتفاق افتد.

برای مثال فرض کنید ابتدا مقدار OCR1A برابر ۲۰۰ است و تایمر از صفر شروع به شماردن می‌کند و زمانی‌که به مقدار ۱۸۰ می‌رسد(یعنی TCNT برابر ۱۸۰ می‌شود) برنامه به دستوری برسد که مقدار OCR1A برابر ۷۰۰ شود آنگاه در لحظه‌ی رسیدن مقدار TCNT به ۲۰۰ هیچ اتفاقی رخ نمی‌دهد و وقفه در لحظه‌ی رسیدن TCNT به ۷۰۰ رخ می‌دهد یعنی مقدار OCR1A به طور فوری مقداردهی می‌شود و در برنامه تاثیر می گذارد.

یک نکته که باید به آن توجه کنیم این است که در مقداردهی به OCR1A حواسمان باشد که این مقداردهی نابه‌جا نباشد. برای مثال فرض کنید که در مثال بالا در لحظه‌ای که مقدار TCNT برابر ۱۸۰ است به OCR1A مقدار ۱۵۰ را بدهیم. آنگاه دیگر این برابری در این سیکل اتفاق نمی‌افتد و مقدار تایمر تا حداکثر خود افزایش می‌یابد و دوباره صفر می‌شود و زمانی‌که به ۱۵۰ برسد وارد وقفه می‌شود. که شاید این اتفاق مطلوب برنامه- نویس نباشد.



همانطور که مشاهده می‌کنید در همان لحظه‌ای که مقدار OCR1Aیا OCR1B تغییر می‌کند مقدار حداکثر TCNT نیز تغییر می‌کند زیرا در این مد مقدار TCNTحداکثر می‌تواند تا مقدار OCR1A یا OCR1B افزایش‌یابد و با تغییر مقدار این دو رجیستر مقدار حداکثر TCNT(یا به عبارتی مقدار حداکثر تایمر) نیز تغییر می‌کند.

در مدهای Normal و CTC تغییر در مقدار OCR1A یا OCR1B به طور آنی رخ می‌دهد و تاثیر خود را در برنامه می گذارد. به جدول زیر که مربوط به تایمر یک می‌باشد توجه کنید:

Timer/Counter Mode of Operation	TOP	Update of OCRnx at	TOVn Flag Set on
Normal	0xFFFF	Immediate	MAX
PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
CTC	OCRnA	Immediate	MAX
Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
PWM, Phase and Frequency Correct	ICRn	BOTTOM	BOTTOM
PWM,Phase and Frequency Correct	OCRnA	BOTTOM	BOTTOM
PWM, Phase Correct	ICRn	TOP	BOTTOM
PWM, Phase Correct	OCRnA	TOP	BOTTOM
CTC	ICRn	Immediate	MAX
(Reserved)	–	–	–
Fast PWM	ICRn	BOTTOM	TOP
Fast PWM	OCRnA	BOTTOM	TOP

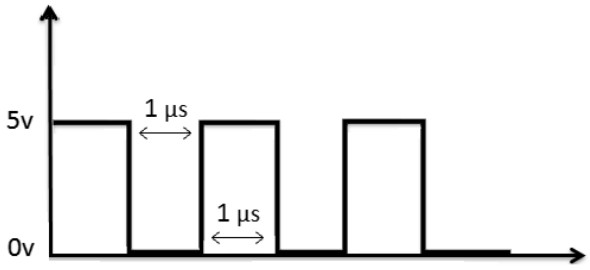
جدول ۷ – ۱: به روز رسانی

در ستون مشخص شده که مربوط به به‌روزرسانی OCR1Aو OCR1B می‌باشد، روبه‌روی مدهای Normal و CTC عبارت Immediate یا فوری نوشته شده است که به معنای عوض کردن مقدار آن در همان لحظه می‌باشد.

اگر به ستون آخر این جدول نگاه کنید لحظه‌ی اتفاق افتادن وقفه‌ی سرریز را نشان می‌دهد که روبه‌روی مدهای Normal و CTC عبارت MAX نوشته شده است که به این معناست که هنگام عبور از حداکثر مقدار تایمر این وقفه اتفاق می‌افتد.

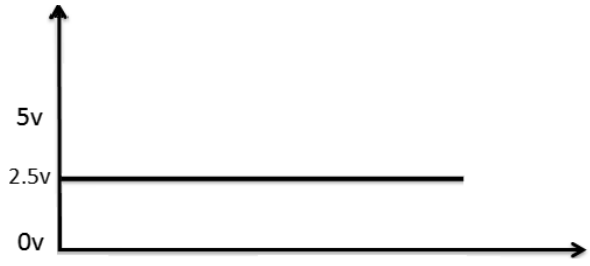
مدهای PWM

PWM مخفف عبارت Pulse Width Modulation و به معنای "مدولاسیون پهنای پالس" می‌باشد که در این مد می‌توان پهنای پالس مربعی تولید شده یا به عبارتی زمان صفر و یک بودن این پالس را تنظیم کرد. همانطور که می‌دانید ولتاژ خروجی میکرو ۰ یا ۵ ولت است، حال فرض کنید قصد داریم موتوری را به چرخش در آوریم. اگر ولتاژ خروجی ۵ ولت باشد موتور با حداکثر سرعت میچرخد و اگر ولتاژ خروجی صفر ولت باشد موتور متوقف می‌شود؛ حال اگر بخواهیم که این موتور با سرعتی متوسط به چرخش درآید چه باید کنیم؟ در بخش تایمرها با تولید موج مربعی آشنا شدیم، حال اگر یک موج مربعی با فرکانس پانصد کیلوهرتز به موتور بدهیم چه اتفاقی می‌افتد؟



شکل ۷-۳۲

این موج تولید شده یک میکروثانیه ۵ ولت و یک میکروثانیه صفر ولت است و موتور این موج را با مقدار متوسط ۲,۵ ولت احساس می‌کند:



شکل ۷-۳۳

اگر بخواهیم ولتاژی با مقدار متوسط ۳ ولت داشته باشیم باید موجی مربعی تولید کنیم که ۱,۲ میکروثانیه پنج ولت و ۰,۸ میکروثانیه صفر ولت باشد (این اعداد می‌تواند بسیار متفاوت باشد برای

مثال برای تولید همان ۳ ولت می‌توان موجی مربعی تولید کرد که ۲,۴ میلی‌ثانیه ۵ ولت و ۱,۶ میلی‌ثانیه ۰ ولت باشد که باز هم مقدار متوسط آن ۳ ولت می‌شود و فقط فرکانس آن تغییر می‌کند)، برای محاسبه‌ی مقدار متوسط و یا محاسبه‌ی زمان یک بودن سیگنال برای رسیدن به ولتاژ متوسط مورد نظر می‌توانیم از تناسب زیر استفاده کنیم:

$$\frac{\text{مقدار متوسط ولتاژ}}{5} = \frac{\text{مدت زمان یک بودن سیگنال}}{\text{زمان یک دوره تناوب}}$$

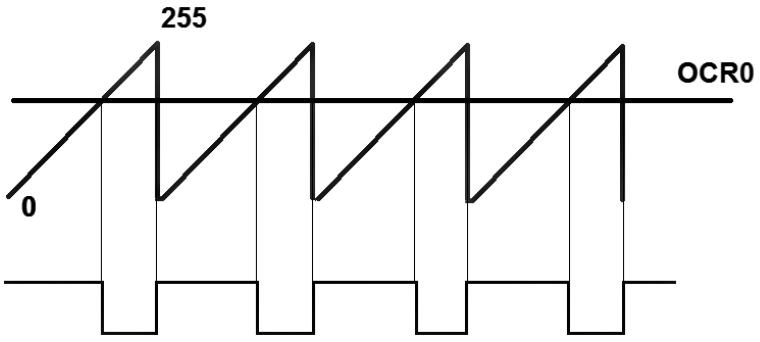
که برای مثال برای تولید مقدار متوسط ۳ ولت در فرکانس ۱ مگاهرتز :

$$\frac{3}{5} = \frac{\text{مدت زمان یک بودن سیگنال}}{1\mu s}$$

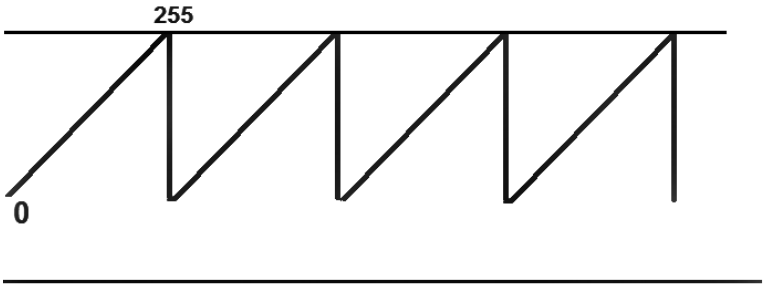
مدت زمان یک بودن سیگنال برابر ۰,۶ میکروثانیه می‌شود پس برای تولید یک موج با این فرکانس و این مقدار متوسط باید شکل موج خروجی ۰,۶ میکروثانیه یک و ۰,۴ میکروثانیه صفر باشد. در واحد تایمر/کانتر امکان تولید موج مربعی با فرکانس و مقدار متوسط دلخواه وجود دارد. با نحوه‌ی تنظیم فرکانس آشنا شدیم، نحوه‌ی تنظیم مقدار متوسط بر روی یک عدد دلخواه پس از آشنایی با مدهای مختلف PWM بررسی می‌شود.

مد Fast PWM

فرض کنید در تایمر صفر مد Fast PWM top=0xFF را انتخاب کرده‌ایم می‌دانیم تایمر صفر از ۰ تا ۲۵۵ را شمارش می‌کند، در تایمر صفر در مد Fast PWM مقداری که درون رجیستر OCR0 قرار دارد دائماً با مقدار رجیستر TCNT0 مقایسه می‌شود و به محض برابری مقدار خروجی تغییر می‌کند. تولید شکل موج (PWM) در این حالت مانند شکل زیر می‌باشد، یعنی زمانی‌که مقدار OCR0 با مقدار تایمر برابر می‌شود خروجی صفر می‌شود و زمانی‌که سرریز رخ می‌دهد خروجی دوباره یک می‌شود و به این‌ترتیب شکل موج مربعی در خروجی ظاهر می‌شود:



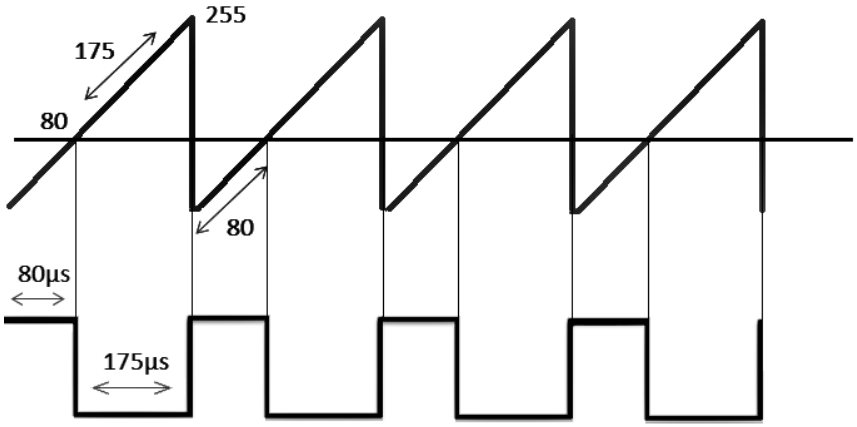
هرچه مقدار OCR0 بیشتر باشد مدت زمان یک بودن خروجی نیز بیشتر می‌شود و زمانی‌که مقدار OCR0 برابر بیشترین مقدار (در اینجا ۲۵۵) باشد خروجی کاملاً یک می‌شود:



شکل ۷-۳۴

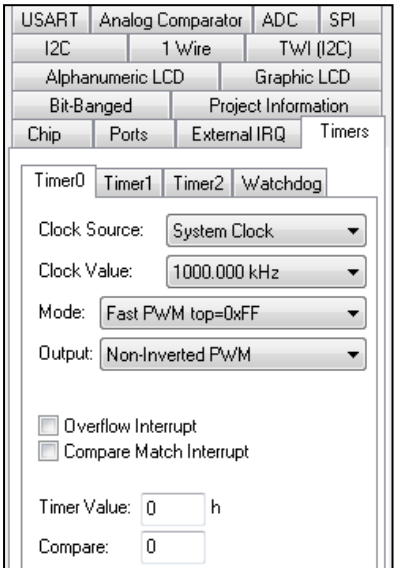
مثال ۱۰: فرض کنید می‌خواهیم یک موج PWM درست کنیم که ۸۰ میکروثانیه یک و ۱۷۵ میکروثانیه صفر باشد:

برای این کار تایمر صفر را روی مدFast PWM قرار می‌دهیم و فرکانس تایمر را روی یک مگاهرتز تنظیم می‌کنیم. چون می‌خواهیم خروجی ۸۰ میکروثانیه یک باشد پس باید مقدار OCR0 در زمان ۸۰ میکروثانیه با مقدار تایمر برابر شود. چون فرکانس یک مگاهرتز است و هر شماره در یک میکروثانیه شماره می‌شود پس مقدار OCR0 باید برابر ۸۰ باشد تا از عدد صفر تا ۸۰ در ۸۰ میکروثانیه طی شود و بعد از ۸۰ میکروثانیه مقدار OCR0 از یک تبدیل به صفر شود سپس مقدار TCNT0 افزایش یافته و در ۱۷۵ شماره‌ی بعد به ۲۵۵ می‌رسد و پس از سرریز دوباره برابر یک می‌شود:



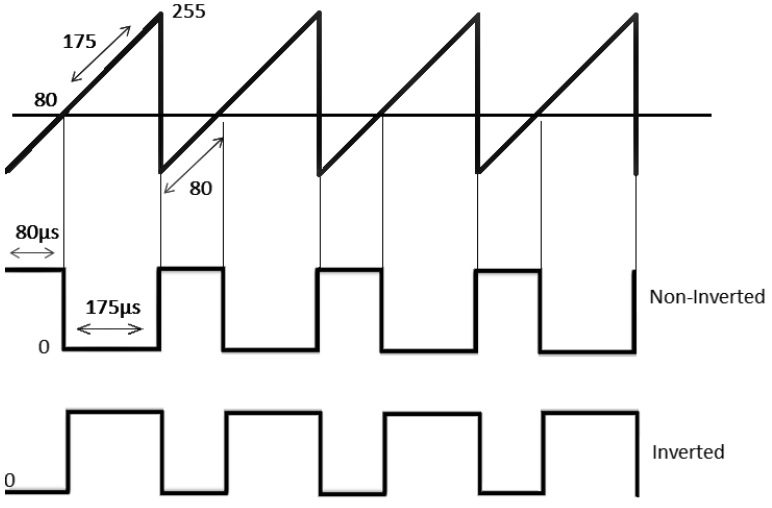
شکل ۷-۳۵

برای نوشتن این برنامه وارد کدویزارد می‌شویم و تنظیمات را مانند شکل زیر انجام می‌دهیم:



شکل ۷-۳۶

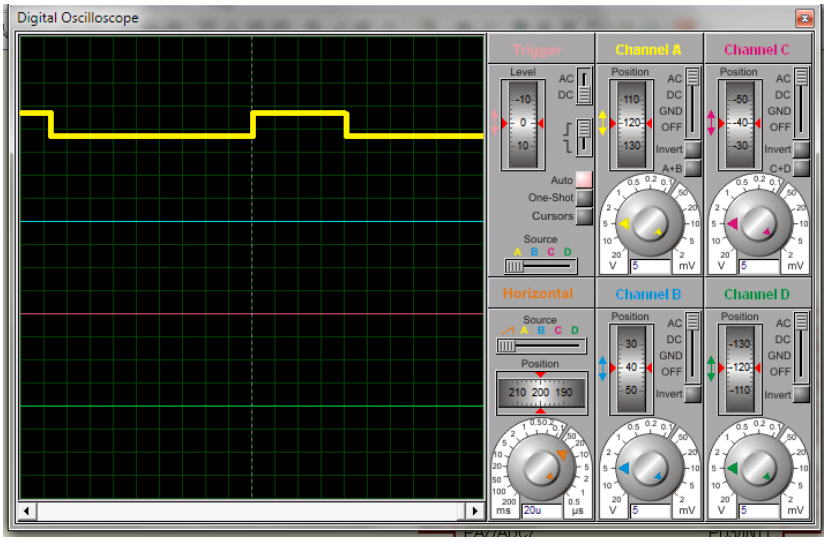
گزینه‌ی Output مربوط به نوع شکل موج خروجی می‌باشد که اگر آن را بر روی حالت Non-Inverted PWM قرار دهیم به این معناست که موج PWM تولید شده بر روی خروجی بدون تغییر اعمال می‌شود (مانند شکل بالا) و اگر آن را بر روی حالت Inverted PWM قرار دهیمPWM را NOT کرده و در خروجی نشان می‌دهد. به شکل زیر نگاه کنید:



شکل ۷-۳۷

بعد از generate کردن کد درون حلقه‌ی while(1) مقدار OCR0 را برابر ۸۰ قرار می‌دهیم. که برای این کار کافیست دستور OCR0 = 80; را تایپ کنیم.

حال عملکرد این کد را در پروتئوس مشاهده می‌کنیم. خروجی تایمر صفر بر روی PORTB.3 یا همان OC0 (پایه‌ی شماره‌ی ۴ در ATmega16) قرار دارد. پس پایه‌ی OC0 میکرو را به یک اسیلوسکوپ متصل می‌کنیم و شکل موج خروجی را مانند شکل زیر مشاهده می‌کنیم:



شکل ۷ - ۳۸

قسمت تقسیم بندی زمانی روی ۲۰ میکروثانیه است (یعنی هر مربع افقی برابر ۲۰ میکروثانیه است) اگر به شکل موج نگاه کنیم می‌بینیم که ۸۰ میکروثانیه(۴ خانه) یک و ۱۷۵ میکروثانیه (کمی کمتر از ۹ خانه) صفر است و مقدار متوسط این موج برابر تقریبا ۱،۵۶ ولت می‌شود:

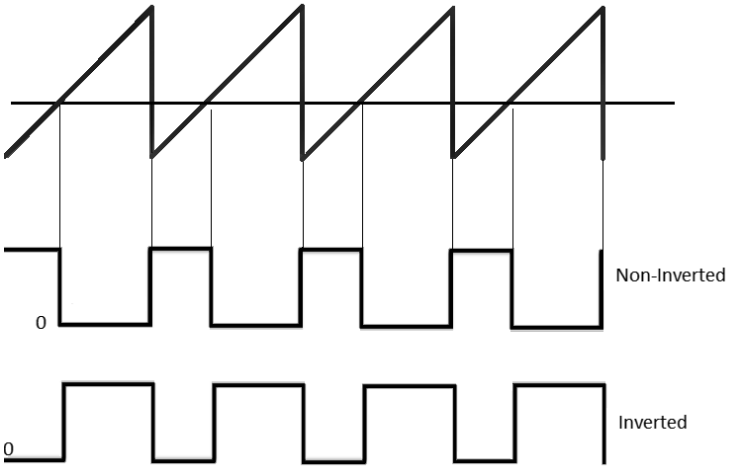
$$\frac{\text{مقدار متوسط ولتاژ}}{5} = \frac{80\mu s}{256\mu s}$$

$$\frac{80}{256} * 5v = 1.56\ v$$

یک نکته‌ی بسیار مهم در مورد مد Fast PWM

همانطور که توضیح داده شد در این مد هنگامی که مقدار تایمر(یا همان مقدار رجیستر TCNT) با مقدار OCR0 برابر می‌شود اولین تغییر و زمان سرریز دومین تغییر در خروجی رخ می‌دهد. اگر PWM در حالت Non-Inverted باشد در لحظه‌ی برابری مقدار تایمر با OCR0 خروجی برابر

صفر و در لحظه‌ی سرریز خروجی برابر یک می‌شود و اگـرPWM در حالـت Inverted باشـد در لحظه‌ی سرریز خروجی برابر صفر و در لحظه‌ی برابری با OCR0 خروجی یک می‌شود:

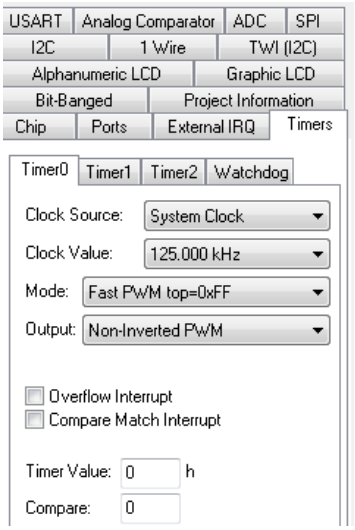


شکل ۷-۳۹

نکته‌ی بسیار مهم این است که در حالت Non-Inverted زمانی‌که می‌گوییم در لحظه‌ی برابـری مقدار تایمر با OCR0خروجی برابر صفر می‌شود منظور در لحظه‌ای است که تایمر از مقدار برابری خارج می‌شود(یعنی برای مثال اگر مقدار OCR0 برابر ۲۰۰ باشد در لحظه‌ای که مقـدار تـایمر از ۲۰۰ به ۲۰۱ می‌رود مقدار خروجی صفر می‌شود) و منظور از اینکـه در لحظـه‌ی سـرریز خروجـی برابر یک می‌شود این است که در لحظه‌ای که مقدار تایمر از حداکثر به صفر باز می‌گردد، خروجـی برابر یک می‌شود.

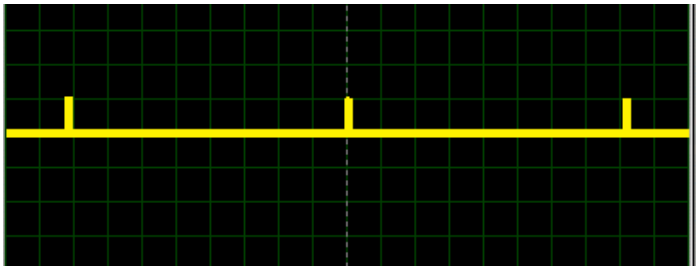
حال فرض کنید مقدار OCR0 را برابر صفر گذاشته باشیم، آن وقت با این تفاسیر زمانی‌که مقـدار تایمر از عدد صفر به عدد یک می‌رود خروجی برابر صفر می‌شود و تا مقدار حداکثر، خروجـی صـفر می‌ماند؛ ولی زمانی‌که تایمر از مقدار حداکثر به صفر سرریز می‌کند خروجی برابـر یـک مـی‌شـود و دوباره بلافاصله بعد از آنکه مقدار تایمر از صفر به یک افزایش یافت خروجی برابـر صـفر مـی‌شـود. یعنی در این حالت خروجی صفر است و فقط به اندازه‌ی یک کلاک در سطح منطقی یک قرار دارد.

مثال ۱۱: برای درک بهتر این موضوع برنامه‌ای می‌نویسیم و مقدار OCR0 را برابر صفر قرار می‌دهیم:



شکل ۷ – ۴۰ : تنظیمات مثال ۱۱

فرکانس را روی عدد دلخواه ۱۲۵ کیلو هرتز و مد را روی حالت Fast PWM قرار می‌دهیم و خروجی را Non-Inv می‌کنیم و مقدار OCR0 را برابر صفر قرار می‌دهیم، آنگاه در پروتئوس شکل موج زیر را مشاهده می‌کنیم:



شکل ۷ – ۴۱

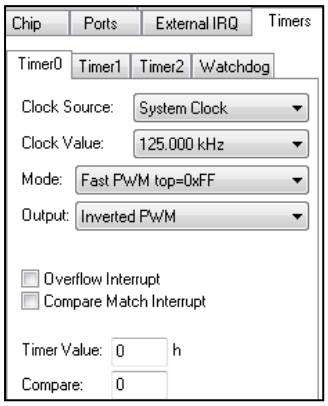
همانطور که مشاهده می‌کنید این شکل موج همیشه صفر است ولی به اندازه یک کلاک، در مقدار یک قرار دارد. این شکل موج دارای ضربه است و برای بسیاری از امور مخرب است.

مد Fast PWM در حالت Non-Inv شکل موج تمام صفر ندارد ولی در مورد شکل موج تمام یک هیچ مشکلی وجود ندارد چون اگر مقدار OCR0 برابر حداکثر باشد این موج در تمام کلاک‌ها یک است و در کلاک مربوط به سرریز هم یک است که در این حالت دارای شکل موجی کاملا یک هستیم.

از این نکته می‌توانیم استفاده کنیم و برای آنکه در مد Fast PWM شکل موج تمام صفر ایجاد کنیم، می‌توانیم مقدار OCR0 را برابر حداکثر مقدار قرار بدهیم و خروجی را روی حالت Inverted بگذاریم، که در این صورت می‌توانیم یک شکل موج تمام صفر یکپارچه داشته باشیم.

مثال ۱۲: می‌خواهیم موجی مربعی در حالت Fast PWM و به صورت Inverted در تایمر صفر ایجاد کنیم:

ابتدا تنظیمات مربوط به کدویزارد را انجام می‌دهیم:

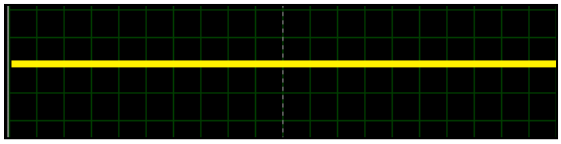


شکل ۷ - ۴۲ : تنظیمات مثال ۱۲

حال اگر بخواهیم ولتاژ خروجی صفر باشد، چون خروجی در حالت Inverted تنظیم شده است باید مقدار OCR0 را برابر ۲۵۵ قرار دهیم:

```
108  while (1)
109      {
110
111      OCR0=255;
112
113      }
114  }
```

اگر شکل موج مربوط به این موج را در پروتئوس مشاهده کنیم می‌بینیم که در این شکل موج ولتاژ خروجی به صورت یک دست صفر ولت است:



شکل ۷ - ۴۳

دقت کنید که در حالت Inverted قادر به تولید موج تمام یک نیستیم چون همانطور که بررسی کردیم در حالت Non-Inv شکل موج یک‌دست صفر ولت نداریم و چون حالت Inverted

معکوس حالت Non-Inv می‌باشد در این حالت ولتاژ یک‌دست ۵ ولت نداریم و این ولتاژ در یک کلاک دارای مقدار صفر است.
توجه داشته باشید که این حالت فقط در مورد مد Fast PWM صادق است و در مدهای دیگر PWM به‌راحتی با قرار دادن OCR0 برابر صفر یک شکل موج تمام صفر و با قرار دادن OCR0 برابر حداکثر، یک شکل موج تمام یک در اختیار خواهیم داشت.

فرکانس PWM در حالت Fast PWM

برای محاسبه‌ی فرکانس در حالت Fast PWM می‌توانیم از فرمول زیر استفاده کنیم:

$$f_{PWM}=\frac{f_{clk_{Io}}}{N*256}$$

می‌توان فرمول بالا را به صورت فرمول ساده‌تر زیر نوشت:

$$f_{PWM}=\frac{f_{clk-timer}}{256}$$

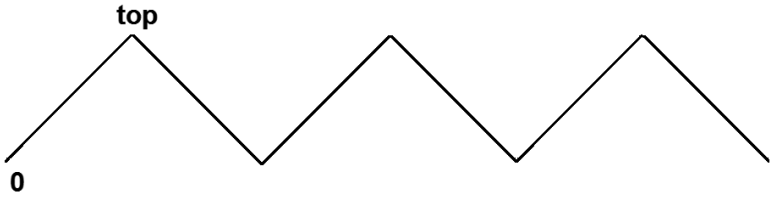
که صورت کسر، فرکانس تنظیم شده در تایمر است و مخرج عدد ۲۵۶ است که همان تعداد کلاک‌های مربوط به تایمر صفر است.

مقدار متوسط سیگنال خروجی در تایمر صفر :

$$\frac{\text{مقدار متوسط ولتاژ}}{5}=\frac{OCR0}{255}$$

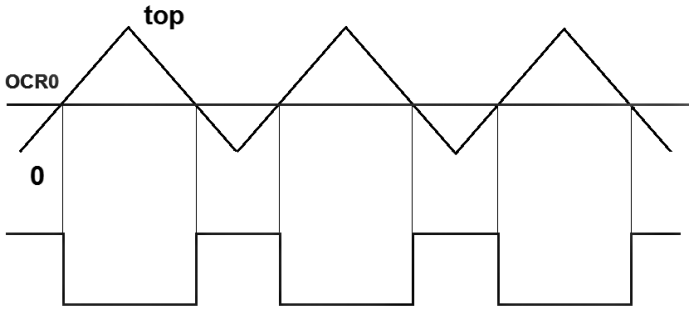
مد phase correct PWM

مدphase correct PWM(یاPWM تصحیح فاز) می‌تواند شکل موجPWM با تفکیک بالاتری را ایجاد کند. شکل موج مربوط به این مد دیگر به صورت دندان اره‌ای نیست و بعد از رسیدن به مقدار حداکثر خود به صفر باز نمی‌گردد و با همان شیبی که از صفر تا حداکثر مقدار خود رفته است از حداکثر به صفر باز می‌گردد. در حقیقت این مد به صورت مثلثی است و دارای تقارن می‌باشد:



شکل ۷ - ۴۴ : مد تصحیح فاز

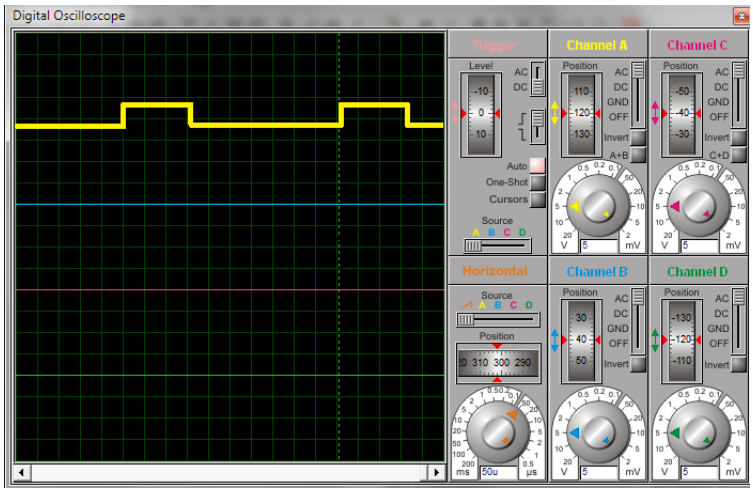
شکل موج خروجی در مدphase correct PWM در تایمر صفر در هنگام برابری مقدار TCNT0 (شمارنده‌ی تایمر صفر) با مقدار OCR0، معکوس می‌شود. به شکل زیر دقت کنید:



شکل ۷-۴۵

در این حالت همانطور که در تصویر پیداست زمان هر دوره‌ی تناوب دو برابر حالت Fast PWM است. در حقیقت فرکانس در حالتphase correct PWM نصف فرکانس در حالت Fast PWM است.

اگر بخواهیم همین مطلب را در شبیه‌ساز هم مشاهده کنیم، در تنظیمات کدویزارد تایمر صفر را روی حالتphase correct PWM و خروجی را روی حالتNon-Inverted PWM قرار می‌دهیم و مانند مثال ۱۰ مقدار OCR0 را برابر ۸۰ می‌گذاریم:



شکل ۷-۴۶

همانطور که در شکل بالا مشاهده می‌کنید با توجه به تقسیم زمان که بر روی ۵۰ میکروثانیه تنظیم شده است دوره‌ی زمانی دو برابر گشته است و خروجی ۱۶۰ میکروثانیه به صورت ۱ و ۳۵۰

میکروثانیه صفر است(یعنی فرکانس نصف حالت قبل می‌باشد) ولی در مقدار متوسط سیگنال خروجی تفاوتی ایجاد نشده و خروجی همان ۱,۵۶ ولت است:

$$\frac{160}{512} * 5v = 1.56\ v$$

محاسبه‌ی فرکانس Phase Correct PWM

برای محاسبه‌ی فرکانس در حالت Phase Correct PWM می‌توانیم از فرمول زیر استفاده کنیم:

$$f_{PWM} = \frac{f_{clk_{Io}}}{N*510}$$

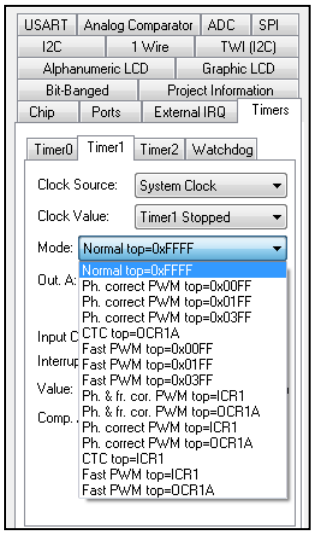
که می‌توان فرمول فوق را به صورت فرمول ساده‌تر زیر نوشت:

$$f_{PWM} = \frac{f_{clk-timer}}{510}$$

که صورت کسر مقدار فرکانس تنظیم شده در تایمر و مخرج کسر عدد ۵۱۰ می‌باشد که همان تعداد کلاک‌های مربوط به تایمر صفر است. (چون در این مد مقدار تایمر با ۲۵۵ کلاک از صفر تا حداکثر و با ۲۵۵ کلاک دیگر از حداکثر به صفر می رسد).

مدهای PWM در تایمر / کانتر یک

تایمر شماره‌ی یک همانطور که در شکل پیداست دارای ۱۲ حالت PWM می‌باشد:



شکل ۷-۴۷

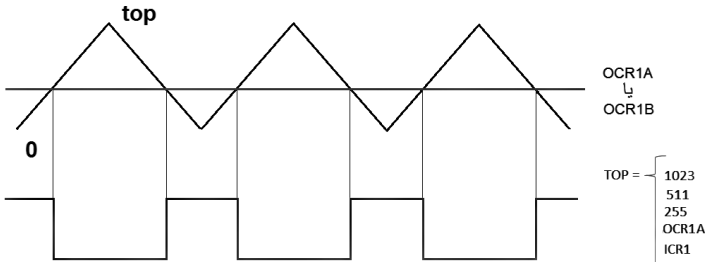
phase correct PWM در تایمر یک

سه مد اول مربوط به phase correct PWM است کـه شـامل سـه حالـت مختلـف مـی‌باشـد: PWM ۸ بیتی و ۹ بیتی و ۱۰ بیتی

حالت اول top=0x00FF است، که در این حالت حداکثر مقدار شمارنده‌ی تایمر عدد ۲۵۵ اسـت که به معنی ۸ بیتی بودن این حالت است(زیرا عدد ۲۵۵ در مبنای دو برابر ۱۱۱۱۱۱۱۱ است) در این حالت حداکثر مقدار OCR1A یا OCR1B می‌تواند برابر ۲۵۵ باشـد و مقـدار بیشـتر از آن معنا ندارد چون بالاتر از حداکثر مقدار تایمر است.

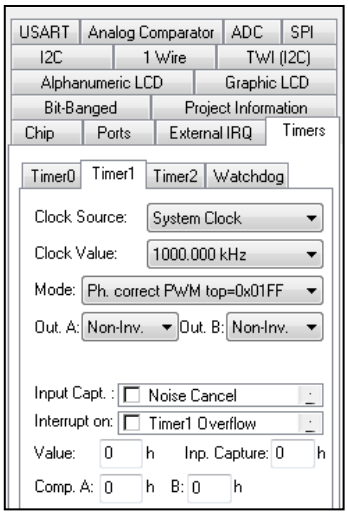
حالت دوم top=0x01FF است که حداکثر مقدار آن ۵۱۱ می‌باشد و مقدارهای بالای ۵۱۱ بـرای OCR1Aیا OCR1B بی‌معنی است.

حالت سوم نیز top=0x03FF می‌باشد که حداکثر مقدار آن ۱۰۲۳ است و مقـادیر بـالای ۱۰۲۳ برای OCR1A یا OCR1B بی معنی می‌باشد، پس باید به این نکته توجه کنیم که اگر در ایـن مدها برنامه‌ای نوشتیم حداکثر مقدار را رعایت کنیم:



شکل ۷-۴۸

مثال ۱۳: فرض کنید می‌خواهیم برنامه‌ای بنویسیم که یک موج مربعی PWM با مقـدار متوسـط ۳,۷۵ ولت در پایه‌های خروجی تولیـد شـود. ایـن برنامـه را در حالـت phase correct PWM top=0x01FF می‌نویسیم:



وارد کدویزارد می‌شویم و تنظیمات مربوطه را انجام می‌دهیم:

خروجی تایمر شماره‌ی یک پایه‌های D4 و D5 می‌باشد که شکل موج خروجی بر روی این دو پایه ظاهر می‌شود، پس این دو پایه را به صورت خروجی تنظیم می‌کنیم.
حال برای آنکه ولتاژ ۳,۷۵ ولت را ایجاد کنیم با یک نسبت تناسب ساده داریم:

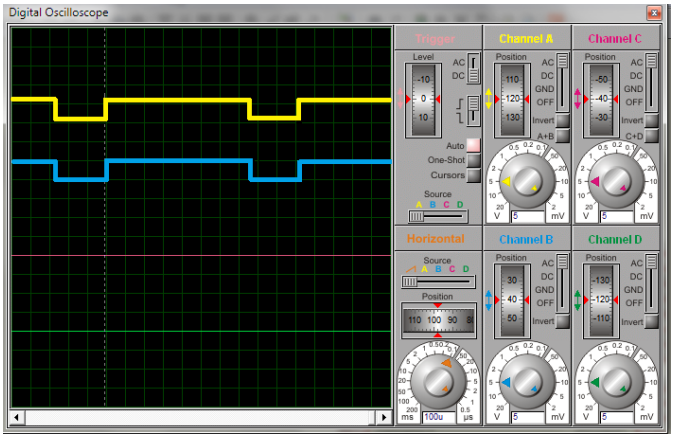
$$\frac{\text{ولتاژ مورد نظر}}{5} = \frac{x}{\text{حداکثر مقدارتایمر}}$$

$$\frac{3.75}{5} = \frac{x}{511}$$

که مقدار x برابر ۳۸۳ بدست می‌آید پس اگر مقدار OCR1A و یا OCR1B را برابر ۳۸۳ قرار دهیم موج مربعی با مقدار متوسط ۳,۷۵ ولت ایجاد می‌شود. پس کد زیر را می‌نویسیم:

```
while (1)
      {

       OCR1A=383;
       OCR1B=383;


      }
```

حال اگر ولتاژ خروجی را مشاهده کنیم به صورت شکل زیر می‌شود:



شکل ۷-۴۹

اگر به شکل نگاه کنیم خروجی ۷,۵ مربع افقی ۵ ولت و ۲,۵ مربع افقی صفر ولت می‌باشد که مقدار متوسط این شکل موج برابر ۳,۷۵ ولت می‌شود: $\frac{7.5}{7.5+2.5} * 5v = 3.75v$

فرکانس PWMدر حالت Phase Correct PWM در تایمر یک

برای محاسبه‌ی فرکانس در حالتPhase Correct PWM در تایمر یک می‌توانیم از فرمول زیر استفاده کنیم:

$$f_{PWM=}\frac{f_{clk_{Io}}}{2*N*TOP}$$

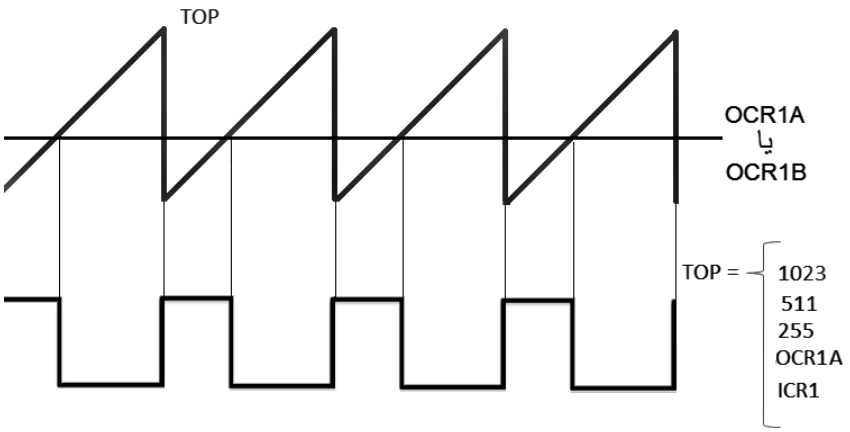
که می‌توان فرمول بالا را به صورت فرمول ساده‌تر زیر هم نوشت:

$$f_{PWM=}\frac{f_{clk-timer}}{2*TOP}$$

که صورت کسر فرکانس تنظیم شده در تایمر و مخرج کسر، دو برابر مقدار حداکثر تایمر است (مثلاً اگر PWM در حالت Phase Correct PWM،Top = 0x03FF باشد مقدار TOP برابر ۱۰۲۳ است.)

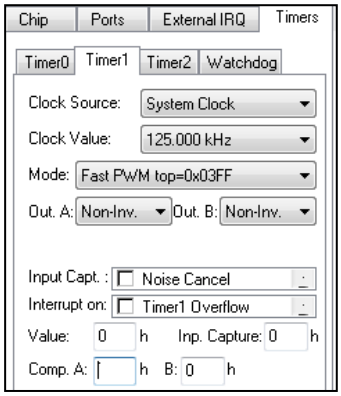
مد Fast PWM در تایمر یک

سه مد بعدی این تایمر مربوط به Fast PWM می‌باشد که مانند مد Phase Correct PWM سه حالت دارد که مقدار حداکثرهای ۲۵۵ و ۵۱۱ و ۱۰۲۳ را شامل می‌شود که در این حالت نیز برای مقداردهی به OCR1A و OCRAB باید مقدار حداکثر را رعایت کنیم:



شکل ۷-۵۰

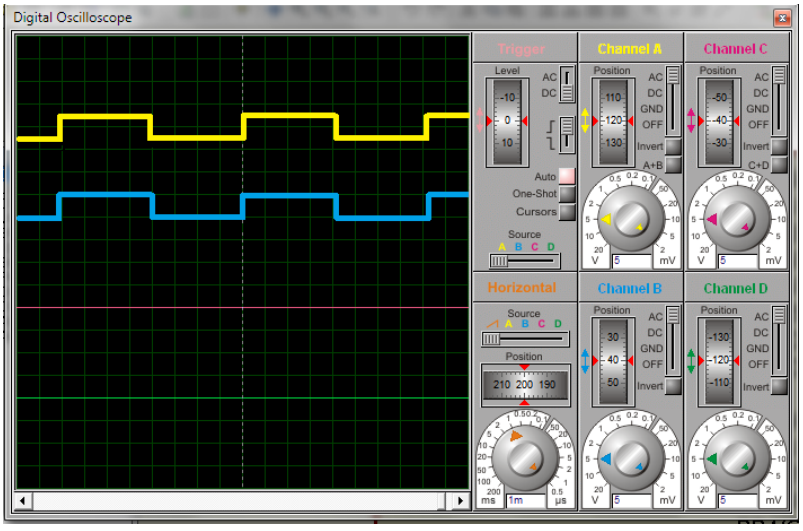
مثال ۱۴: فرض کنید قصد تولید موج PWM با مقدار متوسط ۲,۵ ولت را در مد Fast PWM داریم ابتدا کدویزارد را باز کرده و یکی از مدها را انتخاب می‌کنیم(برای مثال مد Fast PWM top = 03FF را انتخاب می‌کنیم) و فرکانس هم در حالت 125KHz قرار می‌دهیم:



و کد ساده‌ی زیر را برای مقداردهی OCR1A و OCR1B می‌نویسیم:

```
while (1)
      {

       OCR1A=511;
       OCR1B=511;

      }
}
```

شکل موج تولیدی مانند شکل زیر می‌شود:



شکل ۷-۵۱

فرکانس PWM در حالت Fast PWM در تایمر یک

برای محاسبه‌ی فرکانس در حالتFast PWM در تایمر یک می‌توانیم از فرمول زیر استفاده کنیم:

$$f_{PWM}=\frac{f_{clk_{Io}}}{N*(TOP+1)}$$

که می‌توان فرمول بالا را به صورت فرمول ساده‌تر زیر نوشت:

$$f_{PWM}=\frac{f_{clk-timer}}{TOP+1}$$

که صورت کسر، فرکانس تنظیم شده در تایمر است و مخرج، مقدار حداکثر تایمر به علاوه‌ی یک است (مثلاً اگر PWM در حالت Fast PWM Top = 0x01FF باشد مقدارTOP برابر۵۱۱ است که در این صورت مخرج ۵۱۲ می‌شود).

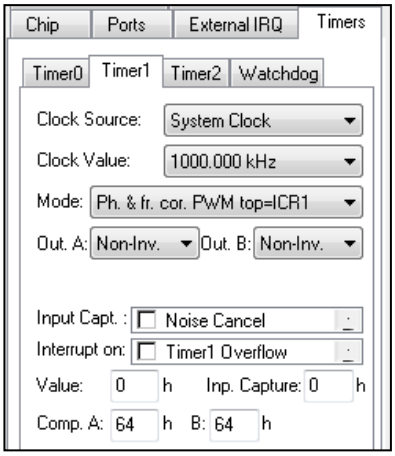
مد تصحیح فاز و فرکانس در تایمر یک

دو مد بعدی تایمر یک مربوط به مد تصحیح فاز و فرکانس می‌باشد که این مد نیز مانند مد تصحیح فاز به صورت دو شیب(یا مثلثی) می‌باشد. عملکرد این مد دقیقاً مانند مد تصحیح فاز است با این تفاوت که به روز رسانی آن با به روز رسانی تصحیح فاز متفاوت است (در این مورد در بخش به روز رسانی به طور کامل صحبت می‌کنیم).

در این مد بر خلاف ۶ مد اول حداکثر مقدار PWM ثابت نیست و این حداکثر می‌تواند به صورت دستور در برنامه نوشته شود و یا حتی در طول برنامه تغییر کند.

در مدCor. PWM top = ICR1A . Ph. &fr حداکثر مقدار شمارنده‌ی تایمر برابر ICR1A است و در مد **Ph. & fr. Cor. PWM top = OCR1A** حداکثر مقدار شمارنده‌ی تایمر برابر OCR1A است.

مثال ۱۵: قصد داریم برنامه‌ای بنویسیم که همان ولتاژ ۳,۷۵ ولت در مثال ۱۳ را تولید کند:



کدویزارد را باز کرده و مد تایمر را روی حالت **Cor. PWM top = ICR1A .Ph. &fr** قرار می‌دهیم و خروجی را روی حالت Non-Inv می‌گذاریم:

حال وارد کد می‌شویم و ICR1A را مقداردهی می‌کنیم. مقدار ICR1A را برابر ۵۱۱ قرار می‌دهیم و مقدار OCR1A و OCR1B را برابر ۳۸۳ قرار می‌دهیم که ولتاژ ۳,۷۵ ولت تولید شود:

```
// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: 1000.000 kHz
// Mode: Ph. & fr. cor. PWM top=ICR1
// OC1A output: Toggle
// OC1B output: Toggle
// Noise Canceler: Off
// Input Capture on Falling Edge
// Timer1 Overflow Interrupt: Off
// Input Capture Interrupt: Off
// Compare A Match Interrupt: Off
// Compare B Match Interrupt: Off
TCCR1A=0x50;
TCCR1B=0x12;
TCNT1H=0x00;
TCNT1L=0x00;
ICR1H=0x01;
ICR1L=0xFF;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;
```

مقدار ۵۱۱ به ICR1A

وکد داخل حلقه‌ی while(1):

```
while (1)
      {

       OCR1A=383;
       OCR1B=383;


      }
```

حال اگر عملکرد همین کد را در پروتئوس مشاهده کنیم دقیقاً همان شکل موجی که در مثال ۱۳ مشاهده کردیم را می‌بینیم.

دو مد بعدی مربوط به phase correct PWM است که مقدار حداکثر آن به جای یک مقدار مشخص می‌تواند یکی از دو رجیستر OCR1A یا ICR1A باشد. دو مد آخر هم حالتی از Fast PWMاست که مقدار حداکثر آن به جای یک مقدار مشخص می‌تواند یکی از دو رجیستر OCR1Aیا ICR1A باشد.

مدهای PWM تایمر دو نیز دقیقاً مانند مدهای PWM در تایمر صفر هستند.

به روز رسانی در مدهای مختلف PWM

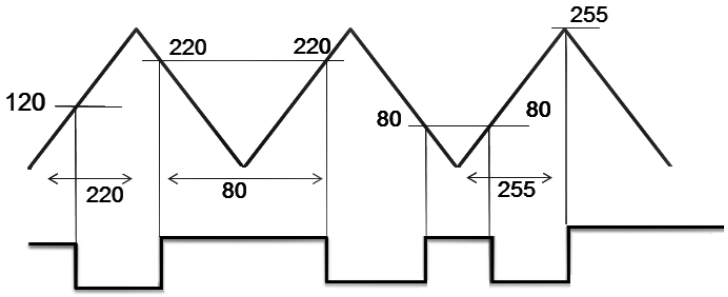
همانگونه که در بخش به‌روزرسانی مـدهای غیـر PWM مشـاهده کـردیم، بـه‌روزرسـانی مـدهای Normal و CTC به صورت لحظه‌ای و فوری بود؛ ولی به‌روزرسانی مدهای مربوط بـه PWM بـه صورت فوری نمی‌باشد. OCR1A و OCR1B در مدهای PWM دارای دو حافظه می باشند یـا به عبارت دیگر دارای بافر مضاعف هستند، یعنی زمانی‌که ما در برنامه مقـدار OCR1A را تغییـر می‌دهیم این مقدار جدید درون یک حافظه‌ی موقت(بافر) نوشته می‌شـود و در زمـانی خـاص ایـن مقدار جدید جایگزین مقدار قبلی می‌شود.به جدول زیر نگاه کنید:

Timer/Counter Mode of Operation	TOP	Update of OCRnx at	TOVn Flag Set on
Normal	0xFFFF	Immediate	MAX
PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
CTC	OCRnA	Immediate	MAX
Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
PWM, Phase and Frequency Correct	ICRn	BOTTOM	BOTTOM
PWM,Phase and Frequency Correct	OCRnA	BOTTOM	BOTTOM
PWM, Phase Correct	ICRn	TOP	BOTTOM
PWM, Phase Correct	OCRnA	TOP	BOTTOM
CTC	ICRn	Immediate	MAX
(Reserved)	–	–	–
Fast PWM	ICRn	BOTTOM	TOP
Fast PWM	OCRnA	BOTTOM	TOP

جدول ۷ - ۲: به روز رسانی

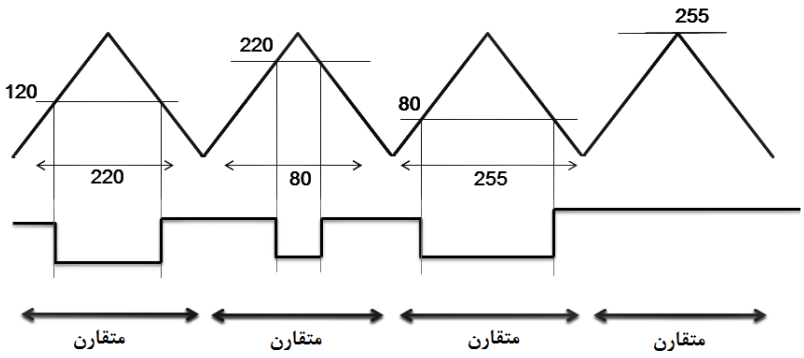
به‌روزرسانی مدهای Phase Correct PWM (یا همان تصحیح فاز) در هنگامی انجام می‌شـود که مقدار شمارنده‌ی تایمر (مقدار رجیستر TCNTn) به حداکثر مقدار خود برسد. برای مثـال اگـر تایمر صفر در حال شمارش باشد و مقدار OCR0 برابر ۱۲۰ باشد و در برنامـه مقـدار آن را تغییـر

بدهیم و برابر ۲۲۰ کنیم، این تغییر در همان لحظه اعمال نمی‌شود و زمانی مقدار OCR0 برابر ۲۲۰ می‌شود که مقدار TCNT0 به حداکثر خود برسد، یعنی زمانی که برابر ۲۵۵ شود، به شکل زیر دقت کنید:



در این تصویر زمانی‌که مقدار TCNT0 بین ۰ تا ۲۵۵ است(یعنی در حال افزایش است) دستور OCR0=220 نوشته شده است ولی این تغییر اعمال نشده تا زمانی‌که مقدار TCNT0 به حداکثر رسیده است و در آن لحظه مقدار جدید، یعنی ۲۲۰، درون OCR0 ریخته شده است و بعد از آن دوباره دستور OCR0=80 نوشته شده ولی باز این مقدار جدید در OCR0 وارد نشده است، این تغییر زمانی اعمال شده است که مقدار TCNT0 به حداکثر مقدار خود رسیده است. به همین دلیل این شکل موج کمی نامتقارن است.در مد Fast PWM نیز تغییرات آنی رخ نمی‌دهد و همانطور که در جدول نوشته شده است زمانی‌که مقدار TCNT به حداقل مقدار خود یا همان صفر رسید تغییرات اعمال می‌شود.

در مد تصحیح فاز و فرکانس (Phase and Frequency Correct) نیز این تغییرات در حداقل مقدار TCNTیعنی در صفر رخ می‌دهد. این تغییر در صفر باعث می‌شود شکل موج این مد نسبت به مد Phase Correct دارای تقارن بهتری باشد. برای مثال به شکل زیر نگاه کنید:



در تصویر قبل تغییرات، زمانی رخ داده است که مقدار تایمر به حداقل مقدار خود رسیده است که این باعث شده که شکل موج در هر سیکل دارای تقارن باشد، یعنی شکل موج از زمان افزایش تایمر تا زمان کاهش و رسیدن به صفر دارای تقارن می‌باشد.
دانستن این به روز رسانی‌ها و انواع شکل موج‌های حاصله در زمان تغییرات OCR0 یا OCR1A یا OCR1B به ما در انتخاب مد PWM در کاربردهای مختلف کمک می‌کند.

تفاوت مدهای PWM

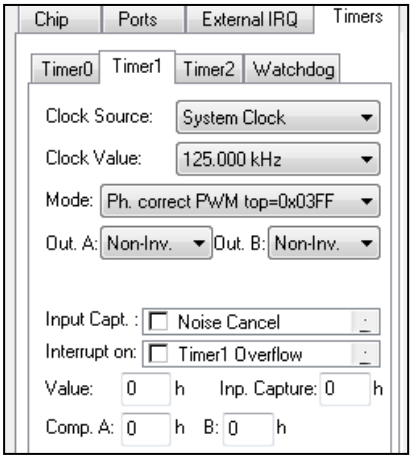
مدهای مربوط به Fast PWM دارای شکل موج دندان اره‌ای هستند در حالی که شکل موج‌های Phase Correct PWM و Phase and Frequency PWM دارای شکل موج مثلثی با دو شیب هستند که به دلیل تقارن بهتر معمولا برای کنترل دور موتورها، این دو مد ترجیح داده می‌شوند.
فرکانس مدFast PWM دو برابر فرکانس دو مد دیگر است و دو مد تصحیح فاز و تصحیح فاز و فرکانس(Phase Correct PWM وPhase and Frequency PWM) دارای تفکیک‌پذیری بهتری هستند.
در مد Fast PWM در حالت Non-Inverted امکان داشتن شکل موج صفر یک‌دست نداریم و شکل موج به اندازه‌ی یک کلاک برابر یک می‌باشد.
مدPhase and Frequency PWM نسبت به مد Phase Correct PWM به دلیل به‌روزرسانی مقدار OCR در حداقل مقدار دارای تقارن بهتری می‌باشد.

PWM کاربردی

تا به اینجا با تایمرها و انواع شکل موج‌های PWM و تفاوت‌های آنها نسبت به یکدیگر آشنا شدیم. حال قصد داریم چند کاربرد آن را بررسی کنیم.
موتور یکی از پرکاربردترین وسایل صنعتی است و به چرخش در آوردن و کنترل سرعت آن از جمله مهمترین کارهایی است که می‌توانیم با میکروکنترلرها انجام دهیم.

مثال ۱۶: قصد داریم یک موتور بسیار کوچک را به چرخش درآوریم:

برای اینکار ابتدا یک برنامه‌ی ساده می‌نویسیم که یک شکل موج PWM ساده تولید شود:

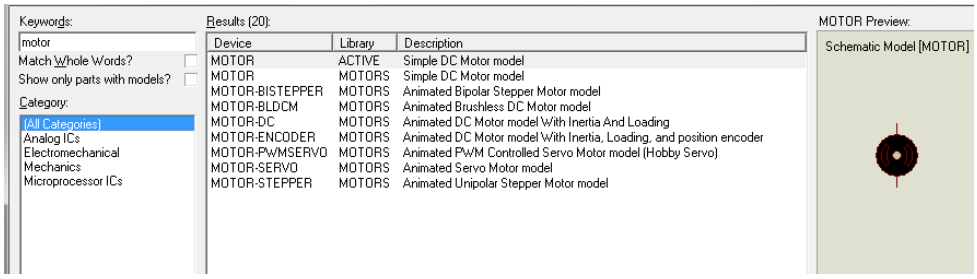


شکل ۷ – ۵۲ : تنظیمات مثال ۱۶

و برای مثال مقدار OCR1A و OCR1B را برابر عدد ۵۱۱ قرار می‌دهیم که مقدار متوسط ولتاژ خروجی برابر ۲٫۵ ولت شود:

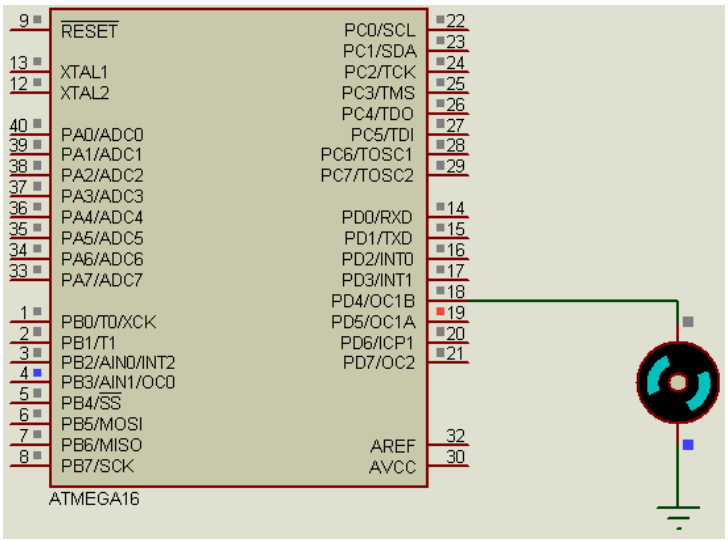
```
while (1)
      {

       OCR1A=511;
       OCR1B=511;

      }
```

سپس برای شبیه‌سازی برنامه‌ی پروتئوس را باز می‌کنیم و مطابق شکل یک موتور DC (موتور جریان مستقیم) را انتخاب می‌کنیم:



شکل ۷ – ۵۳ : انتخاب موتور در پروتئوس

و مدار زیر را در پروتئوس می بندیم:



شکل ۷ - ۵۴

با اجرای برنامه مشاهده می‌کنیم که موتور به آرامی می‌چرخد. اگر مقدار OCR1B را به جای ۵۱۱ برابر ۱۰۲۳ قرار دهیم سرعت موتور بیشتر می‌شود و اگر مقدار آن را کمتر از ۵۱۱ قرار دهیم سرعت آن کمتر می‌شود.

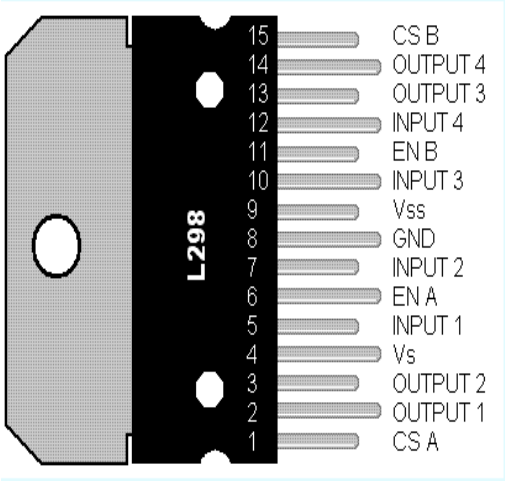
اینگونه می‌توانیم سرعت یک موتور DC را کنترل کنیم ولی بستن موتور به این شکل به میکرو فقط در مورد موتورهای بسیار کوچک جواب می‌دهد و موتورهای DC را اینگونه و مستقیم به میکرو متصل نمی‌کنیم زیرا همانطور که می‌دانید ولتاژ خروجی میکرو ۵ ولت و جریان خروجی آن بسیار کم می‌باشد که توانایی چرخاندن این گونه موتورها را ندارد لذا برای آنکه موتور را کنترل کنیم به یک وسیله به نام درایور نیاز داریم که بتواند این ولتاژ و جریان کمی که توسط میکرو اعمال می‌شود را دریافت کند و متناسب با آن ولتاژ و جریان چند برابر را به موتور تحویل دهد.

یکی از درایورهایی که برای موتور DC استفاده می‌شود آی‌سی L298 است. این درایور می‌تواند شکل موج PWM را از میکرو دریافت کند و همان شکل موج با دامنه و مقدار متوسط چند برابر را به موتور تحویل دهد. برای مثال فرض کنید در حالت PWM تایمر صفر مقدار OCR0 را برابر ۱۲۸ قرار دهیم، میدانیم که چون حداکثر مقدار تایمر صفر ۲۵۵ است مقدار متوسط ولتاژ خروجی برابر نصف حداکثر ولتاژ خروجی میکرو (یعنی ۵ ولت) است که برابر ۲,۵ ولت می‌شود. حال اگر ولتاژ تغذیه‌ی این درایور را برابر ۱۵ ولت قرار بدهیم، همانطور که ولتاژ حداکثر درایور ۳ برابر ولتاژ حداکثر میکرو است، ولتاژ متوسط تحویلی به موتور نیز سه برابر ۲,۵ ولت می‌شود که برابر ۷,۵

ولت می‌باشد که این ولتاژ به موتور می‌رسد(البته علاوه بر افزایش ولتاژ، جریانی که می‌تواند به موتور بدهد نیز تا ۲ آمپر افزایش می‌یابد.)

درایور L298

حال این درایور و پایه‌های آن را بررسی می‌کنیم:



شکل ۷ – ۵۵ : درایور L298

این آی‌سی توانایی درایور دو موتور را به طور هم‌زمان دارد. عملکرد هر یک از پایه‌های این آی‌سی به صورت زیر می‌باشد:

پایه‌های ۱ و ۱۵"Current sensing" نام دارند و برای کنترل جریان موتور استفاده می‌شوند و اگر قصد کنترل جریان را نداشته باشیم این دو پایه را زمین می‌کنیم.

پایه‌های ۲ و ۳ (OUTPUT1 و OUTPUT2) دو خروجی هستند که به دو پایانه‌ی موتور اول متصل می‌شوند.

پایه‌های۱۳و۱۴(OUTPUT3 و OUTPUT4) دو خروجی هستند که به دو پایانه‌ی موتور دوم متصل می‌شوند.

پایه‌ی شماره ۴ پایه‌ی تغذیه‌ی آی‌سی است که برابر حداکثر دامنه‌ی شکل موج می‌باشد.

پایه‌های ۵ و ۷ برای تعیین جهت موتور اول هستند به این صورت که اگر یکی از آنها یک و دیگری صفر باشد موتور در یک جهت می‌چرخد و با عوض کردن جای صفر و یک پایه‌ها جهت چرخش موتور معکوس می‌شود و اگر هر دو پایه صفر یا هر دو یک باشند موتور به چرخش در نمی‌آید.

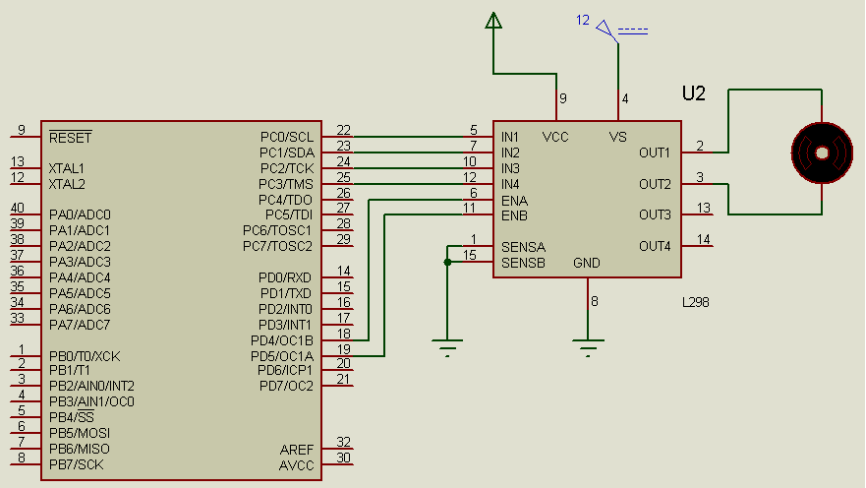
پایه‌های ۱۰ و ۱۲ نیز جهت چرخش موتور دوم را تعیین می‌کنند.

پایه‌های ۶ و ۱۱ (ENABLE) به پایه‌های مربوط بهPWM میکرو متصل می‌شوند.

پایه‌های ۸ و ۹ نیز پایه‌های تغذیه‌ی منطقی خود آی‌سی هستند و به‌ترتیب به زمین و ۵ ولت متصل می‌شوند.

حال برای شبیه‌سازی پروتئوس را باز کرده و همان برنامه‌ی قبل را به وسیله‌ی درایور L298 به موتور متصل می‌کنیم.

پایه‌های این آی‌سی را همانطور که توضیح داده شد وصل می‌کنیم:



شکل ۷-۵۶

همانطور که گفته شد پایه‌های IN1 و IN2 جهت چرخش موتور را تعیین می‌کنند. این پایه‌ها را به POTRTC.0 و PORTC.1 وصل می‌کنیم که جهت چرخش موتور را تعیین کنیم برای مثال اگر POTRTC.0 را یک و PORTC.1 را صفر کنیم(یعنی IN1 یک و IN2 صفر باشد) موتور در جهت پاد ساعتگرد شروع به چرخش می‌کند.

چون فعلاً قصد کنترل جریان موتور را نداریم پایه‌های SENSA و SENSB را به زمین وصل می‌کنیم. پایه‌های ENA و ENB مربوط به PWM مربوط به هرکدام از موتورهاست که به OCR1A و OCR1B متصل می‌کنیم. Vs را نیز به یک منبع DC متصل می‌کنیم.

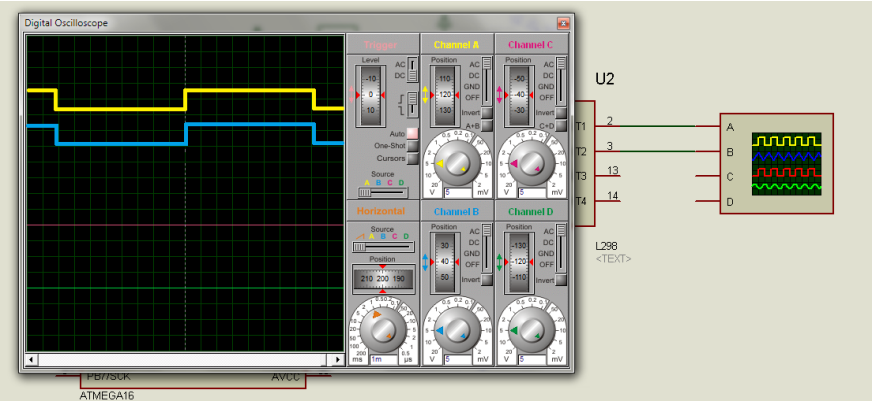
(که در این مثال آن را به یک منبع ۱۲ ولت متصل کرده‌ایم.)

در این برنامه باید پایه‌های مربوط به PORTC را به صورت خروجی تنظیم کنیم و برای چرخاندن موتور اول، می‌توانیم برنامه‌ی زیر را بنویسیم:

```
while (1)
      {

       PORTC.0=1;
       PORTC.1=0;
       OCR1A=511;

      }
```

با اجرای برنامه می‌بینیم که سرعت چرخش موتور بسیار بیشتر از حالتی است که بدون درایور آن را می‌چرخاندیم.

برای مشاهده‌ی شکل موجی که به موتور می رسد می‌توانیم خروجی L298 (پایـه‌هـای OUT1 و OUT2) را به اسیلوسکوپ بزنیم و شکل موج خروجی را مشاهده کنیم:



شکل ۷-۵۷

این شکل موج مربوط به هر یک از پایه‌های خروجی درایور می‌باشد. برای ما اخـتلاف پتانسـیل دو سر موتور اهمیت دارد که باعث چرخش موتور می‌شود. پس برای مشاهده‌ی اختلاف این دو شـکل موج مطابق شکل شماره‌ی بالا ابتدا موج پایین را بـا زدن دکمـه‌ی Invert معکـوس مـی‌کنـیم و سپس با زدن دکمه‌ی A+B اختلاف پتانسیل دو سر موتـور را مشـاهده مـی‌کنیم(مـی‌دانـیم کـه اختلاف پتانسیل دو سر موتور A-B است پس ابتدا A را به A– تبدیل می‌کنیم سپس با B جمع می‌کنیم):

شکل ۷-۵۸

ولتاژ دو سر موتور حدود ۶ ولت افتاده است (زیرا هر خانه‌ی عمودی بر روی حالت ۵ ولت قرار دارد و این شکل موج کمی بیشتر از یک خانه عمودی بالا رفته است.)

و اگر مقدار OCR1A را برای مثال برابر ۹۰۰ قرار دهیم، چون حداکثر این مد PWM ،برابر ۱۰۲۳ است. مقدار متوسط این شکل موج از فرمول زیر بدست می آید:

$$\frac{\text{مقدار متوسط ولتاژ}}{5} = \frac{\text{مقدار } \boldsymbol{OCR1A}}{\text{مقدار حداکثر}}$$

که در حالت TOP =0X03FF و مقدار OCR1A برابر ۹۰۰ داریم:

$$\frac{\text{مقدار متوسط ولتاژ}}{5} = \frac{\mathbf{900}}{\mathbf{1023}}$$

که مقدار متوسط برابر حدود ۴,۴ ولت می‌شود. که اگر ولتاژ تغذیه‌ی درایور ۱۲ ولت باشد ولتاژ ۱۰,۵۶ ولت به موتور می رسد:

$$(\mathbf{4.4 * \frac{12}{5} = 10.56})$$

Watchdog

تب آخر تایمر مربوط می‌شود به Watchdog که به معنای سگ نگهبان است.

فرض کنید یک برنامه‌ی حساس نوشته‌اید و برایتان بسیار مهم است که میکروکنترلر در هنگام اجرای خطوط برنامه هنگ نکند، ولی بنا به شرایط دمایی و یا کم و زیاد شدن ولتاژ و جریان ورودی و یا شرایط دیگری مانند نویز ممکن است میکروکنترلر دچار مشکل شود و هنگ کند. مثلاً اگر این برنامه برای یک ربات پرنده باشد و میکروکنترلر در آسمان هنگ کند بسیار خطرناک است و حتی اگر به کسی هم آسیب نزند به زمین برخورد می‌کند و آسیب می‌بیند پس به یک واحد مانند سگ نگهبان نیاز داریم که هرزمان میکروکنترلر هنگ کرد بفهمد و فوراً به میکرو دستور Reset را بدهد.

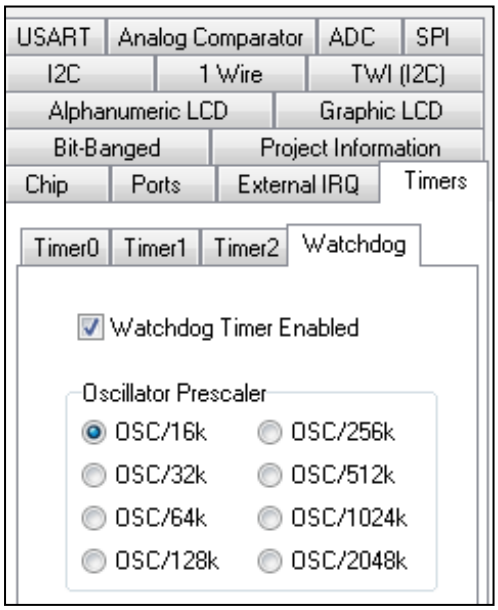
در حالت Watchdog یک شمارنده‌ی داخلی شروع به شماردن می‌کند و زمانی‌که به حداکثر مقدار خود رسید و سرریز اتفاق افتاد به میکرو دستور ریست را می‌دهد.

حال ما در خطوط برنامه دستوری می‌نویسیم که این شمارش را برابر صفر قرار دهد(یعنی به محض رسیدن برنامه به این دستور شمارنده‌ی مربوط به Watchdog را برابر صفر قرار دهد و شمارش از ابتدا شروع شود.)

اگر همه‌ی خطوط برنامه درست و سر وقت اجرا شوند این شمارش به صورت سریع به صفر بازمی‌گردد و هیچگاه به مقدار حداکثر خود نمی‌رسد و سرریز رخ نمی‌دهد و میکرو ریست نمی‌شود، اما زمانی‌که میکرو هنگ کند و خطوط برنامه اجرا نشوند و برنامه شمارش را به صفر بازنگرداند شمارنده‌ی داخلی به حداکثر مقدار خود می‌رسد و سرریز رخ می‌دهد و میکرو ریست می‌شود.

برای صفر کردن شمارنده‌ی داخلی Watchdog باید کد اسمبلی (“wdr”)#asm را در برنامه بنویسیم برای آنکه مطمئن شویم این شمارنده به موقع صفر می‌شود این کد را در تمام حلقه‌ها و وقفه‌های برنامه می‌نویسیم.

برای فعال کردن Watchdog ابتدا تیک مربوط به Watchdog Timer Enable را می‌زنیم:



شکل ۷ – ۵۹ : سگ نگهبان (Watchdog)

میکروکنترلر برای انجام عملیات Watchdog یک کلاک جداگانه و مستقل از کلاک پردازنده(CPU) دارد که کار شمارش را انجام می‌دهد. فرکانس این کلاک در شرایط استاندارد(ولتاژ تغذیه ۵ ولت و دمای ۲۵ درجه) حدود 1MHz می‌باشد که این مقدار می‌تواند با تغییر ولتاژ و جریان ورودی کمتر شود. با تنظیمات مربوط به این واحد می‌توانیم تقسیمی از فرکانس کلاک Watchdog را برای انجام این عملیات انتخاب کنیم. برای مثال اگر حالت OSC/16k را انتخاب کنیم در این حالت فرکانس Watchdog در حالت کاری عادی برابر $\frac{1MHz}{16.384K} = 61Hz$ می‌شود که به این معناست که میکرو در هرثانیه، ۶۱ بار ریست می‌شود(البته اگر ما در برنامه شمارنده‌ی Watchdog را برابر صفر نکنیم یعنی اگر دستور مربوط به صفر شدن شمارنده‌یWatchdog را ننویسیم). در حالت‌های بعدی زمان ریست شدن میکروکنترلر کمی طولانی‌تر است، مثلاً در حالت OSC/2048k میکرو دوثانیه‌ای یک‌بار ریست می‌شود(اگر دستور صفر شدن شمارنده نوشته نشود.)

در جدول زیر مقدار نوعی(Typical)زمان ریست شدن میکرو در صورت صفر نکردن شمارنده‌ی Watchdog آمده است:

فرکانس Watchdog	مقدار حقیقی	زمان ریست(ثانیه) در ولتاژ تغذیه ۳ ولت	زمان ریست(ثانیه) در ولتاژ تغذیه ۵ ولت
OSC/16k	16.384k	0.0173	0.0163
OSC/32k	32.768k	0.0343	0.0325
OSC/64k	65.536k	0.0685	0.065
OSC/128k	131.072k	0. 14	0. 13
OSC/256k	262.144k	0. 27	0. 26
OSC/512k	524.288k	0. 55	0. 52
OSC/1024k	1,048.576k	1.1	1
OSC/2048k	2,097152	2.2	2.1

جدول ۷-۳

برای فعال کردن Watchdog تنها کاری که باید انجام بدهیم زدن تیک Watchdog Timer Enable و انتخاب یکی از حالت‌ها است.

دستوری که برای صفر کردن شمارنده‌ی داخلی Watchdog باید بنویسیم کد اسمبلی #asm("wdr") است که باید این کد را در تمام حلقه‌ها و توابع برنامه بنویسیم، اگر حجم دستورات درون While(1) کم باشد کافیست فقط یک‌بار این کد را درون این حلقه بنوسیم ولی اگر حجم دستورات درون While(1) زیاد باشد می‌توانیم این کد را یک‌بار در ابتدا و یک‌بار در میانه‌ی حلقه بنویسیم.

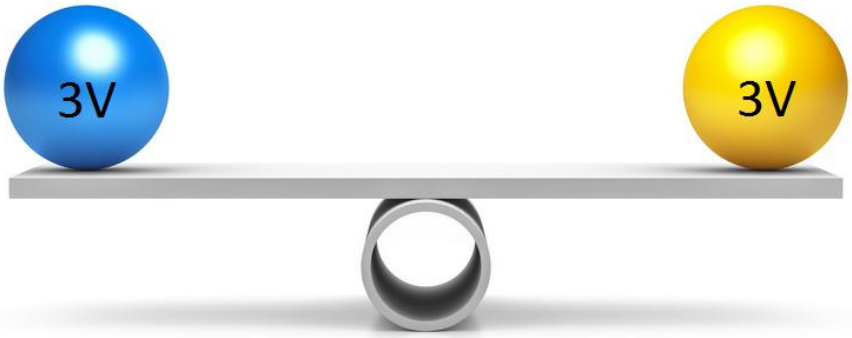
```
131  // Watchdog Timer initialization
132  // Watchdog Timer Prescaler: OSC/16k
133  #pragma optsize-
134  WDTCR=0x18;
135  WDTCR=0x08;
136  #ifdef _OPTIMIZE_SIZE_
137  #pragma optsize+
138  #endif
139
140  while (1)
141        {
142
143          #asm("wdr")
144
145        }
146  }
```

کد مربوط به ریست کردن شمارنده‌ی watchdog

منظور از ریست کردن Watchdog صفر کردن شمارنده‌ی مربوط به تایمر Watchdog است که به مقدار حداکثر خود نرسد و سرریز رخ ندهد.

فصل هشتم

مقایسه کننده‌ی آنالوگ



در این فصل خواهیم خواند:

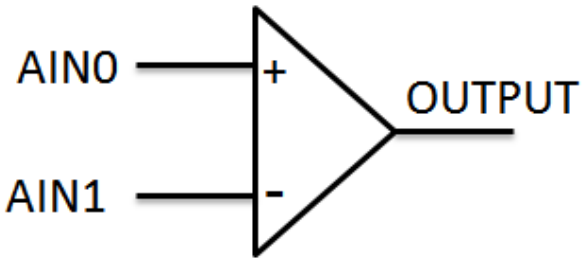
۱. معرفی واحد Analog Comparator
۲. بخش Analog Comparator Interrupt
۳. بخش Analog Comparator Interrupt Capture
۴. بخش Negative Input Multiplexer
۵. بخش Bandgap Voltage Reference

مقایسه‌کننده آنالوگ(Analog Comparator)

مقایسه‌کننده‌ی آنالوگ واحدی در میکروکنترلرهای AVR است که امکان مقایسه‌ی دو ولتاژ آنالوگ(ولتاژی بین ۰ تا ۵ولت) را به ما می‌دهد.شاید ابتدا این نکته به نظر برسد که این مقایسه می‌تواند توسط واحد ADC نیز انجام شود و نیازی به این واحد نیست ولی در ادامه خواهیم دید که این واحد امکانات ویژه‌ای دارد که کاربردهای مخصوص به خود را دارد.

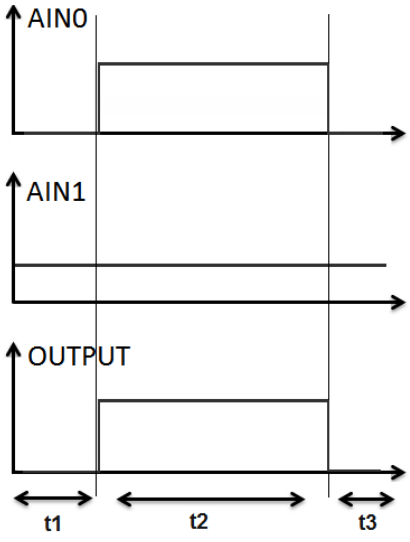
مقایسه‌کننده‌ی آنالوگ می‌تواند ولتاژ پایه‌های AIN0(پایه‌ی PB2) و AIN1(پایه‌ی PB3) را با هم مقایسه کرده و متناسب با آن کارهای مختلفی انجام دهد.

پایه‌ی AIN0 به عنوان پایه‌ی مثبت و پایه‌ی AIN1 به عنوان پایه‌ی منفی در نظر گرفته می‌شود.



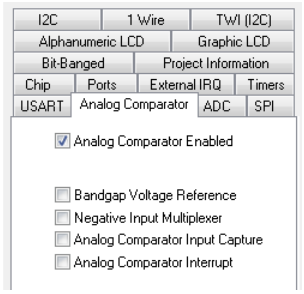
شکل ۸-۱: مقایسه‌کننده‌ی آنالوگ

زمانی‌که ولتاژ پایه‌ی مثبت(AIN0) از ولتاژ پایه‌ی منفی(AIN1) بیشتر باشد خروجی یک و زمانی که ولتاژ پایه‌ی منفی(AIN0) بیشتر از ولتاژ پایه‌ی مثبت(AIN1) باشد خروجی صفر می‌شود. برای مثال شکل زیر را در نظر بگیرید:



شکل ۸-۲: مقایسه دو ورودی مقایسه‌کننده و خروجی مربوطه

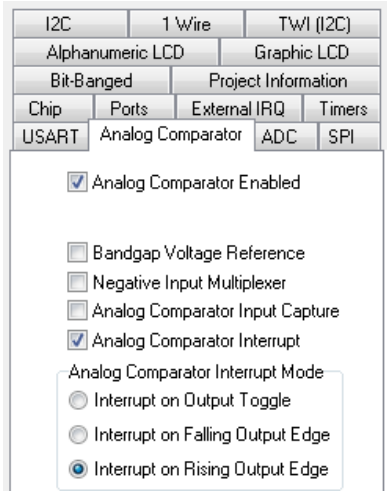
در زمان‌هایی که ولتاژ پایه‌ی AIN1 از AIN0 بیشتر است(زمان‌های t1 و t3) ولتاژ خروجی صفر و زمان‌هایی که ولتاژ پایه‌ی AIN0 از AIN1 بیشتر است(زمان t2) ولتاژ خروجی یک است.
در میکروکنترلرهای AVR قسمتی به عنوان مقایسه‌کننده آنالوگ وجود دارد که دارای چهار حالت می‌باشد. برای شروع کدویزارد را بازکرده و به گزینه‌های موجود در تب Analog Comparator نگاه‌کنید.با زدن گزینه‌ی Analog Comparator Enable مقایسه‌کننده آنالوگ فعال می‌گردد.چهار حالت مختلف وجود دارد که می‌توانیم هر کدام را انتخاب کنیم، این گزینه‌ها را از پایین به بالا یک‌به‌یک توضیح می‌دهیم:



شکل ۸-۳: گزینه‌های مقایسه‌کننده آنالوگ

گزینه‌ی Analog Comparator Interrupt

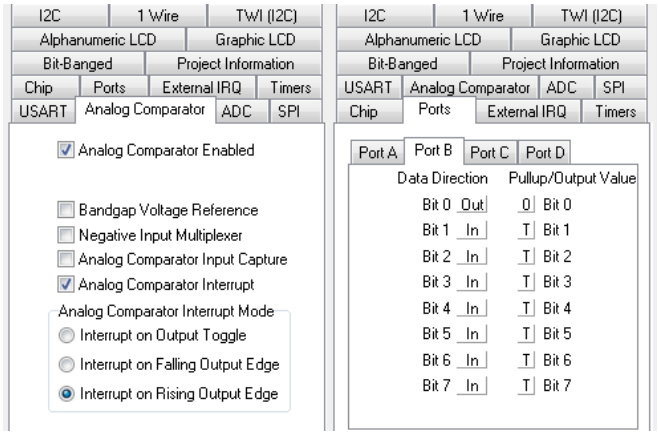
این گزینه مربوط به فعال‌کردن وقفه‌ی مقایسه‌کننده آنالوگ می‌باشد. با زدن این گزینه سه‌حالت برای انتخاب ظاهر می‌شود:



در حالت اول، زمانی‌که مقدار خروجی تغییر کند(یعنی خروجی از یک به صفر و یا از صفر به یک تغییر کند) وقفه‌ی مقایسه‌کننده‌آنالوگ فعال و برنامه وارد وقفه می‌شود.در حالت دوم، تابع وقفه زمانی فعال می‌شود که خروجی لبه‌ی پایین‌رونده را طی کند، یعنی مقدار خروجی از یک به صفر تغییر کند.در حالت سوم، وقفه هنگامی فعال می‌گردد که خروجی، لبه‌ی بالارونده را طی کند(یعنی از صفر به یک تغییر کند).

مثال ۱: برای مثال برنامه‌ای می‌نویسیم که ولتاژهای دو پایه‌ی مربوط به AIN0 و AIN1 را مقایسه کند و اگر ولتاژ پایه‌یAIN0 بیشتر بود یک LED را روشن کند. برای این‌کار ابتدا کدویزارد را باز کرده و مقایسه‌کننده‌آنالوگ را فعال می‌کنیم و آن را

روی حالت وقفه‌ی مربوط به لبه‌ی بالارونده می‌گذاریم و PORTB.0 را خروجی می‌کنیم که LED را به آن پایه نصب کنیم:

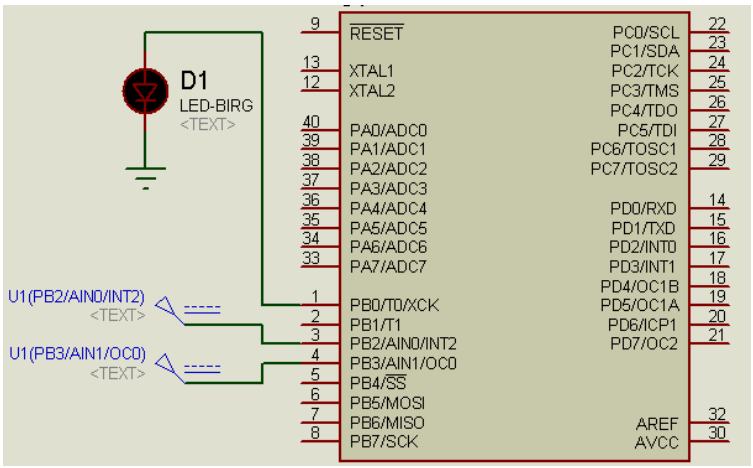


شکل ۸-۵: تنظیمات مثال ۱

سپس درون وقفه‌ی مربوطه دستور روشن کردن PORTB.0 را می‌نویسیم:

```
// Analog Comparator interrupt service routine
interrupt [ANA_COMP] void ana_comp_isr(void)
{
PORTB.0=1;
}
// Declare your global variables here
void main(void)
{
```

حال برای شبیه‌سازی پروتئوس را بازکرده و پس از بستن LED به پایه‌ی B0، دو منبع ولتاژ DC را به دو پایه‌ی AIN0 و AIN1 وصل می‌کنیم:



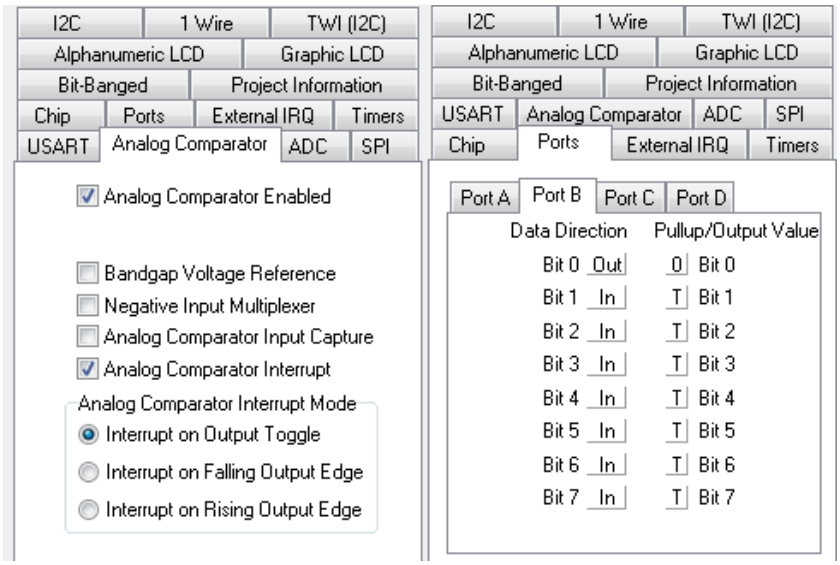
شکل ۸-۶: مدار مثال ۱

برای مقداردهی به منابع ولتاژ DC بر روی آنها دبل کلیک کرده و مقدار آنها را برابر ولتاژ موردنظر قرار می‌دهیم. برای مثال ولتاژ منبع متصل به AIN0 را برابر ۴ولت و ولتاژ منبع متصل به AIN1 را برابر ۳ولت قرار می‌دهیم. در این صورت ولتاژ پایه‌ی مثبت بیشتر از پایه‌ی منفی است و برنامه وارد وقفه شده و LED روشن می‌شود.

به این نکته توجه کنید که در این برنامه، وقفه را بر روی لبه‌ی بالارونده تنظیم کرده‌ایم یعنی زمانی‌که مقدار خروجی از صفر به یک رسید برنامه وارد وقفه شود و در اینجا چون مقدار پایه‌ی مثبت بیشتر از پایه‌ی منفی است به محض شروع برنامه، مقدار خروجی یک می‌شود پس در همین لحظه برنامه وارد وقفه شده و LED را روشن می‌کند.

مثال ۲: برنامه‌ای بنویسیم که هر بار برنامه وارد تابع وقفه شد ولتاژ پایه‌ی B0 یا همان ولتاژ LED، برعکس(Not) شود(یعنی در ابتدای برنامه ولتاژ این پایه صفر است، با ورود به وقفه یک شده و با ورود دوباره به وقفه صفر می‌شود و ...).

کدویزارد را باز می‌کنیم و برای مثال حالت وقفه را بر روی Toggle قرار می‌دهیم که با هر تغییر در خروجی مقایسه‌کننده، برنامه وارد وقفه شود:

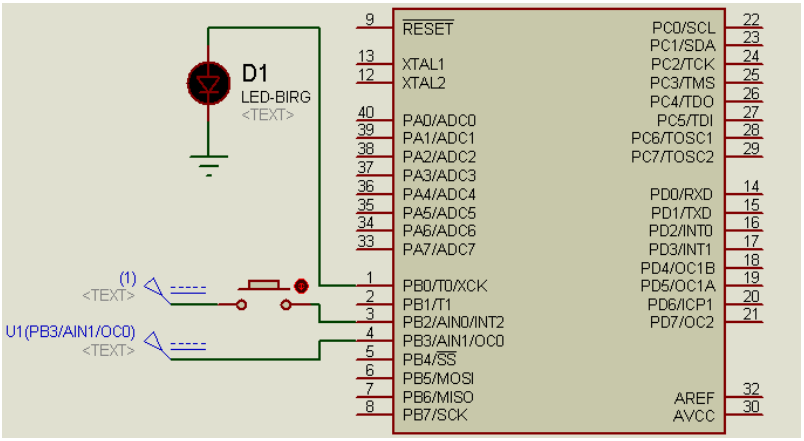


شکل ۸-۷: تنظیمات مثال ۲

سپس برنامه‌ی زیر را درون وقفه می‌نویسیم که هر زمان برنامه وارد وقفه شد ولتاژ پایه‌ی B0 معکوس گردد.

```
// Analog Comparator interrupt service routine
interrupt [ANA_COMP] void ana_comp_isr(void)
{

PORTB.0=~PORTB.0;

}

// Declare your global variables here

void main(void)
{
// Declare your local variables here

```

اکنون مطابق شکل منبع متصل به پایه‌ی AIN0 را با یک کلید(Button) به این پایه وصل می‌کنیم. منبع متصل به AIN1 را روی ۳ولت و منبع متصل به AIN0 را برابر ۴ولت قرار می‌دهیم:



شکل ۸-۸: مدار مثال ۲

در این برنامه وقفه را در حالت Toggle قرار داده‌ایم تا با هر تغییر وارد وقفه شود. در اینصورت در لبه‌ی بالارونده یک بار و در لبه‌ی پایین رونده نیز یک بار دیگر وارد وقفه می‌شود پس در لحظه‌ای که کلید را فشار می‌دهیم ولتاژ پایه‌ی AIN0 برابر ۴ولت می‌شود که چون ولتاژ پایه‌ی AIN1 برابر ۳ولت است، خروجی یک می‌شود پس در همین لحظه برنامه یک‌بار به وقفه رفته و LED را روشن می‌کند. زمانی‌که دستمان را از روی کلید برمی‌داریم ولتاژ پایه‌ی AIN0 از منبع قطع شده و ولتاژ AIN1 از AIN0 بیشتر می‌شود و خروجی صفر می‌گردد. در این لحظه چون خروجی از یک به صفر تغییر می‌کند باز برنامه وارد وقفه شده و LED را خاموش می‌کند.

گزینه‌ی Analog Comparator Input Capture

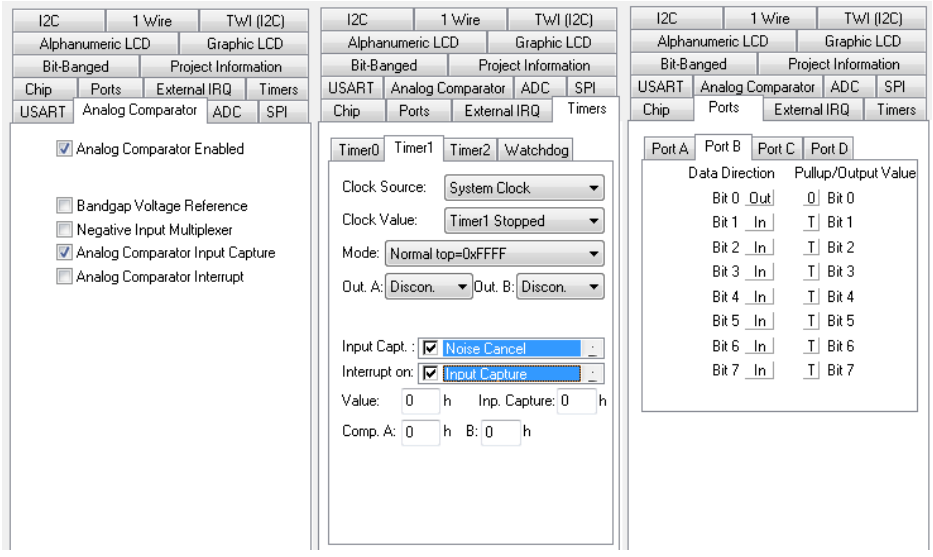
در این حالت خروجی مقایسه‌کننده آنالوگ به Input-Capture مربوط به تایمر یک از داخل متصل می‌شود و واحد Input-Capture با خروجی مقایسه‌کننده آنالوگ همانند یک سیگنال ورودی برخورد می‌کند. برای مثال اگر در تایمر یک وقفه‌ی مربوط به‌Input-Capture را فعال کنیم با هر بار صفر شدن این خروجی برنامه وارد وقفه‌ی Input-Capture می‌شود.

اکنون برنامه‌ی زیر را در نظر بگیرید که مقایسه‌کننده آنالوگ را روی حالت Analog-Comparator Input Capture قرار داده‌ایم و در تایمر یک وقفه‌ی مربوط به Input-Capture را فعال کرده‌ایم.

مثال ۳: حال برای مثال برنامه‌ای می‌نویسیم که هر زمان ورودی AIN0 کمتر از AIN1 شد برنامه وارد وقفه Input-Capture شود و ولتاژ یک LED را معکوس کند(یعنی اگر LED خاموش است آن را روشن و اگر روشن است آن را خاموش کند).

دقت کنید که زمانی برنامه وارد وقفه‌ی Input-Capture می‌شود که ورودی پایه‌ی مربوط آن صفر شود و چون در حالت Analog Comparator Input Capture ورودی این واحد به خروجی واحد مقایسه‌کننده آنالوگ متصل شده است، با صفر شدن خروجی مقایسه‌کننده، ورودی مربوط به واحد Input-Capture نیز صفر می‌شود و برنامه وارد وقفه‌ی Input-Capture می‌شود.

تنظیمات کدویزارد را مطابق شکل زیر انجام می‌دهیم:

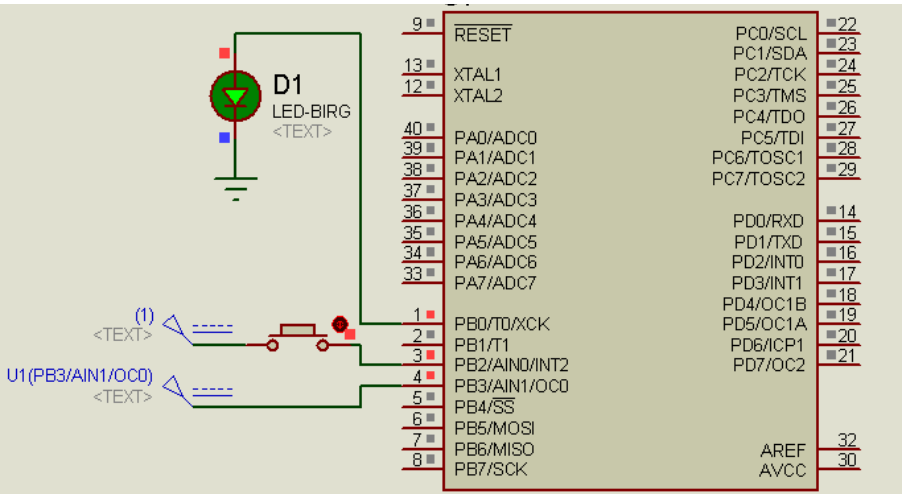


شکل ۸-۹: تنظیمات مثال ۳

سپس کد زیر را درون وقفه‌ی مربوط به Input-Capture می‌نویسیم:

```
// Timer1 input capture interrupt service routine
interrupt [TIM1_CAPT] void timer1_capt_isr(void)
{
  PORTB.0=~PORTB.0;
}
```

و در آخر برای شبیه‌سازی مدار آن‌را در پروتئوس می‌کشیم:

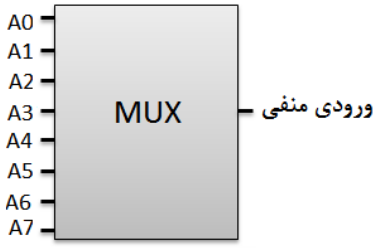


شکل ۸-۱۰: مدار مثال ۳

در ابتدای برنامه که کلید باز است ولتاژ پایه‌ی AIN0 کمتر از پایه‌ی AIN1 بوده و خروجی صفر می‌شود و لذا ورودی Input-Capture صفر است و برنامه وارد وقفه شده و LED روشن می‌شود. با فشار دادن کلید، خروجی یک شده بنابراین ورودی Input-Capture یک می‌شود که هیچ اتفاقی نمی‌افتد ولی به محض برداشتن دستمان از کلید، خروجی دوباره صفر و برنامه وارد وقفه شده و LED خاموش می‌شود. دوباره با فشار دادن و رها کردن کلید LED دوباره روشن می‌شود.

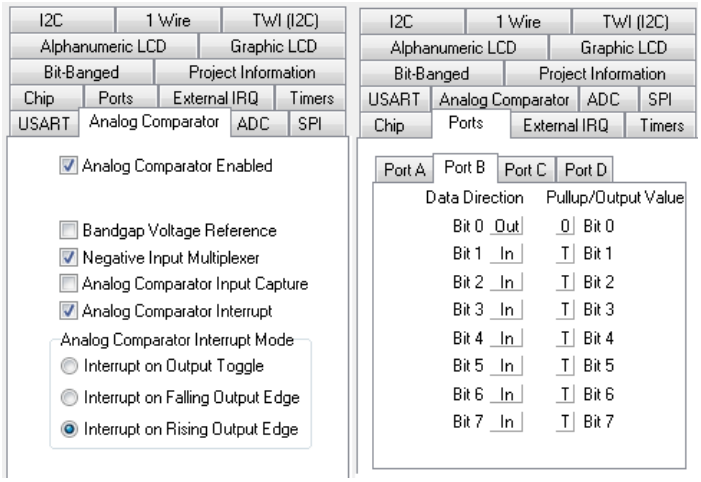
گزینه‌ی Negative Input Multiplexer

یکی از امکاناتی که این واحد دارد این است که می‌توان ورودی یکی از پایه‌های ADC را به عنوان پایه‌ی منفی(به جای AIN1) در نظر گرفت، البته این کار تنها زمانی امکان‌پذیر است که واحد ADC غیرفعال باشد. در این مد یک مالتی‌پلکسر وجود دارد که انتخاب می‌کند کدام پایه‌ی ADC به پایه‌ی منفی مقایسه‌کننده متصل باشد. در شکل روبرو مالتی‌پلکسر(MUX) می‌تواند یکی از پایه‌های ADC را به پایه‌ی منفی مقایسه‌کننده متصل کند.اینکه کدام پایه به ورودی منفی مقایسه‌کننده متصل باشد را ما در



برنامه می‌نویسیم پس این امکان را به ما می‌دهد تا در صورت نیاز هر زمان که خواستیم ورودی منفی مقایسه‌کننده را تغییر دهیم. برای درک بهتر موضوع، این قسمت را با مثالی توضیح می‌دهیم.

مثال ۴: فرض کنید پایه‌ی شماره‌ی یک ADC را به یک منبع ولتاژ ۳ولت و پایه‌ی AIN0 را به یک منبع ۴ولت متصل کرده‌ایم. می‌خواهیم زمانی‌که ولتاژ پایه‌ی AIN0 بیشتر از ولتاژ پایه‌ی A1 شد یک LED روشن شود.تنظیمات کدویزارد را مانند شکل زیر انجام می‌دهیم:



شکل ۸-۱۲: تنظیمات مثال ۴

علاوه بر گزینه‌ی Negative Input Multiplexer، باید گزینه‌ی Analog Comparator Interrupt را نیز انتخاب کنیم که هر زمان خروجی یک شد برنامه به وقفه برود و LED را روشن کند(همانطور که گفته شد واحد ADC باید غیرفعال باشد). برای آنکه پایه‌ی A1 به ورودی منفی مقایسه‌کننده متصل شود باید دستور ADMUX=1; را بنویسیم. به کمک این دستور مالتی‌پلکسر متوجه می‌شود که باید کدام ورودی ADC را به پایه‌ی منفی متصل کند(می‌توانستیم این دستور را به صورت هگزادسیمال نیز بنویسیم ;ADMUX=0x01). نکته‌ای که باید به آن توجه کنیم این است که چون نام رجیستر مربوط به مالتی‌پلکسر، ADMUX است باید این اسم به همین صورت و با حروف بزرگ نوشته شود(برای انتخاب پایه‌های دیگر نیز می‌توانیم دستور را به همین صورت بنویسیم برای مثال برای انتخاب ADC شماره‌ی ۵ به عنوان پایه‌ی منفی مقایسه‌کننده باید دستور ADMUX=5 یا ADMUX=0x05 را بنویسیم).

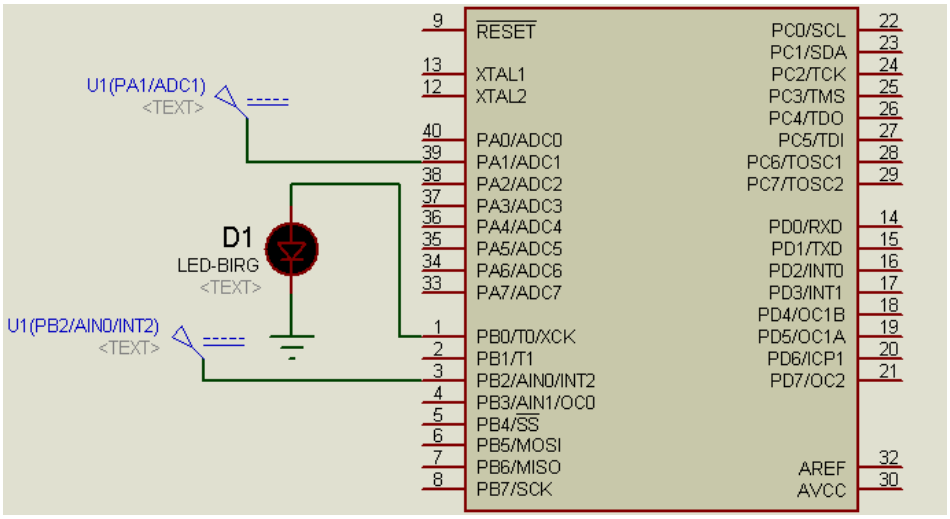
درون وقفه کد ساده‌ی روبرو را می‌نویسیم:

```
interrupt [ANA_COMP] void ana_comp_isr(void)
{
 PORTB.0=1;
}
```

و درون While(1) نیز دستور روبرو را می‌نویسیم:

```
while (1)
      {
       ADMUX=1;
      }
```

سپس مدار زیر را در پروتئوس می‌بندیم و مقدار منبع DC متصل به پایه‌ی A.1 را برابر عدد دلخواهی مانند ۳ولت قرار می‌دهیم و مقدار منبع DC متصل به پایه‌ی AIN0 را برابر ۴ولت قرار می‌دهیم، سپس برنامه را اجرا می‌کنیم. در این حالت چون ولتاژ پایه‌ی AIN0 از پایه‌ی منفی(A.1) بیشتر است LED روشن می‌شود و اگر منبع AIN0 را برابر ۲ولت(یعنی کمتر از پایه‌ی منفی) انتخاب کنیم، برنامه به وقفه نمی‌رود و LED روشن نمی‌شود.



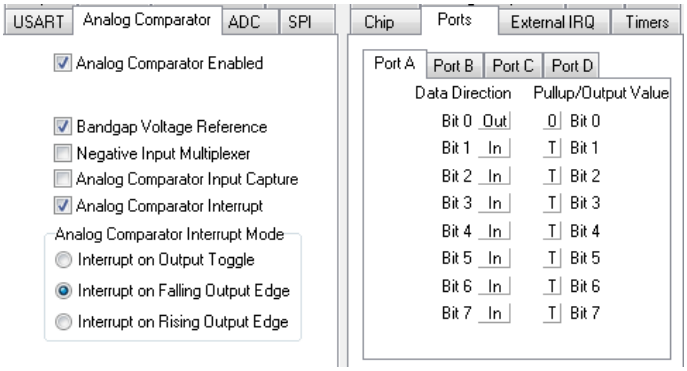
شکل ۸-۱۳: مدار مثال ۴

گزینه‌ی Bandgap Voltage Reference

اگر این گزینه را انتخاب کنیم یک ولتاژ مرجع داخلی(1.22v) به پایه‌ی مثبت مقایسه‌کننده متصل می‌شود و می‌توانیم به پایه منفی نیز توسط پایه‌ی AIN1 ولتاژ بدهیم تا عملیات مقایسه بین این پایه و ولتاژ مرجع(۱,۲۲ ولت) صورت گیرد.

مثال ۵: برای مثال برنامه‌ای می‌نویسیم که هنگامی ولتاژ پایه‌ی AIN1 برابر ۲ولت شد یک LED روشن شود. برای این کار ابتدا گزینه‌ی Bandgap Voltage Reference را انتخاب

می‌کنیم که پایه‌ی مثبت به ولتاژ داخلی وصل شود و نیز وقفه‌ی مقایسه‌کننـده را در حالـت لبـه‌ی پایین‌رونده قرار می‌دهیم و این بار LED را به پایه‌ی A0 متصل می‌کنیم:

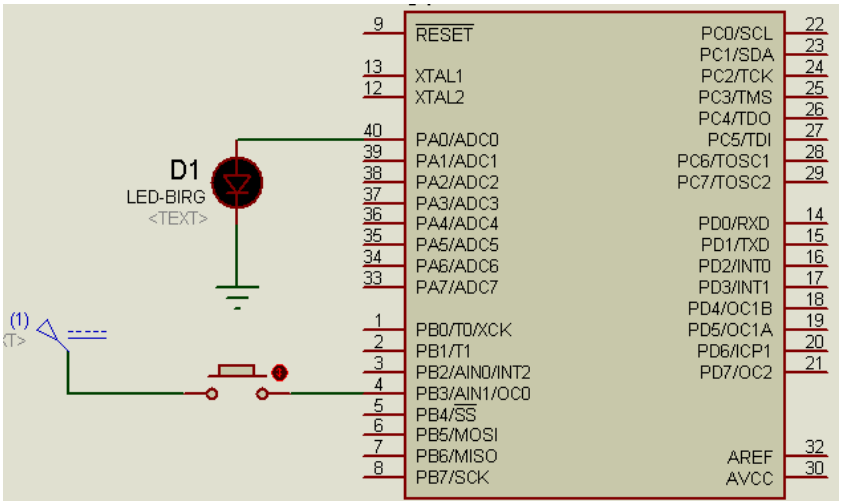


شکل ۸-۱۴: تنظیمات مثال ۵

کد روبرو را درون وقفه می‌نویسیم:

```
interrupt [ANA_COMP] void ana_comp_isr(void)
{
PORTA.0=1;
}
```

سـپس مـدار زیـر را در پروتئـوس می‌بندیم و مقدار منبع ولتاژ متصل به پایه‌ی منفی را برابر ۲ولت قرار می‌دهیم:

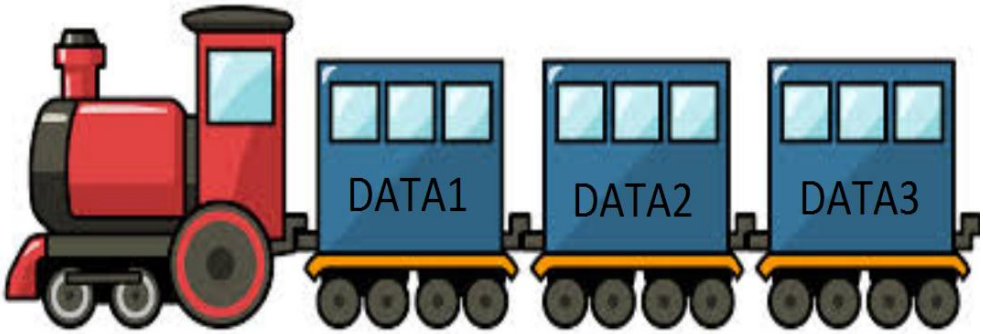


شکل ۸-۱۵: مدار مثال ۵

زمانی که کلید باز است ولتاژ پایه‌ی مثبت ۱,۲۲ ولت است که بیشـتر از پایـه‌ی منفـی بـوده پـس خروجی یک می‌شود، زمانی‌که کلید را وصل کنیم ولتاژ پایه‌ی منفی از پایه‌ی مثبت بیشتر شـده و خروجی صفر می‌شود، در این لحظه خروجی یک لبه‌ی پایین رونده را طی کرده و برنامه وارد وقفـه شده و LED روشن می‌شود.اکنون با گزینه‌ها و حالت‌های مختلف مقایسه‌کننده‌آنالوگ آشنا شدیم و می‌توانیم از این گزینه ها به صورت ترکیبی و بنا به نیاز استفاده کنیم.

فصل نهم

ارتباط سریال

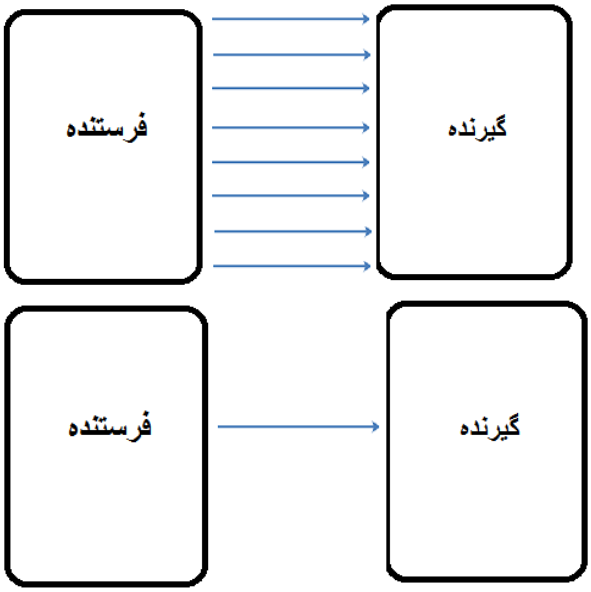


در این فصل خواهیم خواند:

۱. معرفی ارتباط سریال
۲. ارتباط همگام یا سنکرون
۳. ارتباط ناهمگام یا آسنکرون
۴. نحوه‌ی ارسال اطلاعات در ارتباط ناهمگام
۵. کنترل جریان داده

ارتباط سریال

ارسال اطلاعات بین دو وسیله به دو صورت انجام می‌شود: به صورت سریال و یا به صورت موازی. در ارتباط موازی اطلاعات به صورت موازی ارسال می‌شود و تمام بیت‌ها به طور همزمان به گیرنده ارسال می‌شوند و در ارتباط سریال اطلاعات به صورت بیت‌بیت و پشت سر هم ارسال می‌شوند:



شکل ۹-۱: ارتباط سریال و موازی

ممکن است ابتدا این‌طور به نظر برسد که "سرعت انتقال اطلاعات در ارتباط موازی بیشتراست، چون به صورت همزمان همه‌ی بیت‌ها فرستاده می‌شود درحالیکه در ارتباط سریال این کار بیت‌بیت انجام می‌شود. اگر در ارتباط موازی با زده شدن یک کلاک ۸ بیت ارسال شود، همین هشت بیت در ارتباط سریال به ۸ کلاک نیاز دارد"، ولی این گفته به این سادگی‌ها درست نمی‌باشد زیرا گاهی سرعت ارسال اطلاعات در ارتباط سریال به حدی بالا می‌رود که همان ۸ کلاک نیز بسیار سریع‌تر از یک کلاک مربوط به ارتباط موازی زده می‌شود.

امروزه اکثر ارتباط‌ها به صورت سریال انجام می‌شود به این دلیل که در ارتباط سریال از تعداد سیم کمتری استفاده می‌شود در صورتی که برای ارتباط موازی تعداد سیم‌های بیشتری مورد نیاز است. فرض‌کنید خطوط مخابراتی به جای استفاده از ارتباط سریال از ارتباط موازی استفاده می‌کرد، در آن صورت تمام شهر سیم‌کشی می‌شد و دیگر مسی در بازار پیدا نمی‌شد چون همه‌ی آن مصرف سیم‌های ارتباطی می‌شد.

در ارتباط سریال یک سیم وظیفه‌ی انتقال داده را به عهده دارد و سیم دیگر نیز برای آنکه زمین(GND) فرستنده و گیرنده را به هم متصل‌کند وجود دارد. می‌دانید که هر ولتاژی نسبت به یک مبدا خوانده می‌شود، زمین مدار همان مبدا است که داده‌ها نسبت به آن حساب می‌شوند. مثلاً وقتی می‌گوییم این سیگنال پنج ولت است یعنی نسبت به مبدا پنج ولت بالاتر است بنابراین باید مبدا در فرستنده و گیرنده یکسان باشد تا داده‌ای که در طرف فرستنده به‌عنوان یک شناخته می‌شود در طرف گیرنده هم به‌عنوان یک شناخته شود.

موضوع دیگری که در ارتباط سریال بسیار اهمیت دارد سرعت دریافت و ارسال داده‌ها است.

فرض‌کنید شما در یک زمین تنیس هستید و دوست شما هم در آن سمت قرار دارد به‌طوری که دیواری بین شما قرار دارد و همدیگر را نمی‌بینید و شما قصد دارید که تمام توپ‌ها را از سمت خود به سبدی که در کنار دوستتان است انتقال بدهید، پس یکی‌یکی توپ‌ها را برای دوستتان می‌فرستید و دوستتان توپ‌ها را داخل سبد می‌گذارد.

مساله‌ای که بسیار اهمیت دارد این است که سرعت شما در پرتاب توپ و سرعت دوستتان در دریافت توپ برابر باشد. اگر سرعت شما بیشتر باشد دوستتان فرصت جمع‌آوری توپ‌ها را پیدا نمی‌کند و تعدادی از توپ‌ها از دست می‌روند و اگر سرعت دوستتان بیشتر از شما باشد باز هم دچار مشکل می‌شوید چون دوست شما انتظار دارد توپ‌ها را با سرعت خودش دریافت‌کند که در این حالت چون هنوز توپی نیامده و او از زمان ارسال توپ بعدی خبر ندارد نمی‌تواند توپ‌ها را دریافت کند.

در ارسال و دریافت اطلاعات هم همین وضعیت است و فرستنده و گیرنده باید با هم هماهنگ باشند.

سرعت ارتباط سریال با معیار بیت‌برثانیه سنجیده می‌شود(bps) و نرخ‌بیتی نامیده می‌شود. به‌عنوان مثال پورت سریال کامپیوتر می‌تواند سرعت‌هایی نظیر ۱۱۰، ۳۰۰، ۲۴۰۰، ۹۶۰۰بیت-برثانیه و... را پشتیبانی کند.

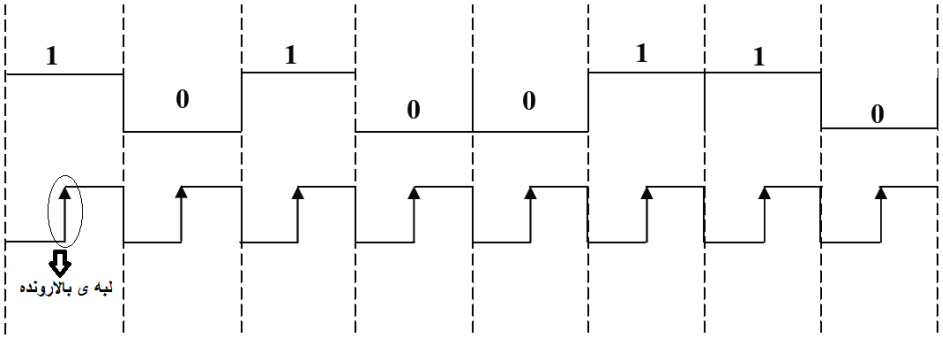
گفتیم فرستنده و گیرنده باید با هم هماهنگ باشند حال این هماهنگی می‌تواند به دو صورت باشد:

۱- همگام(سنکرون)۲- غیرهمگام(آسنکرون)

ارتباط همگام(سنکرون)

فرض‌کنید زمانی‌که توپ‌ها را به سمت دوستتان پرتاب می‌کنید، هر توپی را که پرتاب کردید بلند بگویید "بگیر" در آن صورت دوستتان متوجه می‌شود که الان باید توپ را دریافت‌کند.

در ارتباط همگام فرستنده همراه با داده یک کلاک نیز می‌فرستد. گیرنده با دریافت این کلاک اطلاعات را دریافت می‌کند، یا اگر بخواهیم بهتر بگوییم زمانی‌که فرستنده یک لبه‌ی بالارونده می‌فرستد گیرنده اطلاعات را برمی‌دارد. به شکل زیر دقت کنید:

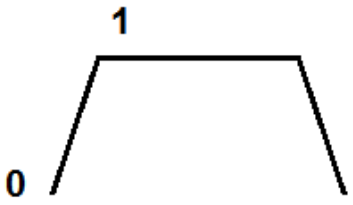


شکل ۹-۲: دریافت اطلاعات در ارتباط سنکرون

سطر اول مربوط به اطلاعاتی است که فرستنده می‌فرستد و سطر دوم مربوط به کلاک است که میکرو با فرستادن هر داده ارسال کرده و گیرنده با دریافت لبه‌ی بالارونده اطلاعات را برمی‌دارد.

سوالی که پیش می‌آید این است که چرا این لبه‌ی بالارونده وسط هر داده‌ی ارسالی فرستاده می‌شود و چرا اول هر داده لبه‌ی بالارونده فرستاده نمی‌شود؟

به این دلیل که ما قادر به تولید یک موج مربعی با زاویه‌ی قائم نیستیم، فقط می‌توانیم یک موج شبیه به زاویه‌ی قائم پدید بیاوریم. دقیق‌ترین جایی که مطمئن هستیم موج به حالت درست خود رسیده است وسط آن می‌باشد.



شکل ۹-۳: شکل واقعی یک موج مربعی

اگر به شکل بالا نگاه کنید می‌بینید که اول و آخر این شکل موج دقیقاً یک نیست و عددی بین صفر و یک است و جایی که دقیقاً یک است وسط این شکل موج است. به همین خاطر فرستنده لبه‌ی بالارونده را وسط شکل موج می‌فرستد که گیرنده درآنجا اطلاعات را دریافت‌کند.

از این نوع ارتباط قبلاً بیشتر استفاده می‌شد و هنوز هم ارتباط بعضی از صفحه کلیدها با کامپیوتر به کمک ارتباط سریال به صورت همگام است که توسط یک رابط ۲۵پین این ارتباط برقرار می‌شود.

ارتباط ناهمگام(آسنکرون)

یکی از مزیت‌های این ارتباط به ارتباط همگام این است که نحوه‌ی هماهنگ‌سازی با ارسال کلاک انجام نمی‌شود و در تعداد سیم‌ها صرفه‌جویی می‌شود. این نوع ارتباط کاربرد بسیار وسیعی دارد مثلاً پورت COM کامپیوتر(که ۹ پین دارد)، کنترل از راه دور تلویزیون و...

اما این نوع هماهنگ‌سازی بدون کلاک چطور اتفاق می‌افتد؟

به کمک یکسان‌سازی سرعت نرخ ارسال. یعنی سرعت ارسال داده‌ها توسط فرستنده و دریافت آنها به وسیله‌ی گیرنده باید از قبل بین فرستنده و گیرنده توافق شده باشد. این دریافت و ارسال اطلاعات به صورت ناهمگام باید در یک قاب مشخص و از پیش تعیین شده اتفاق بیافتد که گیرنده دچار خطا نشود.

نحوه‌ی ارسال اطلاعات در ارتباط ناهمگام

۱- زمانی‌که خط بیکار می‌باشد، یعنی هیچ اطلاعاتی ارسال نمی‌شود خط برابر یک است(یعنی سطح ولتاژ خط برابر سطح ولتاژ یک است مثلاً برابر ۵ولت است).
۲- برای شروع ارسال اطلاعات یک بیت صفر به‌عنوان بیت شروع برای آنکه به فرستنده بفهماند که اطلاعات قرار است ارسال بشود فرستاده می‌شود.
۳- بعد از آن ۸بیت اطلاعات فرستاده می‌شود(این اطلاعات می‌تواند به جای ۸ بیت در یک قاب ۵ یا ۶ یا ۷بیتی هم ارسال شود)
۴- بعد از آن ۸بیت داده یک بیت به‌عنوان بیت توازن فرستاده می‌شود(نقش این بیت در پیدا کردن خطاست که توازن می‌تواند فرد یا زوج یا آنکه قاب بدون بیت توازن باشد).

همانطور که گفتیم بیت توازن برای پیدا کردن خطا استفاده می‌شود، برای مثال اگر حالت توازن فرد را انتخاب کنیم(odd - parity) به این معناست که جمع داده‌هایمان باید عددی فرد باشد مثلاً اگر در بیت‌های ارسالی دو عدد ۱ داشته باشیم برای آنکه جمع فرد شود، بیت توازن ۱ می‌شود ولی اگر سه بیت ۱ فرستادیم بیت توازن صفر می‌شود تا مجموع همچنان فرد باقی بماند یا برای مثال در حالت Even parity وقتی تعداد یک‌های بسته‌ی اطلاعاتی فرد است بیت توازن ۱ می‌شود که تعداد یک‌های بسته‌ی اطلاعاتی را زوج کند و زمانی‌که تعداد یک‌های بسته‌ی اطلاعاتی زوج است بیت توازن صفر می‌شود که تعداد بیت‌های یک زوج باقی بماند.

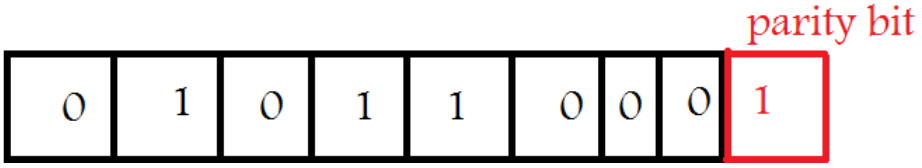
برای مثال جدول زیر چند حالت بیت توازن را بررسی کرده است:

7 bits of data	(count of 1 bits)	8 bits including parity	
		even	odd
0000000	0	00000000	00000001
1010001	3	10100011	10100010
1101001	4	11010010	11010011
1111111	7	11111111	11111110

جدول ۹-۱: بیت‌های توازن

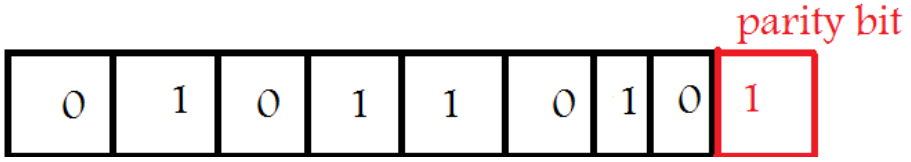
این روش برای تشخیص خطاست به این معنا که اگر حالت توازن فرد انتخاب شود دریافت‌کننده انتظار دریافت بسته‌ی فرد را دارد که اگر طی یک خطا یا نویز در شبکه یکی از صفرها یک شد یا یکی از یک‌ها صفر شد بسته‌ی دریافتی زوج می‌شود و دریافت‌کننده می‌فهمد که این بسته نامعتبر است.

برای مثال فرض‌کنید داده‌های زیر را در قالب Even parity ارسال کرده‌ایم:



شکل ۹-۴: داده‌ی ارسالی در حالت توازن زوج

در اثر اغتشاشات و عوامل محیطی(نویز و تداخل امواج و...) بیت ۲ از حالت صفر به یک تبدیل شده است:



شکل ۹-۵: داده‌ی دریافتی در اثر نویز

در این حالت گیرنده متوجه می‌شود که تعداد بیت‌های ۱ فرد شده است(۵ عدد یک) و این در قالب Even parity نامعتبر می‌باشد پس آن را به صورت داده‌ی خطا تشخیص داده و پردازشی روی آن انجام نمی‌دهد.

۵- بعد از بیت توازن دو بیت یک به‌عنوان پایان ارسال اطلاعات یا برگرداندن خط به حالت بیکار(یک) فرستاده می‌شود.

توقف	توقف										بیت شروع	خط بیکار
یک	یک	توازن	D7	D6	D5	D4	D3	D2	D1	D0	صفر	یک

جدول ۹-۲: دو بیت یک به‌عنوان پایان ارسال اطلاعات

سوالی که ممکن است پیش‌آید این است که چرا ما در هر بسته اطلاعاتی فقط ۸بیت داده می‌فرستیم و بیشتر نمی‌فرستیم؟

- فرض‌کنید که فرستنده بیت شروع را فرستاد و با ارسال بیت شروع گیرنده و فرستنده آماده‌ی تبادل اطلاعات شدند و فرستنده تک‌تک بیت‌ها را با نرخ مثلاً ۹۶۰۰ بیت‌برثانیه فرستاد و گیرنده با همین نرخ و با یک فاصله‌ی زمانی کوتاه نسبت به فرستنده(به دلیل اینکه وسط شکل موج ارسالی را دریافت‌کند) دریافت را شروع کرد. آیا تا پایان زمانی‌که فرستنده داده‌ای را بفرستد، گیرنده وسط آن شکل موج ارسالی را دریافت می‌کند؟

پاسخ این است که اگر سرعت‌های فرستنده و گیرنده دقیقاً یکسان باشد بله. ولی سرعت‌ها دقیقاً یکسان نیست چون در طبیعت هیچ دو عنصر یکسانی وجود ندارد و هیچ دو کریستالی پیدا نمی‌شود که دقیقاً سرعت برابری داشته باشند پس ۹۶۰۰ بیت‌برثانیه‌ی فرستنده کمی با ۹۶۰۰ بیت‌برثانیه‌ی گیرنده تفاوت دارد پس بعد از گذشت مدت زمانی، دیگر گیرنده وسط داده‌ی ارسالی را دریافت نمی‌کند و دچار خطا می‌شود، حتی ممکن است فاصله‌ی زمانی فرستنده و گیرنده با گذشت زمان به بیش از چند بیت برسد به همین خاطر تعداد بیت‌های ارسالی در هر بسته باید محدود باشد(حداکثر ۸ بیت) که گیرنده دچار اشتباه نشود و ارسال و دریافت اطلاعات به درستی انجام شود.

کنترل جریان داده

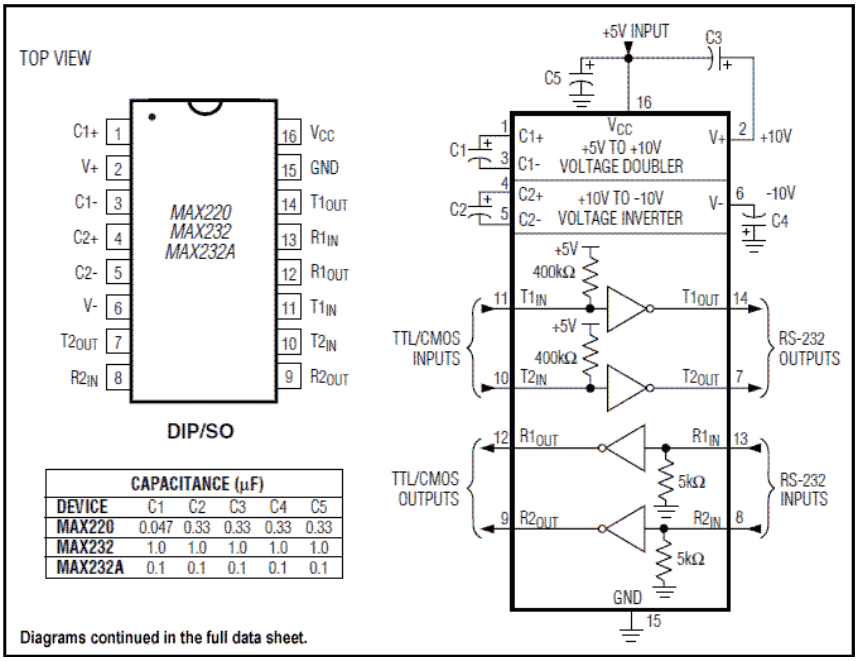
یکی از مهمترین پارامترها که باید در ارتباط سریال رعایت شود کنترل جریان داده است. فرض‌کنید شما توپ‌ها را برای دوستتان می‌فرستید، دوستتان هم همه‌ی توپ‌ها را دریافت کرده و با سرعت سبد را پر می‌کند. سرعت شما و دوستتان خیلی زیاد و برابر است ولی اگر سبد پر بشود

تا سبد بعدی برسد ممکن است شما دهها توپ را پرتاب کرده باشید ولی سبدی نباشد که توپ‌ها را درونش بریزید.
همین اتفاق می‌تواند در ارتباط سریال هم مشکل ایجاد کند و بافر موقت مربوط به ذخیره‌ی اطلاعات پر بشود. برای رفع این مشکل یک روش سخت‌افزاری و یک روش نرم‌افزاری وجود دارد.
روش نرم‌افزاری به این شکل است که زمانی‌که بافر گیرنده در شرف پر شدن باشد و نتواند اطلاعات بیشتری را ذخیره کند یک کاراکتر ویژه به نام Xoff برای فرستنده ارسال می‌کند و فرستنده با دریافت این کاراکتر ارسال اطلاعات را متوقف می‌کند تا زمانی‌که گیرنده دوباره آمادگی خود را اعلام کند، اعلام آمادگی گیرنده با ارسال یک کاراکتر ویژه به نام Xon است که به فرستنده ارسال می‌کند.
اگر به پشت کامپیوتر خود نگاه کنید یک پورت ۹ پایه می‌بینید که به پورت COM یا DB9 معروف است:

شکل ۹-۶: درگاه com در کامپیوتر

سازمان صنایع الکترونیک(EIA) در سال ۱۹۶۰ برای استانداردسازی ارتباط سریال بین وسایل مختلف پروتکلی پیشنهاد داد به نام پروتکل EIA232 یا RS232.
تا اینجای کار با استاندارد TTL کار می‌کردیم و ولتاژ منطقی به این شکل بود که اگر ولتاژ بین ۰ تا ۰٫۸ولت بود به عنوان صفر منطقی و اگر بین ۳٫۵ تا ۵ولت بود به عنوان یک منطقی حساب می‌شد، ولی ولتاژهای منطقی درRS232 با TTL فرق می‌کند، چون در RS232 ولتاژ بین 3- تا 25- ولت به‌عنوان یک منطقی و ولتاژ بین 3+ تا 25+ به عنوان صفر منطقی حساب می‌شود.
برای ارتباط میکروکنترلر(که سطح ولتاژ TTL دارد) با RS232 باید از یک تراشه به نام Linear-Drive استفاده کنیم که ولتاژ RS232 را به TTL و ولتاژTTLرا به RS232 تبدیل کند، Max232وMax233 دو نمونه از این تراشه‌ها هستند.

اگر به دیتاشیت Max232 نگاهی بیاندازیم:

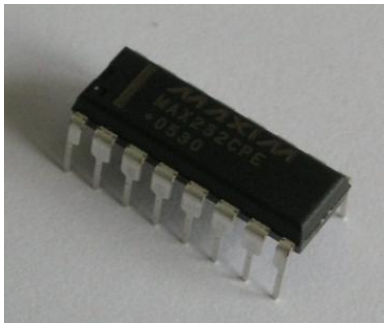


CAPACITANCE (µF)					
DEVICE	C1	C2	C3	C4	C5
MAX220	0.047	0.33	0.33	0.33	0.33
MAX232	1.0	1.0	1.0	1.0	1.0
MAX232A	0.1	0.1	0.1	0.1	0.1

شکل ۹-۷: دیتاشیت max232

می‌بینیم که ۱۶پایه دارد و به توصیه‌ی دیتاشیت باید تعدادی خازن بین پایه‌ها قرار بدهیم. اما چیزی که مهمتر است این می‌باشد که این آی‌سی دو تبدیل از TTL به RS232 دارد و دو تبدیل از RS232 به TTL که همانطور که در شکل مشخص است پایه‌های ۱۰ و ۱۱ ورودی TTL هستند و پایه‌های ۷ و ۱۴ خروجی همان اطلاعات با سطح منطقی RS232 و پایه‌های ۸ و ۱۳ ورودی RS232 و پایه‌های ۹ و ۱۲ خروجی همان اطلاعات با سطح منطقی TTL هستند.

شکل زیر تصویری از این آی‌سی را نشان می دهد:



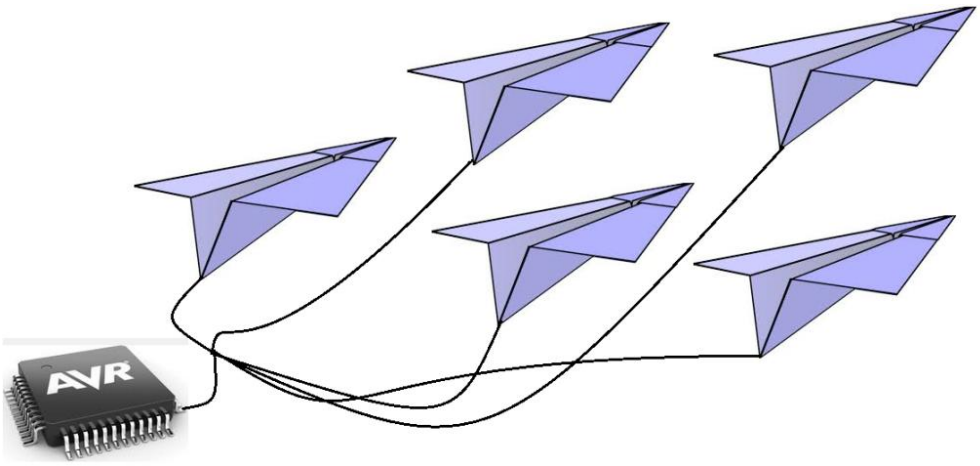
شکل ۹-۸: آی‌سی max232

ارتباط سریال به چندین سیستم مختلف برای برقراری ارتباط تقسیم می‌شود از قبیل USART، I2C، SPI، USB که هرکدام از این سیستم‌ها دارای مزیت‌هایی نسبت به همدیگر هستند که

بسته به فاصله، تعداد وسیله‌ها برای ارتباط، نوع وسیله و... هر کدام از این سیستم‌ها انتخاب می‌شوند.

فصل دهم

ارتباط USART



در این فصل خواهیم خواند:

۱. معرفی ارتباط USART
۲. تنظیمات اولیه‌ی USART در کدویزارد
۳. حالت Transmitter در ارتباط آسنکرون
۴. دریافت اطلاعات
۵. ارتباط USART در حالت سنکرون
۶. وقفه‌ی ارتباط USART
۷. یک گام فراتر

ارتباط USART

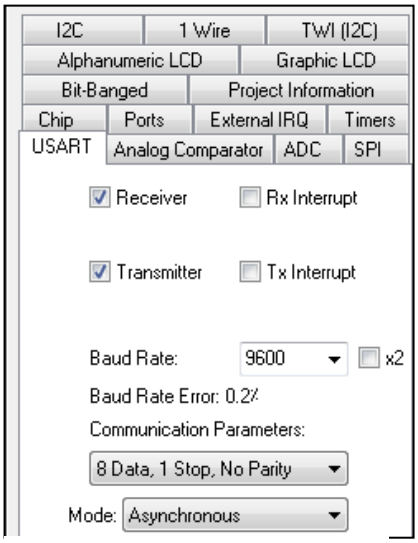
یکی از لازمه‌های هر میکروکنترلر این است که بتواند با دنیای خارج از خود ارتباط برقرار کند، یعنی اینکه بتواند با میکروکنترلرها، کامپیوتر و آی‌سی‌های مختلف در ارتباط باشد.
تاکنون با ارتباط سریال آشنا شدیم و دیدیم که این ارتباط به دو نوع هم‌زمان و غیرهم‌زمان(سنکرون و آسنکرون) تقسیم می‌شود.
ارتباط USART یک نوع ارتباط سریال است که توانایی برقراری ارتباط سنکرون و آسنکرون را دارا می‌باشد. واژه‌ی USART از اول کلمه‌های Universal Synchronous Asynchronous Receiver Transmitterگرفته شده که به معنای داشتن ارتباط سنکرون و آسنکرون و توانایی دریافت و ارسال اطلاعات است.
نوعی از این ارتباط که فقط قادر به ارتباط غیرهمزمان یا همان آسنکرون است به نام UART معروف است که مخففUniversal Asynchronous Receiver Transmitter می‌باشد.
بعضی از میکروکنترلرها فقط از ارتباط UART پشتیبانی می‌کنند و قادر به برقراری ارتباط سنکرون نیستند، ولی خانواده‌ی AVRهای ATmega دارای سخت‌افزار لازم جهت برقراری ارتباط سنکرون و آسنکرون می‌باشند.
همانطور که قبلاً گفته شد ارتباط سنکرون همراه با سیمی که داده را منتقل می‌کند یک سیم هم به عنوان کلاک از فرستنده به گیرنده دارد که مشخص می‌کند که گیرنده چه زمانی اطلاعات را دریافت کند ولی ارتباط آسنکرون سیمی به عنوان کلاک نداشته و برای آنکه گیرنده دچار خطا نشود سرعت و نرخ ارسال اطلاعات بین فرستنده و گیرنده به توافق می‌رسد و بسته‌ای که به گیرنده فرستاده می‌شود در یک قالب مشخص است که با یک بیت ‘صفر’ شروع می‌شود و بعد از آن بیت صفر، ۸ بیت داده ارسال می‌شود(که در قاب‌های مختلف بین ۵ تا ۸ بیت می‌تواند متفاوت باشد) و بعد از آن یک بیت توازن برای تشخیص خطا ارسال می‌شود (که می‌تواند از نوع توازن فرد یا زوج یا حتی قاب بدون بیت توازن باشد) بعد از بیت توازن یک یا دو بیت ۱ به عنوان بیت توقف ارسال می‌شود.
مزیت ارتباط آسنکرون به سنکرون این است که یک سیم اضافه به عنوان کلاک ندارد.

USART در حقیقت یک پروتکل ارتباطی است که می‌تواند با کامپیوترها، میکروکنترلرها و انواع سنسورها ارتباط برقرار کند. برای استفاده از این نوع ارتباط در AVR، وارد کدویزارد شده و یک پروژه‌ی جدید ایجاد می‌کنیم. برای این ارتباط تبی با نام USART در کدویزارد وجود دارد.

تنظیمات اولیه‌ی USART در کدویزارد

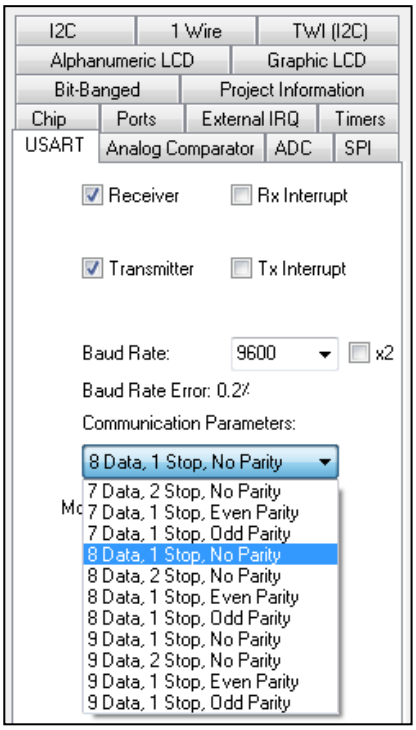
در قدم اول روی تب USART کلیک می‌کنیم:

دو گزینه‌ی Receiver و Transmitter همانطور که از نامشان پیداست به ترتیب مربوط به دریافت و ارسال اطلاعات هستند.



شکل ۱۰-۱: ارتباط USART

گزینه‌ی Baud Rate مربوط به نرخ ارسال اطلاعات بوده که بر حسب بیت‌برثانیه می‌باشد. ما در مثال‌های این کتاب Baud Rate را بر روی 9600 تنظیم می‌کنیم. در کنار این اعداد یک گزینه به نام 2* نیز وجود دارد که اگر تیک آن را بزنیم سرعت ارسال اطلاعات را دو برابر می‌کند.قسمت communication parameters مربوط به قاب ارسال اطلاعات است، همانطور که قبلاً اشاره شد این قاب می‌تواند همراه بیت توازن در حالت‌های توازن زوج یا فرد یا حتی بدون بیت توازن باشد، همچنین نحوه‌ی ارسال(یا همان قاب ارسال اطلاعات) می‌تواند حالت‌های مختلفی داشته باشد:



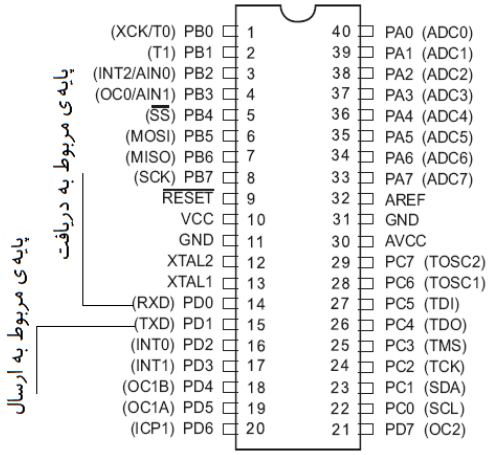
ما فعلاً با حالت 8 Data، 1 stop، No Parity کار می‌کنیم یعنی بسته‌ها حاوی ۸بیت داده هستند و بیت توازن ندارند و آخر هر بسته یک بیت ۱ به عنوان بیت توقف(یا به عبارتی بیتی که بین بسته های ارسالی فاصله می‌اندازد) ارسال می‌شود. به این دلیل ۸بیت داده انتخاب شد که کاراکترهای موجود در کامپیوتر همگی ۸بیتی هستند(در این مورد در ادامه بیشتر توضیح می‌دهیم).

گزینه‌ی Mode هم نوع ارتباط را مشخص می‌کند که این ارتباط می‌تواند سنکرون(یعنی با یک سیم اضافه به عنوان کلاک) یا آسنکرون باشد. ما در ابتدای این فصل با مد آسنکرون کار می‌کنیم و در ادامه به بررسی مد سنکرون می‌پردازیم.

حالت Transmitter در ارتباط آسنکرون

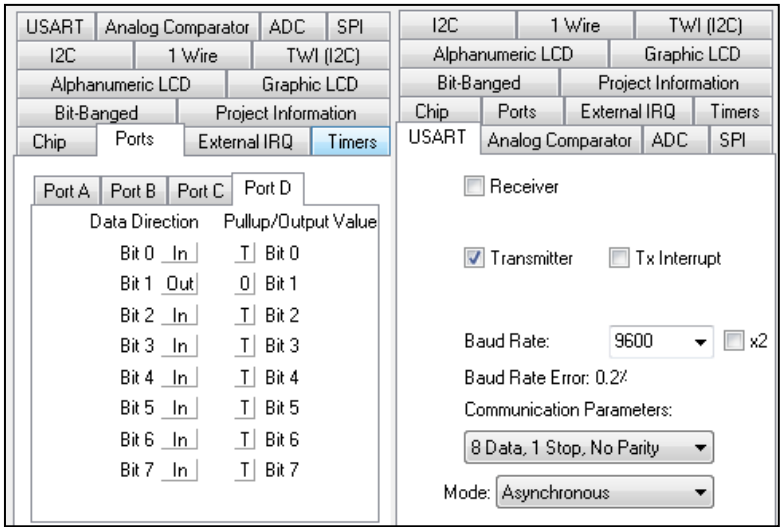
مثال۱: به‌عنوان اولین مثال قصد داریم یک برنامه‌ی ساده بنویسیم که یک کلمه را روی خروجی مربوط به ارتباط USART میکروکنترلر ارسال کند پس در تب مربوط به USART تیک گزینه‌ی Transmitter را می‌زنیم.

پایه‌ی PD1 در مد USART وظیفه‌ی ارسال اطلاعات را دارد که نام دیگر این پایه TX می‌باشد و پایه‌ی PD وظیفه‌ی دریافت اطلاعات را دارد که نام دیگر آن RX می‌باشد:



شکل ۱۰-۳: پایه‌های مربوط به ارتباط USART

در تنظیمات کدویزارد PD1 را به صورت خروجی تنظیم می‌کنیم:



شکل ۱۰-۴: تنظیمات مثال ۱

کدنویسی در کدویژن:

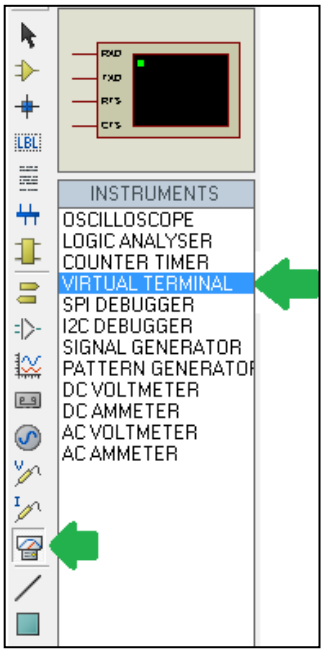
ابتدا کتابخانه‌ی delay.h را بالای برنامه اضافه می‌کنیم تا بتوانیم در طول برنامه از دستور delay_ms() استفاده کنیم.

برای فرستادن اطلاعات می‌توانیم از چندین تابع استفاده کنیم، اولین و ساده‌ترین تابع putchar() می‌باشد که کافیست به وسیله‌ی تک‌کوتیشن (') حرف مورد نظر را درون آن بنویسیم.

برنامه‌ی زیر همانطور که مشاهده می‌کنید ۶ حرف را پشت سر هم ارسال می‌کند(که آخرین حرف فاصله است) و بخاطر آنکه این حروف آرام‌آرام ارسال شوند که بتوانیم در پروتئوس ارسال داده‌ها را مشاهده کنیم، یک تاخیر ۵۰۰ میلی‌ثانیه‌ای آخر برنامه اضافه می‌کنیم:

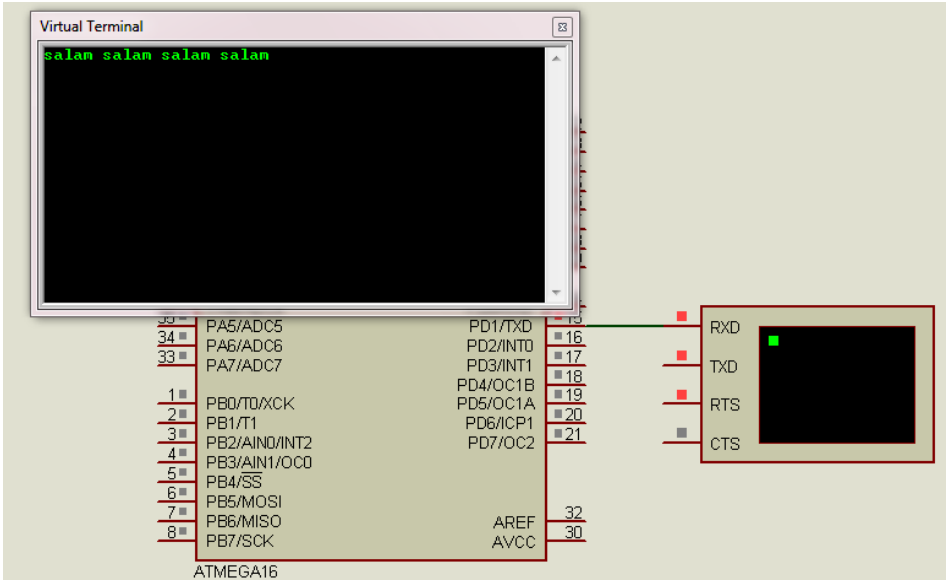
```
while (1)
      {
        putchar('s');
        putchar('a');
        putchar('l');
        putchar('a');
        putchar('m');
        putchar(' ');
        delay_ms(500);
      }
```

حال برای شبیه‌سازی برنامه‌ی پروتئوس را باز کرده و نحوه‌ی عملکرد برنامه را مشاهده‌کنیم. پروتئوس دارای یک شبیه‌ساز است که می‌تواند اطلاعاتی که میکروکنترلر ارسال می‌کند را آشکار سازد. برای انتخاب این شبیه‌ساز باید مانند شکل روبرو VIRTUL TERMINAL را انتخاب کنیم.



پایه‌ی PD1 یا همان TX را به پایه‌ی RX ترمینال مجازی متصل می‌کنیم زیرا میکروکنترلر اطلاعات را از طریق پایه‌ی TX می‌فرستد وVIRTUL-TERMINALاین اطلاعات را از طریق پایه‌ی RX دریافت می‌کند.

حال برنامه را اجرا می‌کنیم:



شکل ۱۰-۶: مشاهده‌ی اطلاعات ارسالی از میکرو توسط ترمینال مجازی

برای درک بهتر موضوع به یک مثال دیگر توجه کنید:

مثال ۲: مانند مثال قبل یک پروژه‌ی جدید باز می‌کنیم و همان تنظیمات را برای USART انجام می‌دهیم و پایه‌ی PD1 را نیز به صورت خروجی تنظیم می‌کنیم.

این بار یک متغیر صحیح (یا integer) به نام a را بالای تابع اصلی و به صورت عمومی(public) تعریف می‌کنیم:

```
24  #include <mega16.h>
25  #include <stdio.h>
26  #include <delay.h>
27
28  int a;
29
30  // Declare your global variables here
31
32  void main(void)
33  {
34  // Declare your local variables here
35
```

این بار از دستور printf() برای ارسال داده‌ها استفاده می‌کنیم که در فصل آشنایی با زبان C با آن آشنا شدیم، این دستور می‌تواند کلمات و رشته‌های مورد نظر را ارسال کند.

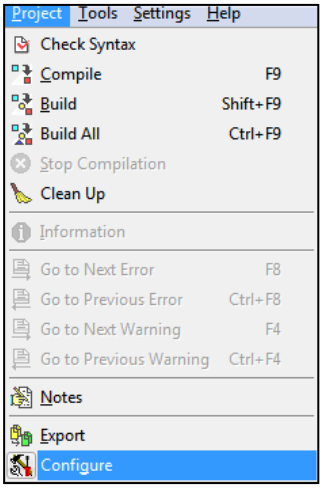
```
while (1)
      {
       printf("salam");
        delay_ms(500);
      }
```

برنامه‌ی روبرو دقیقاً همان برنامه‌ی مثال قبل است با این تفاوت که به جای دستور putchar() از دستور printf() استفاده کردیم(این نکته را بخاطر داشته باشید که تابع printf() بسیار تابع سنگین‌تر و حجیم‌تری نسبت به تابع putchar() است و در برنامه‌هایی که تعداد خطوط بسیار زیاد است و ممکن است حجم برنامه زیاد شود بهتر است از putchar استفاده کنیم).

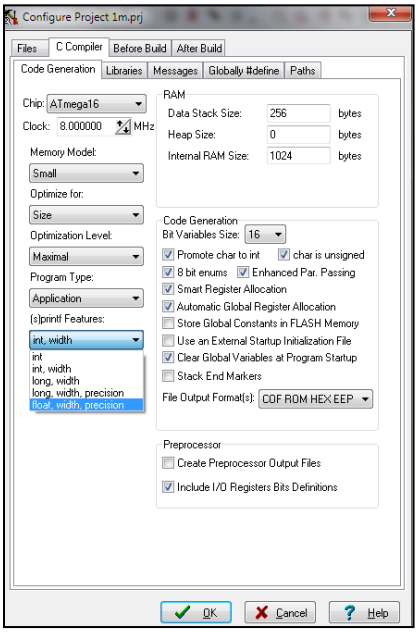
اگر بخواهیم کنار کلمه‌ی ارسالی یک عدد هم ارسال کنیم، درون دستور printf از %d استفاده می‌کنیم. ضمناً از زبان C به یاد داریم که برای فرستادن عدد اعشاری (float) از %f و برای حرف (char) هم از %c استفاده می‌کنیم.

به این نکته دقت کنید که برای استفاده از %f باید در کدویژن تغییرات زیر را انجام دهیم:

مطابق تصویر روبرو گزینه‌ی Project را انتخاب کرده و بر روی گزینه‌ی Configure کلیک می‌کنیم:



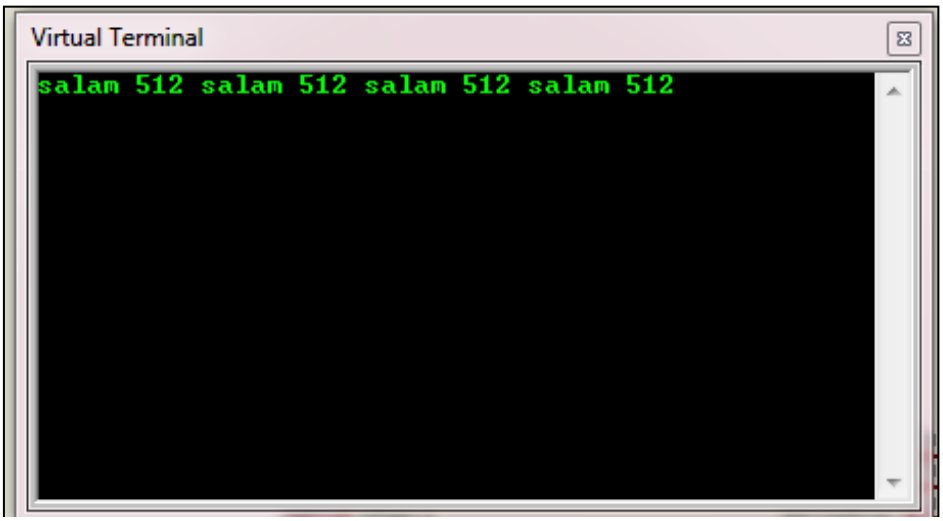
در پنجره‌ی باز شده مطابق شکل روبرو وارد تب C compiler شده و گزینه‌ی Features printf(s) را روی حالت float, width, precision قرار می‌دهیم:



مثال۳: برای مثال در کد زیر یک متغیر به نام a را بالای تابع while(1) مقداردهی کرده‌ایم و درون while(1) آن را همراه با عبارت "salam" هر ۵۰۰ میلی‌ثانیه ارسال می‌کنیم:

```
145   a=512;
146   while (1)
147         {
148          printf("salam %d",a);
149           delay_ms(500);
150         }
151   }
```

که این برنامه به صورت زیر اجرا می‌شود:



شکل۱۰-۹: اجرای برنامه‌ی مثال ۳

دریافت اطلاعات

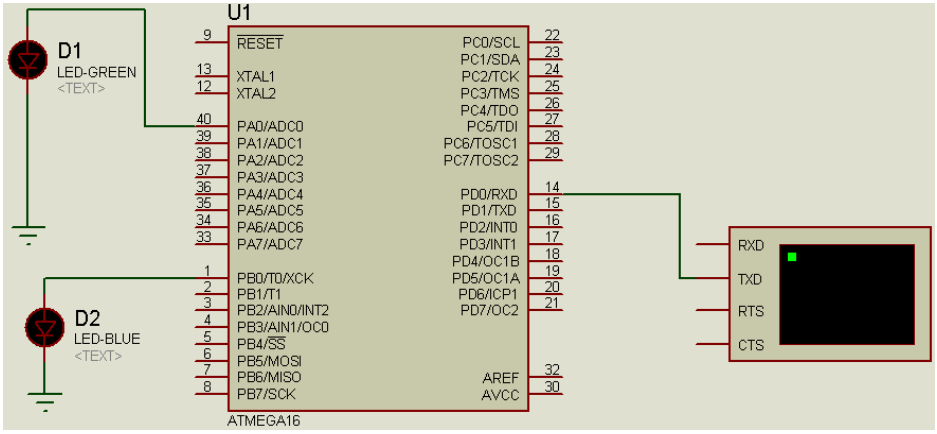
حال اگر بخواهیم توسط کامپیوتر یا میکرکنترلرهای دیگر اطلاعاتی بفرستیم تا میکروکنترلر آنها را دریافت کند و متناسب با آن کاری را انجام دهد، باید از گزینه‌ی Receiver استفاده کنیم.حال به مثال بعد توجه کنید.

مثال ۴: می‌خواهیم برنامه‌ای بنویسیم که اگر به ورودی USART کاراکتر w ارسال شد پایه‌ی A0 و اگر کاراکتر s ارسال شد پایه‌ی B0 روشن شود:

ابتدا یک متغیر از نوع کاراکتر به نام a تعریف می‌کنیم، سپس کد را به صورت زیر می‌نویسیم:

```
while (1)
      {
       a=getchar();
       if(a=='w')
         PORTA.0=1;
       if(a=='s')
         PORTB.0=1;
      }
```

دستور a=getchar() کاراکتر ورودی را از پایه‌ی RX میکرو دریافت کرده و درون متغیر a می‌ریزد حال اگر a برابر 'w' بود(یعنی کاراکتر ارسالی به میکرو w بود) A0 و اگر 's' بود B0 روشن می‌شود. این مدار را در پروتئوس به صورت زیر می‌کشیم:



شکل ۱۰-۱۰: مدار مثال ۴

مانند شکل RX میکرو را به TX ترمینال مجازی وصل کرده و برنامه را اجرا می‌کنیم.

فقط باید حواسمان باشد که اول بر روی صفحه‌ی سیاه‌رنگ ترمینال کلیک کنیم بعد حروف را وارد کنیم، البته در صفحه‌ی سیاه رنگ حروف ارسالی را نمی‌توانیم ببینیم ولی ترمینال آنها را ارسال می‌کند.یکی از دستورات پرکاربرد دیگر برای دریافت اطلاعات دستور gets(1,2); می‌باشد که در محل (۱) نام متغیری که می‌خواهیم رشته درون آن ذخیره شود و در محل (۲) تعداد کاراکترهای دریافتی را می‌نویسیم برای مثال دستور gets(s,5) ۵ کاراکتر پشت سر هم را دریافت کرده و آن را درون رشته‌ی s می‌ریزد.

ارسال اطلاعات برروی کامپیوتر

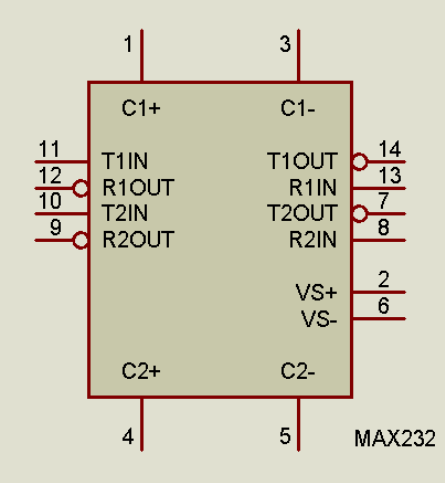
چطور می‌توانیم این اطلاعات را بر روی کامپیوتر بفرستیم؟

یکی از راه‌های ارسال اطلاعات به کامپیوتر پروتکل RS232 است که در فصل قبل با آن آشنا شدیم و دیدیم که سطح منطقی RS232 با سطح منطقی میکرو(TTL) فرق دارد و برای آنکه بخواهیم این دو را با هم مرتبط کنیم باید از یک مبدل استفاده کنیم که مبدل Max232 یکی از این مبدل‌هاست.

مثال ۵: اکنون در این مثال می‌خواهیم با استفاده از همین مبدل تعدادی کاراکتر را به RS232 ارسال کنیم. با یک برنامه‌ی خیلی ساده شروع می‌کنیم که عبارتی مانند"salam" را به RS232 ارسال کند پسUSART را در حالت فرستنده قرار می‌دهیم و کد زیر را برای ارسال کلمه‌ی salam می‌نوسیم:

```
146   while (1)
147         {
148           printf("salam");
149           delay_ms(500);
150         }
```

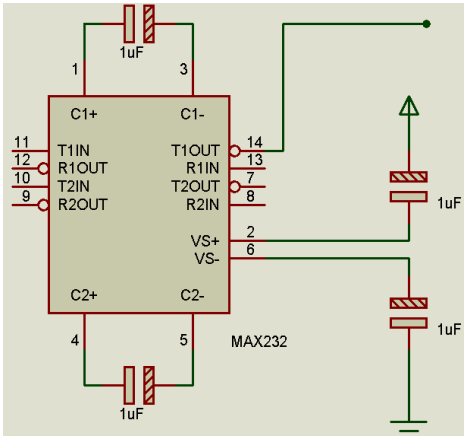
پروتئوس را باز می‌کنیم و مطابق شکل زیر آی‌سیMax232 را انتخاب می‌کنیم:



شکل۱۰-۱۱: آی‌سی MAX232 در پروتئوس

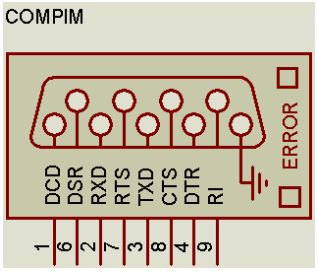
همانطور که توضیح دادیم و در شکل نیز از نام پایه‌ها معلوم است پایه‌ی ۱۱ ورودی TTL یعنی ورودی سیگنال میکروکنترلر و پایه‌ی ۱۴ خروجی همان سیگنال تبدیل شده به سطح منطقی RS232 است.

همانطور که در فصل قبل برگه‌ی اطلاعاتی(datasheet) Max232 را مشاهده کردیم، خازن‌های توصیه‌شده توسط دیتاشیت را نصب می‌کنیم:

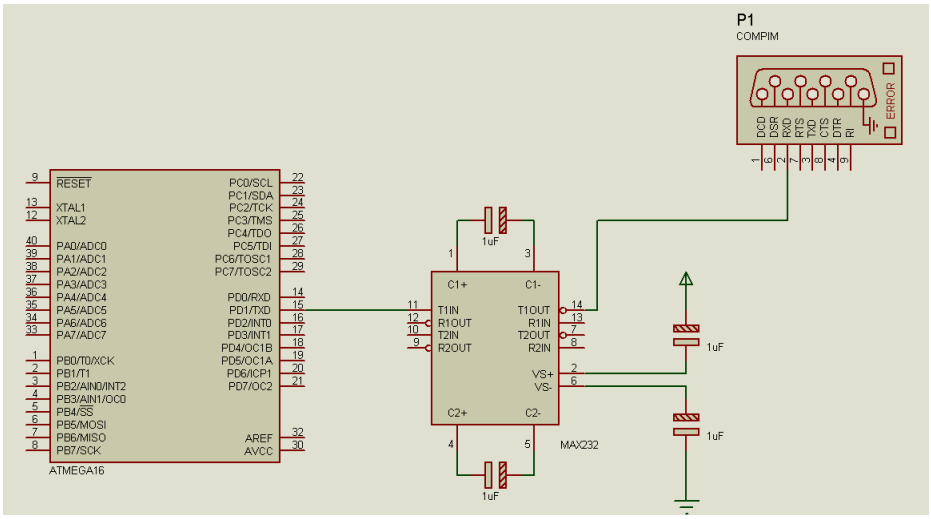


شکل ۱۰-۱۲: اتصال خازن‌های MAX232

برای آوردن درگاه RS232 باید عبارتCOMPIM را از مجموعه قطعات پروتئوس انتخاب کنیم:



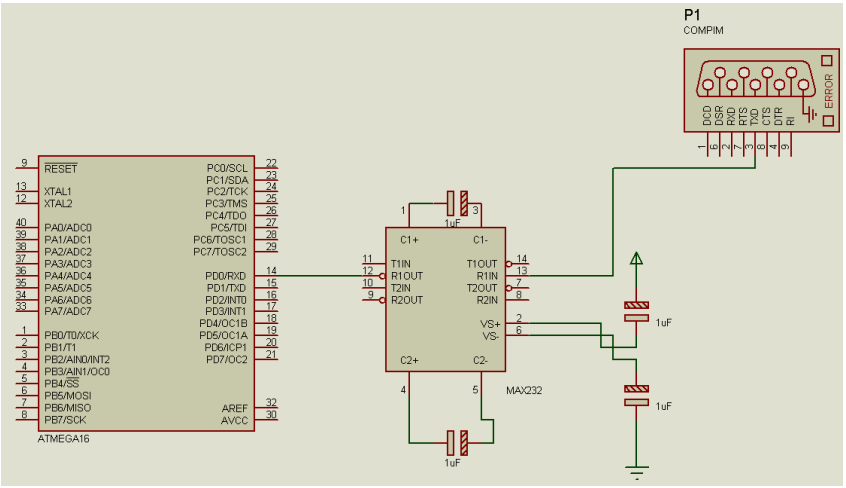
سپس یک میکرو ATmega16 انتخاب می‌کنیم و مدار را شبیه به شکل زیر می‌بندیم:



شکل ۱۰-۱۴: مدار مثال ۵

با این کار سیگنال خروجی از میکرو که در سطح منطقی TTL است بـه سـطح منطقـی RS232 تبدیل می‌شود و می‌توان آن را به کامپیوتر متصل کرد.

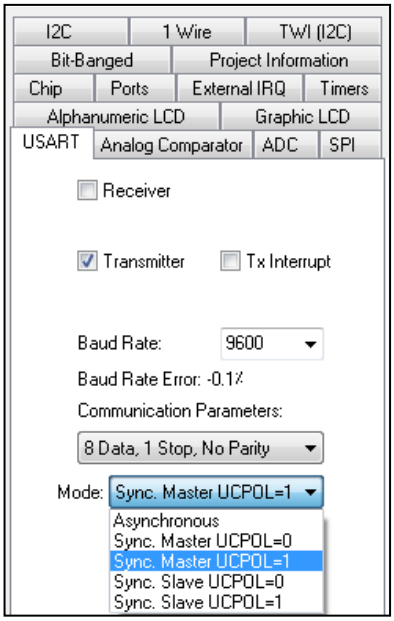
اگر می‌خواستیم سیگنال ورودی را از کامپیوتر به میکرو وصل کنیم باید همانند شـکل زیـر وصـل می‌کردیم:



شکل ۱۰-۱۵: ارسال اطلاعات از کامپیوتر به میکرو

ارتباط USART در حالت سنکرون

تا الان با ارتباط USART در حالت آسنکرون (غیرهمزمان) آشنا شدیم حال می‌خواهیم با نحوه‌ی برقـراری ارتبـاط USART در حالـت سـنکرون آشـنا شویم.



گفتیم در ارتباط سنکرون علاوه بر اینکه یـک سـیم بـه عنوان سیم داده بین فرستنده و گیرنده وجود دارد یک سیم دیگر به عنوان کلاک هم بین آنهـا وجـود دارد. در قدم اول کدویزارد را باز کرده و مد سـنکرون را بررسـی می‌کنیم:

ابتدا فرستنده(Transmitter) یا گیرنـده(Receiver) بودن میکرو را مشخص می‌کنیم و مانند مـد آسـنکرون نرخ ارسال اطلاعات را انتخاب می‌کنیم، نکته‌ای که باید به آن توجه کنیم این است که در مد سـنکرون قابلیـت دو برابر کردن نرخ ارسال وجود ندارد.

پس از انتخاب قالب بسته‌ی ارسال(مثلا 8 Data، 1 Stop، No Parity) روی گزینه‌ی Mode کلیک کرده و از گزینه‌های موجود یکی را انتخاب می‌کنیم:

مد اول: ارتباط آسنکرون می‌باشد.

مد دوم: مربوط به ارتباط سنکرون است و در حالت Master(به معنای مدیر یا رئیس) می‌باشد که مربوط به فرستنده است و میکروکنترلری را که از آن در حالت فرستنده استفاده می‌کنیم روی مد Master قرار می‌دهیم.

مد سوم: مربوط بهMaster است با این تفاوت که در مد دوم UCPOL=0 و در مد سوم UCPOL=1 است.

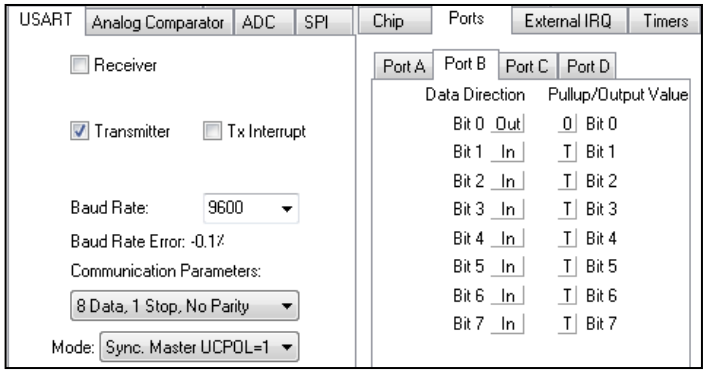
در ارتباط سنکرون گفتیم که یک سیم به عنوان کلاک است و در حقیقت لحظه‌ی ارسال و دریافت اطلاعات را مشخص می‌کند، در حالت Master، UCPOL=0 به این معناست که ارسال اطلاعات در لبه‌ی پایین رونده‌ی هر کلاک انجام شود و UCPOL=1 به این معناست که ارسال اطلاعات در لبه‌ی بالارونده‌ی هر کلاک انجام شود.

مد چهارم و پنجم: مربوط بهSlave بوده(به معنی فرمانبردار (گیرنده‌ی اطلاعات)) و برای زمانی است که میکرو به عنوان گیرنده(Receiver) استفاده می‌شود که در این حالت UCPOL=0 به این معناست که دریافت اطلاعات در لبه‌ی پایین رونده‌ی هر کلاک انجام شود و UCPOL=1 به معنای دریافت اطلاعات در لبه‌ی بالارونده‌ی هر کلاک است.

پایه‌ی مربوط به کلاک، پایه‌ی PB0 یا XCK است که در حالت فرستده(Master) این پایه باید به صورت خروجی و در حالت گیرنده(Slave) باید به عنوان ورودی تنظیم شود.

مثال ۶: برنامه‌ای بنویسید که از دو میکروکنترلر یکی به عنوان فرستنده و دیگری به عنوان گیرنده استفاده شود. میکروکنترلر فرستنده یک کاراکتر مانند 's' را در حالت سنکرون بفرستد و میکروکنترلر گیرنده کاراکتر دریافتی را بر روی LCD نمایش دهد:

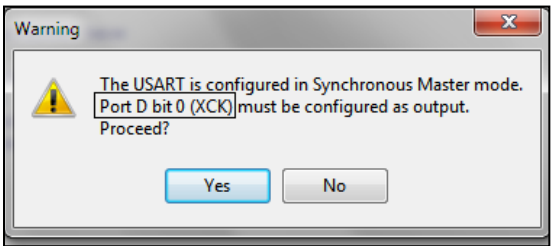
برای اینکار ابتدا کدویزارد را باز کرده و در مرحله‌ی اول تنظیمات مربوط به میکروکنترلر فرستنده را انجام می‌دهیم:



در این مثال مد را در حالت سنکرون و لبه‌ی بالارونده انتخاب می‌کنیم که داده‌ها در لبه‌ی بالارونده ارسال شوند، پایه‌ی PB0 را در حالت خروجی قرار می‌دهیم چون سیگنال کلاک از این پایه خارج می‌شود.

قبلاً مشاهده کردیم زمانی که در کدویزارد پایه‌ای را که باید خروجی باشد خروجی قرار نمی‌دادیم خود کدویزارد پیام اخطاری می‌داد و خودش آن پایه را در حالت خروجی قرار می‌داد.

ولی در این مورد دقت کنید که در این نسخه کدویژن(CodeVisionAVR V2.05.3) اشتباهاً خود کدویزارد پایه‌ی PD0 را به عنوان خروجی در نظر می‌گیرد **که این اشتباه است** و شما خودتان پایه‌ی PB0 را خروجی کنید.

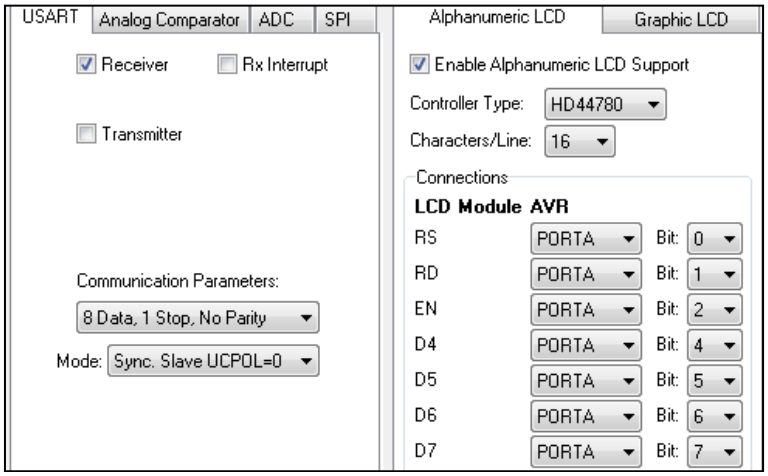


شکل ۱۰-۱۸: پیام اخطار کدویزارد(که اشتباه می‌باشد)

حال وارد برنامه شده و کد مربوط به ارسال کاراکتر 's' را می‌نویسیم:

```
146  while (1)
147        {
148          putchar('s');
149        }
```

سپس تنظیمات مربوط به میکروکنترلر گیرنده را انجام می‌دهیم:



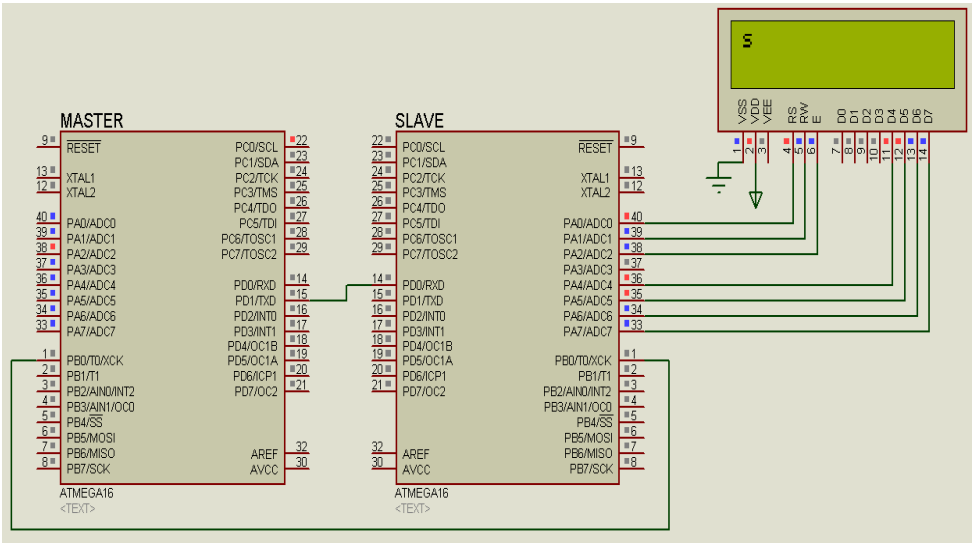
شکل ۱۰-۱۹: تنظیمات گیرنده در مثال ۶

در این بخش قسمت مربوط به نرخ دریافت اطلاعات(Baud Rate) وجود ندارد، چون ارتباط به صورت سنکرون(همزمان) است و هر زمان که فرستنده کلاک را ارسال کند در لبه‌ی پایین‌رونده‌ی کلاک داده‌ها دریافت می‌شوند(چون مد Slave UCPOL=0 انتخاب شده است). ضمناً پایه‌ی PB0 باید روی حالت ورودی(PIN) باشد چون سیگنال کلاک به این پایه وارد می‌شود. LCD را نیز فعال می‌کنیم تا کاراکتر دریافتی را روی آن نشان دهیم.

سپس کد مربوط به گیرنده را می نویسیم:

```
146  while (1)
147  {
148    c=getchar();
149    lcd_gotoxy(0,0);
150    lcd_putchar(c);
151  }
```

و مدار مربوط به این ارتباط را در پروتئوس می‌کشیم:

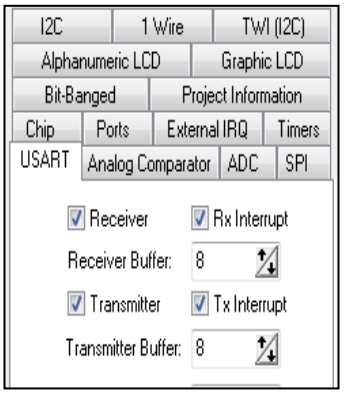


شکل ۱۰-۲۰: مدار مثال ۶

کد مربوط به فرستنده را در میکروکنترلری که نامش را Master گذاشتیم و کد مربوط به گیرنده را در میکروکنترلر Slave، لود(Load) می‌کنیم و فرکانس هر دو را روی ۸ مگاهرتز تنظیم می‌کنیم.

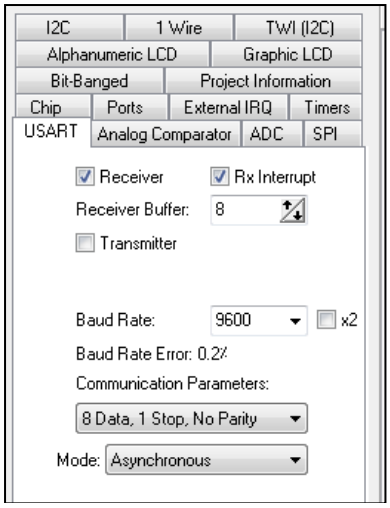
وقفه‌ی ارتباط USART

ارتباط USARTدر این میکروکنترلرها وقفه‌هایی برای ارسال و دریافت داده در نظر گرفته است:



وقتی تیک Rx-interrupt را می‌زنیم وقفه‌ی دریافت را فعال کرده‌ایم. این وقفه بدین صورت می‌باشد که یک بافر دریافت(Buffer-Receiver) وجود دارد که می‌توان آن را به مقدار دلخواه تنظیم‌کرد که در شکل بالا روی مقدار ۸ تنظیم شده است و بدین معناست که وقتی ۸ داده توسط این بافر دریافت شد وارد وقفه شده و یک سری عملیات انجام - دهد.

مثال۷: فرض کنید می‌خواهیم میکروکنترلر در حالت دریافت باشد و هر زمان ۸داده دریافت کرد، LED را که به پایه‌ی A0 متصل است روشن کند، برای این کار ابتدا تنظیمات را مطابق شکل زیر در کدویزارد انجام می‌دهیم:



شکل ۱۰-۲۲: تنظیمات مثال ۷

PORTA را نیز به صورت خروجی تنظیم می‌کنیم. پس از آن که کد را Generate کردیم مشاهده می‌کنیم که کدویزارد یک بافر نرم‌افزاری با نام عمومی rx_buffer را تولید می‌کند که در آن متغیر عمومی rx_wr_inde همواره شماره بایتی را که برای آخرین بار داده در آن قرار گرفته در خود نگه می‌دارد. برای وقفه‌ی سریال مربوط به ارسال داده‌ها نیز یک بافر با نام عمومی tx_buffer و متغیر عمومی tx_wr_index برای تغییر اندیس آن تولید می‌شود.

حال در خطی که دستور شرطی(if) زیر قرار گرفته است دستور مورد نظرمان را قرار می‌دهیم:

```
if (rx_wr_index == RX_BUFFER_SIZE)  rx_wr_index=0;
```

دستورمان را که مربوط به روشن شدن پایه‌ی A0 است، به صورت زیر می‌نویسیم:

```
if (rx_wr_index == RX_BUFFER_SIZE) {  PORTA.0=1;
rx_wr_index=0;}
```

خط مربوط به این دستور را در برنامه مشاهده می‌کنید:

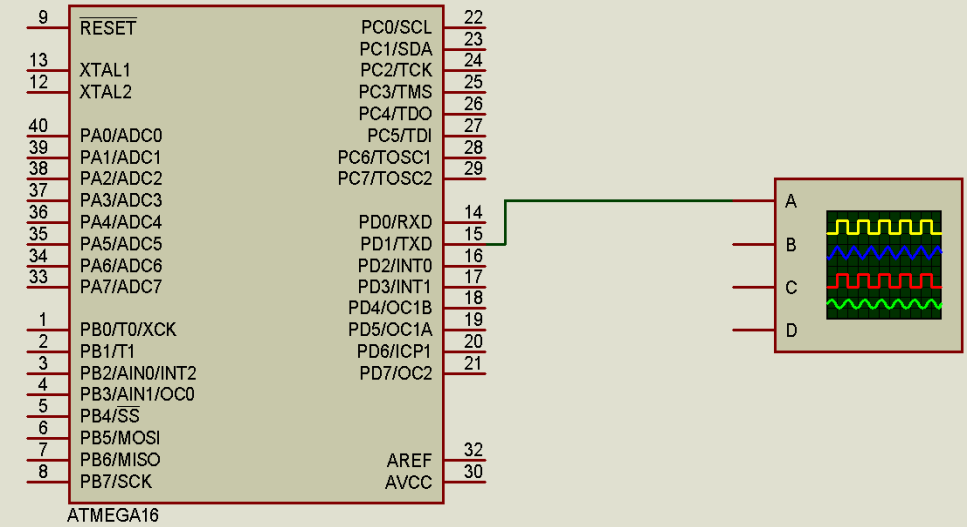
```
// USART Receiver interrupt service routine
interrupt [USART_RXC] void usart_rx_isr(void)
{
char status,data;
status=UCSRA;
data=UDR;
if ((status & (FRAMING_ERROR | PARITY_ERROR | DATA_OVERRUN))==0)
   {
   rx_buffer[rx_wr_index++]=data;
#if RX_BUFFER_SIZE == 256
   // special case for receiver buffer size=256
   if (++rx_counter == 0) rx_buffer_overflow=1;
#else
   if (rx_wr_index == RX_BUFFER_SIZE){PORTA.0=1; rx_wr_index=0; }
   if (++rx_counter == RX_BUFFER_SIZE)
      {
      rx_counter=0;
      rx_buffer_overflow=1;
      }
#endif
   }
}
```

می‌توان این کد را GENERATE کرده و مشاهده کنیم که اگر در پنجره‌ی Virtual Terminal ۸بار کاراکتری را ارسال کنیم، در بار هشتم LED روشن می‌شود. برای مثال اگر در تنظیمات کدویزارد اندازه بافر سایز را ۱۶ می‌گذاشتیم باید ۱۶ کاراکتر را ارسال می‌کردیم تا وقفه رخ دهد و LED روشن شود. البته کارهای بسیار پیچیده‌تری با این وقفه می‌توان انجام داد که متناسب با هر برنامه می‌تواند متفاوت باشد.

یک گام فراتر

تا اینجا با نحوه‌ی این ارتباط و دریافت و ارسال اطلاعات آشنا شدیم. حال می‌خواهیم ببینیم زمانی‌که یک داده را ارسال می‌کنیم، دقیقاً چه سیگنالی ارسال می‌شود که توسط گیرنده درست تشخیص داده می‌شود و همچنین اینکه می‌گوییم ارتباط USART در حالت آسنکرون دارای یک قاب مخصوص به خود است و اطلاعات را در آن قاب می‌فرستد به چه معناست؟

مثال۸: ابتدا یک کد ساده می‌نویسیم که یک کاراکتر مانند**'a'**را ارسال کند و در پروتئوس به جای آنکه پایه‌ی TX میکرو را به یک Virtual-Terminal وصل کنیم آن را به یک اسیلوسکوپ وصل می‌کنیم که دقیقاً شکل موج ارسالی را مشاهده کنیم:



شکل ۱۰-۲۳: مشاهده‌ی شکل موج ارسالی در اسیلوسکوپ

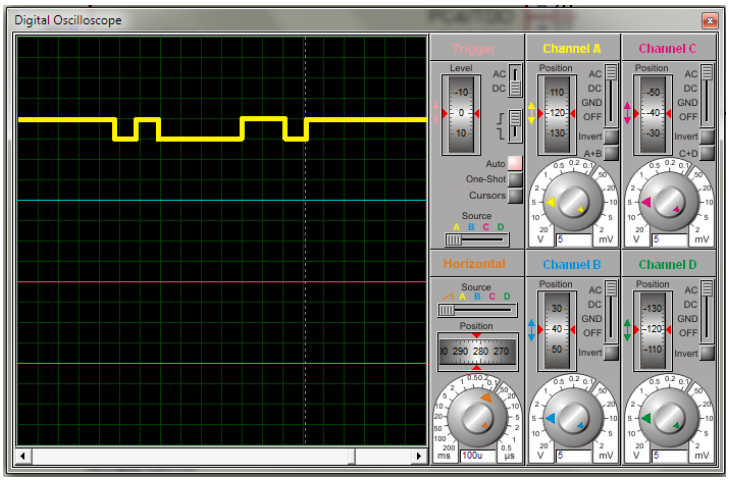
فقط قبل از آنکه برنامه را اجرا کنیم لازم است که یک مقدمه کوتاه در مورد کاراکترها در کامپیوتر بدانیم.

هر کاراکتر در کامپیوتر دارای یک کد اسکی مخصوص به خود است. اسکی یا ASCII مخفف **American Standard Code for Information Interchange** می‌باشد برای مثال کد اسکی مربوط به کاراکتر 'S'، ۱۱۵ است که در مبنای دو برابر ۰۱۱۱۰۰۱۱ می‌شود و کامپیوتر هر زمان که این شماره را ببیند آن را به عنوان کاراکتر 'S' می شناسد.

در جدول انتهای فصل کد اسکی انواع کاراکترها موجود است که این کدها در مبنای ۱۰ (دسیمال) و ۲(باینری)نوشته شده‌اند.

برای مثال کد دسیمال حرف 'a' برابر عدد ۹۷ در مبنای ده است که در مبنای دو برابر ۰۱۱۰۰۰۰۱ می‌شود.تمام این حروف در مبنای دو عددی ۸بیتی هستند و هر حرفی را می‌توان با ۸بیت نمایش داد. حال به برنامه باز می‌گردیم و حرف 'a' را توسط میکرو ارسال می‌کنیم و

سیگنال ارسالی را مشاهده می‌کنیم. اگر برنامه را اجرا کنیم بر روی اسیلوسکوپ موج زیر را مشاهده می‌کنیم:



شکل ۱۰-۲۴: مشاهده‌ی شکل موج حرف 'a' در اسیلوسکوپ

همانطور که در جدول انتهای این فصل آمده است، کاراکتر 'a' در مبنای دو به شکل ۰۱۱۰۰۰۰۱ است و اگر به قاب ارسال اطلاعات در حالت آسنکرون نگاهی بیندازیم:

توقف	توقف									بیت شروع	خط بیکار
یک	یک	توازن	D7	D6	D5	D4	D3	D2	D1	D0	صفر	یک

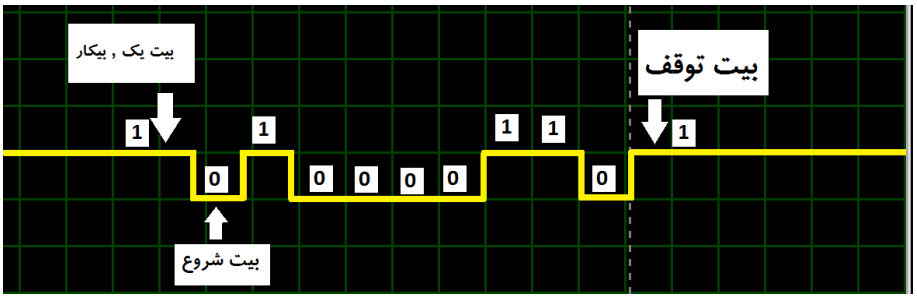
جدول ۱۰-۱: قاب ارسال اطلاعات در حالت آسنکرون

این بسته برای کاراکتر 'a' باید به فرم زیر باشد:

توقف								بیت شروع	خط بیکار
1	0	1	1	0	0	0	0	1	0

جدول ۱۰-۲: قاب ارسال برای کاراکتر 'a'

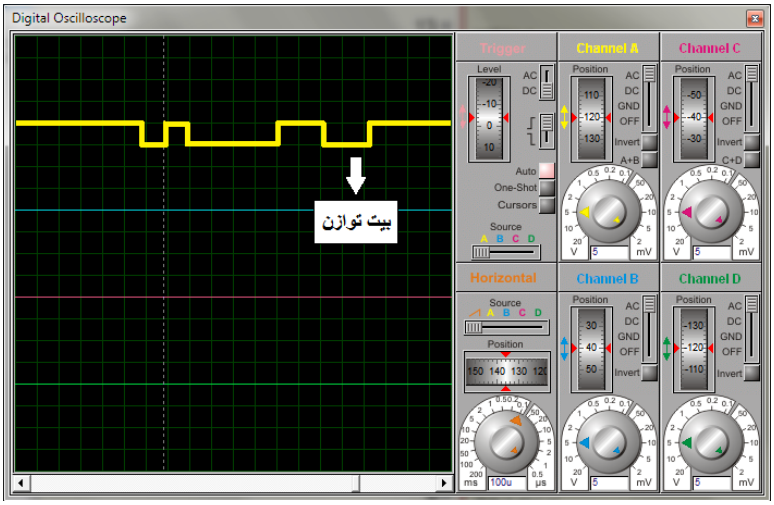
و چون حالت ارسال را بر روی No parity قرار داده بودیم بیت توازن هم ندارد. حال اگر به شکل زیر نگاه کنیم دقیقاً همین بسته را مشاهده می‌کنیم:



فرض کنید بخواهیم همین کاراکتر را در حالت توازن زوج ارسال کنیم، آن وقت باید یک بیت دیگر به عنوان بیت توازن به این بسته اضافه شود و مقدارش هم باید صفر باشد(چون تعداد یک‌های این بسته‌ی ارسالی خودش زوج است).

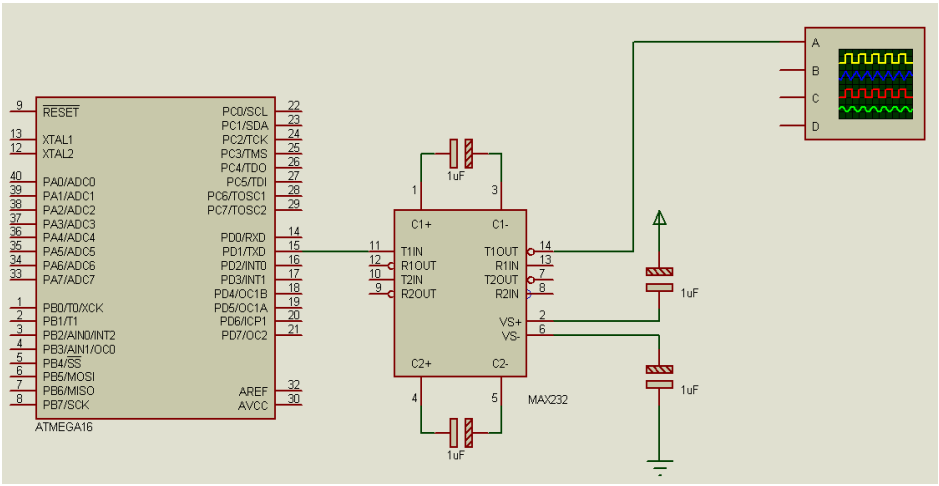
مثال ۹: در تنظیمات کدویزارد مد را روی حالت ۸بیت داده(data) و یک بیت توقف و تـوازن زوج می‌گذاریم.

در شبیه‌ساز پروتئوس شکل موج به صورت زیر در مشاهده می‌شود:



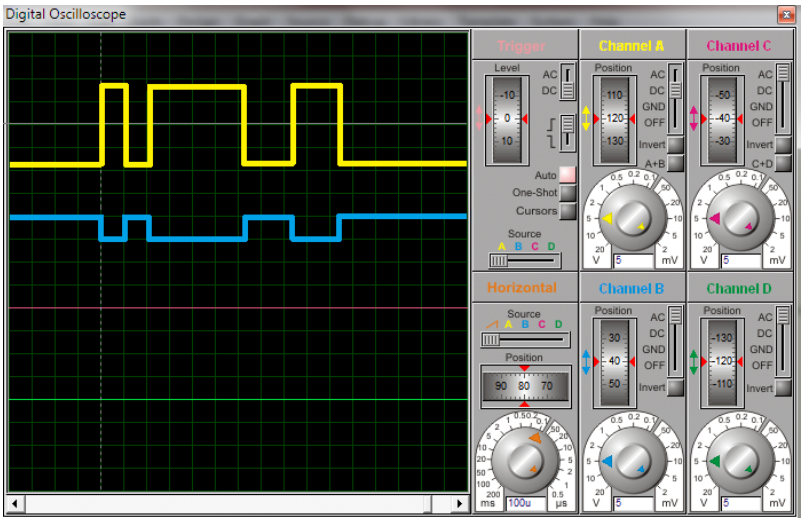
شکل ۱۰-۲۶: حالت توازن زوج

اگر بخواهیم همین کاراکتر را در سطح منطقی RS232 ببینیم، ابتـدا مـدار زیـر را در پروتئـوس می‌کشیم سپس برنامه را اجرا می‌کنیم.



شکل ۱۰-۲۷: مدار مثال ۹ برای ارسال اطلاعات به کامپیوتر

پایه‌ی دیگری از اسیلوسکوپ را به خروجی پایه‌ی TX می‌زنیم تا روی اسیلوسکوپ هر دو شکل موج مربوط به کاراکتر 'a' را در دو سطح منطقی TTL و RS232 به طور همزمان ببینیم و با هم مقایسه کنیم:



شکل ۱۰-۲۸: بسته‌ی اطلاعاتی کاراکتر 'a' در سطح منطقی TTL و RS232

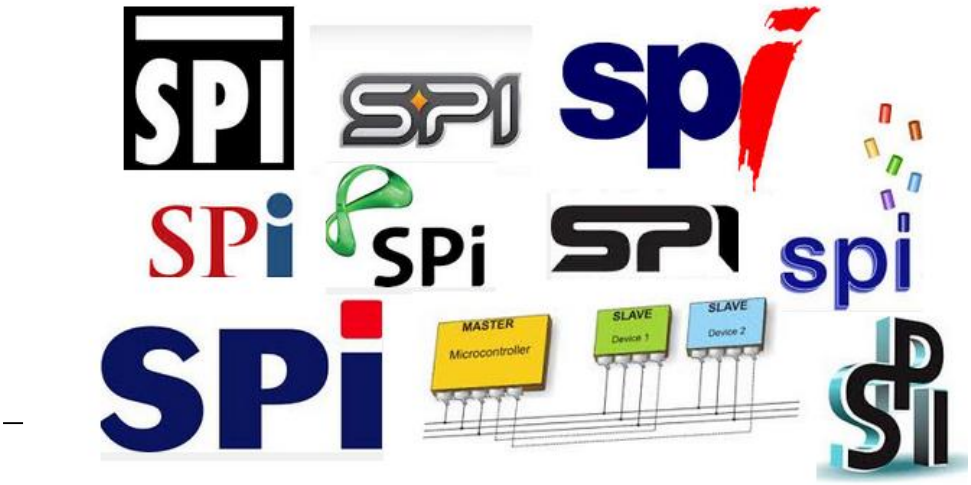
همانطور که در شکل پیداست Max232 سطح منطقی TTL را به RS232 تبدیل کرده است. اگر دقت کنید 5ولت را به تقریباً 9- ولت و 0ولت را به تقریباً 9+ولت تبدیل کرده است(همانطور که قبلاً اشاره شد سطح منطقی RS232 برای ولتاژهای بین 0 تا 25-ولت به عنوان یک منطقی و برای ولتاژ بین 3+ تا 25+ به عنوان صفر منطقی است).

در آخرین صفحه هم جدول کدهای اسکی به همراه نمایش باینری آنها را مشاهده می‌کنیم.

Decimal	Binary	ASCII	Decimal	Binary	ASCII	Decimal	Binary	ASCII	Decimal	Binary	ASCII
0	00000000	NUL	32	00100000	SP	64	01000000	@	96	01100000	`
1	00000001	SOH	33	00100001	!	65	01000001	A	97	01100001	a
2	00000010	STX	34	00100010	"	66	01000010	B	98	01100010	b
3	00000011	ETX	35	00100011	#	67	01000011	C	99	01100011	c
4	00000100	EOT	36	00100100	$	68	01000100	D	100	01100100	d
5	00000101	ENQ	37	00100101	%	69	01000101	E	101	01100101	e
6	00000110	ACK	38	00100110	&	70	01000110	F	102	01100110	f
7	00000111	BEL	39	00100111	'	71	01000111	G	103	01100111	g
8	00001000	BS	40	00101000	(	72	01001000	H	104	01101000	h
9	00001001	HT	41	00101001	)	73	01001001	I	105	01101001	i
10	00001010	LF	42	00101010	*	74	01001010	J	106	01101010	j
11	00001011	VT	43	00101011	+	75	01001011	K	107	01101011	k
12	00001100	FF	44	00101100	,	76	01001100	L	108	01101100	l
13	00001101	CR	45	00101101	-	77	01001101	M	109	01101101	m
14	00001110	SO	46	00101110	.	78	01001110	N	110	01101110	n
15	00001111	SI	47	00101111	/	79	01001111	O	111	01101111	o
16	00010000	DLE	48	00110000	0	80	01010000	P	112	01110000	p
17	00010001	DC1	49	00110001	1	81	01010001	Q	113	01110001	q
18	00010010	DC2	50	00110010	2	82	01010010	R	114	01110010	r
19	00010011	DC3	51	00110011	3	83	01010011	S	115	01110011	s
20	00010100	DC4	52	00110100	4	84	01010100	T	116	01110100	t
21	00010101	NAK	53	00110101	5	85	01010101	U	117	01110101	u
22	00010110	SYN	54	00110110	6	86	01010110	V	118	01110110	v
23	00010111	ETB	55	00110111	7	87	01010111	W	119	01110111	w
24	00011000	CAN	56	00111000	8	88	01011000	X	120	01111000	x
25	00011001	EM	57	00111001	9	89	01011001	Y	121	01111001	y
26	00011010	SUB	58	00111010	:	90	01011010	Z	122	01111010	z
27	00011011	ESC	59	00111011	;	91	01011011	[	123	01111011	{
28	00011100	FS	60	00111100	<	92	01011100	\	124	01111100	|
29	00011101	GS	61	00111101	=	93	01011101	]	125	01111101	}
30	00011110	RS	62	00111110	>	94	01011110	^	126	01111110	~
31	00011111	US	63	00111111	?	95	01011111	_	127	01111111	DEL

فصل یازدهم

ارتباط SPI



در این فصل خواهیم خواند:

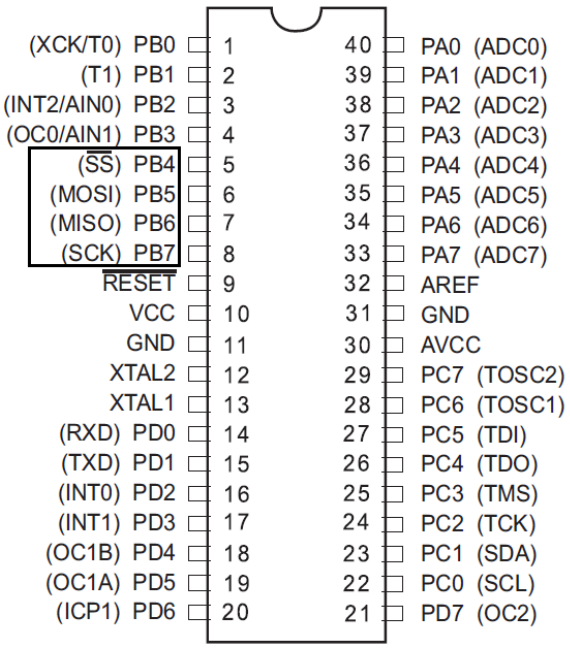
۱. آشنایی مقدماتی با ارتباط SPI
۲. نحوه‌ی برقراری ارتباط در واحد SPI
۳. تنظیمات اولیه‌ی بخش SPI در کدویزارد
۴. تنظیمات بخش SPI Type
۵. تنظیمات بخش SPI Clock Rate
۶. تنظیمات بخش Data Order
۷. مدهای ارتباط SPI
۸. تابع کار با SPI در کدویژن

ارتباط SPI

در این فصل قصد داریم با ارتباط SPI که زیرمجموعه‌ای از ارتباط‌های سریال می‌باشد آشنا شویم و بفهمیم که چگونه می‌توان با استفاده از این ارتباط وسایل (Device) مختلف و میکروکنترلرهای دیگر را با همدیگر مرتبط سازیم.

آشنایی مقدماتی با ارتباط SPI

اول از همه باید بدانیم SPI مخفف چه چیزی است و به چه معنا می‌باشد؟ ارتباط SPI یکی از ارتباطات استاندارد سریال سنکرون می‌باشد و مخفف **S**erial-**P**eripheral-**I**nterface می‌باشد و معنای لغوی آن هم در زبان فارسی "ارتباط سریال با دستگاه‌های جانبی" می‌باشد. این ارتباط برای اولین بار توسط شرکت موتورولا طراحی گردید و به شدت هم مورد استقبال قرار گرفت و در صنعت جای خود را به سرعت پیدا کرد چرا که با استفاده از این ارتباط می‌توان اطلاعات با حجم بالا را با سرعت بسیار زیاد منتقل کرد. در برقرارسازی این ارتباط از ۴ پایه‌ی میکروکنترلر به نام‌های SCK،MOSI،MISO و SS استفاده می‌شود که در شکل زیر این پایه‌ها را مشاهده می‌کنید:



شکل ۱۱-۱ : پایه‌های مربوط به ارتباط SPI در ATmega16

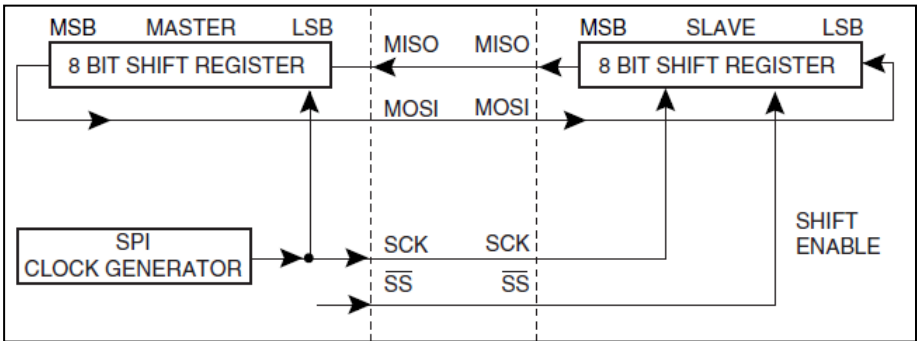
نحوه‌ی برقراری ارتباط در واحد SPI

حال با خواندن این مقدمه به سراغ اصل مطلب می‌رویم و اینکه این ارتباط چگونه برقرار می‌شود و اطلاعات را منتقل می‌سازد و هر کدام از این پایه‌ها چه هستند؟

همیشه در امر برقراری ارتباط بین دو نفر، یک نفر گوینده و نفر مقابل شنونده می‌باشد، در ارتباط SPI نیز همچین حالتی وجود دارد که یک دستگاه به عنوان فرستنده‌ی اطلاعات(همان گوینده) و دستگاه دیگر دریافت‌کننده‌ی اطلاعات(همان شنونده) می‌باشد که به فرستنده‌ی اطلاعات Master و به گیرنده‌ی اطلاعات Slave می‌گویند، همچنین در برقراری ارتباط، گوینده نیز باید متوجه شود که شنونده پیام او را فهمیده است پس باید پیام شنونده را دریافت کند، برای این منظور دو پایه در میکروکنترلر وجود دارد، یک پایه که حاوی اطلاعاتی می‌باشد که از فرستنده‌ی اطلاعات یا Master (گوینده)خارج و به دریافت‌کننده‌ی اطلاعات یا Slave (شنونده) وارد می‌شود که به این پایه MOSI(**M**aster **O**ut **S**lave **I**n) می‌گویند(پیامی که گوینده به شنونده می‌دهد) و پایه‌ی دیگر پایه‌ای می‌باشد که حاوی اطلاعاتی می‌باشد که از دریافت‌کننده‌ی اطلاعات یا Slave خارج می‌شود و به فرستنده‌ی اطلاعات Master وارد می‌شود که به این پایه MISO (**M**aster **I**n **S**lave **O**ut) می‌گویند (پیامی که شنونده به گوینده می‌دهد). نحوه‌ی سنکرون(هم‌زمان) کردن این اطلاعات هم توسط پایه‌ی تولیدکننده‌ی کلاک ارتباط SPI انجام می‌گیرد که به این پایه، پایه‌ی SCK می‌گویند.

حال سنکرون کردن اطلاعات به چه معنی می‌باشد؟ شما فرض کنید می‌خواهید در یک زمان اطلاعات را بگویید و طرف مقابل هم‌زمان اطلاعات شما را دریافت کند و پاسخ شما را بدهد، در میکروکنترلرهای AVR برای اینکه هم‌زمان‌سازی انجام گیرد از کلاک استفاده می‌شود بدین ترتیب که برای مثال در لبه‌ی بالارونده‌ی کلاک عمل ارسال اطلاعات و در لبه‌ی پایین‌رونده عمل دریافت اطلاعات انجام بگیرد. با هر کلاک در ارتباط SPI یک بیت داده بین Master و Slave و برعکس جابجا می‌شود. پایه‌ی دیگری که در برقراری این ارتباط نقش ایفا می‌کند پایه‌ی SS (Slave Select) یا انتخاب‌کننده‌ی دریافت‌کننده می‌باشد، حال کاربرد این پایه چیست؟ این پایه زمانی که صفر است دریافت‌کننده (Slave) متوجه می‌شود که باید اطلاعات را دریافت کند، کاربرد دیگر این پایه را در مثالی که اکنون می‌گوییم متوجه می‌شوید. شما فرض کنید یک گوینده (Master) هستید و در مقابلتان چندین نفر شنونده (Slave)، برای اینکه بتوانید مشخص کنید روی صحبتتان با کدامیک از این شنوندگان است باید به او خطاب کنید که با "تو" دارم صحبت می‌کنم، در میکروکنترلر این پایه هر موقع برای دستگاهی که نقشSlave را دارد صفر شود یعنی آن دستگاه باید آماده‌ی دریافت اطلاعات از Master باشد. با توجه به این توضیحات شکل این

پروتکل ارتباطی را که در دیتاشیت میکروکنترلر ATmega16 هم موجود است، را در شکل زیر مشاهده می‌کنیم. به نکاتی که در ادامه آمده توجه کنید:



شکل ۱۱-۲ : پروتکل **SPI**

نکته۱: همانطور که در شکل مشاهده می‌کنید پایه‌ی $\overline{SS}$ یک علامت بار در بالای خود دارد و این علامت بدین معناست که این پایه Active Low می‌باشد یعنی اینکه با صفر شدن فعال می‌گردد.

نکته۲: پایه‌ی SS در Master لزومی ندارد حتماً پایه SS میکروکنترلر باشد و هر پایه‌ی دیگری از میکروکنترلر می‌تواند باشد ولی در Slave باید حتماً پایه‌ی SS باشد زیرا این پایه قرار است مشخص کند که دریافت‌کننده(Slave) چه زمانی آماده‌ی دریافت اطلاعات از فرستنده (Master) باشد که این فرمان را از هر کدام از پایه‌های میکروکنترلری که در حالت Master است به میکروکنترلرSlave می‌توان داد(با صفر شدن آن پایه).

نکته۳: به جهت سیگنال MOSI توجه کنید، همانطور که مشاهده می‌کنید این سیگنال از Master خارج و به Slave وارد می‌شود.

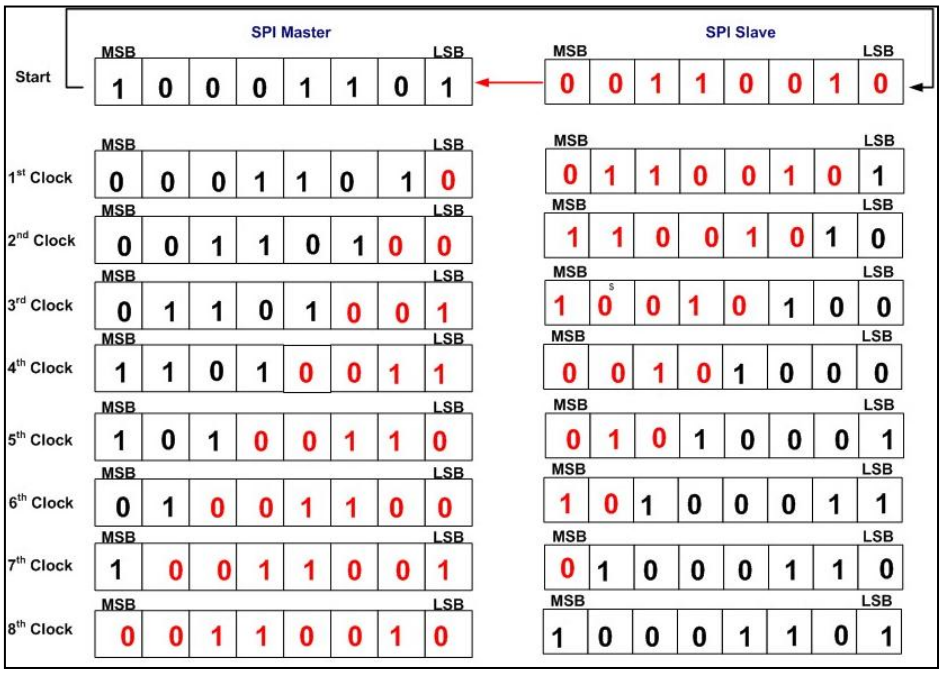
نکته۴: به جهت سیگنال MISO توجه کنید، همانطور که مشاهده می‌کنید این سیگنال از Slave خارج و به Master وارد می‌شود.

نکته۵: با توجه به شکل متوجه می‌شوید که سیگنال SCK هم به Master و هم به Slave وارد می‌شود و همانطور که گفته شد برای همزمان‌سازی(سنکرون‌سازی) ارسال و دریافت اطلاعات می‌باشد.

نکته۶: همانطور که در شکل مشاهده می‌کنید در هر بار تبادل اطلاعات Master و Slave می‌توانیم ۸ بیت(۱بایت) اطلاعات را تبادل کنیم.

اکنون با یک مثال با موضوعات مطرح شده در بالا بیشتر آشنا می‌شوید. فرض کنید دو میکروکنترلر داریم که یکی از آن‌ها به صورت Master و میکروکنترلر دیگر به صورت Slave تنظیم شده است و قصد ارسال عدد ۸ بیتی 0b10001101 (این عدد در مبنای ۱۶ برابر 0x8D و در مبنای ده

برابر ۱۴۱ می‌باشد)را توسط میکروکنترلر Master به میکروکنترلر Slave را داشته باشیم و همزمان قصد ارسال عدد باینری ۸ بیتی 0b00110010 (این عدد در مبنای ۱۶ برابر 0x32 و در مبنای ده برابر ۵۰ می‌باشد) را توسط میکروکنترلر Slave به میکروکنترلر Master داشته باشیم ، در این صورت جابجای داده‌ها بین Master و Slave در ارتباط SPI به شکل زیر انجام می‌گیرد:



شکل ۱۱ –الف :جابجایی داده بین Master و Slave

همانطور که در شکل مشاهد می‌کنید با هر کلاکی که توسط SCK صورت می‌گیرد یک بیت داده از Master به Slave و برعکس جابجا می‌گردد و با ۸ کلاک تمامی ۸ بیت داده بین Master و Slave جابجا می‌گردد. در ادامه در قالب ۴ مُد ارتباطی با نحوه‌ی اینکه چگونه بوسیله کلاک SCK هم عمل خواندن و هم عمل ارسال داده توسط Master و Slave صورت می‌گیرد، آشنا می‌شویم.

با این توضیحاتی که تا به اینجای کار گفته شد تا حدودی با این ارتباط آشنا شدید، با این مقدار آشنایی به قسمت تنظیمات کدویزارد ارتباط SPI می‌رویم تا دقیق‌تر با این ارتباط و نحوه‌ی تنظیمات آن در کدویزارد آشنا شویم.

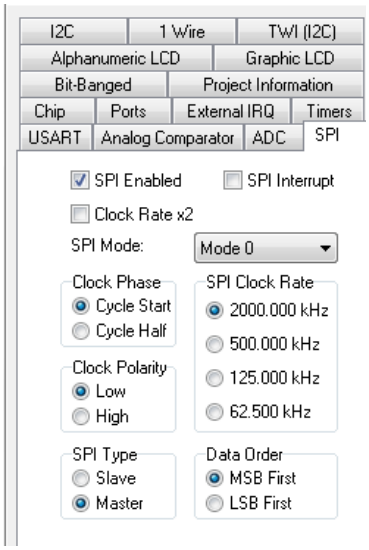
کاربرد هر کدام از ۴ پایه‌ی ارتباط SPI را در جدول ۱-۱ به صورت خلاصه مشاهده می‌کنید:

Pin	Direction, Master SPI	Direction, Slave SPI
MOSI	User Defined	Input
MISO	Input	User Defined
SCK	User Defined	Input
$\overline{SS}$	User Defined	Input

جدول۱۱-۱ : کارکرد هر کدام از پایه‌ها در پروتکل SPI

تنظیمات اولیه‌ی بخش SPI در کدویزارد

در بخش کدویزارد در تب SPI با فعال کردن ارتباط SPI با صفحه‌ای مانند شکل زیر روبرو می‌شویم:



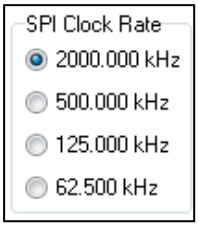
تنظیمات بخش SPI Type

در این بخش شما می‌توانید تعیین کنید که میکروکنترلر به صورت Master (فرستنده) یا Slave (دریافت‌کننده) باشد:

شکل ۱۱-۳

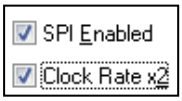
تنظیمات بخش SPI Clock Rate

در این بخش کاربر می‌تواند فرکانس مربوط به کلاک سنکرون‌ساز یا همان SCK را تنظیم کند که بسته به فرکانس‌های کاری میکروکنترلر می‌تواند دارای فرکانس‌های f/4،f/16،f/64،f/128 باشد (که در اینجا چون فرکانس کاری میکروکنترلر f=8Mhz می‌باشد دارای مقادیر۲مگاهرتز، ۵۰۰کیلوهرتز، ۱۲۵کیلوهرتز و ۶۲,۵کیلوهرتز می‌باشد) و تنظیمات آن به صورت پیش‌فرض روی فرکانس f/4 می‌باشد:



شکل ۱۱-۴

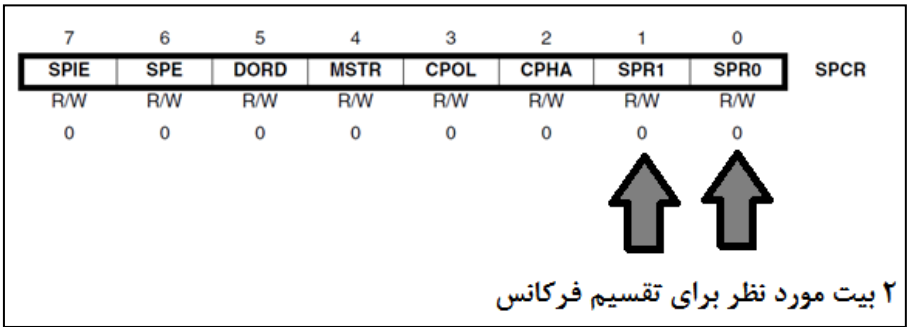
با زدن تیک گزینه‌ی Clock-Ratex2 می‌توان فرکانس کلاک را دو برابر کرد و انتقال اطلاعات با سرعت دوبرابر انجام گیرد:



شکل ۱۱-۵

یعنی در این حالت توانایی تولید فرکانس‌های f/2،f/8،f/32،f/64 را دارا می‌باشیم. البته برای اینکه مقسم‌های کلاک تولید فرکانس کلاک ارتباط SPI را دقیق‌تر بشناسیم بایستی ابتدا بیت‌های رجیستر SPCR (SPI Control Register)، دوبیت SPR1،SPR0 و بیت SPI2x (گزینه‌ی دو برابرکننده) را بشناسیم.

رجیستر کنترل ارتباط SPI را مطابق شکل زیر مشاهده می‌کنیم:



شکل ۱۱-۶ : رجیسترکنترلSPI

حال نحوه‌ی تقسیم فرکانس کلاک میکروکنترلر را در ارتباط SPI را مطابق جدول ۱۱-۲ مشاهده می‌کنیم:

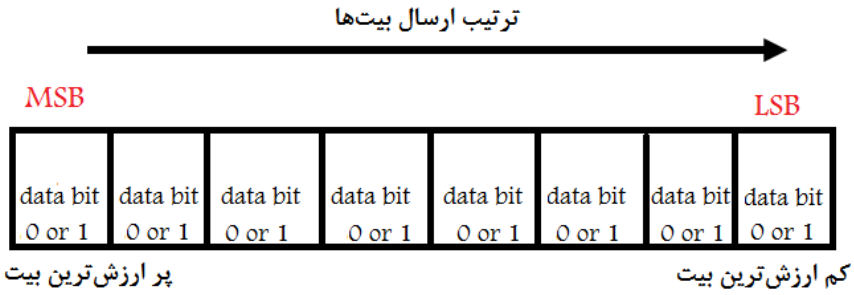
SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

جدول۱۱-۲ : مقسم کلاک ارتباط **SPI**

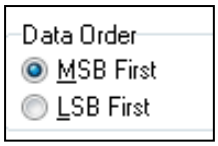
نکته: تولید کننده‌ی کلاک SCK در ارتباط SPI همیشه Master می‌باشد.

تنظیمات بخش Data order

در این بخش مشخص می‌کنیم که در هنگام تبادل ۱ بایت اطلاعات (۸ بیت) ارسال اطلاعات از پرارزش‌ترین بیت (MSB) به سمت کم‌ارزش‌ترین بیت (LSB) صورت بگیرد که در این صورت باید تیک MSB First را بزنیم:

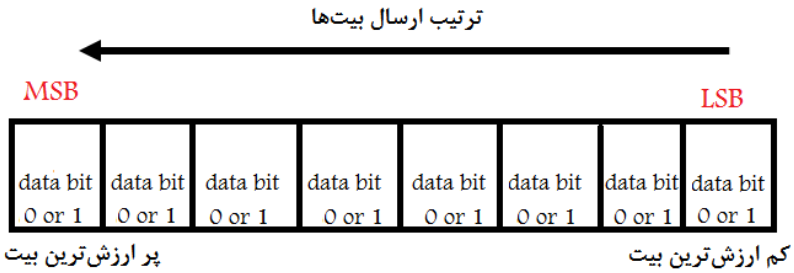


شکل ۱۱- ۷ : ارسال اطلاعات از بیت پرارزش به کم ارزش

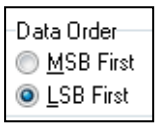


شکل ۱۱-۸

و یا اینکه اطلاعات از کم‌ارزش‌ترین بیت (LSB) به سمت پرارزش‌ترین بیت (MSB) صورت بگیرد که در این صورت باید تیک LSB First را بزنیم:



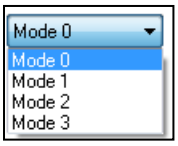
شکل ۱۱- ۹ : ارسال اطلاعات از بیت کم‌ارزش به پرارزش



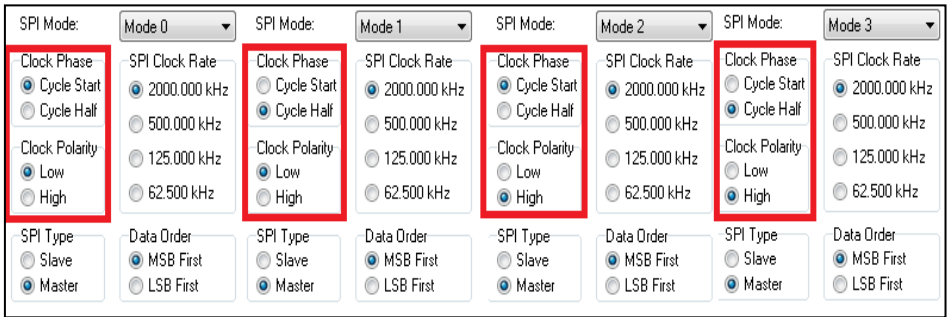
شکل ۱۱-۱۰

تا اینجای کار با ۳ تنظیم ساده‌ی ارتباط SPI یعنی SPI Type، SPI Clock Rate ، Data order آشنا شدیم، در ادامه با مُدهای ارتباطی SPI آشنا می‌شویم که درک آن بسیار در فهم ارتباط SPI مفید می‌باشد.

مدهای ارتباط SPI

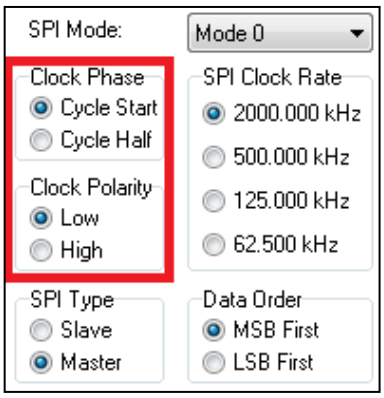


همانطور که در بخش تنظیمات کدویزارد مشاهده می‌کنید برای SPI ۴ مُد تعیین شده است که با انتخاب هر کدام از این ۴ مُد تیک‌های مربوط به بخش‌های Clock Phase و Clock Polarity عوض می‌شود.



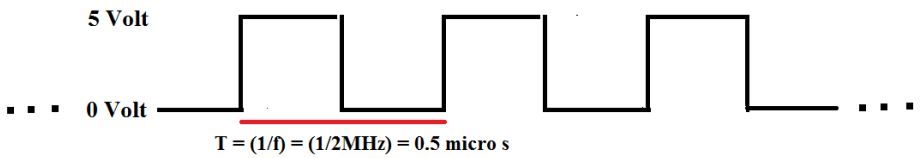
شکل ۱۱-۱۱ : مُدهایارتباطSPIدرکدویزارد

حال هر کدام از این مُدها چه هست و بیانگر چه چیزی می‌باشند؟
با انتخاب هر یک از مُدها در حقیقت فرمت ارسال داده را مشخص می‌کنیم و اینکه چه زمانی در ارتباط بوسیله‌ی کلاک SCK عمل خواندن داده و چه موقع عمل ارسال داده صورت بگیرد. در مُد صفر (Mode 0) همانطور که مشاهده می‌کنید، Clock Phase به صورت Cycle Start و Clock Polarity به صورت Low تنظیم می‌شود،مانند شکل زیر:



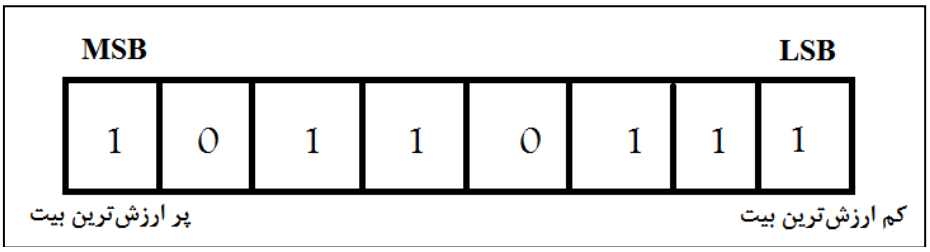
شکل ۱۱-۱۲ : مُدصفردرارتباطSPI

در این مُد از انتقال داده،کلاک مربوط به سنکرون‌سازی (SCK) هنگامی که در حالت بیکار (Idel) می‌باشد یعنی اینکه انتقال اطلاعاتی صورت نمی‌گیرد به صورت صفر یا Low می‌باشد (تنظیمات مربوط به Clock Polarity) و موقعیت لبه‌ی سیگنال SCK نسبت به هر بیت داده‌ی ارسالی (هر بار ارسال اطلاعات ۱ بایت(8 bits)داده جابجا می‌شود) در ابتدای سیکل یا Cycle Start می‌باشد (تنظیمات مربوط به Clock Phase). اگر با این توضیحات کمی سردرگم شده‌اید کمی صبر کنید در ادامه به صورت کامل با این مفاهیم آشنا می‌شویم.
هر کلاک SCK در میکروکنترلر به صورت یک موج مربعی می‌باشدکه فرکانس آن را در بخش SPI Clock Rate تنظیم می‌کنیم که در اینجا به صورت پیش‌فرض بر روی ۲ مگاهرتز می‌باشد یعنی هر تناوب ۰,۵ میکروثانیه طول می‌کشد همانند شکل زیر:



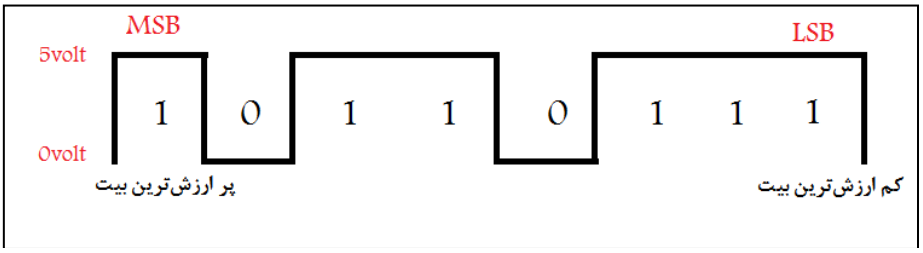
شکل ۱۱-۱۳ : کلاکSPI

همانطور که گفته شد در هر بار انتقال اطلاعات ۱ بایت (۸بیت) داده منتقل می‌شود یعنی می‌توانیم عددی بین ۰ تا ۲۵۵ را منتقل کنیم و با هر کلاک SCK یک بیت داده جابجا می‌شود(که در مجموع به ۸ کلاک نیاز داریم)، حال فرض کنید می‌خواهیم عدد ۱۸۳ را از طریق ارتباط SPI منتقل بسازیم، این عدد در مبنای ۲ (Binary) به صورت ۸ بیتی به شکل زیر در می‌آید:



شکل ۱۱-۱۴ : عدد ۱۸۳ در مبنای ۲

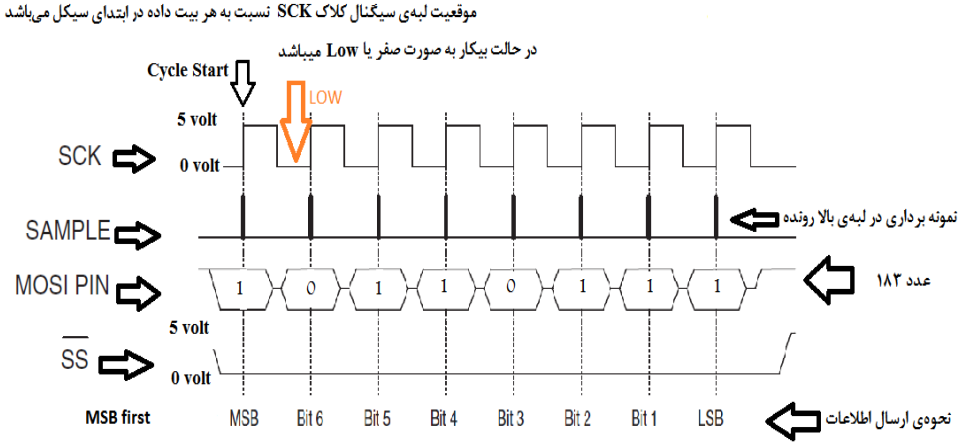
حال شکل موج مربعی (با لبه‌های صفر و ۵ ولت) این عدد را مشاهده می‌کنید که قرار است از پایه MOSI میکروکنترلری که در حالت Master تنظیم شده است خارج گردد، همانند شکل زیر:



شکل ۱۱-۱۵

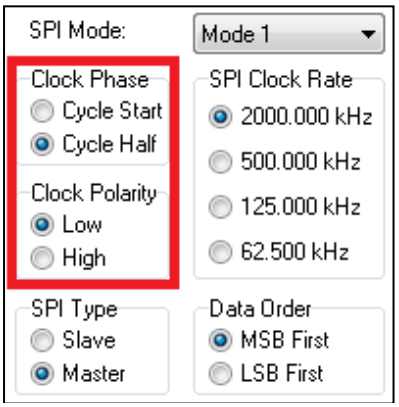
اکنون می‌خواهیم بفهمیم در تنظیم کردن Mode و Data order در Mode 0 و Msb First، اینکه کلاک مربوط به سنکرون‌سازی یا SCK هنگامی که بیکار است (یعنی اینکه انتقال اطلاعاتی صورت نمی‌گیرد) به صورت صفر یا Low می‌باشد و موقعیت لبه‌ی سیگنال SCK نسبت به هر بیت داده‌ی ارسالی در ابتدای سیکل یا Cycle Start می‌باشد به چه معنا می‌باشد؟

در شکل زیر مفهوم آنچه را که گفتیم مشاهده می‌کنید:



شکل ۱۱-۱۶ : ارتباط SPI در مُد صفر

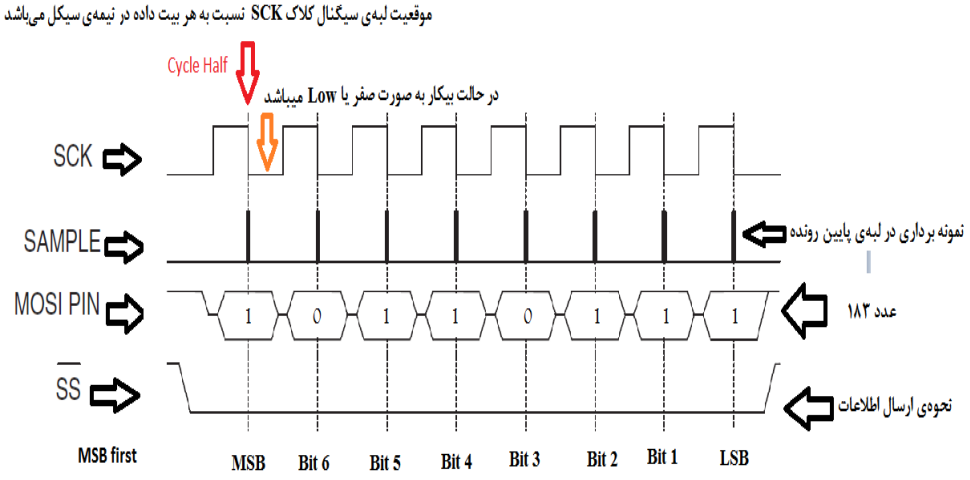
حال فرض کنید تنظیمات به صورت Mode1 و Msb First باشد دراین حالت نحوه‌ی تنظیمات Clock Phase و Clock Polarity به مانند شکل زیر می‌باشد:



شکل ۱۱-۱۷

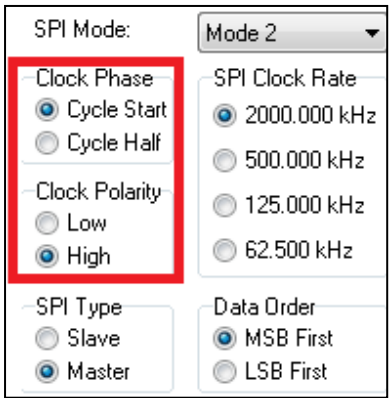
در این حالت کلاک مربوط به سنکرون‌سازی (SCK) هنگامی که در حالت بیکار می‌باشد یعنی اینکه انتقال اطلاعاتی صورت نمی‌گیرد به صورت صفر یا Low می‌باشد (تنظیمات مربوط به Clock Polarity) و موقعیت لبه‌ی سیگنال SCK نسبت به هر بیت داده‌ی ارسالی در نیمه‌ی سیکل یا Cycle Half می‌باشد (تنظیمات مربوط به Clock Phase).

در شکل زیر توضیحات گفته شده را مشاهده می‌کنید:



شکل ۱۱-۱۸ : ارتباط SPI در مُد ۱

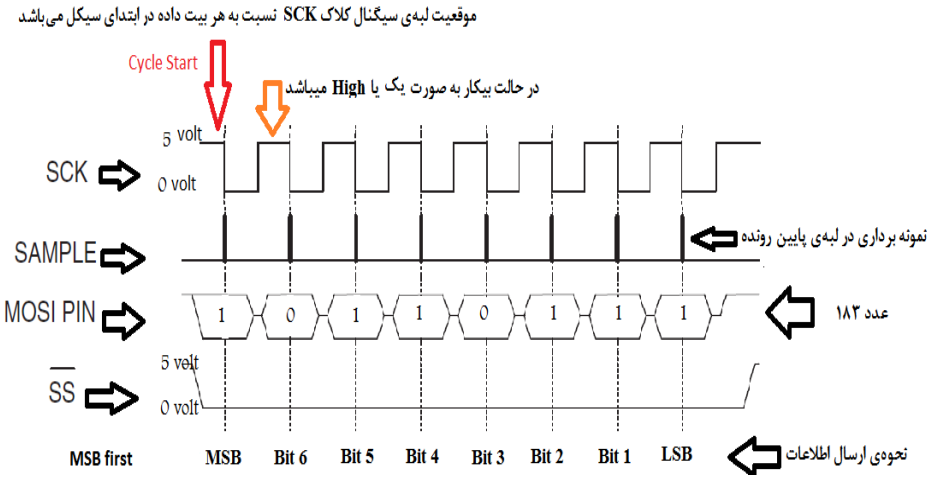
حال فرض کنید تنظیمات به صورت Mode2 و Msb First باشد دراین حالت نحوه‌ی تنظیمات Clock Phase و Clock Polarity مانند شکل زیر می‌باشد:



شکل ۱۱-۱۹

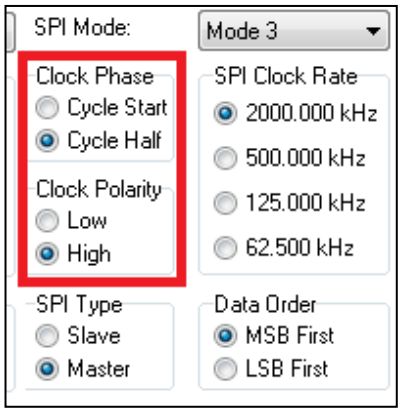
در این حالت کلاک مربوط به سنکرون‌سازی(SCK) هنگامی که در حالت بیکار می‌باشد به صورت یک یا High می‌باشد (تنظیمات مربوط به Clock Polarity) و موقعیت لبه‌ی سیگنال SCK نسبت به هر بیت داده‌ی ارسالی در ابتدای سیکل می‌باشد (تنظیمات مربوط به Clock Phase).

به شکل زیر دقت کنید:



شکل ۱۱-۲۰ : ارتباط SPI در مُد ۲

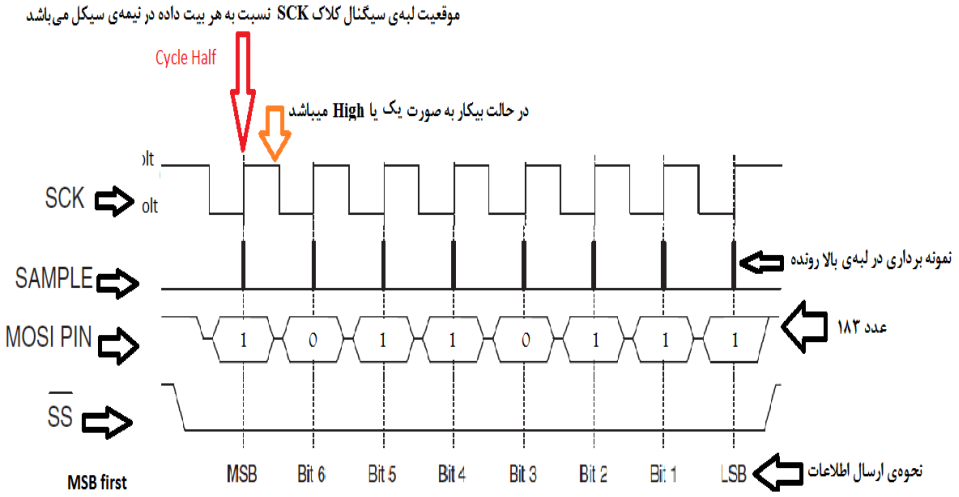
و در آخرین حالت از ۴ مُد گفته شده فرض کنید تنظیمات به صورت Mode3 و Msb First باشد دراین حالت نحوه‌ی تنظیمات Clock Phase و Clock Polarity مانند شکل زیر می‌باشد:



شکل ۱۱-۲۱

در این حالت کلاک مربوط به سنکرون‌سازی (SCK) هنگامی که در حالت بیکار می‌باشد به صورت یک یا High می‌باشد (تنظیمات مربوط به Clock Polarity) و موقعیت لبه‌ی سیگنال SCK نسبت به هر بیت داده‌ی ارسالی (هر بار ارسال اطلاعات ۱ بایت (8bits)داده جابجا می‌شود)در نیمه‌ی سیکل یا Cycle Half می‌باشد (تنظیمات مربوط به Clock Phase).

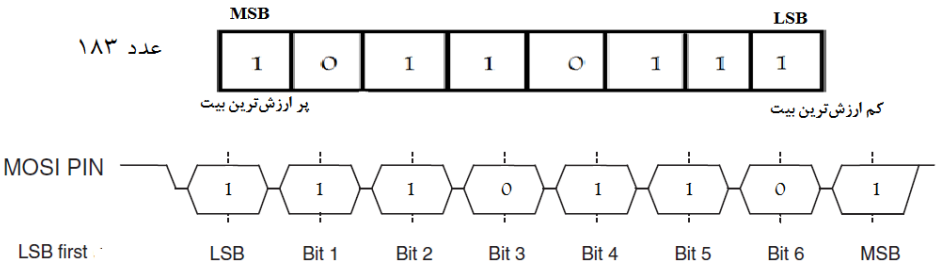
به شکل زیر توجه کنید:



شکل ۱۱-۲۲ : ارتباط SPI در مُد ۳

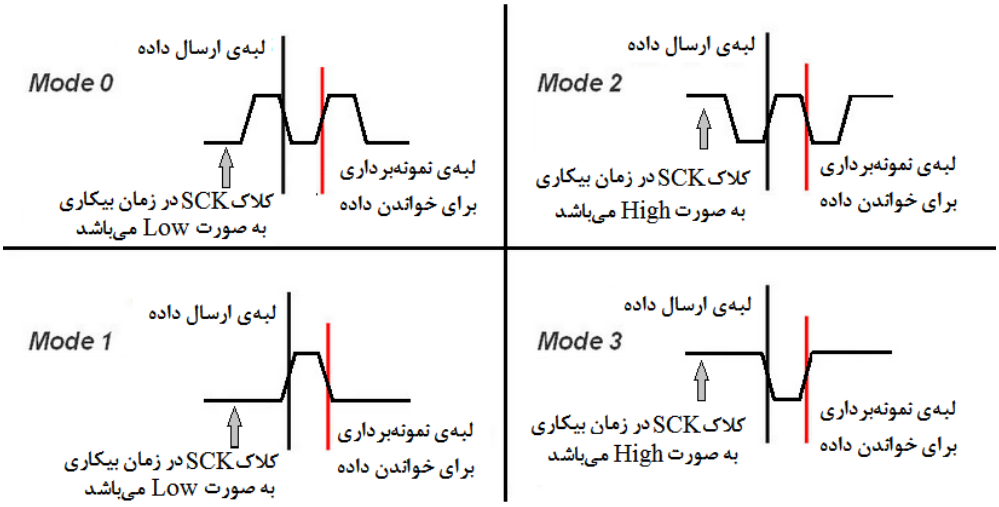
تنها یک نکته از توضیحات قسمت‌های بالا باقی می‌ماند اینکه اگر در هر ۴ شکل به جای اینکه تنظیمات Data Order به صورت MSB First باشد به صورت LSB First می‌بود نحوه‌ی ارسال اطلاعات چگونه می شد؟

پاسخ: ارسال اطلاعات مانند شکل زیر انجام می‌گرفت:



شکل ۱۱-۲۳ : فرمت ارسال اطلاعات در LSB First

خلاصه: اکنون آنچه از ۴ مُد ارتباط و زمان ارسال و دریافت داده با توجه به کلاک SCK در ارتباط SPI را خواندیم در شکل زیر خلاصه می‌کنیم:



شکل ۱۱-ب : خلاصه ۴ مُد ارتباط SPI

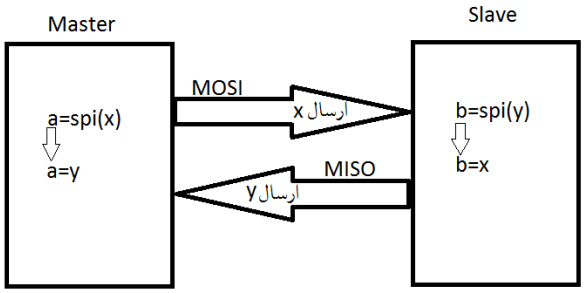
تابع کار با SPI در کدویژن

قبل از هر چیز باید تابعی که در کدویژن برای برقراری این ارتباط می‌باشد و ارسال Data را انجام می‌دهد، معرفی کنیم:

Spi(unsigned char data)

این تابع یک بایت داده (۸ بیت) را می‌فرستد و همزمان یک بایت (۸ بیت) داده دریافت می‌کند.
یعنی برای مثال در برنامه‌ی خود با کدویژن یک متغیر صحیح مانند a را تعریف کنیم و در میکروکنترلر Master بنویسیم a=spi(x) با این دستور عدد دلخواه x (که x می‌تواند عددی بین صفر تا ۲۵۵ باشد)توسطMaster و از طریق پایه‌ی MOSI ارسال می‌گردد و هر داده‌ای که از Slave توسط پایه‌ی MISO ارسال می‌گردد درون متغیر a ریخته می‌شود، به همین ترتیب اگر این دستور (a=spi(x)) برای میکروکنترلر Slave نوشته شود عدد x توسط Slave و از طریق

پایه‌ی MISO ارسال می‌گردد و هر داده‌ای که از Master توسط پایه‌ی MOSI ارسال می‌گردد درون متغیر a ریخته می‌شود ، برای اینکه بیشتر متوجه شوید به تصویر زیر دقت کنید:

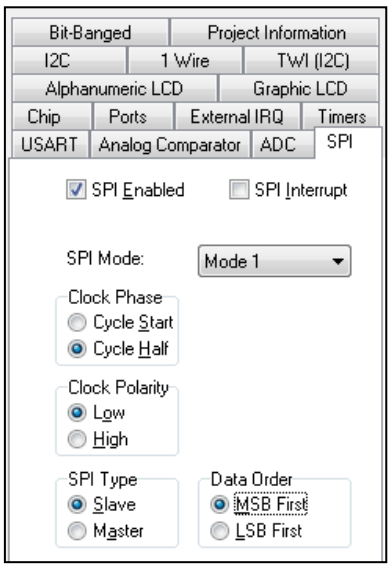


شکل ۱۱-ج : فرمت تابع **Spi()** در کدویژن

همانطور که در شکل مشخص است با نوشتن دستور a=spi(x) در میکروکنترلر Master داده‌ی دلخواه x توسط پایه‌ی MOSI به Slave ارسال می‌گردد و درون متغیر a داده‌ای که توسط پایه‌ی MISO از Slave ارسال شده قرار می‌گیرد و به همین ترتیب با نوشتن دستور b=spi(y) در میکروکنترلر Slave، داده‌ی دلخواه y توسط پایه‌ی MISO به Master ارسال می‌گردد و درون متغیر b داده‌ای که توسط پایه‌ی MOSI از Master ارسال شده قرار می‌گیرد.

تا اینجای کار تقریباً با مفهوم SPI آشنا شدید با استفاده از چند مثال تسلط شما در این بخش کامل می‌شود.

مثال۱: می‌خواهیم برنامه‌ای بنویسیم که در آن یک میکروکنترلر به صورت Master و نحوه‌ی ارسال اطلاعات به صورت Mode1 باشد و این میکروکنترلر عدد ۱۸۳ را بفرستد:



هدف: می‌خواهیم نحوه‌ی نمایشی را که در مُد ۱ توضیح داده شد را با استفاده از اسیلوسکوپ نمایش دهیم.

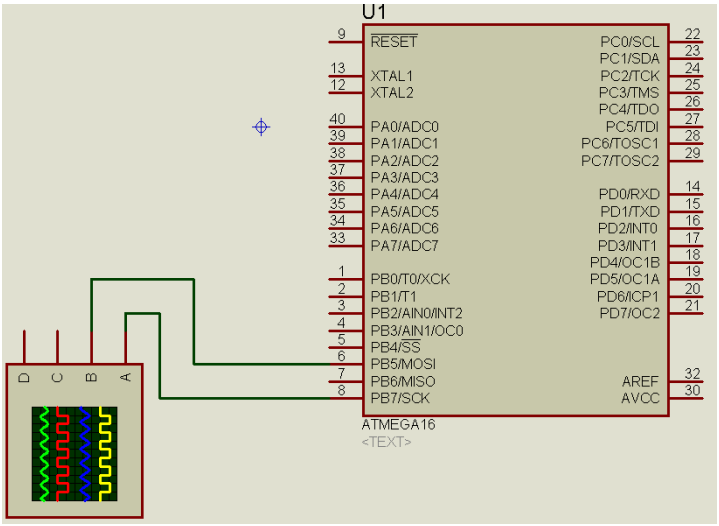
ابتدا تنظیمات کدویزارد را مطابق شکل روبرو انجام می‌دهیم:

حال باید کدی بنویسیم که با استفاده از تابع Spi عدد ۱۸۳ را روی باس Spi قرار دهد (یعنی داده را آماده‌ی ارسال از طریق ارتباط Spi کند):

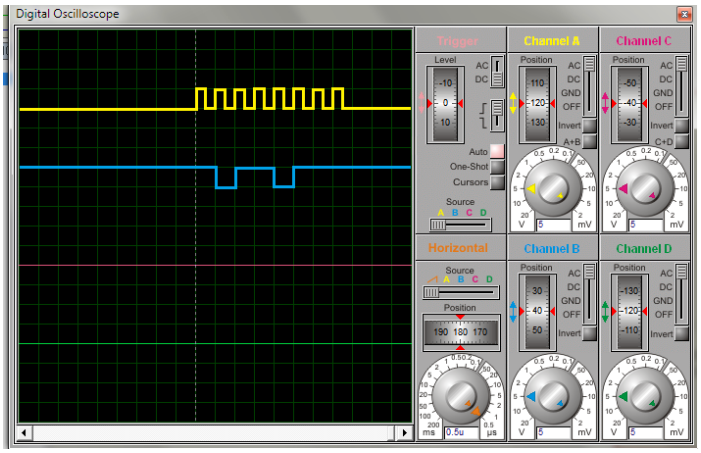
```
while (1)
      {
      spi(183);
      delay_ms(10);
      }
```

از یک تاخیر ۱۰ میلی‌ثانیه‌ای هم استفاده می‌کنیم که بتوانیم شکل موج را به خوبی بر روی اسیلوسکوپ نمایش دهیم.

می‌توانیم این مدار را در پروتئوس مانند شکل زیر شبیه‌سازی کنیم:

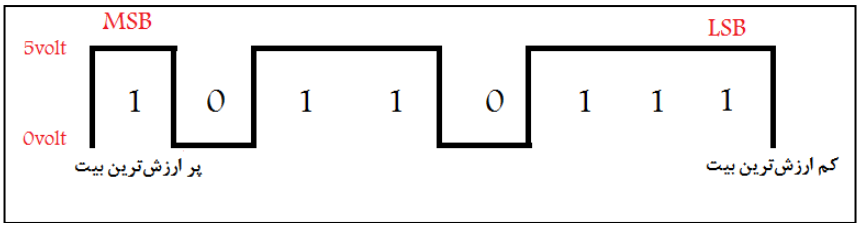


شکل ۱۱-۲۴ : مدار مثال ۱ در پروتئوس



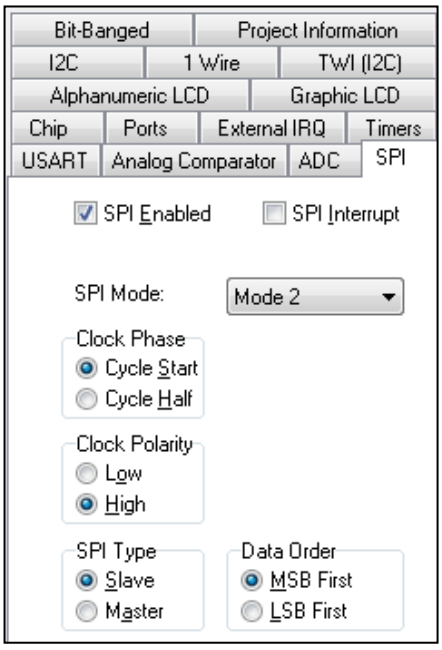
شکل ۱۱-۲۵ : شکل موج پایه‌های SPI

شکل موج قبلی مربوط به پایه‌ی SCK می‌باشد که به ازای هر بیت جابجا شده یک کلاک خورده است(در مجموع ۸ کلاک) و شکل موج پایینی مربوط به پایه‌ی MOSI می‌باشد(عدد ۱۸۳) و همانطور که مشاهده می‌کنیم همان شکل موجی است که کمی بالاتر توضیح داده شد:



شکل ۱۱-۲۶

مثال۲: می‌خواهیم برنامه‌ای بنویسیم که در آن یک میکروکنترلر به صورت Master باشد، نحوه‌ی ارسال اطلاعات به صورت Mode2 باشد و میکروکنترلر عدد ۳۴ را بفرستد:

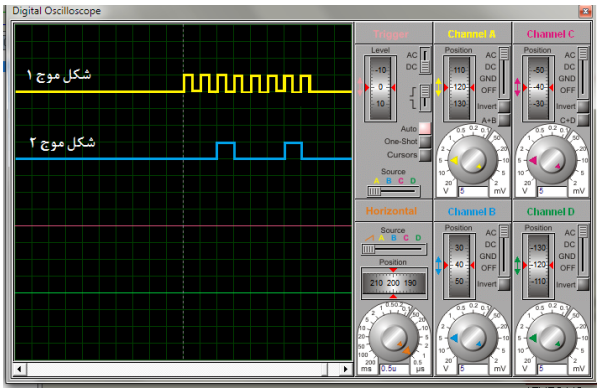


هدف: می‌خواهیم نحوه‌ی نمایشی را که در مُد ۲ گفته شد را با استفاده از اسیلوسکوپ نمایش دهیم. ابتدا تنظیمات کدویزارد را مانند شکل روبرو انجام می‌دهیم:

کد زیر را درون حلقه‌ی while(1) می‌نویسیم:

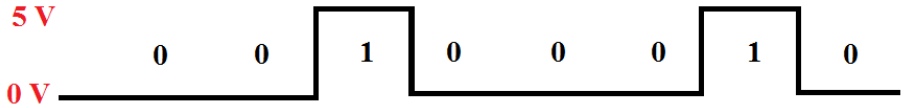
```
while (1)
      {
      spi(34);
      delay_ms(10);
      }
```

شکل موج پایه‌ها در بخش شبیه‌سازی مطابق زیر می‌باشد:



شکل ۱۱-۲۷ : شکل موج پایه‌های SPI در مثال ۲

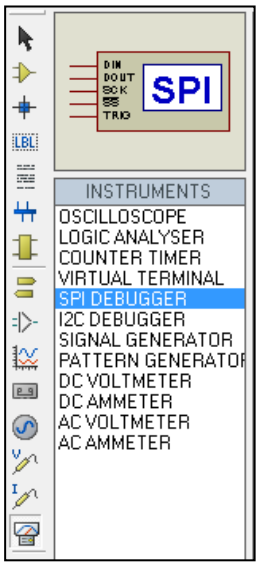
همانطور که در شکل بالا مشاهده می‌کنید شکل موج ۲ مربوط به داده‌ی ۳۴ می‌باشد که در مبنای دو به صورت زیر می‌شود که با شکل موج ۲ یکسان است:



عدد ۳۴

شکل ۱۱-۲۸

البته دستگاهی در پروتئوس تحت عنوان Spi Debugger وجود دارد که در بخش Instruments ()موجود می‌باشد و می‌توان اطلاعات ارسالی توسط ارتباط Spi را بوسیله‌ی آن مشاهده کرد، این وسیله را مطابق شکل روبرومی‌توانید پیدا کنید:



در شکل زیر این دستگاه را مشاهده می‌کنید:

پایه‌ی DIN: نام این پایه مخفف Data Input می‌باشد و همانطور که از نام آن مشخص است بوسیله‌ی این پایه داده‌ی وارد شده به Spi Debugger دریافت می‌گردد. برای مثال اگر بخواهیم داده‌ی ارسالی توسط یک میکروکنترلر Master را بخوانیم پایه‌ی MOSI میکروکنترلر را به پایه‌ی DIN دستگاه متصل می‌کنیم.

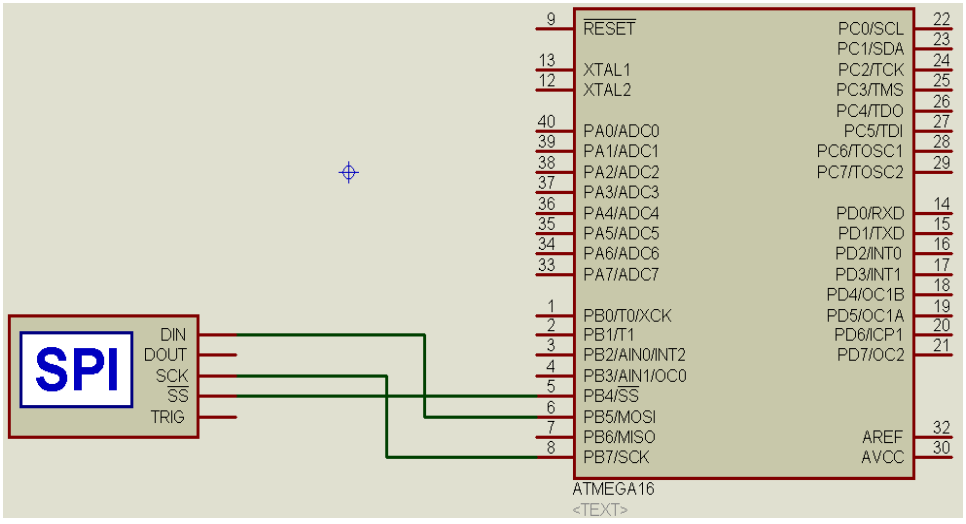
پایه‌ی DOUT: نام این پایه مخفف Data Output می‌باشد و همانطور که از نام آن مشخص است بوسیله‌ی این پایه داده‌ای از Spi Debugger ارسال می‌گردد.

پایه SCK: مربوط به کلاک سنکرون‌سازی ارتباط SPI می‌باشد. اگر SPI Debugger به صورت Slave باشد این پایه به صورت ورودی و اگر به صورت Master باشد به صورت خروجی و تولید کننده‌ی فرکانس سنکرون‌ساز می‌باشد.

پایه SS: همان پایه‌ی Slave Select می‌باشد.

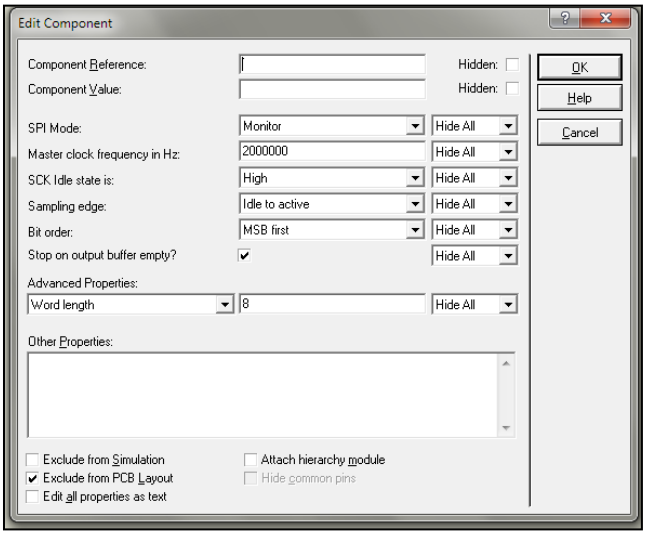
پایه TRIG: مربوط تریگر کردن داده‌های SPI می‌باشد.

حال برای اینکه داده توسط SPI Debugger را در این مثال مشاهده کنیم ابتدا مدار را مطابق شکل زیر می‌بندیم:



شکل ۱۱-۲۹

بر روی Spi Debugger یک دبل کلیک می‌کنیم و صفحه‌ی زیر باز می‌گردد و تنظیمات زیر را انجام می‌دهیم:



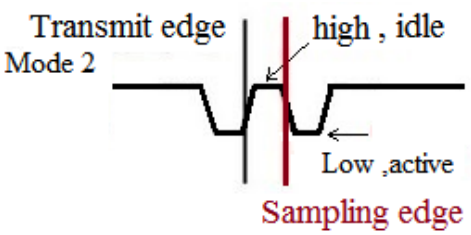
شکل ۱۱- د : تنظیمات SPI Debugger

بخش SPI-Mode: در این بخش می‌توان تعیین کرد که Spi Debugger به صورت Master ، Slave و یا یک Monitor رفتار کند که در صورت انتخاب به صورت مانیتور تمامی اطلاعات رد و بدل شده را نمایش می‌دهد.در این مثال Spi Debugger را به صورت Monitor تنظیم می‌کنیم.

بخش Master Clock frequency in Hz: در این بخش فرکانس سنکرون‌سازی ارتباط را مشخص می‌کنیم که در این مثال ۲ مگاهرتز می‌باشد(در کدویزارد این فرکانس کاری SPI تنظیم شده بود).

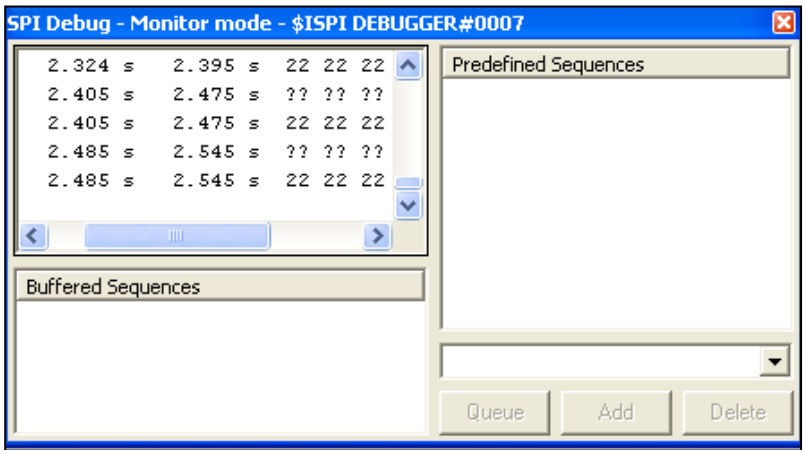
بخش SCK Idle State is: در این بخش وضعیت کلاک سنکرون‌ساز ارتباط را در حالت بیکاری مشخص می‌کنیم که همانطور که گفته شد در Mode2 به صورت High می‌باشد.

بخش Sampling edge: این بخش لبه‌ی نمونه‌برداری را مشخص می‌کند که در مثال ۲ به صورت Mode2 می‌باشد و همانطور که گفته شد در Mode2 نمونه‌برداری در لبه‌ی پایین رونده صورت می‌گیرد. همانند شکل روبرو به صورت Idle to active بایستی تنظیم گردد:



بخش Bit-Order: در این بخش ترتیب ارسال داده مشخص می‌گردد که در مثال ۲ به صورت MSB first تنظیم شده است.

حال مشاهده‌ی اطلاعات مثال۲ (ارسال عدد ۳۴) از طریق Spi Debugger:



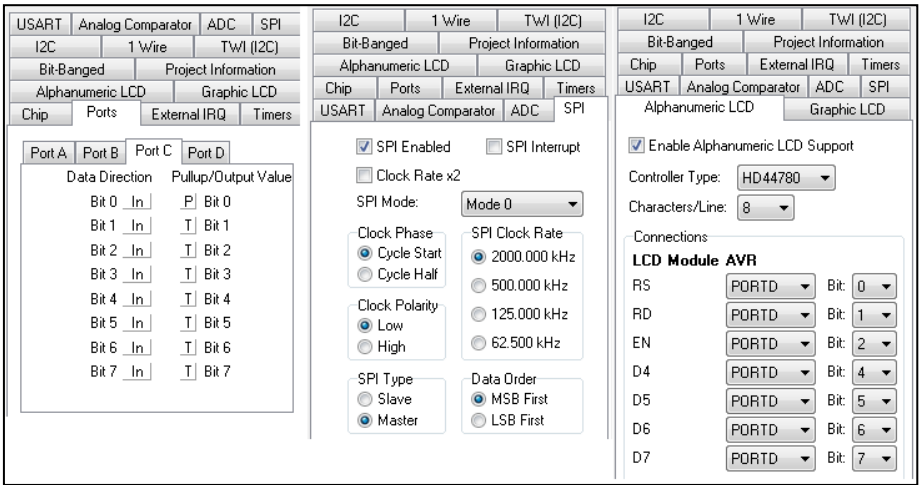
شکل ۱۱-۳۰ : نمایش عدد ۳۴ توسط SPI Debugger در مبنای ۱۶

همانطور که مشاهده می‌کنید داده‌ای که در حال دریافت می‌باشد به صورت ؟؟ مشخص شده زیرا هیچ داده‌ی مشخصی برای دریافت توسط میکروکنترلر Master وجود ندارد و داده‌ی ارسال شده از پایه‌ی MOSI توسط میکروکنترلر به صورت ۲۲ مشخص شده برابر عدد ۳۴ در مبنای هگزادسیمال می‌باشد.

نکته:در Spi Debugger اعداد را در مبنای ۱۶ (Hex) نمایش می‌دهد.

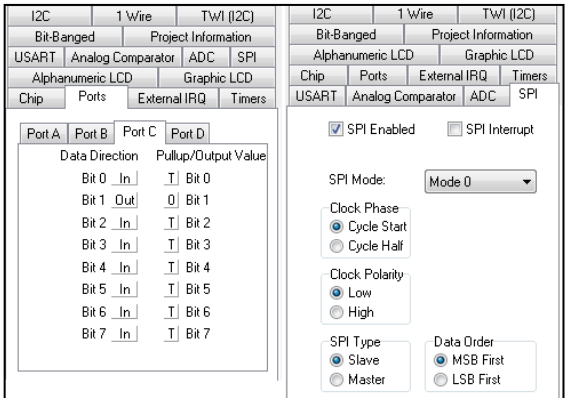
مثال۳: می‌خواهیم برنامه‌ای بنویسیم که در آن ۲ میکروکنترلر یکی به صورت Master و دیگری به صورت Slave کار می‌کند. زمانیکه پایه C0 میکروکنترلر Master که به صورت ورودی و Pull-up تنظیم شده است زمین شد میکروکنترلر Master عدد ۳۵ را روی پایه‌ی MOSI باس ارتباطی Spi قرار دهد و زمانیکه میکروکنترلر Slave عدد ۳۵ را دریافت کرد ،ال ای دی متصل به پایه C1 خود را روشن کند و میکروکنترلر slave عدد ۱۸ را برای میکروکنترلر Master بفرستد و تا قبل از دریافت عدد ۳۵ میکروکنترلر Slave عدد صفر را برای Master ارسال کند. در نهایت میکروکنترلر Master عدد ارسال شده توسط میکروکنترلر Slave را بر روی LCD که به پورت D آن متصل است نمایش دهد:

تنظیمات کدویزارد میکروکنترلرMaster مانند شکل زیر انجام می‌گیرد:



شکل ۱۱-۳۱ : تنظیمات کدویزارد میکروکنترلر Master درمثال۳

و تنظیمات کدویزارد میکروکنترلر Slave:



شکل ۱۱-۳۲ : تنظیمات کدویزارد میکروکنترلر Slave در مثال۳

کد میکروکنترلرMaster: ابتدا متغیر صحیح a و رشته‌ی r را تعریف می‌کنیم:

```
#include <mega16.h>
// Alphanumeric LCD functions
#include <alcd.h>
#include <delay.h>
#include <stdio.h>
// SPI functions
#include <spi.h>
// Declare your global variables here
int a;
char r[12];
void main(void)
{
// Declare your local variables here
```

سپس کد زیر را در حلقه‌ی While(1) برای ارسال داده‌ها در میکروکنترلر Master را می‌نویسیم:

```
while (1)
      {
      if(PINC.0==0)
      a=spi(35);
      lcd_clear();
      sprintf(r,"%d",a);
      lcd_puts(r);
      }
```

در کد فوق با دستور شرطی if زمانیکه پین C.0 که به صورت Pull-Up نیز تنظیم شده است، زمین شود دستور a=spi(35) اجرا می‌گردد، همانگونه که در بالاتر نیز آموختیم با دستور a=spi(35)، میکروکنترلر Master عدد ۳۵ را بر روی باس SPI قرار می‌دهد و یا به عبارتی دیگر فرمانی از طریق پایه‌ی MOSI به Slave صادر می‌کند و همزمان درون متغیر a داده‌ای را از Slave بر روی خط MISO دریافت می‌کند و می‌خواند و همانطور که در کد مشاهده می‌کنید با دستور Sprintf() این داده‌ی دریافتی از Slave توسط میکروکنترلر Master بر روی LCD نمایش داده می‌شود.

کد میکروکنترلر Slave: ابتدا متغیر صحیح b را تعریف می‌کنیم:

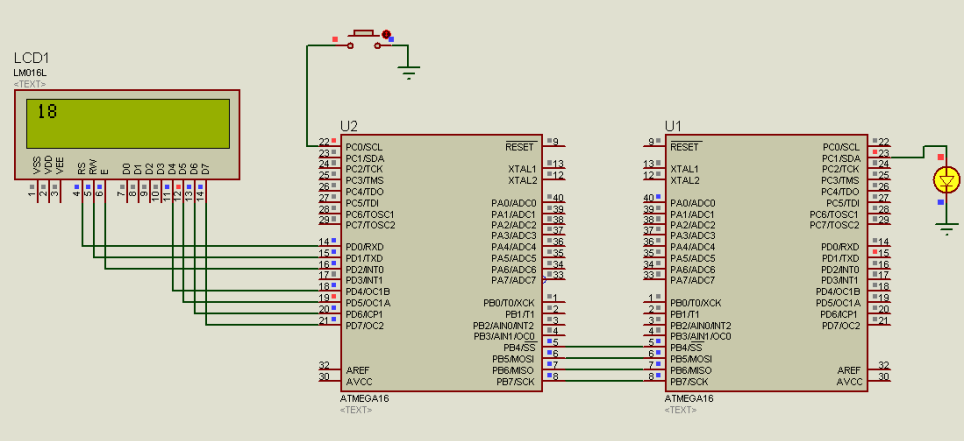
```
#include <mega16.h>
#include <stdio.h>
// SPI functions
#include <spi.h>
// Declare your global variables here
int b;
void main(void)
{
// Declare your local variables here
```

سپس درون حلقه‌ی while(1) کد زیر را می‌نویسیم:

```
while (1)
      {
      b=spi(0);
      if(b==35)
      {
      PORTC.1=1;
      spi(18);
      }
      }
```

همانطور که در کد قبل مشاهده می‌کنید ابتدا دستور b=spi(0) نوشته شده است. با این دستور داده‌ای که از Master به سمت Slave توسط پایه‌ی MOSI صادر می‌شود، درون متغیر صحیح b ذخیره می‌گردد و میکروکنترلر Slave عدد صفر را به سمت میکروکنترلر Master از طریق پایه‌ی MISO ارسال می‌کند. زمانیکه Slave عدد ۳۵ را دریافت کرد دستور عبارت شرطی if اجرا می‌گردد و PORTC.1 برابر یک می‌گردد و Slave عدد ۱۸ را برای Master ارسال می‌کند.

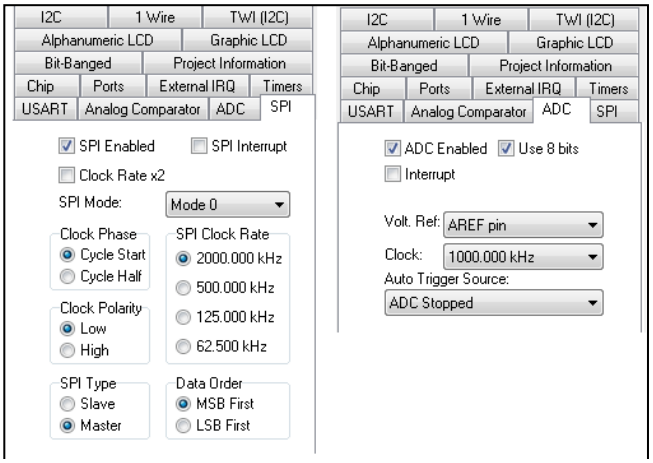
و در نهایت مدار بسته شده در شبیه‌سازی پروتئوس:



شکل ۱۱-۳۳ : مدار بسته شده‌ی مثال ۳

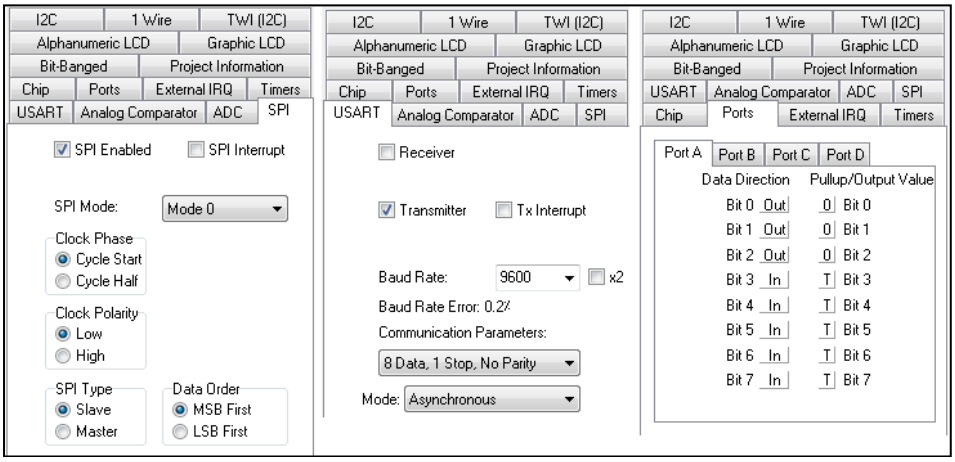
مثال ۴: به عنوان آخرین مثال از این بخش قصد داریم برنامه‌ای بنویسیم که در آن دو میکروکنترلر با یکدیگر بوسیله‌ی ارتباط SPI متصل شده‌اند و میکروکنترلر Master ولتاژ پایه‌ی A0 خود را توسط واحد ADC خود (که به صورت ۸ بیتی تنظیم شده است) می‌خواند و آن را به میکروکنترلر Slave می‌دهد و این میکروکنترلر ولتاژ را بوسیله‌ی ۳ LED که به پایه‌های A0 تا A2 متصل است ناحیه‌بندی می‌کند بدین ترتیب که اگر ولتاژ پایه‌ی A0 میکروی Master کمتر از ۲ ولت باشد LED آبی پایه‌ی A0 را روشن می‌کند و اگر بین ۲ تا ۴ ولت بود LED زرد پایه‌ی A1 را روشن می‌کند و اگر بزرگتر از ۴ ولت باشد LED قرمز پایه‌ی A2 را روشن می‌کند و علاوه بر این‌ها ولتاژ خوانده شده از میکروکنترلر Master توسط میکروکنترلر Slave با ارتباط USART ارسال می‌شود:

تنظیمات میکروکنترلر Master:



شکل ۱۱-۳۴ : تنظیمات میکروکنترلر Master مثال۴

تنظیمات میکروکنترلر Slave:



شکل ۱۱-۳۵ : تنظیمات میکروکنترلر Slave مثال۴

کد زیر را برای میکروکنترلر Master مـی‌نویسـیم کـه بـا دسـتور (spi(read_adc(0)) ، مقـدار ADC خوانده شده از پایه‌ی A0 میکروکنترلر Master به میکروکنترلر Slave ارسال می‌گردد:

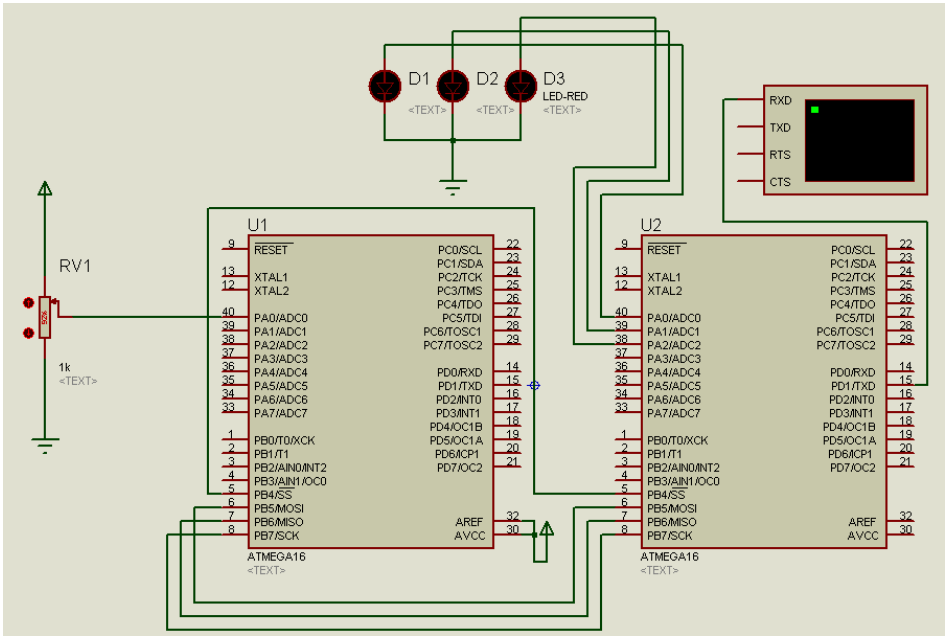
```
while (1)
      {

       spi(read_adc(0));
      }
```

کد روبرو را برای میکروکنترلر slave می‌نویسیم:

```
while (1)
      {
      a=spi(0);
      printf("X=%d",a);
      if(a<102){
      PORTA.0=1;
      PORTA.1=0;
      PORTA.2=0;
      }

      if(a>=102 && a<204){
      PORTA.0=0;
      PORTA.1=1;
      PORTA.2=0;
      }

       if(a>=204){
      PORTA.0=0;
      PORTA.1=0;
      PORTA.2=1;
      }
      }
```

در کد روبرو همانطور که مشاهده می‌کنید متغیر صحیح a توسط مقداری که از میکروکنترلر Master به Slave ارسال می‌گردد پر می‌گردد که همان مقدار ADC خوانده شده از پایه‌ی A.0 میکروکنترلر Master می‌باشد. دستور spi(0) هم عدد صفر را برای میکروکنترلر Master ارسال می‌کند که در این مثال مقدار این عدد می‌تواند دلخواه باشد و در طراحی ما تاثیری ندارد.

حال مدار زیر را در پروتئوس می‌بندیم (از یک پتانسیومتر برای تغییر ولتاژ پایه‌یA0 استفاده شده است. این قطعه را با نام Pot-Hg در پروتئوس بیابید):



شکل ۱۱-۳۶ : مدار بسته شده در مثال۴

فصل دوازدهم

ارتباط I2C و کار با eeprom

در این فصل خواهیم خواند:

۱. آشنایی مقدماتی با ارتباط I2C
۲. مشخصات کلی ارتباط I2C
۳. حالت شروع در ارسال داده
۴. حالت توقف در ارسال داده
۵. شکل بسته‌ی آدرس
۶. شکل بسته‌ی داده
۷. تنظیمات I2C در کدویزارد
۸. توابع کار با I2C در کدویژن
۹. eepromهای خانواده‌ی AT24C
۱۰. ویژگی‌های خانواده‌ی AT24C
۱۱. حافظه‌های ۲ و ۱۰۲۴ کیلوبیتی این خانواده
۱۲. تابع برای خواندن از حافظه در کدویژن
۱۳. تابع برای نوشتن بر روی حافظه در کدویژن
۱۴. آشنایی با EEPROM داخلی ATmega16
۱۵. خواندن و نوشتن بر روی EEPROM داخلی ATmega16

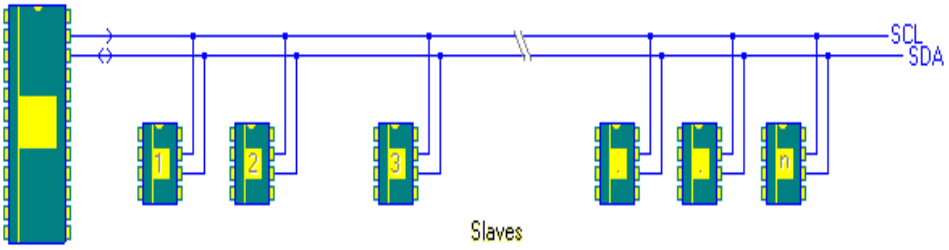
ارتباط I2C و کار با حافظه‌های eeprom

در این فصل قصد داریم با پروتکل ارتباطی I2C و همچنین نحوه‌ی استفاده از آن برای برقراری ارتباط با حافظه‌های eeprom آشنا شویم.
ارتباط I2C یک ارتباط دوسیمه است و زیرمجموعه‌ی ارتباط‌های سریال می‌باشد که توانایی ارسال و دریافت اطلاعات به صورت سریال از طریق دو سیم را دارا می‌باشد. پروتکل I2C برای اولین بار در سال ۱۹۸۰ توسط شرکت Philips ابداع گشت و هدف اولیه‌ی آن برقراری ارتباطی ساده بین یک CPU با سایر وسایل الکتریکی(Device) در یک دستگاه تلویزیون بود.امروزه شرکت‌های بزرگ دنیا از این پروتکل بهره می‌برند و تمامی شرکت‌ها سعی بر سازگار کردن دستگاه‌های خود با این پروتکل ارتباطی دارند. واژه‌ی I2C مخفف **I**nter **I**C **B**us می‌باشد که به معنای خط ارتباط داخلی برای مدارات مجتمع می‌باشد.

از جمله مزایای ارتباط I2C می‌توان به موارد زیر اشاره کرد:
۱. پروتکل I2C هم به صورت نرم‌افزاری و هم سخت‌افزاری قابلیت کنترل دارد.
۲. وسایل (Device) مختلف به راحتی می‌توانند به این باس ارتباطی اضافه شوند.
۳. مصرف بسیار کم جریان(در حد نانو آمپر)
۴. حفاظت ارتباط در برابر انواع نویزها مثل نویزهای سوزنی
۵. کارکرد مناسب در طیف وسیعی از تغییرات دمایی
۶. قابلیت کارکرد در طیف وسیعی از ولتاژ تغذیه
۷. به علت اینکه این ارتباط به وسیله‌ی تنها دو سیم برقرار می‌شود در طراحی مدارات چاپی(PCB مخفف **P**rinted **C**ircuit **B**oard) حجم سیم‌ها به طور قابل توجهی کاهش می‌یابد و در تولیدات انبوه هزینه‌های تولید مدار کاهش می‌یابد.

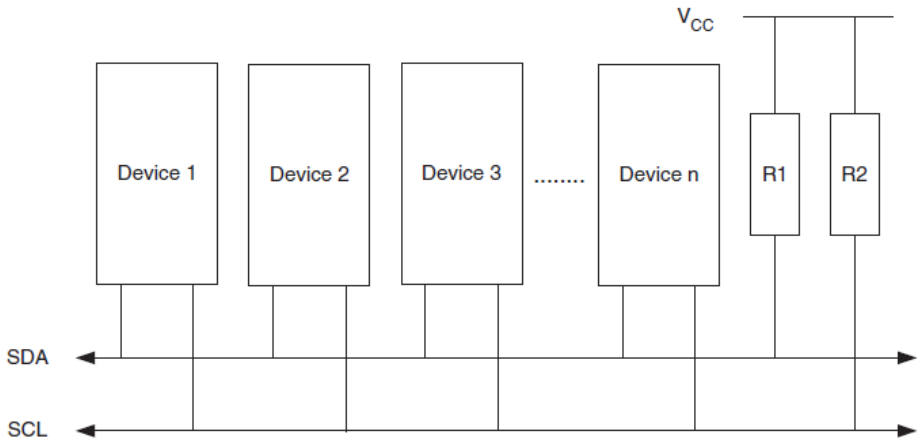
سوال:
حال ارتباط I2C چگونه با دو سیم برقرار می‌شود و کارکرد هر کدام از این سیم‌ها چیست؟
باس (Bus) ارتباطی I2C دارای دو خط فعال برای برقراری ارتباط به نام‌های SDA(Serial Data Line)و SCL(Serial Clock Line) می‌باشد که به وسیله‌ی خط SDA آدرس و داده منتقل می‌شوند و بوسیله‌ی خط SCL هم حالت‌های شروع ارسال یا دریافت، توقف ارسال یا

دریافت و همچنین حالت شروع مجدد مشخص می‌شود. در شکل زیر نمایشی از ارتباط چندین وسیله با یکدیگر را با پروتکل I2C مشاهده می‌کنید:



شکل ۱۲-۱ : برقراری ارتباط وسایل گوناگون با یکدیگر از طریق پروتکل I2C

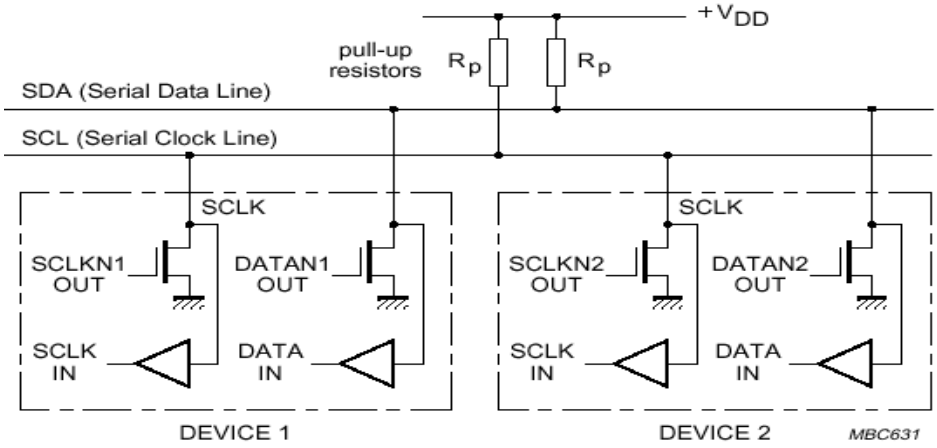
عمدتاً پروتکل ارتباطی I2C این امکان را به کاربران میکروکنترلرها می‌دهد که تا ۱۲۸ وسیله‌ی مختلف جانبی را به وسیله‌ی تنها دو سیم (SDA و SCL) به میکروکنترلر مرتبط سازد. تنها سخت‌افزار خارجی مورد نیاز در این ارتباط یک مقاومت Pull-up برای هر یک از خطوط ارتباطی است که مدار پیشنهادی شکل ۱۲-۲ در Datasheet میکروکنترلر (ATmega16) آمده است:



شکل ۱۲-۲ : مدار پیشنهادی برای ارتباط I2C

مقاومت‌های R2،R1 مقاومت‌های Pull-Up می‌باشند. در این حالت وقتی فرمانی روی خط نباشد خط از لحاظ منطقی High (یک) می‌باشد.

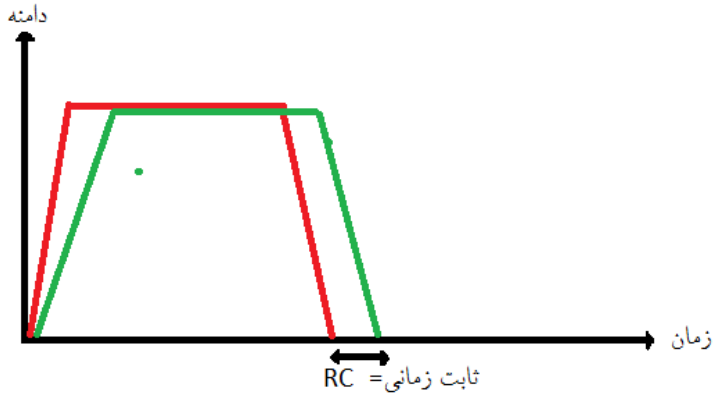
در تمامی وسایلی که از پروتکل ارتباط دوسیمه حمایت می‌کنند برای جلوگیری از تاثیر نامطلوب قطعات معیوب بر روی باس ارتباطی از دو تکنیک Open-drain-output یا Open-collector-output استفاده می‌کنند، مطابق شکل زیر:



شکل ۱۲-۳ : تکنیک Open-drain-output و Open-collector-output

این تکنیک موجب می‌شود که وقتی یک وسیله خراب شد به طور کامل از خط ارتباطی خارج شود و هیچ اثری بر روی باس(Bus) ارتباطی نگذارد.

البته نکته‌ی منفی این تکنیک این می‌باشد که در خطوط طولانی ارتباطی موجب می‌شود که یک ظرفیت خازنی در خط به وجود آید که این ظرفیت خازنی با مقاومت‌های Pull-Up ،یک ثابت‌زمانی(RC) ایجاد می‌کنند که موجب می‌شود سیگنال‌های ارسالی روی خط دچار تاخیر و اعوجاج شوند.



شکل ۱۲-۴ : ایجاد تاخیر در سیگنال

مشخصات کلی ارتباط I2C

این ارتباط دارای ویژگی‌های فنی زیر می‌باشد:

۱. این ارتباط می‌تواند تا فرکانس ۳,۴ Mbit/s تبادل اطلاعات کند.

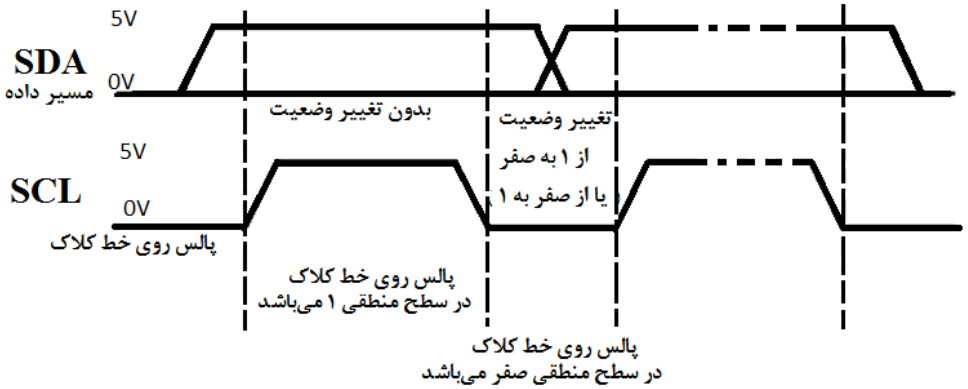
۲. رنج تغذیه بسته به نوع وسایل جانبی مرتبط می‌تواند از ۲,۵ ولت تا ۵,۵ ولت باشد.

۳. رنج دمایی برای کارکرد مناسب بسته به شرایط ساخت می‌تواند از ۴۰ تا ۸۵ درجه سانتیگراد و بعضاً از ۰ تا ۱۲۰ درجه سانتیگراد می‌تواند تغییر کند.

حال بایستی ببینیم که فرمت انتقال اطلاعات با دو سیم در این پروتکل چگونه صورت می‌گیرد.

سطح منطقی بستگی به Vcc دارد ولی در کار با میکروکنترلرهای AVR سطح منطقی ۱ را ۵ ولت و سطح منطقی صفر را صفر ولت در نظر می‌گیریم(هر بیت می‌تواند ۰ یا ۱ باشد و با توجه به رنج منبع تغذیه تعیین می‌گردد که فرض ما در این کتاب بر منبع تغذیه ۵ ولت می‌باشد).

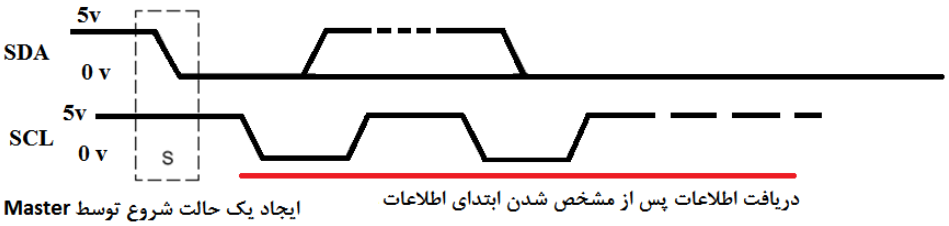
در پروتکل I2C ارسال اطلاعات به صورت بیت به بیت می‌باشد و همراه هر بیت داده‌ای که انتقال پیدا می‌کند یک پالس در خط SCL آن را همراهی می‌کند، هر زمان بیت‌های منتقل شده در مسیر انتقال داده(SDA) تغییر وضعیت دهند(یعنی از ۱ به ۰ یا از ۰ به ۱ برود) بایستی مسیر SCL در سطح منطقی صفر باشد و وقتی تغییر رخ نمی‌دهد کلاک موجود در خط SCL باید در سطح منطقی یک باشد مطابق شکل زیر:

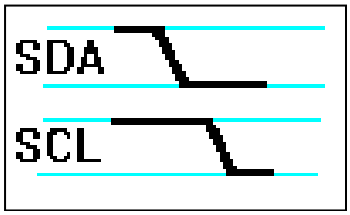


شکل ۱۲- ۵ : فرمت ارسال اطلاعات با ۲ سیم

حالت شروع یا Start در ارسال داده

برای اینکه فرستنده‌ی اطلاعات (Master) گیرنده‌ی اطلاعات (Slave) را متوجه سازد که آغاز بسته‌ی اطلاعاتی از کجا می‌باشد باید Master یک حالت شروع (Start) را ایجاد کند. این حالت بدین‌ترتیب صورت می‌گیرد که یک تغییر وضعیت از سطح منطقی ۱(High) به سطح منطقی صفر(Low) بر روی خط SDA انجام گیرد و اگر در این حالت خط SCL در سطح منطقی ۱(High) باشد،گیرنده (Slave) متوجه می‌شود که فرستنده (Master) یک حالت شروع یا Start را ایجاد کرده است و آماده‌ی دریافت اطلاعات بعد از آن باشد. شکل حالت Start را مطابق شکل زیرمشاهده می‌کنید:



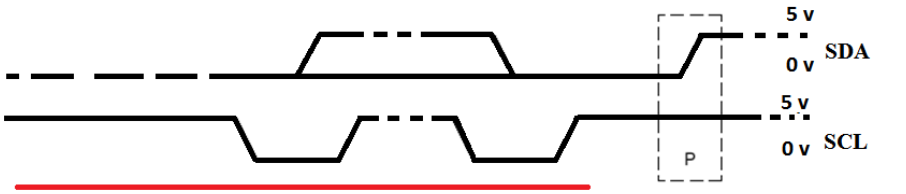


شکل ۱۲-۶ : حالت شروع

حالت توقف یا Stop در ارسال داده

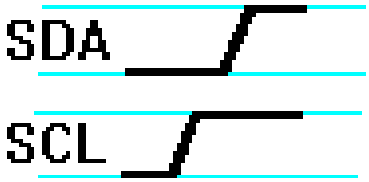
برای اینکه فرستنده (Master) گیرنده (Slave) را متوجه سازد که پایان بسته‌ی اطلاعاتی کجاست باید Master یک حالت توقف (Stop) را ایجاد کند. این حالت بدین‌ترتیب صورت می‌گیرد که یک تغییر وضعیت از سطح منطقی صفر(Low) به سطح منطقی یک (High) بر روی خط SDA انجام گیرد و خط SCL هم در سطح منطقی ۱(High) باشد، در این حالت گیرنده

(Slave) متوجه می‌شود که فرستنده (Master) یک حالت توقف یا Stop را ایجاد کرده است و دریافت اطلاعات پایان یافته است.



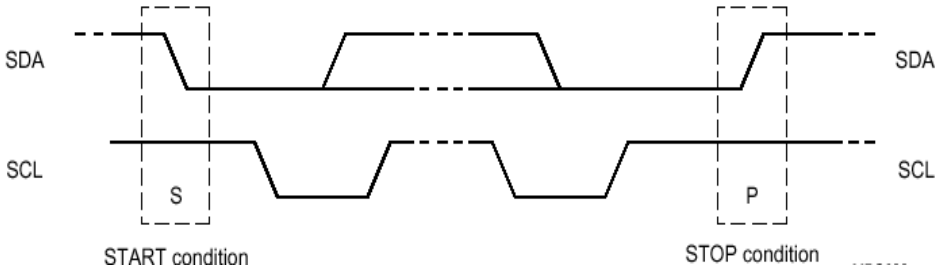
ایجاد یک حالت توقف توسط Master

دریافت اطلاعات پس از مشخص شدن ابتدای اطلاعات



شکل ۱۲-۷ : حالت توقف

شکل کامل ارسال یک بسته‌ی اطلاعاتی توسط Master به Slave را در شکل زیر مشاهده می‌کنید:



شکل ۱۲-۸ : نحوه‌ی ارسال بسته‌ی اطلاعاتی

در ارتباط با وسایل جانبی که از پروتکل I2C حمایت می‌کنند نیاز به یک آدرس و داده می‌باشد، برای مثال وقتی یک میکروکنترلر را با حافظه‌ی eeprom از طریق ارتباط I2C مرتبط می‌سازیم زمانی که می‌خواهیم یک داده را روی حافظه بنویسیم باید مشخص کنیم که داده (داده‌های ۸ بیتی یا یک بایتی) را در کدام خانه‌ی حافظه بریزیم پس نیاز به آدرس می‌باشد. برای خواندن از حافظه نیز دقیقاً نیاز به این می‌باشد که از کدام خانه‌ی حافظه داده‌ی مورد نیازمان را برداریم، فقط

جهت ارسال داده فرق می‌کند (در نوشتن روی حافظه جهت انتقال داده از میکروکنترلر به eeprom و در خواندن از حافظه جهت انتقال داده از eeprom به میکروکنترلر می‌باشد)، البته در ادامه نحوه‌ی ارتباط با eepromهای سری at24c را به طور کامل توضیح خواهیم داد.

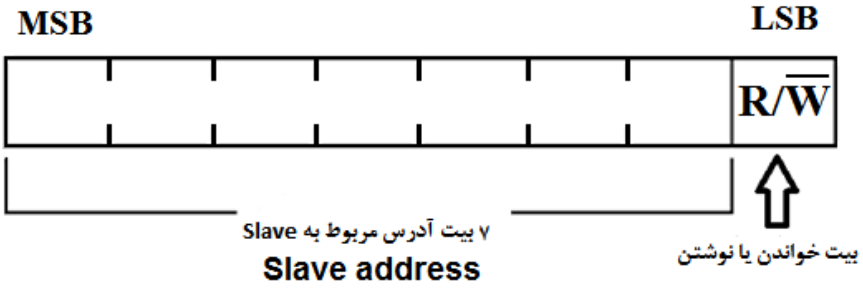
شکل بسته‌ی آدرس

بسته‌های آدرسی که از طریق دو سیم منتقل می‌شوند بدین صورت هستند که پس از ایجاد یک حالت شروع (Start) بایستی ۷ بیت آدرس را از طریق خط SDA منتقل کنند(۰ تا ۱۲۷) و سپس یک بیت مربوط به خواندن (Read) یا نوشتن (Write) را بفرستیم که اگر بیت خواندن یا نوشتن (R/W) صفر باشد یعنی در حالت نوشتن (Write) و وقتی یک باشد در حالت خواندن (Read) می‌باشیم.

بعد از ارسال این ۸ بیت، دریافت‌کننده باید با زمین کردن خود(سطح منطقی صفر) یعنی ایجاد یک بیت صفر به فرستنده بگوید که آدرس را دریافت کرده‌ام و آدرس‌دهی شده‌ام و باتوجه به بیت خواندن یا نوشتن (R/W) آماده‌ی ارسال اطلاعات به آدرس مورد نظر(بیت R/w صفر باشد) یا دریافت اطلاعات از آدرس مورد نظر (بیت R/w یک باشد) می‌باشم که به بیت نهم، بیت تصدیق یا بیت Acknowledge می‌گوییم که به اختصار آن را بیت Ack می‌گوییم.

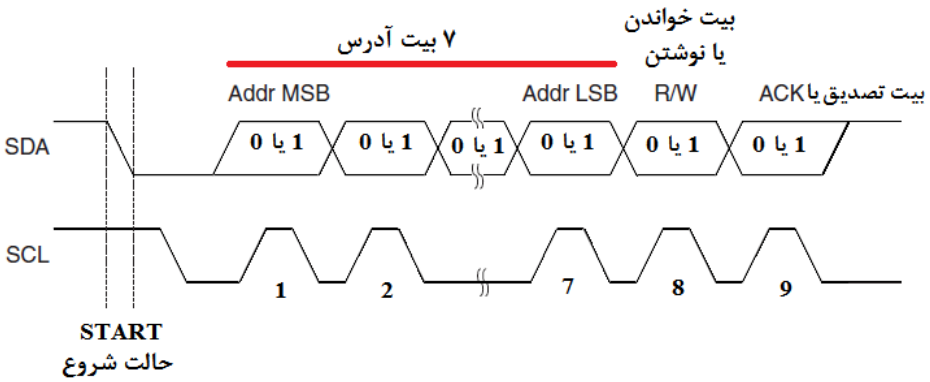
حال اگر Slave بیت تصدیق را ۱ کند یعنی اینکه آدرس‌دهی نشده است و آمادگی تبادل اطلاعات از آدرس مورد نظر را ندارد در این شرایط Master متوجه می‌شود که باید یک حالت شروع مجدد را ایجاد و دوباره Slave را آدرس‌دهی کند.

تا اینجا متوجه شدیم که بسته‌های آدرس ۹ بیتی هستند (۷ بیت آدرس، ۱بیت خواندن یا نوشتن و ۱ بیت تصدیق) اکنون شکل گفته‌های بالا را در شکل ۱۲-۹ و ۱۲-۱۰ مشاهده می‌کنید:



شکل ۱۲-۹ : بیت‌های بسته‌ی آدرس

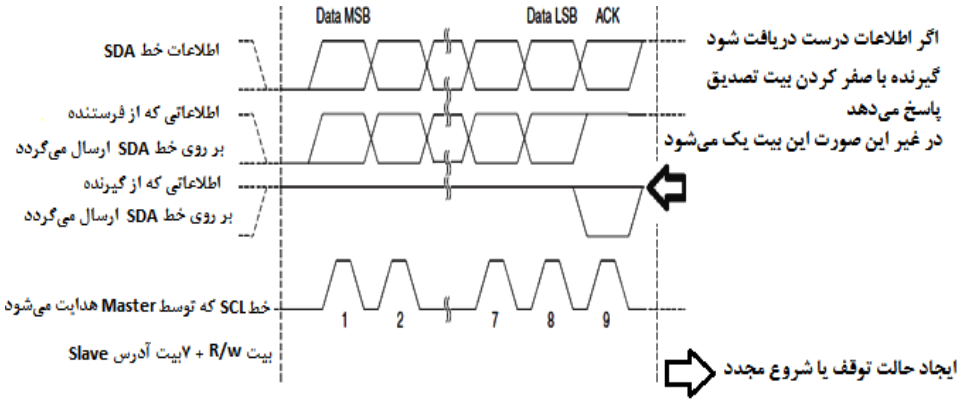
و شکل موج پایه‌های SDA و SCL برای آدرس‌دهی:



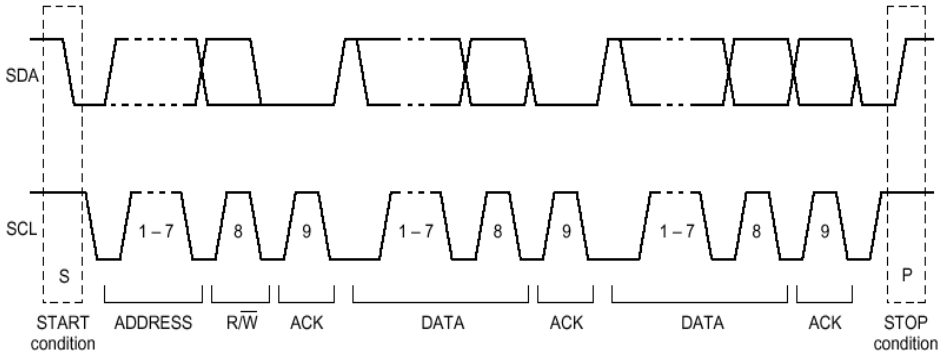
شکل ۱۲- ۱۰ : شکل موج پایه‌های SDA و SCL برای آدرس‌دهی

شکل بسته‌ی داده

فرمت بسته‌های داده همانند بسته‌های آدرس به صورت ۹ بیتی می‌باشند، با این تفاوت که دارای ۸ بیت داده و یک بیت تصدیق می‌باشد. زمانیکه دریافت‌کننده، اطلاعات را به طور کامل دریافت کرد با زمین کردن خود(صفر منطقی) پاسخ می‌دهد، یعنی بیت تصدیق (ACK) صفر می‌شود و فرستنده (Master) را متوجه می‌سازد که دریافت اطلاعات کامل است ولی اگر بیت ACK از لحاظ منطقی بالا یا به عبارتی یک باشد Master را متوجه می‌سازد که دریافت کامل نبوده و بایستی اطلاعات (۸بیت یا یک بایت داده) مجدداً ارسال گردد. به شکل زیر دقت کنید:



و شکل زیر مربوط به ارسال آدرس و داده با هم می‌باشد:

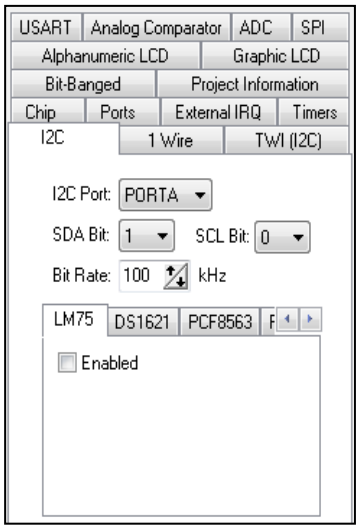


شکل ۱۲- ۱۱ : شکل ارسال آدرس و داده با یکدیگر

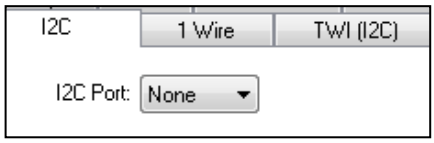
تا به اینجای کار با نحوه‌ی ارسال اطلاعات از طریق دو سیم در پروتکل I2C آشنا شدیم در ادامه با تنظیمات کدویزارد و توابعی که در کدویژن برای کار با I2C وجود دارد و همچنین نحوه‌ی کار با eepromها آشنا می‌شویم.

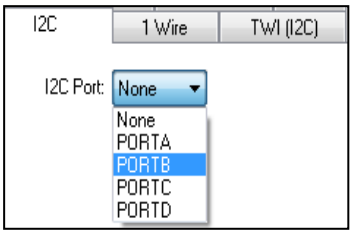
تنظیمات I2C در کدویزارد

تنظیمات مربوط به I2C در کدویزارد بسیار ساده می‌باشد، ابتدا به شکل روبرو توجه کنید:

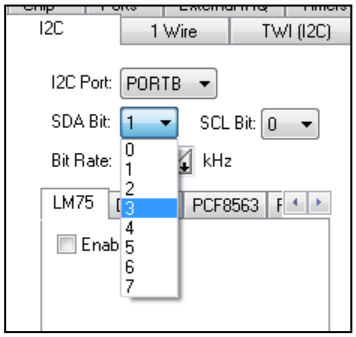


در صفحه‌ی تنظیمات I2C با باز کردن آن ابتدا با شکل زیر مواجه می‌شویم:



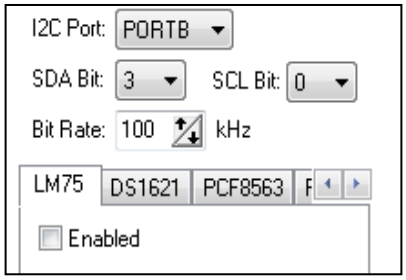


در ایـن بخـش شـما مـی‌توانیـد پـورتی (PORT) کـه می‌خواهید دو سیم SCL و SDA باشند را انتخاب کنید:



بـرای مثـال PORTB را انتخـاب کـردیم بعـد از آن بایـد بیت‌های مربـوط بـه پایـه‌هـای SCL و SDA را مشـخص کنیم(شکل روبرو):
که بیت ۳ را برای SDA و بیت صفر را برای SCL انتخـاب کردیم.

اسامی قطعاتی که در پایین آن مشاهده می‌کنید مربوط به ارتبـاط I2C بـا قطعـات نوشـته شـده می‌باشد. در کدویژن کتابخانه‌های این قطعات برای استفاده‌ی راحت از آنها وجود دارد:



گزینه‌ی Bit Rate هم مربوط به نرخ ارسال بیت‌ها در ثانیه می‌باشد که در میکروکنترلرهای سری ATmega16 حداکثر می‌تواند با نرخ ۱۰۰ کیلو هرتز کار کند.
همانطور که مشاهده شد تنظیمات این ارتباط به سـادگی انجـام گرفـت. بـا انجـام ایـن تنظیمـات میکروکنترلر به صورت Master عمل می‌کند و آماده‌ی ارتباط با سایر وسـایل جـانبی مـی‌باشـد. تنظیمات مربوط به بخش(TWI(I2C برای زمانی اسـت کـه مـی‌خـواهیم دو میکروکنترلـر را بـا یکدیگر مرتبط سازیم یا اینکه میکروکنترلر در حالت Slave باشد.

توابع کار با I2C در کدویژن

این توابع که در کتابخانه‌ی I2C.h وجود دارد کار با پروتکل I2C را بسیار ساده می‌کند، این توابع عبارتند از:

۱)i2c_init();

با نوشتن این تابع تنظیمات مربوط به ارتباط I2C انجام می‌گیرد.

۲) i2c_start();

با نوشتن این تابع در حقیقت یک حالت شروع ایجاد می‌شود که در بالا توضیح داده شده است(حالت start).

۳) i2c_stop();

با نوشتن این تابع در حقیقت یک حالت توقف ایجاد می‌شود که در بالا توضیح داده شده است(حالت stop).

۴) i2c_write(unsigned char data);

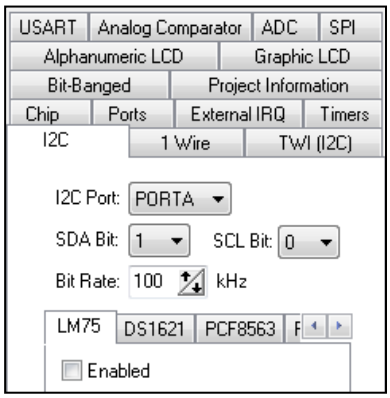
به‌وسیله‌ی این دستور می‌توان یک بایت داده را روی باس ارتباطی نوشت و در صورتی که دریافت‌کننده (Slave) آن را دریافت کند مقدار ۱ و در غیر این صورت صفر را برمی‌گرداند.

۵) i2c_read(unsigned char ack);

این دستور یک بایت را از روی باس ارتباطی می‌خواند و زمانیکه داده را خواند یک پالس تصدیق ایجاد می‌کند.اکنون به سراغ یک مثال می‌رویم که مفاهیم گفته شده را در شبیه‌سازی هم مشاهده کنیم.

مثال۱: می‌خواهیم برنامه‌ای بنویسیم که در آن یک میکروکنترلر به صورت Master باشد و نحوه‌ی ارسال اطلاعات به صورت پروتکل I2C و میکروکنترلر عدد ۱۸۳ را ارسال کند:

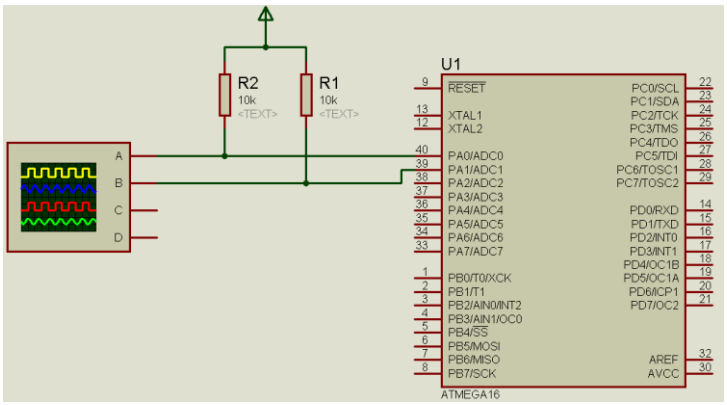
هدف: می‌خواهیم شکل موج‌های تولید شده روی پایه‌های SCL و SDA را با استفاده از اسیلوسکوپ نشان دهیم.



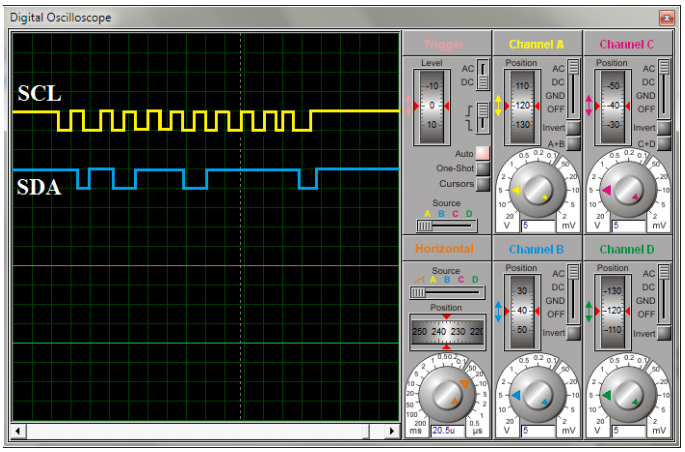
ابتدا تنظیمات کدویزارد را مطابق شکل روبرو انجام می‌دهیم (می‌خواهیم پایه‌های صفر و یک PORTA را به ترتیب برای خط SCL و SDA و Bit Rate=100 kHZ تنظیم کنیم):

```
while (1)
      {
      i2c_init();
      i2c_start();
      i2c_write(183);
      i2c_stop();
      delay_ms(10);
      }
```

حال کدنویسی را در کدویژن انجام می‌دهیم:
همانطور که مشاهده می‌کنید با دستور i2c_init() ابتدا تنظیمات مربوط به ارتباط I2C را انجام دادیم، سپس با دستور i2c_start() یک حالت شروع (اعلام آغاز بسته‌ی اطلاعاتی) را ایجاد کردیم، سپس با دستور i2c_write(183) عدد ۱۸۳ را روی باس I2C قرار دادیم و با دستور i2c_stop() یک حالت توقف ایجاد کردیم (اعلام پایان بسته‌ی اطلاعاتی)، در آخر هم از یک تاخیر ۱۰ میلی‌ثانیه‌ای استفاده کرده‌ایم که در شبیه‌سازی شکل موج به خوبی توسط اسیلوسکوپ مشاهده شود.
حال کد فوق را در نرم‌افزار پروتئوس شبیه‌سازی می‌کنیم و سپس شکل موج پایه‌های SCL و SDA را در شکل ۱۲-۱۳ مشاهده می‌کنیم:



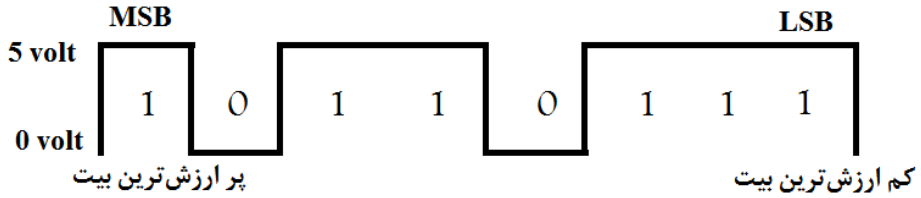
شکل ۱۲-۱۲: مدار بسته شده در پروتئوس



شکل ۱۲- ۱۳: شکل موج پایه‌های SCL و SDA

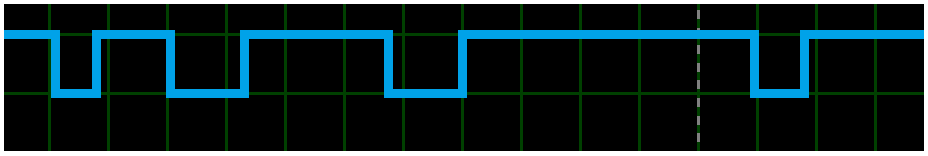
شکل موج اول مربوط به پایه‌ی SCL و شکل دوم مربوط به پایه‌ی SDA می‌باشد.

بررسی دقیق‌تر شکل موج شبیه‌سازی: همانطور که در فصل SPI هم مشاهده کردیم شکل موج مربوط به عدد ۱۸۳ به صورت شکل زیر می‌باشد:

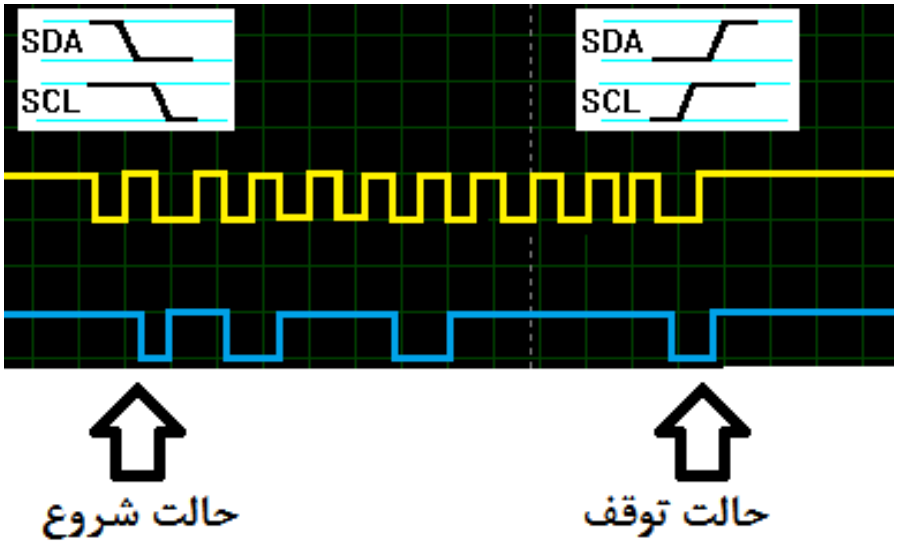


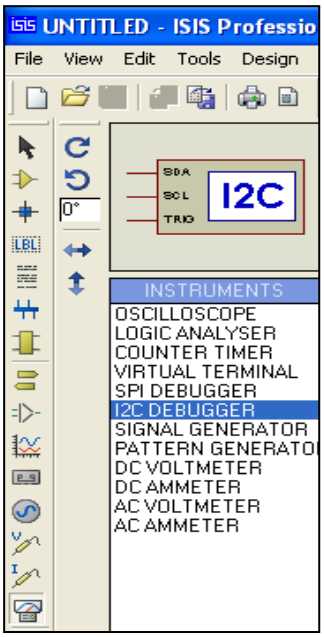
شکل ۱۲-۱۴

مشاهده می‌کنید شکل موج ۲ هم مطابق شکل زیر می‌باشد:



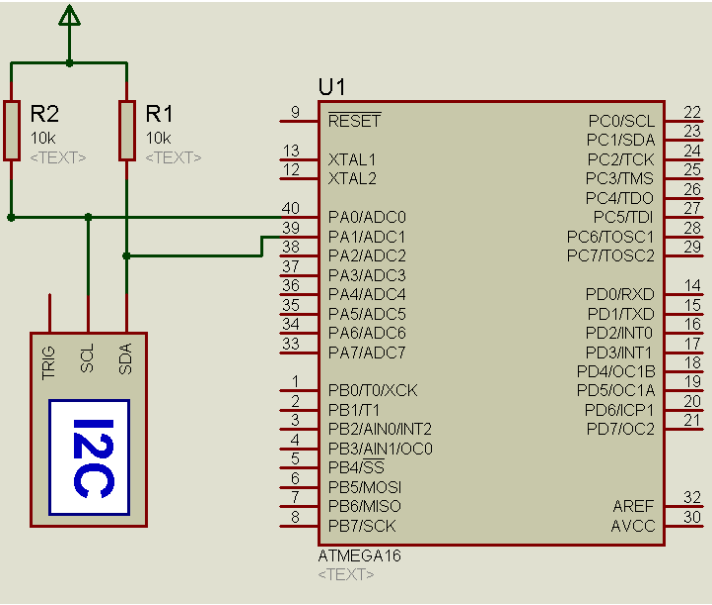
در شکل زیر نیز حالات شروع و توقف را در شکل موج پایه‌ها مشاهده می‌کنید:



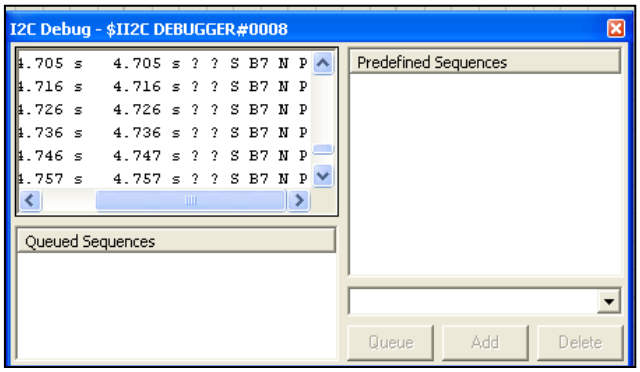


در نرم‌افزار پروتئوس دستگاهی به نام I2C Debugger وجود دارد که می‌توان اطلاعات موجود روی بـاس I2C را بوسـیله‌ی آن مشاهده کرد، تصویر روبرو مکان این دستگاه را در پروتئوس نشان می‌دهد:

حال اطلاعات ارسالی همین مثال با استفاده از I2C Debugger را در شکل ۱۲-۱۶ صفحه‌ی بعد مشاهده می‌کنیم:



شکل ۱۲-۱۵ : استفاده از I2C Debugger



شکل ۱۲-۱۶ : داده‌های دریافتی از I2C DEBUGGER

همانطور که مشاهده می‌کنید فرم بسته‌ی اطلاعاتی به صورت زیر می‌باشد:

S B7 N P

شکل ۱۲- ۱۷ : فرم بسته‌ی اطلاعاتی

S: یعنی حالت شروع(Start).

B7: عدد هگزادسیمال ۱۸۳ می‌باشد که روی باس I2C می‌باشد.

N: به معنای تصدیق نشدن بسته‌ی اطلاعاتی یا NACK(NOT Acknowledge) که با یک شدن این بیت همراه می‌باشد زیرا میکروکنترلر را به وسیله‌ی جانبی که به عنوان Slave باشد متصل نکردیم تا با زمین کردن خود دریافت بسته‌ی اطلاعاتی را تصدیق کند.

P: به معنای ایجاد حالت توقف (Stop).

با این مثال کاملاً توضیحات گفته شده در مورد این پروتکل ارتباطی و نحوه‌ی ایجاد آن را متوجه شدید، در ادامه با eepromهای خانواده‌ی at24c آشنا می‌شویم و نحوه‌ی کار با این خانواده از eepromها را می‌آموزیم.

eeprom های خانواده‌ی AT24C

در این بخش قصد داریم نحوه‌ی کار با این خانواده از eepromهارا بیاموزیم و با نحوه‌ی آدرس‌دهی آنها آشنا شویم. برای رسیدن به این هدف نحوه‌ی کار با دو حافظه‌ی at24c02 و at24c1024 را که به ترتیب حافظه‌های ۲ کیلوبیتی و ۱۰۲۴ کیلوبیتی از این خانواده می‌باشند را با دو مثال ساده و آموزنده فرا می‌گیریم.

ویژگی‌های eeprom خانواده‌ی AT24C

مهم‌ترین ویژگی‌های این خانواده عبارتند از:

- مصرف پایین و کار با ولتاژهای استاندارد
- حافظه‌هایی با صفحه‌بندی ۸ بایتی در حافظه‌های با حجم ۱,۲کیلوبیتی، صفحه‌بندی ۱۶ بایتی در حافظه‌های با حجم۱۶،۸،۴و۶۴کیلوبیتی، صفحه‌بندی۳۲بایتی در حافظه‌های با حجم۳۲و ۶۴کیلوبیتی، صفحه‌بندی۶۴بایتی در حافظه‌های با حجم ۱۲۸و۲۵۶کیلوبیتی، صفحه‌بندی ۱۲۸ بایتی برای حافظه‌های با حجم ۵۱۲ کیلوبیتی و در نهایت صفحه‌بندی ۲۵۶بایتی برای حافظه‌های با حجم ۱مگابیتی.
- از دیگر ویژگی‌های این خانواده قابلیت ارتباط I2C، قابلیت کار با فرکانس ۱۰۰ کیلو هرتز (ماکزیمم فرکانس کاری میکروکنترلر هم در مُد کاری I2C، ۱۰۰ کیلوهرتز می‌باشد)
- قابلیت یک بیلیون بار نوشتن بر روی حافظه و حفظ اطلاعات به مدت ۱۰۰ سال .

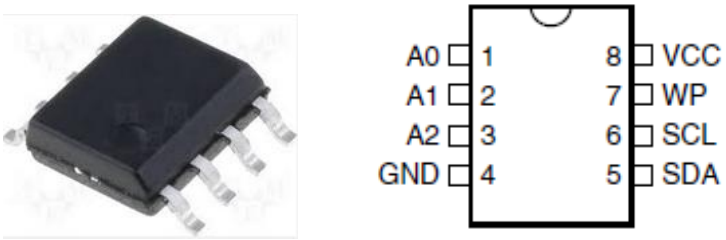
در شکل زیر ویژگی‌های کامل این IC که در دیتاشیت آنها موجود می‌باشد را مشاهده می‌کنید:

Features

- Low-voltage and Standard-voltage Operation
  - 2.7 (V_{CC} = 2.7V to 5.5V)
  - 1.8 (V_{CC} = 1.8V to 5.5V)
- Internally Organized 128 x 8 (1K), 256 x 8 (2K), 512 x 8 (4K), 1024 x 8 (8K) or 2048 x 8 (16K)
- Two-wire Serial Interface
- Schmitt Trigger, Filtered Inputs for Noise Suppression
- Bidirectional Data Transfer Protocol
- 100 kHz (1.8V) and 400 kHz (2.7V, 5V) Compatibility
- Write Protect Pin for Hardware Data Protection
- 8-byte Page (1K, 2K), 16-byte Page (4K, 8K, 16K) Write Modes
- Partial Page Writes Allowed
- Self-timed Write Cycle (5 ms max)
- High-reliability
  - Endurance: 1 Million Write Cycles
  - Data Retention: 100 Years
- Automotive Devices Available
- 8-lead JEDEC PDIP, 8-lead JEDEC SOIC, 8-lead Ultra Thin Mini-MAP (MLP 2x3), 5-lead SOT23, 8-lead TSSOP and 8-ball dBGA2 Packages
- Die Sales: Wafer Form, Waffle Pack and Bumped Wafers

شکل ۱۲-۱۸ : برگه‌ی اطلاعات فنی حافظه‌های eeprom

در شکل زیر فرم بسته‌بندی DIP این ICها را مشاهده می‌کنید:



برخی از مهم‌ترین ویژگی‌های خانواده‌ی AT24C

۱- کارکرد پایه‌ها (مطابق جدول ۱۲-۱):

نام پایه	عملکرد پایه	Function
A0 - A2	ورودی آدرس (برای نصب چند حافظه بر روی یک باس)	Address Inputs
SDA	خط ارسال داده به صورت سریال	Serial Data
SCL	خط ایجاد پالس برای ارتباط	Serial Clock Input
WP	محافظت از نوشته‌ها Write protect	Write Protect
NC	به جایی وصل نمی‌شود	No Connect
GND	زمین	Ground
VCC	منبع تغذیه	Power Supply

جدول ۱۲-۱ : کارکردهرپایه

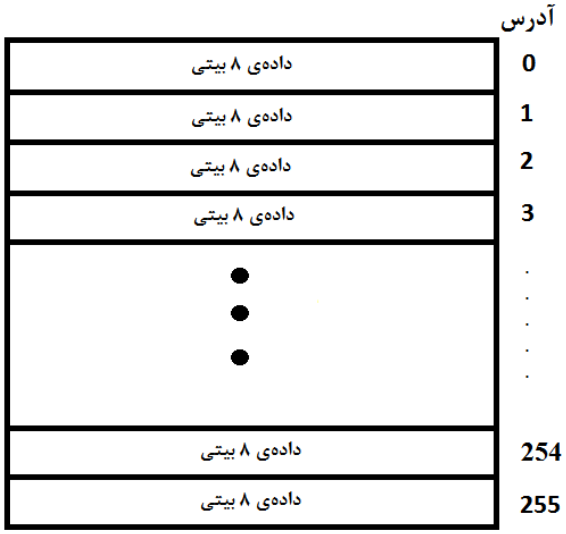
۲- پارامترهای الکتریکی مجاز:

Operating Temperature	–55°C to +125°C
Storage Temperature	–65°C to +150°C
Voltage on Any Pin with Respect to Ground	–1.0V to +7.0V
Maximum Operating Voltage	6.25V
DC Output Current	5.0 mA

شکل ۱۲-۱۹ : پارامترهای مجاز

حافظه‌های خانواده AT24C02 (حافظه‌ی ۲ کیلو بیتی)

این حافظه از خانواده‌ی AT24C دارای حافظه ۲ کیلوبیتی یا به عبارتی ۲۰۴۸ بیتی می‌باشند (۱۰۲۴*۲ چون هر کیلوبیت ۱۰۲۴ بیت می‌باشد) که به صورت 256*8 بیت آدرس‌دهی می‌شود(ماکزیمم آدرس ۲۵۶) می‌باشد. در شکل زیر بلوک‌بندی این حافظه را مشاهده می‌کنید:

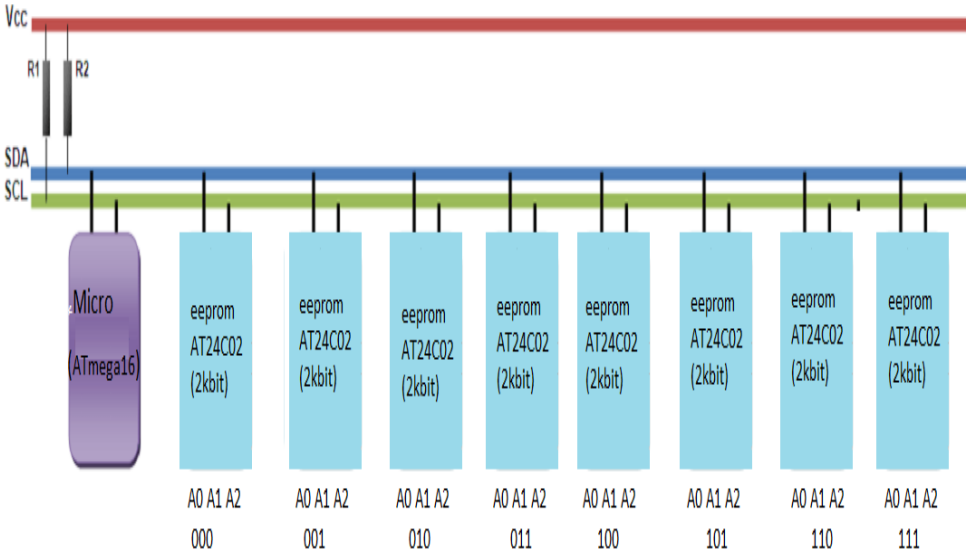


شکل ۱۲-۲۰ : بلوک‌بندی حافظه AT24C02

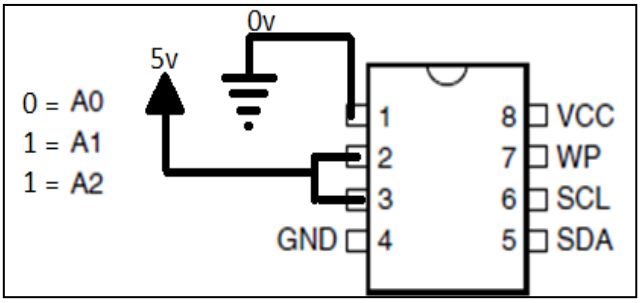
همانطور که در مشخصات پایه‌های خانواده‌ی AT24C مشاهده کردید ۳ پایه به نام‌های A2،A1،A0 وجود دارد که با این ۳ پایه می‌توان بسته به نوع تراشه تعدادی از آنها را روی باس I2C نصب کرد برای اینکه بیشتر متوجه شوید به جدولی از آدرس تراشه‌های ۱،۲،۴،۸و۱۶ کیلوبیتی توجه کنید:

1K/2K	1	0	1	0	A_2	A_1	A_0	R/W
	MSB							LSB
4K	1	0	1	0	A_2	A_1	P0	R/W
8K	1	0	1	0	A_2	P1	P0	R/W
16K	1	0	1	0	P2	P1	P0	R/W

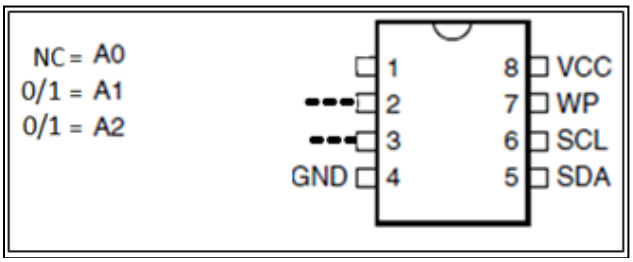
این آدرس برای تمامی eepromهای خانواده‌ی AT24C با ۴ بیت ۱۰۱۰ شروع می‌شود، ۳ بیت بعدی بسته به حجم حافظه می‌تواند متفاوت آدرس‌دهی شود و بیت آخر هم بیت خواندن و نوشتن می‌باشد و مشخص می‌کند که از حافظه قرار است بخوانیم یا بر روی آن بنویسیم. برای مثال از نحوه‌ی آدرس‌دهی این تراشه‌ها همانطور که در حافظه‌های ۱و ۲ کیلوبیتی مشاهده می‌کنید می‌توان ۳ بیت آدرس برای آی‌سی‌های خارجی مشابه دیگر داد، این بدین معنی است که با ۳ بیت A2،A1،A0 به تعداد حداکثر ۸ آی‌سی خارجی ۱ یا ۲ کیلوبیتی می‌توان روی خط I2C قرار داد و با این ۳ بیت آنها را همانند شکل زیر آدرس‌دهی کرد:



فقط باید توجه داشت که بیت‌های مربوط به آدرس ورودی(بیت‌های A) باید به صورت خارجی آدرس آنها مشخص شود، چون میکروکنترلر زمانی در حال فرمان دادن می‌باشد که مشخص کند می‌خواهد روی کدامیک از حافظه‌ها بخواند یا بنویسد، برای مثال در eeprom که آدرس آن به صورت ۰۱۱ می‌باشد(حافظه‌ی چهارم روی خط از سمت چپ در شکل بالا) باید پایه ها را به صورت شکل متصل کرد:



در حافظه های ۴ کیلوبیتی همانطور که مشاهده می‌کنید دوبیت A2،A1 برای آدرس‌دهی آی‌سی خارجی می‌باشد، این بدین معنی است که حداکثر می‌توان از ۴ حافظه‌ی ۴ کیلوبیتی بر روی باس I2C استفاده کرد و آنها را آدرس‌دهی کرد. بیت P0 هم مربوط به آدرس صفحه می‌باشد، در این حالت پایه‌ی A0 را نباید به جایی وصل کرد (NC یا Not Connect) مطابق شکل زیر:



شکل ۱۲-۲۱

منظور از آدرس صفحه این می‌باشد که در این حالت بسته به اینکه آدرس صفحه (P) چند بیتی می‌باشد هر صفحه به 2^P قسمت تقسیم می‌شود، برای مثال حافظه‌ی ۴کیلوبیتی که ۱ بیت آدرس صفحه دارد هر صفحه به دو بخش تقسیم می‌شود و در ۸ کیلوبیت که دوبیت آدرس صفحه دارد(P1،P0) هر صفحه به ۴ قسمت و به همین ترتیب حافظه‌ی ۱۶ کیلوبیتی هم به ۸ قسمت تقسیم می‌شود، در بقیه‌ی حافظه‌های این خانواده با حجم‌های متفاوت نیز همین روند وجود دارد، یعنی اینکه بیت‌های A مربوط به آدرس قطعاتیست که می‌توان روی خط قرار داد و بیت‌های P مربوط به آدرس صفحه می‌باشد.

اکنون برنامه‌ای می‌نویسیم که نحوه‌ی کار با این حافظه‌ها را متوجه شوید.

در قسمت Help کدویژن تابعی برای خواندن و نوشتن از حافظه نوشته شده که این توابع به قرار زیر می باشند.

تابع برای خواندن از حافظه در کدویژن

```
/* read a byte from the EEPROM */
unsigned chareeprom_read(unsigned char address) {
unsigned char data;
i2c_start();
i2c_write(eeprom_BUS_ADDRESS);
i2c_write(address);
i2c_start();
i2c_write(eeprom_BUS_ADDRESS | 1);
data=i2c_read(0);
i2c_stop();
return data;
}
```

این تابع بدین ترتیب عمل می‌کند که در ابتدا حالت Start را آغاز می‌کند بعد از آن در بخش i2c_write(EEPROM_BUS_ADDRESS) آدرس تراشه را می‌نویسیم(با توجه به توضیحات قبل که گفته شد) سپس در قسمت i2C_write(address) آدرس مورد نظر داده‌ای را که می‌خواهیم بخوانیم را می‌نویسیم، بعد از اینکه تراشه و آدرس داده‌ی مورد نظر به درستی آدرس‌دهی شد یک حالت شروع مجدد را ایجاد می‌کنیم و بعد از آن با دستور i2c_write(EEPROM_BUS_ADDRESS|1) بیت آخر آدرس تراشه را که بیت مربوط به خواندن یا نوشتن(R/W) می‌باشد یک می‌کنیم تا عمل خواندن را انجام دهیم(Read=1 و Write=0 می‌باشد) بعد از آن اولین بسته‌ی اطلاعاتی را که روی باس i2c وجود دارد با دستور i2c_read(0) می‌خوانیم و درون متغیر data می‌ریزیم سپس حالت توقف را با دستور i2s_stop() ایجاد می‌کنیم و در نهایت آن را به جایی که فراخوانی کرده باز می‌گردانیم.

تابع برای نوشتن بر روی حافظه در کدویژن

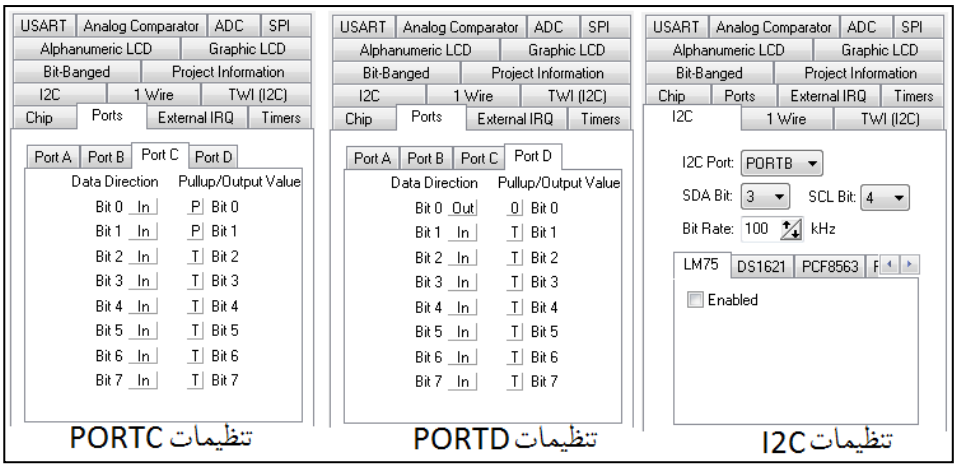
```
/* write a byte to the          */
void eeprom_write(unsigned char address، unsigned char data) {
i2c_start();
i2c_write(EEPROM_BUS_ADDRESS);
i2c_write(address);
i2c_write(data);
i2c_stop();
/* 10ms delay to complete the write operation */
delay_ms(10);
}
```

این تابع بدین ترتیب عمل می‌کند که در ابتدا حالت Start را آغاز می‌کند بعد از آن در بخش i2c_write(EEPROM_BUS_ADDRESS) آدرس تراشه را می‌نویسیم، سپس در قسمت i2C_write(address) آدرس مورد نظرحافظه‌ای که می‌خواهیم بر روی آن داده‌ی مورد نظرمان را بنویسیم قرار می‌دهیم سپس با دستور i2c_write(data) یک بایت داده‌ی مورد نطر خود را می‌نویسیم و در آخر حالت توقف را با دستور i2s_stop() ایجاد می‌کنیم حال با یک مثال موضوع کاملاً روشن می‌گردد.

مثال۲: برنامه‌ای بنویسید که با زمین کردن پایه‌ی C0 (که به صورت pull-up تنظیم شده است) عدد ۵۵ را بر روی خانه‌ی ۱۳۶یک AT24C02 بنویسد و با زمین کردن پایه‌ی C1 (که به صورت pull-up تنظیم شده است) داده‌ی موجود در خانه‌ی ۱۳۶ حافظه‌ی AT24C02 را بخواند و آن را در متغیر x بریزد بعد از آن با نوشتن یک دستور شرطی که اگر این متغیر x

مقداری برابر ۵۵ داشت(مقداری که درون حافظه ریختیم)LED پایه‌ی D0 را روشن کند (نحوه‌ی کار با حافظه‌ی eeprom AT24C02):

ابتدا تنظیمات مربوط به کدویزارد را مانند شکل زیر انجام می‌دهیم:

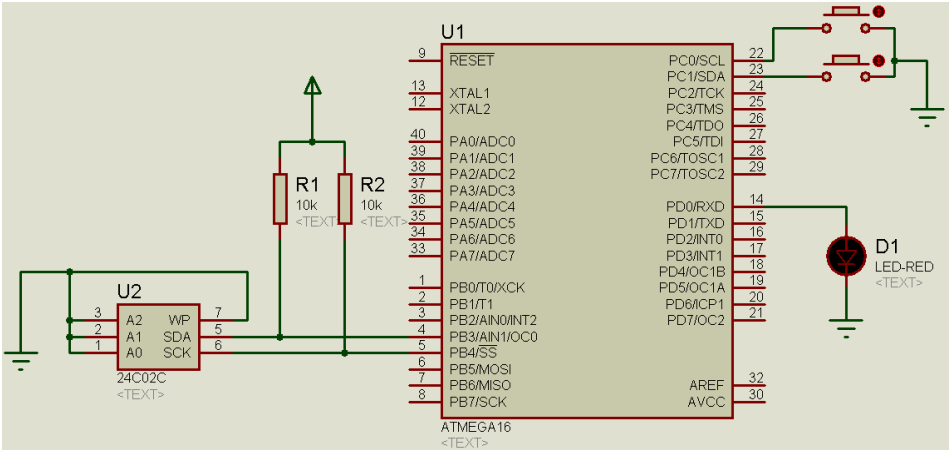


شکل ۱۲-۲۲ : تنظیمات مثال ۲

حال کد زیر را درون حلقه‌ی while(1) می‌نویسیم:

```
while (1)
      {
      if(PINC.0==0)
      {
      i2c_init();
      i2c_start();
      i2c_write(0xa0);
      i2c_write(136);
      i2c_write(55);
      i2c_stop();
      delay_ms(10);
      }
      if(PINC.1==0)
      {
      i2c_start();
      i2c_write(0xa0);
      i2c_write(136);
      i2c_start();
      i2c_write(0xa0|1);
      x=i2c_read(0);
      i2c_stop();
      }
      if(x==55)
      PORTD.0=1;
      }
}
```

مدار زیر را در پروتئوس می‌بندیم:

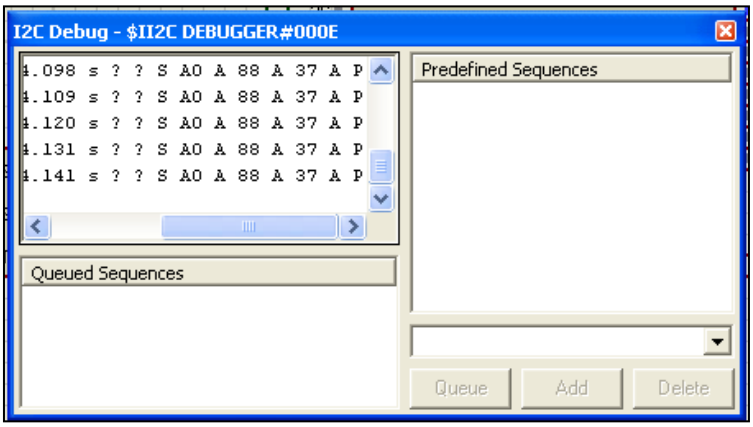


شکل ۱۲-۲۳ : مدار بسته شده در پروتئوس

همانطور که در شکل بالا مشاهده می‌کنید، سه پایه‌ی A2،A1،A0 زمین شده است یعنی آدرس این تراشه برای ۳ بیت آخر ۰۰۰ می‌باشد.

اگر شبیه‌سازی را انجام دهیم: ابتدا کلید پایه‌ی C0 را می‌زنیم، بعد از آن پایه‌ی C1 را زمین می‌کنیم و مشاهده می‌کنیم که LED روشن می‌شود.

حال داده‌ها را در هنگام نوشتن و خواندن با دستگاه I2C Debugger مشاهده می‌کنیم. ابتدا وقتی پایه‌ی C0 را زمین می‌کنیم یعنی عملیات نوشتن را انجام می‌دهیم داده‌ی I2C Debugger را می‌بینیم:



شکل ۱۲-۲۴ : داده‌های دریافتی از I2C DEBUGGER

که داده‌ی آن به صورت زیر می‌باشد:

S A0 A 88 A 37 A P

S: یعنی حالت شروع یا Start

A0: آدرس تراشه که برابر A0 بود در مبنای HEX

A: بیت تصدیق(ACK)

88: این همان عدد هگزادسیمال ۱۳۶ است که مربوط به آدرس حافظه می‌باشد.

A: بیت تصدیق(ACK)

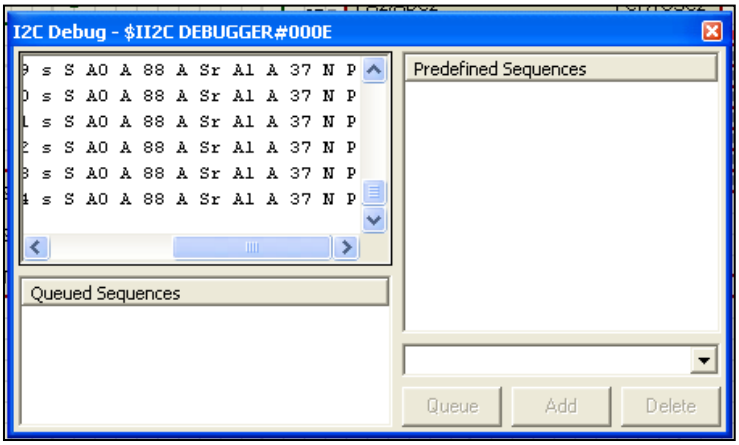
37: همان عدد هگزادسیمال ۵۵ است که مربوط به داده‌ی نوشته شده بر روی حافظه می‌باشد.

A: بیت تصدیق(ACK)

P: ایجاد حالت توقف یا stop

این دستورات را با کد نوشته شده در بخش write مقایسه کنید.

حال وقتی پایه‌ی C1 را زمین می‌کنیم یعنی عملیات نوشتن را انجام می‌دهیم داده‌ی I2C Debugger را می‌بینیم:



شکل ۱۲- ۲۵ : داده‌های دریافتی از I2C DEBUGGER

که داده‌ی آن به صورت زیر است:

S A0 A 88 A Sr A1 A 37 N P

S: یعنی حالت شروع یا Start

A0: آدرس تراشه که A0 بود در مبنای HEX

A: بیت تصدیق(ACK)

88: این همان عدد هگزادسیمال ۱۳۶ است که مربوط به آدرس حافظه می‌باشد.

A : بیت تصدیق(ACK)

Sr: ایجاد حالت شروع مجدد یا start repeated.

A1: آدرس تراشه است که در مبنای HEX بیان می‌شود، بیت آخر آن یک شده یعنی آماده‌ی خواندن اطلاعات است.

A: بیت تصدیق(ACK)

37: همان عدد هگزادسیمال ۵۵ است که مربوط به داده‌ی نوشته شده بر روی حافظه می‌باشد.

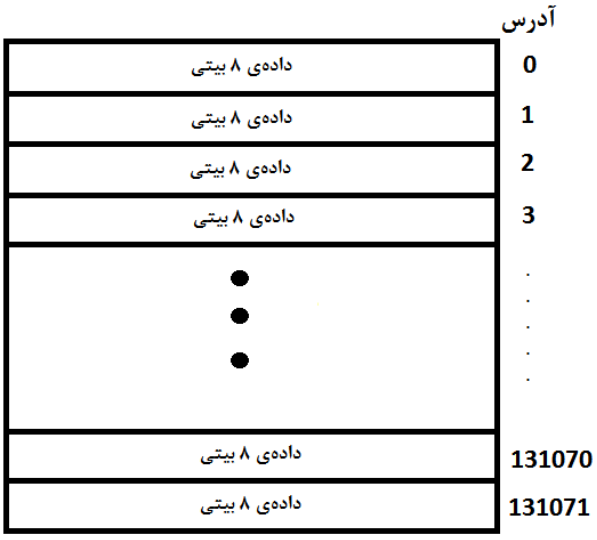
N:مربوط به بیت عدم تصدیق (NACK)

P:ایجاد حالت توقف یا stop

توجه مهم: این دستورات را با کد نوشته شده در بخش Read مقایسه کنید.

حافظه‌های خانواده AT24C1024 (حافظه‌ی ۱۰۲۴ کیلو بیتی)

این حافظه همانطور که گفته شد یک حافظه‌ی ۱۰۲۴ کیلوبیتی یا به عبارتی۱۰۲۴* ۱۰۲۴ بیتی(۱۰۴۸۵۷۶بیتی) می‌باشد و نحوه‌ی بلوک‌بندی حافظه‌ی آن به صورت ۱۳۱۰۷۲ کلمه‌ی ۸ بیتی می‌باشد. در شکل زیر نحوه‌ی بلوک‌بندی حافظه را مشاهده می‌کنید:



شکل ۱۲-۲۶ : بلوک‌بندی حافظه AT24C1024

همانطور که مشاهده می‌کنید آدرس آخرین خانه‌ی حافظه ۱۳۱۰۷۱ می‌باشد که در مبنای دو می‌شود:

1111111111111111

دارای ۱۶ بیت آدرس می‌باشد، پس باید آدرس‌های خود را به صورت دو آدرس ۸ بیتی بدهیم(زیرا در کدویژن دستور i2c_write(8bit) توانایی نوشتن‌های ۸ بیتی را داراست).
نحوه‌ی آدرس‌دهی تراشه برای این حافظه به شکل زیر می‌باشد(مطابق برگه‌ی مشخصات فنی (Datasheet) حافظه):

1	0	1	0	0	A_1	P_0	R/W

MSB LSB

که با یک بیت A1 می‌توان به تعداد دو حافظه‌ی ۱۰۲۴ کیلوبیتی بر روی یک باس SPI را آدرس‌دهی نمود. بیت P0 هم مربوط به آدرس صفحه می‌باشد یعنی هر صفحه (Page) را به دو بخش تقسیم می‌کند و بر روی آن می نویسد(این حافظه به صورت داخلی دارای پیکره‌بندی به صورت ۵۱۲ صفحه‌ی ۲۵۶ بایتی می‌باشد).

مثال۳: همان مثال قبلی را در نظر بگیرید با این تفاوت که یک حافظه‌ی ۱۰۲۴ کیلو بیتی را آدرس‌دهی می‌کنیم (نحوه‌ی کار با حافظه ی eeprom AT24C1024):
تنظیمات کدویزارد همانند مثال قبل می‌باشد. کد زیر را در while(1) می‌نویسیم:

```
while (1)
      {
      if(PINC.0==0)
      {
      i2c_init();
      i2c_start();
      i2c_write(0xa0);
      i2c_write(136);
      i2c_write(13);
      i2c_write(55);
      i2c_stop();
      delay_ms(10);
      }
      if(PINC.1==0)
      {
      i2c_start();
      i2c_write(0xa0);
      i2c_write(136);
      i2c_write(13);
      i2c_start();
      i2c_write(0xa0|1);
      x=i2c_read(0);
      i2c_stop();
      }

      if(x==55)
      PORTD.0=1;
      }
}
```

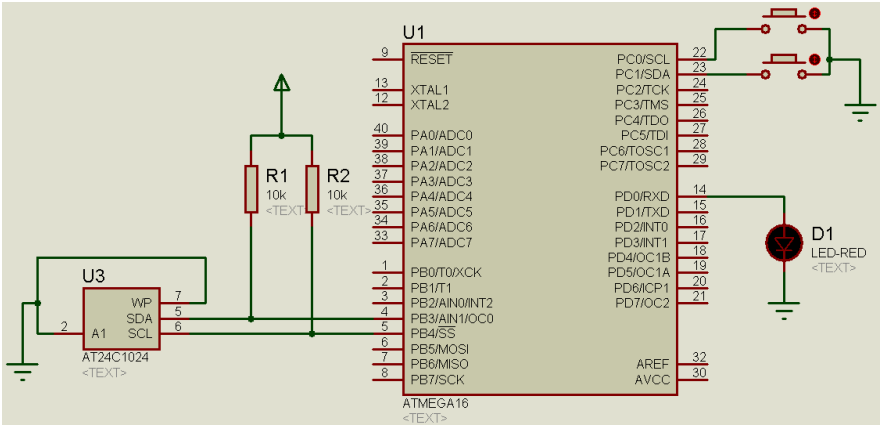
i2c_write(136)

1	0	0	0	1	0	0	0	0	0	0	0	1	1	0	1

آدرس‌دهی به صورت ۱۶ بیتی
i2c_write(136)
i2c_write(13)

i2c_write(13)

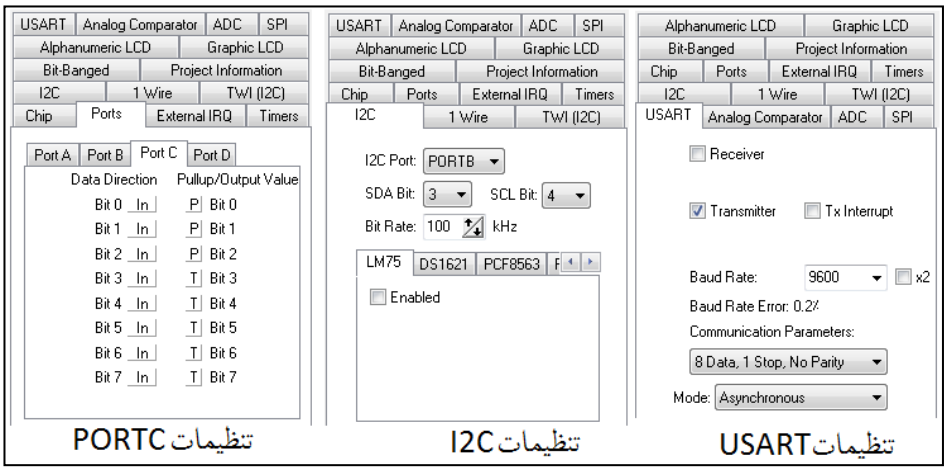
مدار بسته شده در نرم‌افزار پروتئوس مطابق شکل می‌باشد:



شکل ۱۲-۲۷ : مدار بسته شده‌ی مثال ۳

مثال۴: در این مثال به عنوان مثال آخر قصد داریم روی حافظه‌ی AT24C1024 زمانی که پایه‌ی C0 (که به صورت pull-up تنظیم شده است) را زمین کردیم عبارت "salam" را بنویسیم(در این مثال می‌خواهیم از خانه‌ی ۳۱۲۴۲ حافظه برای کلمه‌ی s تا خانه‌ی ۳۱۲۴۶ برای کلمه‌ی m استفاده کنیم) و وقتی پایه‌ی C1 (که به صورت pull-up تنظیم شده است) را زمین کردیم داده‌ها را از حافظه بخواند و به ترتیب در متغیرهای x0 تا x4 بریزد (x4=m، x3=a، x2=l، x1=a، x0=s) همچنین زمانیکه پایه‌ی C2 (که به صورت pull-up تنظیم شده است) را زمین کردیم این عبارات را به صورت USART ارسال کند:

تنظیمات کدویزارد مطابق تصویر زیر می‌باشد:



کد نوشته شده در کدویژن:

ابتدا ۴ متغیر x0 تا x4 را تعریف می‌کنیم:

```
#include <i2c.h>
#include <delay.h>
#include <stdio.h>

// Declare your global variables here
unsigned char x0,x1,x2,x3,x4;

void main(void)
{
// Declare your local variables here
```

حال کد زیر را درون حلقه‌ی while(1) می‌نویسیم:

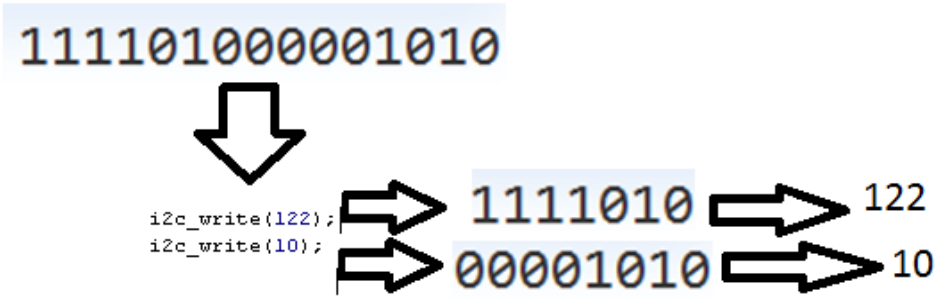
```
while(1)
  {
   if(PINC.0==0)
   {
          i2c_init();
          i2c_start();
          i2c_write(0xa0);
          i2c_write(122);
          i2c_write(10);
          i2c_write('s');
          i2c_stop();
          delay_ms(10);
          i2c_start();
          i2c_write(0xa0);
          i2c_write(122);
          i2c_write(11);
          i2c_write('a');
          i2c_stop();
          delay_ms(10);
          i2c_start();
          i2c_write(0xa0);
          i2c_write(122);
          i2c_write(12);
          i2c_write('l');
          i2c_stop();
          delay_ms(10);
          i2c_start();
          i2c_write(0xa0);
          i2c_write(122);
          i2c_write(13);
          i2c_write('a');
          i2c_stop();
          delay_ms(10);
          i2c_start();
          i2c_write(0xa0);
          i2c_write(122);
          i2c_write(14);
          i2c_write('m');
          i2c_stop();
          delay_ms(10);
          }
   if(PINC.1==0)
          {
          i2c_start();
```

```
        i2c_write(0xa0);
        i2c_write(122);
        i2c_write(10);
        i2c_start();
        i2c_write(0xa0|1);
        x0=i2c_read(0);
        i2c_stop();
        i2c_start();
        i2c_write(0xa0);
        i2c_write(122);
        i2c_write(11);
        i2c_start();
        i2c_write(0xa0|1);
        x1=i2c_read(0);
        i2c_stop();
        i2c_start();
        i2c_write(0xa0);
        i2c_write(122);
        i2c_write(12);
        i2c_start();
        i2c_write(0xa0|1);
        x2=i2c_read(0);
        i2c_stop();
        i2c_start();
        i2c_write(0xa0);
        i2c_write(122);
        i2c_write(13);
        i2c_start();
        i2c_write(0xa0|1);
        x3=i2c_read(0);
        i2c_stop();
        i2c_start();
        i2c_write(0xa0);
        i2c_write(122);
        i2c_write(14);
        i2c_start();
        i2c_write(0xa0|1);
        x4=i2c_read(0);
        i2c_stop();
        }
if(PINC.2==0)
        {
        putchar(x0);
        putchar(x1);
        putchar(x2);
        putchar(x3);
        putchar(x4);
        delay_ms(1000);
        }
        }
```

عدد ۳۱۲۴۲ در مبنای ۲ به شکل زیر محاسبه می‌شود:

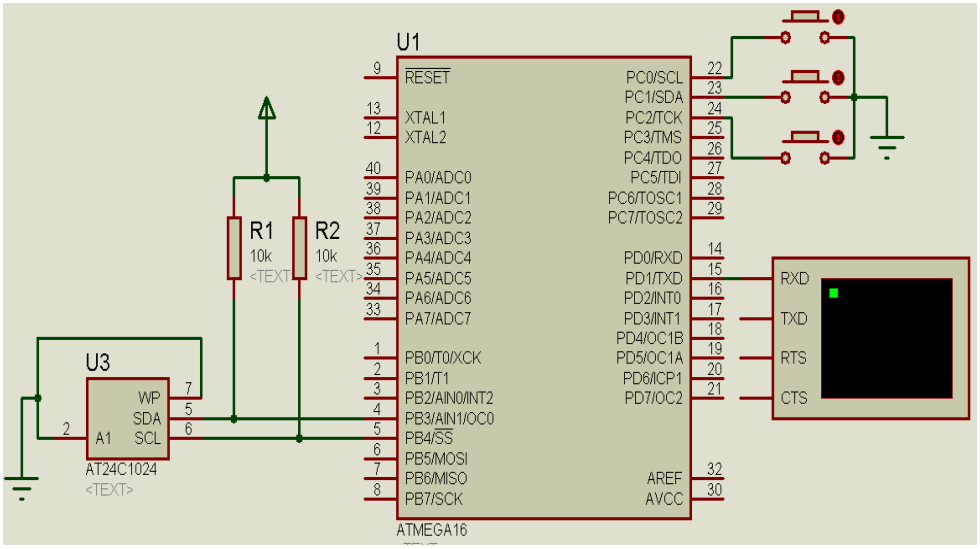
```
111101000001010
```

که به صورت دو آدرس ۸ بیتی با دو دستور زیر داده شده است:



و بقیه خانه‌های حافظه هم با توجه به شکل بالا مشخص است.

نتیجه‌ی شبیه‌سازی: بعد از اینکه به ترتیب سه کلید را زدیم، داده‌ای که توسط ارتباط USART ارسال می‌شود را مشاهده می‌کنیم که همان salam است. در زیر مدار بسته شده را مشاهده می‌کنیم:



شکل ۱۲-۲۸ : شبیه‌سازی مثال۴

کار با EEPROM داخلی میکروکنترلر

همانطور که می‌دانید میکروکنترلرهای ATmega16 دارای سه نوع حافظه‌ی داخلی می‌باشند؛ حافظه‌یFlash، حافظه‌ی RAM و حافظه‌یEEPROM. حافظه‌ی Flash برای ثبت اطلاعاتی اعم از کد و برنامه‌ی نوشته شده، مقادیر ثابت و مقادیر اولیه‌ی متغیرها است که در زمان پروگرام کردن ، درون میکروکنترلر ذخیره می‌شود. حافظه‌ی RAM برای نگهداری اطلاعات و پردازش آنها در حین اجرای برنامه به کار می‌رود، تعداد خواندن و نوشتن در این حافظه تقریباً نامحدود و بسیار سریع است ولی با خاموش شدن میکروکنترلر و قطع مدار اطلاعات درون این حافظه پاک می‌شود. حافظه‌ی دیگری که درون میکروکنترلر قرار دارد به نام EEPROM است که حافظه‌ای دائمی می‌باشد و با خاموش شدن میکروکنترلر و قطع مدار پاک نمی‌شود و خواندن و نوشتن روی این حافظه از نظر تعداد دارای محدودیت می‌باشد و نمی‌توان به طور نامحدود از آن استفاده کرد.

میکروکنترلر ATmega16 دارای حافظه‌ی EEPROM داخلی به حجم ۵۱۲ بایت می‌باشد که قابلیت صد هزار بار خواندن و نوشتن را دارد. کار با این حافظه مانند حافظه‌ی RAM به صورت مستقیم و سریع نیست و برای استفاده از آن باید با رجیسترهای میکروکنترلر کار کنیم. میکروکنترلر ATmega16 دارای سه رجیستر به نام‌های EEAR و EEDR و EECR می‌باشد که با این سه رجیستر می‌توانیم با EEPROM داخلی کار کنیم.

حجم EEPROM داخلی این میکروکنترلر برابر ۵۱۲ بایت است که به معنی آن است که این حافظه از ۵۱۲ خانه‌ی حافظه‌ی یک بایتی تشکیل شده است؛ یعنی می‌توان ۵۱۲ کاراکتر درون آن نوشت (چون هر کاراکتر ۸ بیت یا یک بایت است). برای ذخیره‌ی هر داده ابتدا باید خانه‌ی حافظه‌ای که قصد نوشتن درون آن را داریم مشخص کنیم سپس داده را درون آن بریزیم. برای مشخص کردن خانه‌ی حافظه از رجیستر EEAR (EEPROM Address Register) استفاده می‌کنیم. این رجیستر از دو بایت پرارزش و کم ارزش (EEARH و EEARL) تشکیل شده که برای مقداردهی به آن هم می‌توانیم به صورت هگزادسیمال و هم به صورت عدد در مبنای ده آن را مقداردهی کنیم. برای مثال اگر آدرس موردنظر خانه‌ی شماره ۳۲۰ باشد می‌نویسیم:

EEAR=320;

و یا اگر بخواهیم به صورت هگزادسیمال مقداردهی کنیم می‌نویسیم:

EEARL=0X40; EEARH=0X01;

پس از مشخص کردن آدرس حافظه، اطلاعاتی که می‌خواهیم ذخیره کنیم را درون رجیستر EEDR (EEPROM Data Register) می‌ریزیم که این کار را نیز به راحتی با یک دستور تساوی انجام می‌دهیم. مثلاً در دستور زیر متغیر a را درون این رجیستر قرار داده‌ایم:

EEDR=a;

نکته‌ای که باید به آن توجه کنیم این است که هر خانه‌ی حافظه فقط یک بایت است و نمی‌توانیم عددی بزرگتر از ۲۵۵ را درون یک خانه بنویسیم (چون عدد ۲۵۵ در مبنای دو برابر ۱۱۱۱۱۱۱۱ می‌باشد که بزرگترین عددی است که می‌توان در ۸ بیت یا یک بایت نشان داد).

برای خواندن و نوشتن در حافظه EEPROM ابتدا باید با رجیستر EECR آشنا شد.

رجیستر EECR که بر گرفته از EEPROM Control Register می‌باشد برای کنترل خواندن و نوشتن در حافظه‌ی EEPROM است. همانطور که در تصویر زیر مشاهده می‌کنید این رجیستر از ۸ بیت تشکیل شده که تنها ۴ بیت نخست آن به کار می‌آید و باقی بیت‌ها به صورت رزرو هستند. بیت شماره صفر (EERE) که مخفف EEPROM Read Enable است مخصوص خواندن از حافظه می‌باشد و هرگاه این بیت را یک کنیم عمل خواندن از EEPROM صورت می‌گیرد.

دو بیت بعدی یعنی بیت شماره ۱ و ۲ (EEWE و EEMWE) که برگرفته از EEPROM Write Enable و EEPROM Master Write Enable می‌باشند مخصوص نوشتن روی حافظه هستند و برای نوشتن داده روی حافظه باید با ترتیبی خاص این بیت‌ها را یک کنیم.

7	6	5	4	3	2	1	0	
–	–	–	–	EERIE	EEMWE	EEWE	EERE	EECR
R	R	R	R	R/W	R/W	R/W	R/W	
0	0	0	0	0	0	X	0	

نوشتن روی حافظه

برای نوشتن یک متغیر درون حافظه باید مراحل زیر را انجام دهیم:

۱- آدرس حافظه‌ی مورد نظر را روی رجیستر EEAR می‌نویسیم.
۲- داده‌ی مورد نظر را روی رجیستر EEDR می‌نویسیم.
۳- بیت شماره‌ی ۲ رجیستر EECR یعنی EEMWE را برابر یک قرار می‌دهیم.
۴- بیت شماره‌ی ۱ رجیستر EECR یعنی EEWE را برابر یک قرار می‌دهیم.

برای آنکه بیت شماره‌ی ۲ رجیستر EECR را برابر یک قرار دهیم می‌توانیم بنویسیم EECR=0X04 (زیرا عدد هگزادسیمال 0X04 معادل عدد ۴ در مبنای ۱۰ و عدد ۱۰۰ در مبنای ۲ است. با برابر قرار دادن این رجیستر با عدد 0X04 دو بیت اول این رجیستر برابر صفر و بیت سوم آن برابر یک می‌شود و بیت‌های دیگر آن هم همگی صفر می‌شوند).برای انجام مرحله‌ی چهارم یعنی یک کردن بیت شماره‌ی ۱ رجیستر EECR می‌توانیم آن را برابر عدد 0X02 قرار بدهیم، ولی با این کار بیت شماره‌ی ۲ دوباره صفر می‌شود. پس اگر بخواهیم این بیت را یک کنیم بدون آنکه باقی بیت‌ها تغییر کنند می‌توانیم این رجیستر را با عدد 0X02 ، OR (یای منطقی)کنیم.

	7	6	5	4	3	2	1	0
EECR	–	–	–	–	EERIE	EEMWE	EEWE	EERE
				OR				
0X02	0	0	0	0	0	0	1	0
				=				
	–	–	–	–	EERIE	EEMWE	1	EERE

با این کار می‌توانیم داده‌ی مورد نظرمان را در حافظه‌ی EEPROM ذخیره کنیم. فقط نکته‌ای که وجود دارد این است که برای آنکه بخواهیم دوباره روی حافظه بنویسیم باید صبر کنیم تا بیت EEWE که یک کردیم دوباره به صورت سخت‌افزاری برابر صفر بشود. پس باید قبل از نوشتن روی حافظه چک کنیم که آیا EEWE (یا بیت دوم رجیستر EECR) صفر هست یا خیر و اگر صفر نبود صبر کنیم تا صفر شود، بعد از آن مراحل نوشتن را شروع کنیم. برای اینکار می‌توانیم قبل از مراحل فوق دستور ;(while((EECR&0X02) == 0x02 را بنویسیم. زمانی که EECR و عدد 0x02 ، And می‌شوند تمام بیت‌ها صفر می‌شوند و تنها بیت شماره‌ی یک برابر EEWE باقی می‌ماند. اگر این بیت هنوز یک باشد پاسخ and همان عدد 0x02 می‌شود و برنامه درون while می‌ماند تا زمانی که EEWE صفر شود و تساوی درون آن دیگر برابر نباشد. به شکل زیر دقت کنید:

7	6	5	4	3	2	1	0	
–	–	–	–	EERIE	EEMWE	EEWE	EERE	EECR

And

0	0	0	0	0	0	1	0	0X02

=

0	0	0	0	0	0	EEWE	0

مثال۵: فرض کنید می‌خواهیم متغیر a را درون خانه‌ی ۵۲ حافظه بنویسیم. طبق مراحل بالا کد زیر را می‌نویسیم:

```
while((EECR&0X02) == 0X02);
EEAR = 52 ;
EEDR = a ;
EECR = 0X04 ;
EECR = (EECR | 0X02);
```

خواندن از حافظه EEPROM

۱- آدرس حافظه‌ی مورد نظر را روی رجیستر EEAR می‌نویسیم.
۲- بیت شماره‌ی صفر رجیستر EECR یعنی EERE را برابر یک قرار می‌دهیم.
۳- EEDR را برابر متغیر مورد نظر قرار می‌دهیم.

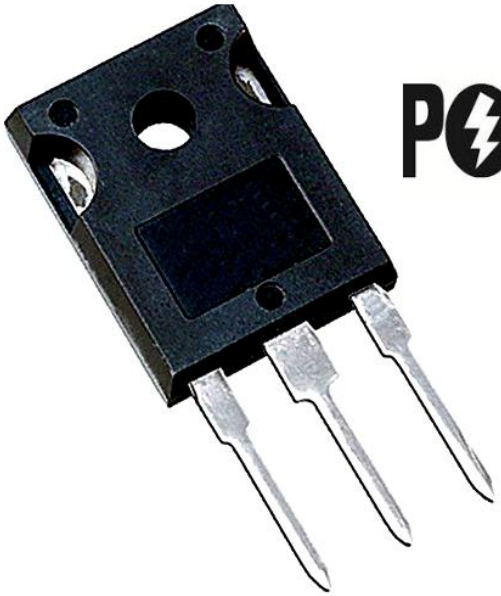
در خواندن نیز مانند نوشتن قبل از مراحل فوق باید چک کنیم که نوشتن قبلی روی حافظه تمام شده باشد یعنی بیت دوم رجیستر EECR صفر شده باشد. پس اینجا نیز از دستور **while((EECR&0X02) == 0X02);** استفاده می‌کنیم.

مثال۶ : فرض کنید می‌خواهیم آدرس ۱۰۵ حافظه را بخوانیم و آن را درون متغیر s بریزیم. طبق مراحل فوق کد زیر را می‌نویسیم:

```
while((EECR&0X02) == 0X02);
EEAR = 105 ;
EECR = 0X01 ;
s =  EEDR ;
```

فصل سیزدهم

کنترل توان



در این فصل خواهیم خواند:

۱. روش‌های کنترل توان
۲. عملکرد ترانزیستور در حالت کلیدزنی
۳. معرفی کلی ترانزیستورهای قدرت
۴. آپ–امپ و دیود زنر

توان

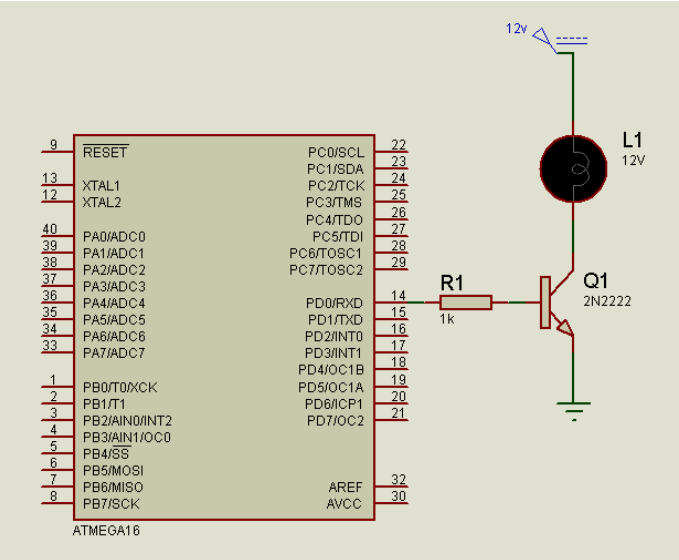
تا به اینجا نحوه‌ی استفاده از میکروکنترلر و کار با پایه‌های خروجی را آموختیم و در بیشتر مثال‌های خود از LEDها استفاده کردیم که عناصری با توان مصرفی پایینی هستند. حال فرض کنید به جای کار با یک عنصر توان پایین مانند LED قصد کنترل وسیله‌ای را دارید که توان بالاتر یعنی ولتاژ و جریان بیشتری احتیاج دارد. در ساده‌ترین حالت فرض کنید شما تعدادی چراغ رنگی دارید و قصد دارید با حالت‌های مختلف آن‌ها را روشن و خاموش کنید، پس به یک برنامه نیاز دارید تا در زمان‌های مختلف حالت‌های مختلفی برای پایه‌های میکروکنترلر داشته باشد.

اگر پایه‌های میکروکنترلر را به چراغ‌ها وصل کنید، میکروکنترلر توانایی لازم برای روشن و خاموش کردن این چراغ‌ها را ندارد زیرا ولتاژ خروجی میکروکنترلر ۵ ولت و حداکثر جریان خروجی آن ۲۵ میلی‌آمپر می‌باشد، حتی اگر بخواهید همان LED را روشن و خاموش کنید حداکثر دو LED را به صورت سری می‌توانید کنترل کنید پس برای آنکه وسایل با ولتاژ و جریان بیشتر را کنترل کنیم (وسایل توان بالا) باید از میکروکنترلر فقط به عنوان سوئیچ استفاده کنیم یعنی اگر قصد کنترل یک بار بزرگ را داشته باشیم باید مداری طراحی کنیم که میکروکنترلر وظیفه‌ی خاموش و روشن کردن کلید متصل به بار را داشته باشد (منظور از بار، وسیله‌ای است که توان زیادی مصرف می‌کند که در اینجا چون ولتاژ باتری ثابت است، طبق رابطه‌ی $P = VI$ توان متناسب با جریان است پس منظور از بار بزرگ وسیله‌ای است که جریان زیادی مصرف می‌کند).

عملکرد ترانزیستورها در حالت کلیدزنی

فرض کنید یک لامپ ۱۲ ولتی در اختیار دارید و قصد دارید روشن و خاموش شدن آن را به وسیله‌ی میکروکنترلر کنترل کنید. همانطور که می‌دانید ولتاژ خروجی میکروکنترلر ۵ ولت است و نمی‌توان لامپ را به صورت مستقیم به آن متصل کرد پس باید مداری طراحی کنیم که میکروکنترلر تنها وظیفه‌ی روشن و خاموش کردن کلید متصل به لامپ را داشته باشد. مدار شکل ۱۳-۱ را در نظر بگیرید که در آن یک سر لامپ به منبع ۱۲ولت و سر دیگر آن به وسیله‌ی یک ترانزیستور به زمین متصل شده است، در این حالت ترانزیستور نقش یک کلید را ایفا می‌کند. زمانی‌که ترانزیستور روشن است جریان امیتر و کلکتور تقریباً با هم برابر می‌شوند (یعنی هرجریانی که از کلکتور عبور می‌کند از امیتر هم می‌گذرد) پس در این حالت جریان از باتری به سمت لامپ جاری می‌شود سپس وارد کلکتور ترانزیستور شده و از امیتر آن وارد زمین شده و لامپ روشن می‌شود و زمانی‌که ترانزیستور خاموش باشد کلکتور و امیتر به هم متصل نیستند پس مدار قطع و لامپ خاموش است.

خاموش و روشن شدن ترانزیستور توسط میکروکنترلر تعیین می‌شود یعنی زمانی‌که ولتاژ خروجی میکروکنترلر برابر ۵ ولت است (یا به عبارتی پایه‌ی خروجی میکروکنترلر یک است) بیس تحریک شده و ترانزیستور روشن می‌شود و زمانی‌که ولتاژ خروجی میکروکنترلر صفر است ترانزیستور خاموش است:

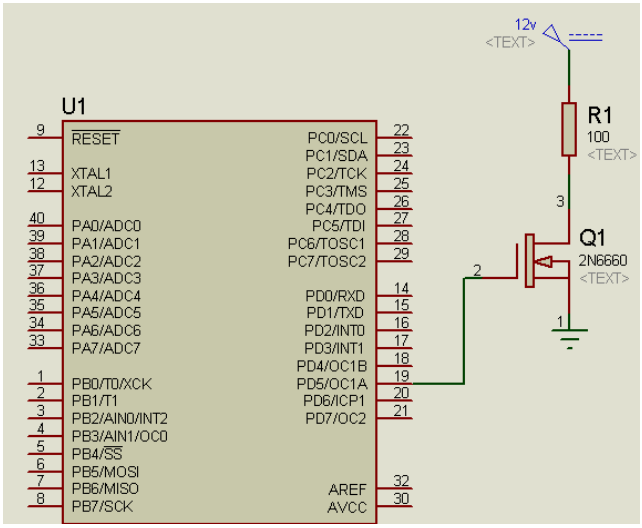


شکل ۱۳-۱ : عملکرد میکروکنترلر به عنوان تحریک ترانزیستور

در مدار فوق از ترانزیستور BJT به عنوان کلید استفاده کردیم، این ترانزیستورها برای روشن شدن به جریان بیس احتیاج دارند یعنی زمانی‌که ولتاژ خروجی میکروکنترلر را یک می‌کنیم بیس این ترانزیستورها جریانی از میکروکنترلر می‌کشند و روشن می‌شوند و اگر از یک مقاومت برای محدود کردن جریان آن استفاده نکنیم ترانزیستور می‌سوزد.

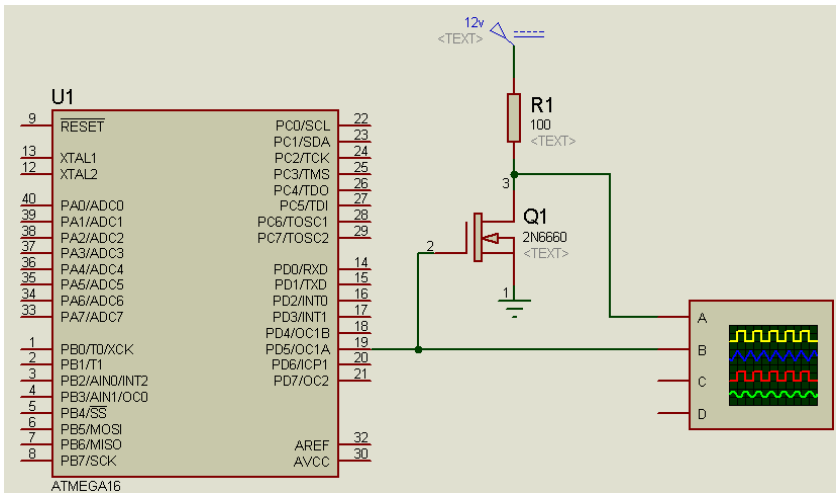
نوع دیگری از ترانزیستور که به عنوان کلید استفاده می‌شود ماسفت است. ماسفت یک نوع ترانزیستور می‌باشد که دارای ۳ پایه به نام‌های گیت، درین و سورس است. ماسفت‌ها دارای دو نوع N-Channel و P-Channel هستند. ماسفت N-Channel زمانی روشن می‌شود که ولتاژ گیت آن از ولتاژ سورس به اندازه‌ای مشخص بیشتر باشد، این اندازه می‌تواند از ماسفتی تا ماسفت دیگر متفاوت باشد که این مقدار درون برگه‌ی مشخصات فنی (Datasheet) هر ماسفت نوشته شده است. در حقیقت با اعمال این ولتاژ کانالی بین سورس و درین باز شده و درین و سورس به هم متصل می‌شوند یا به عبارتی اگر ماسفت را به عنوان کلید در نظر بگیریم با اعمال این ولتاژ کلید بسته می‌شود (در ماسفت‌های معمولی این مقدار می‌تواند حدوداً بین ۲ تا ۵ ولت باشد که به راحتی می‌توان توسط میکروکنترلر این ماسفت ها را روشن و خاموش کرد). ماسفت P-Channel نیز زمانی روشن می‌شود که ولتاژ سورس آن از ولتاژ گیت به اندازه‌ای مشخص بیشتر باشد.

برای مثال فرض‌کنید می‌خواهیم به باری با مقاومت ۱۰۰ اهم ولتاژی مربعی (PWM) بدهیم، چون این بار جریانی حدود 120mA می‌خواهد نمی‌توان آن را مستقیماً به میکروکنترلر متصل کرد پس از یک ماسفت نوع n استفاده می‌کنیم که در حالت کلیدزنی کار کند (در پروتئوس یک ماسفت به نام 2N660 را انتخاب می‌کنیم، از هر ماسفت یا ترانزیستور دیگری هم باتوجه به جریان قابل تحملش می‌توانستیم استفاده کنیم):



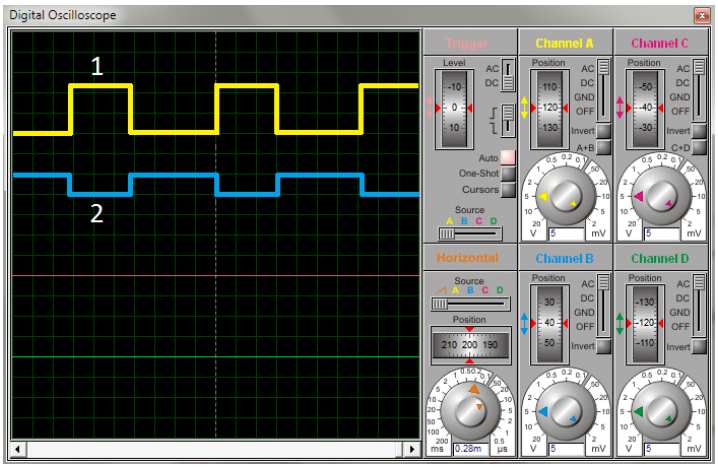
شکل ۱۳-۲: اتصال ماسفت به میکروکنترلر بعنوان سوییچ

برای آنکه ولتاژخروجی میکروکنترلر و ولتاژی که توسط کلید (ماسفت) به پایین بار می‌رسد را مشاهده کنیم یک پایه‌ی اسیلوسکوپ را به پایه‌ی خروجی میکروکنترلر و یک پایه‌ی دیگر آن را به پایین بار وصل می‌کنیم:



شکل ۱۳-۳: مشاهده‌ی عملکرد ماسفت با اسیلوسکوپ

با اجرای برنامه ولتاژها را به کمک اسیلوسکوپ مشاهده می‌کنیم:

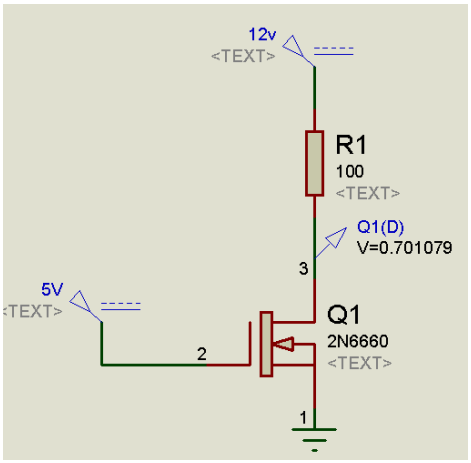


شکل ۱۳-۴: ولتاژ گیت و درین ماسفت

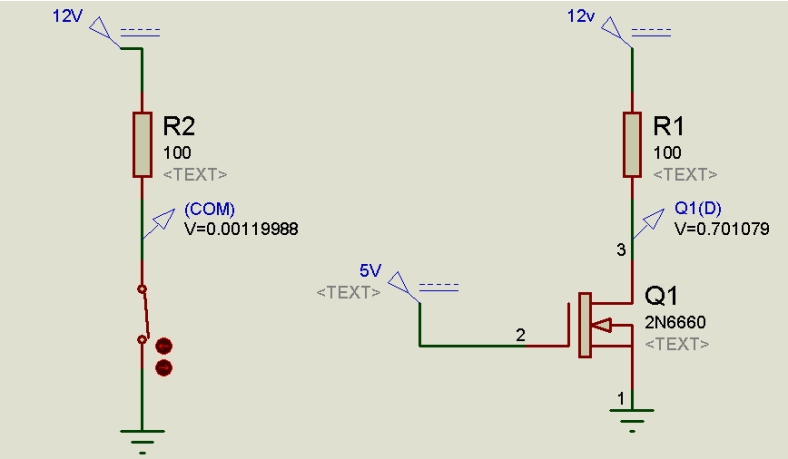
همانطور که در تصویر بالا مشخص است میکروکنترلر ولتاژ ۵ ولت را به گیت ماسفت اعمال می‌کند (شکل موج ۲) و ولتاژی که توسط عملیات کلید زنی (سوئیچینگ) ماسفت به پایین بار می‌رسد حدود ۱۶ ولت است (شکل موج ۱).

به این نکته توجه کنید که در این حالت ترانزیستور هیچ‌گونه تقویتی انجام نمی‌دهد و فقط مانند یک کلید عمل می‌کند زیرا اصلاً ترانزیستور در این حالت نمی‌تواند ولتاژ DC را تقویت کند (که این ولتاژ مربعی هم در واقع یک ولتاژ DC است که به طور پشت سر هم صفر و ۵ ولت می‌شود)، ترانزیستور فقط در حالت تفاضلی (دیفرانسیل) است که می‌تواند ولتاژ DC را تقویت کند (که ما در این فصل قصد تقویت ولتاژ را نداریم و برای تحویل ولتاژ به بار از عملیات کلیدزنی(سوئچینگ) استفاده می‌کنیم).

برای مشاهده‌ی همین عملکرد به طور جداگانه مدار زیر را می‌بندیم و به گیت ماسفت ولتاژ ۵ ولت را اعمال می‌کنیم و توسط ولتمتر ولتاژ پایین بار را می‌خوانیم:

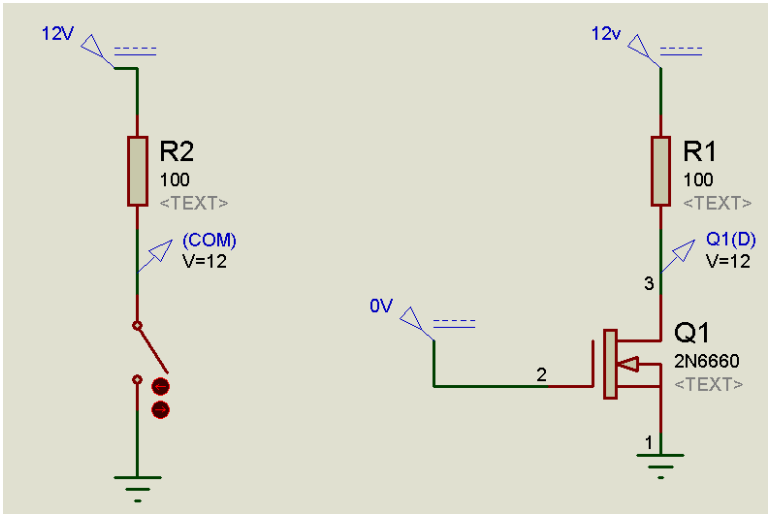


همانطور که در شکل زیر مشخص است ولتاژ پایین بار نزدیک صفر ولت است و همانند آن است که به جای ماسفت یک کلید گذاشته‌ایم و آن کلید را بسته‌ایم:



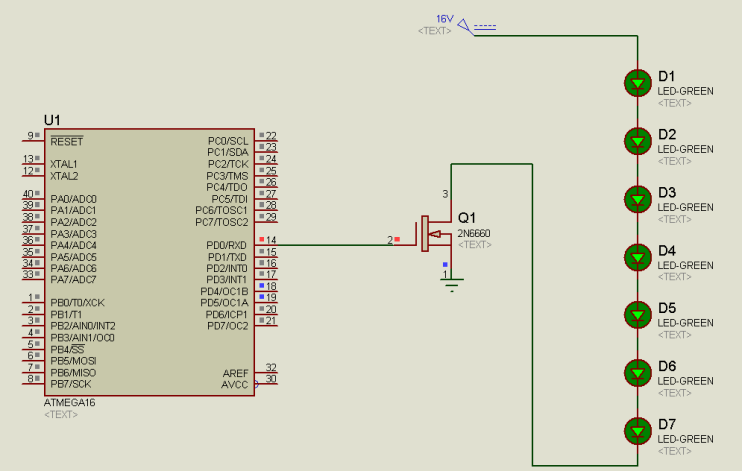
شکل ۱۳-۶: ماسفت معادل کلید(در حالت کلید بسته)

و اگر به گیت ولتاژ صفر ولت را اعمال کنیم همانند آن است که کلید را باز کرده‌ایم:



شکل ۱۳-۷: ماسفت معادل کلید(درحالت کلید باز)

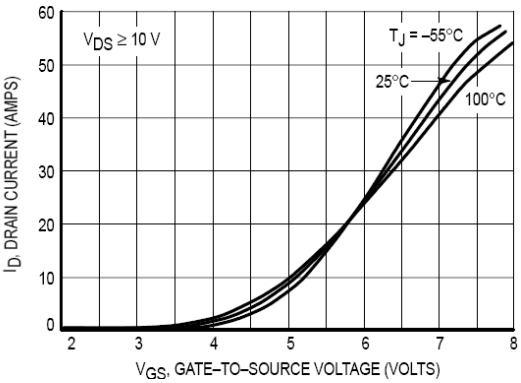
برای مثال فرض کنید می‌خواهیم تعدادی LED را به صورت سری روشن و خاموش کنیم، برای انجام این کار می‌توانیم مداری مانند شکل ۱۳-۸ را ببندیم:



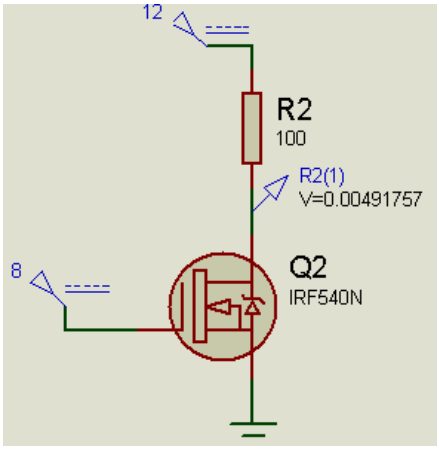
شکل ۱۳-۸: کنترل تعدادی LED به صورت سری

برای استفاده از ترانزیستور به عنوان کلید باید جریان عبوری از آن را در نظر بگیریم، برای مثال ترانزیستورهای عادی (BJT یا MOSFET) جریانی در حدود چند صد میلی‌آمپر را تحمل می‌کنند و اگر بخواهیم توسط میکروکنترلر بارهایی با جریان چند آمپر را کنترل کنیم باید از ترانزیستورهای قدرت استفاده کنیم. ترانزیستورهای قدرت از نوع BJT و MOSFET نیز وجود دارد که تحمل جریان‌هایی تا چند صد آمپر را دارند.

ما در اینجا به معرفی کوتاهی از ماسفت‌های قدرت (Power Mosfet) می‌پردازیم، برای مثال یکی از ماسفت‌های قدرت موجود در بازار IRF540N است که اگر به برگه‌ی مشخصات فنی (datasheet) آن رجوع کنیم، نمودار زیر را مشاهده می‌کنیم که جریان درین را بر حسب ولتاژ گیت-سورس رسم کرده است که طبق شکل در ولتاژهای کمتر از ۴ولت کاملاً جریان درین صفر است یعنی ماسفت خاموش و کانال قطع است و در ولتاژهای بالای ۷ولت ماسفت روشن و کانال کاملاً باز است:



پس برای آنکه این ماسفت قدرت را روشن کنیم ولتاژی حدود ۸ ولت بین گیت و سورس آن اعمال می‌کنیم و برای خاموش کردن آن ولتاژ بین گیت-سورس را کمتر از ۲ ولت قرار می‌دهیم.
همانطور که می‌دانید ولتاژ خروجی میکروکنترلرها ۵ ولت است و این ماسفت‌ها با ۵ ولت روشن نمی‌شوند. برای مثال شکل زیر یک ماسفت قدرت کانال N را نشان می‌دهد که چون سورس آن زمین شده است اگر ولتاژ ۸ ولت به گیت آن اعمال کنیم در حالت کلیدزنی (حالت روشن کلید) قرار می‌گیرد:



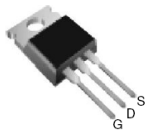
شکل ۱۳-۱۰: ماسفت قدرت

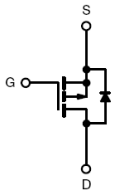
برای استفاده از هر ماسفتی باید ابتدا به دیتاشیت آن مراجعه و ولتاژ موردنیاز برای روشن و خاموش شدن آن را مشاهده‌کنیم، برای مثال IRF9640 یک ماسفت قدرت کانال P است؛ اگر به دیتاشیت آن نگاهی بیندازیم:

Power MOSFET

PRODUCT SUMMARY		
V_{DS} (V)	- 200	
$R_{DS(on)}$ (Ω)	V_{GS} = - 10 V	0.50
Q_g (Max.) (nC)	44	
Q_{gs} (nC)	7.1	
Q_{gd} (nC)	27	
Configuration	Single	

TO-220AB





P-Channel MOSFET

FEATURES

- Dynamic dV/dt Rating
- Repetitive Avalanche Rated
- P-Channel
- Fast Switching
- Ease of Paralleling
- Simple Drive Requirements
- Compliant to RoHS Directive 2002/95/EC

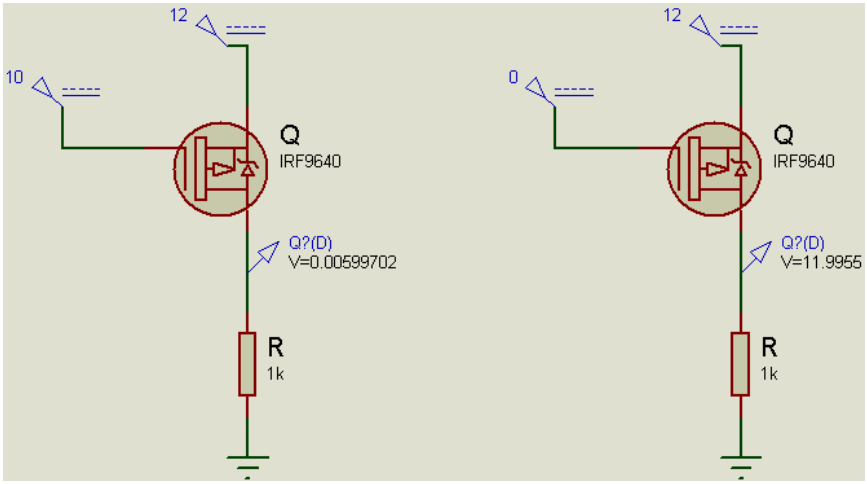
Available
RoHS* COMPLIANT

DESCRIPTION

Third generation Power MOSFETs from Vishay provide the designer with the best combination of fast switching, ruggedized device design, low on-resistance and cost-effectiveness.
The TO-220AB package is universally preferred for all commercial-industrial applications at power dissipation levels to approximately 50 W. The low thermal resistance and low package cost of the TO-220AB contribute to its wide acceptance throughout the industry.

شکل ۱۳-۱۱: دیتاشیت یک ماسفت قدرت

ولتاژ گیت-سورس مورد نیاز برای روشن شدن ماسفت 10 v- نوشته شده است، یعنی ولتاژ سورس باید ۱۰ ولت بیشتر از ولتاژ گیت باشد تا ماسفت روشن شود پس سورس را باید به بیشترین ولتاژ مدار وصل کنیم (به مثبت باتری). برای مثال اگر باتری ۱۲ ولتی در اختیار داشته باشیم زمانی‌که ولتاژ گیت کمتر از ۲ولت باشد اختلاف پتانسیل سورس و گیت ۱۰ ولت می‌شود و ماسفت روشن می‌شود و زمانی‌که ولتاژ گیت بیشتر از ۲ولت باشد ماسفت خاموش می‌شود:



شکل ۱۳-۱۲: عملکرد ماسفت قدرت

برای آنکه این ماسفت‌ها را به وسیله‌ی میکروکنترلر و ولتاژ ۵ ولت خروجی آن کنترل کنیم باید به نحوی این ولتاژ ۵ ولت را به ولتاژی حدود ۱۲ ولت تبدیل کنیم. آپ‌امپ(Op.Amp) یا تقویت کننده‌ی عملیاتی قطعه‌ای الکترونیکی است که می‌تواند این کار را انجام دهد.

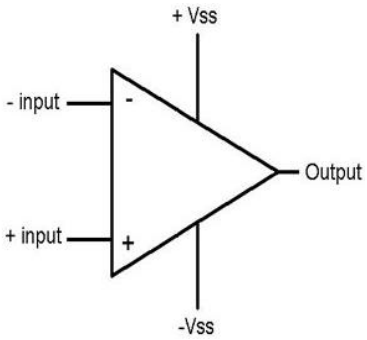
قطعات کاربردی (آپ‌امپ(Op Amp) و دیود زنر)

آپ-امپ می‌تواند به دو صورت بسته شود: با فیدبک و بدون فیدبک(در این بحث ما از نوع بدون فیدبک آن استفاده می‌کنیم).

آپ‌امپ دارای دو ورودی و یک خروجی و دو ولتاژ تغذیه می‌باشد:

ولتاژ خروجی آپ‌امپ از رابطه‌ی زیر بدست می‌آید:

$$V_O = A * (V_+ - V_-)$$

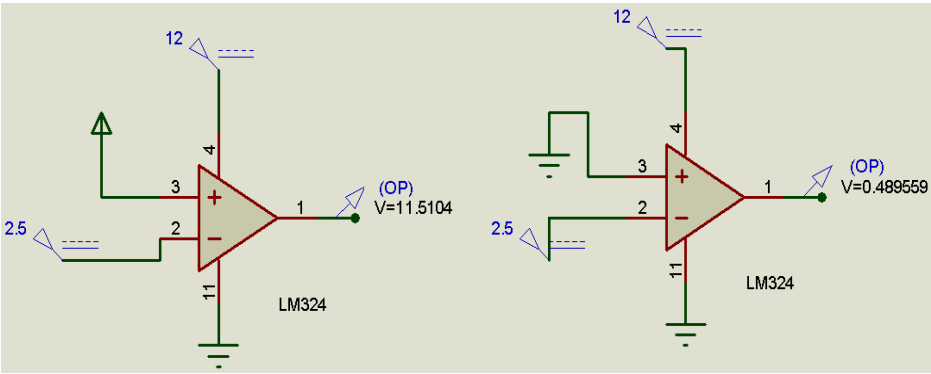


شکل ۱۳-۱۳: تقویت کننده‌ی عملیاتی (آپ -امپ)

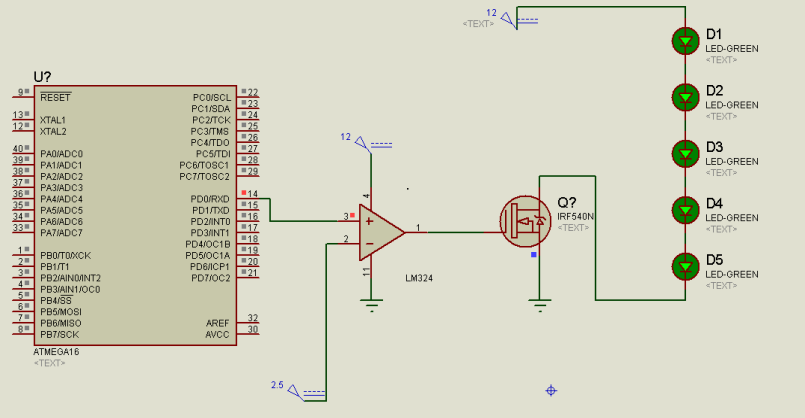
که A گین آپامپ می‌باشد و عدد بسیار بزرگی است که در حالت ایده‌آل بی‌نهایت و در حالت غیر ایده‌آل مثلاً عددی حدود ۶۰۰۰ می‌باشد (گین هر آپامپ در دیتاشیت آن نوشته شده است).حال سوالی که ممکن است پیش بیاید این است که اگر ولتاژ پایه‌ی مثبت از پایه‌ی منفی بیشتر باشد مثلاً پایه‌ی مثبت ۱ولت از پایه‌ی منفی بیشتر باشد ولتاژ خروجی چه عددی می‌شود، آیا ۶۰۰۰ ولت می‌شود؟

$$V_O = A * (V_+ - V_-) = 6000 * 1 = 6000$$

خیر. در این حالت چون ولتاژ خروجی از ولتاژ تغذیه‌ی آپامپ بیشتر است، آپامپ به اشباع می‌رود و ولتاژ خروجی برابر ولتاژ تغذیه می‌شود پس در حالت بدون فیدبک اگر ولتاژ پایه‌ی مثبت از ولتاژ پایه‌ی منفی بیشتر باشد ولتاژ خروجی برابر ولتاژ تغذیه می‌شود و اگر ولتاژ پایه‌ی منفی از ولتاژ پایه‌ی مثبت بیشتر باشد ولتاژ خروجی برابر ولتاژ تغذیه منفی (-Vss) می‌شود که اگر ولتاژ تغذیه‌ی منفی را صفر ولت انتخاب کنیم، زمانی‌که ولتاژ پایه‌ی منفی بیشتر از پایه‌ی مثبت باشد ولتاژ خروجی برابر صفر ولت می‌شود. برای مثال اگر بخواهیم مداری طراحی کنیم که ولتاژ ۵ ولت میکروکنترلر را به ۱۲ ولت تبدیل کند، خروجی میکروکنترلر را به پایه‌ی مثبت آپامپ وصل کرده و ولتاژ پایه‌ی منفی آپامپ را برابر ۲,۵ ولت قرار می‌دهیم در این صورت اگر پایه‌ی مثبت که به میکروکنترلر متصل است برابر ۵ ولت باشد ولتاژ پایه‌ی مثبت بیشتر از پایه‌ی منفی است و ولتاژ خروجی برابر ولتاژ تغذیه می‌شود و زمانی‌که ولتاژ خروجی میکروکنترلر صفر باشد چون پایه‌ی منفی ۲,۵ ولت بوده و ولتاژ پایه‌ی منفی بیشتر از پایه‌ی مثبت است ولتاژ خروجی صفر ولت می‌شود. یعنی اگر ورودی ۵ ولت باشد خروجی ۱۲ ولت و اگر ورودی صفر ولت باشد خروجی صفر ولت می‌شود. در شبیه‌ساز پروتئوس می‌توانید از آپامپ LM324 استفاده کنید، که پایه‌ی شماره‌ی ۴ آن پایه‌ی تغذیه‌ی Vss و پایه‌ی ۱۱ آن Vss– می‌باشد که به زمین متصل می‌شود:

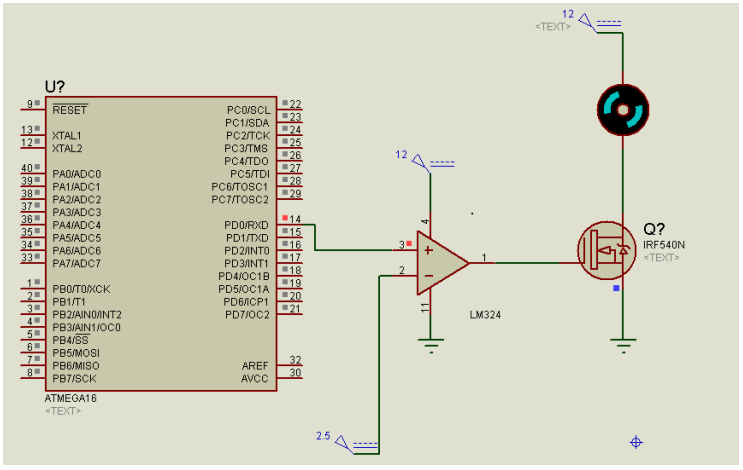


اگر بخواهیم یک مثال بسیار ساده بزنیم می‌توانیم تعدادی LED را با استفاده از ماسفت قدرت و با تقویت ولتاژ میکروکنترلر به وسیله‌ی آپ‌امپ روشن کنیم:

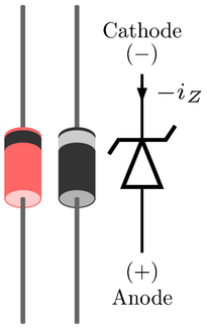


شکل ۱۳-۱۵: کنترل تعدادی LED به کمک آپ‌امپ و ماسفت قدرت

و یا اگر بخواهیم یک موتور را در یک جهت به چرخش در بیاوریم می‌توانیم از آپ‌امپ استفاده کنیم:



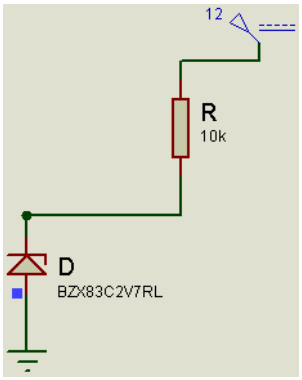
شکل ۱۳-۱۶: کنترل چرخش یک موتور به وسیله‌ی آپ‌امپ و ماسفت قدرت



برای تولید ولتاژ ۲٫۵ ولت باید از یک دیود زنر استفاده کنیم. دیود زنر یک نوع دیود است که اگر به صورت برعکس بایاس شود (یعنی جریان را به جای آنکه از سمت آند به سمت کاتد برقرار کنیم از سمت کاتد به آند برقرار کنیم) ولتاژی برابر ولتاژ شکست در دو سر آن قرار می‌گیرد.

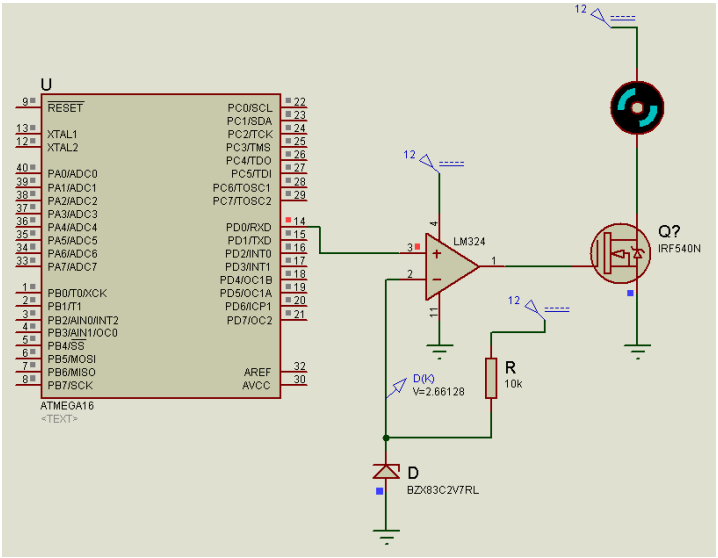
دیود زنر اگر به صورت مستقیم بایاس شود ولتاژ دو سر آن حدود ۰٫۷ ولت می‌شود و اگر به صورت معکوس بایاس شود ولتاژ دو سر آن برابر ولتاژ شکست خواهد شد. ولتاژ شکست هر دیود زنر متفاوت است مثلاً در این مثال از یک دیود زنر به نام BZX83C2V7 استفاده می‌کنیم که ولتاژ شکست آن ۲٫۷ ولت است (ولتاژ شکست هر دیود زنر در دیتاشیت آن موجود می‌باشد).

برای آنکه دیود زنر به صورت معکوس بایاس شود باید یک جریان از سمت کاتد به آند آن جاری شود، در آن صورت ولتاژ ۲٫۷ ولت بین کاتد و آند قرار می‌گیرد. برای این کار آند را به زمین و کاتد را با یک مقاومت به ولتاژ مثبت (در اینجا ۱۲ ولت) متصل می‌کنیم که جریانی از سمت کاتد به آند جاری شود:



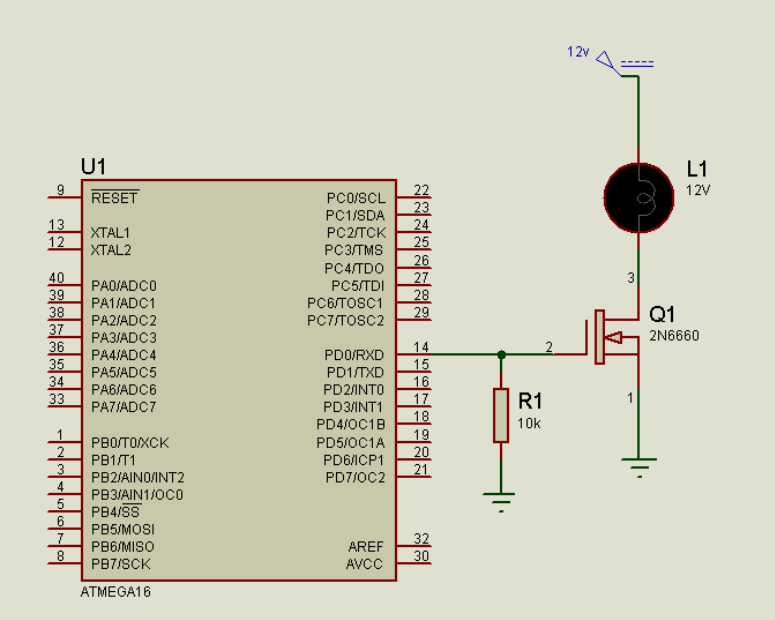
شکل ۱۳-۱۸: بایاس یک دیود زنر

در این‌صورت ولتاژ کاتد ۲٫۷ ولت می‌شود که می‌توان آن را به پایه‌ی منفی آپ‌امپ متصل کرد:



شکل ۱۳-۱۹: کنترل موتور با استفاده از دیود زنر و آپ‌امپ و ماسفت قدرت

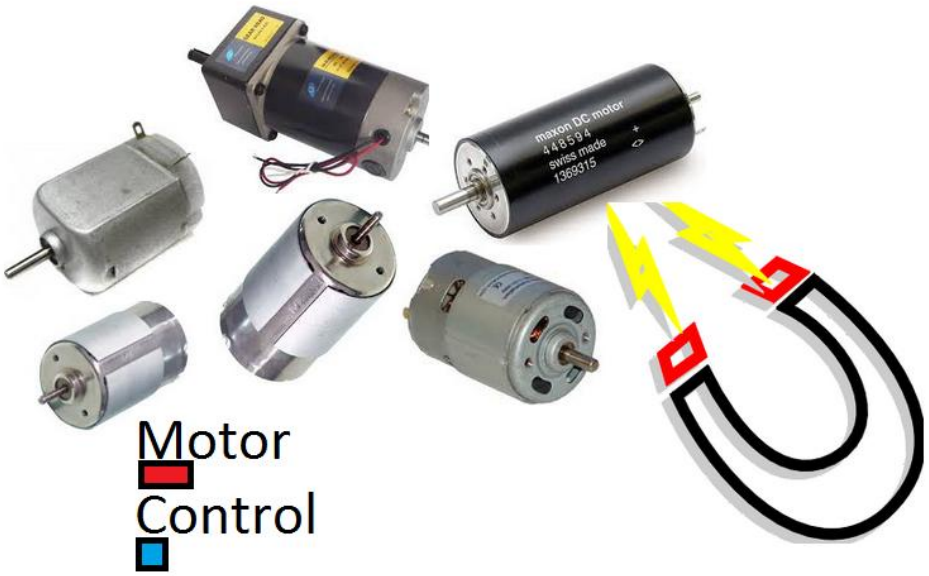
نکته‌ای که باقی می‌ماند این است که اگر از ماسفت کانال N به عنوان کلید استفاده کردیم باید حتماً پایه‌ی گیت آن را با یک مقاومت به زمین متصل کنیم(شکل ۱۳-۲۰). اگر این کار را انجام ندهیم زمانی که ولتاژ ورودی به گیت ماسفت قطع باشد ولتاژ گیت برابر ولتاژ هوا می‌شود که با هر نویز می‌تواند کم و زیاد شود که در این صورت ماسفت به صورت پشت سر هم خاموش و روشن می‌شود که باعث می‌شود ماسفت به شدت داغ شود و در نهایت بسوزد.



شکل ۱۳- ۲۰: اتصال گیت با یک مقاومت به زمین

فصل چهاردهم

کنترل موتور

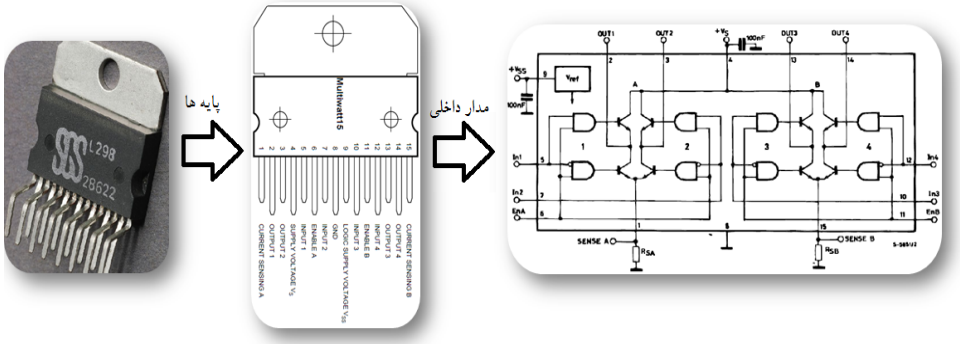


در این فصل خواهیم خواند:

۱. آشنایی با پل H
۲. طراحی درایور با مدل پل H
۳. نکات ایمنی در طراحی

کنترل موتور

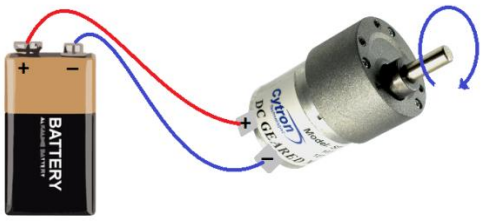
برای کنترل هر موتور به یک درایور نیاز داریم که این درایور باید هم جهت چرخش موتور و هم سرعت آن را کنترل‌کند. تا اینجا با درایور L298 آشنا شدیم و نحوه‌ی استفاده از آن را آموختیم ، ولی این درایور حداکثر جریان خروجی ۲ آمپر را تحمل می‌کند و جریان بیشتر از آن را نمی‌تواند تحویل دهد، لذا اگر قصد کار با موتور بزرگتر را داشته باشیم (جریان‌کشی موتور بیشتر باشد) نمی‌توانیم از L298 استفاده‌کنیم و باید یک درایور موتور طراحی‌کنیم. یکی از راه‌ها برای کنترل موتورDC استفاده از پلH (H-bridge) می‌باشد. در شکل زیر یک درایور L298 به همراه پایه‌ها و مدار داخلی آن مشاهده می‌کنید:



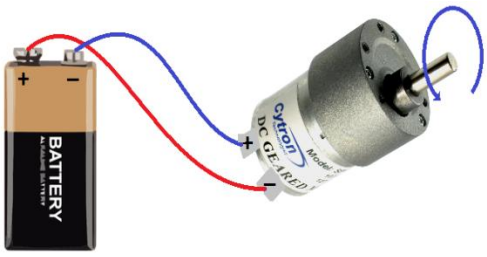
شکل ۱۴-۱ : درایور L298

پلH (H-Bridge)

اکثر موتورهای DC توانایی چرخیدن در دو جهت مختلف را دارا می‌باشند که این جهت بستگی به نوع اتصال موتور به باتری دارد. همانطورکه می‌دانید باتری دارای دو سر مثبت و منفی است () و موتورهای DC هم دارای دو سر مثبت و منفی می‌باشند(البته به صورت قراردادی تعیین می‌شود)، اگر سر مثبت باتری را به سر مثبت موتور و سر منفی باتری را به سر منفی موتور متصل کنیم موتور در جهت رو به جلو (قرارداد: رو به جلو معادل ساعتگرد است) می‌چرخد، و اگر این سرها را جابه‌جا متصل کنیم، جهت چرخش موتور هم برعکس می‌شود.

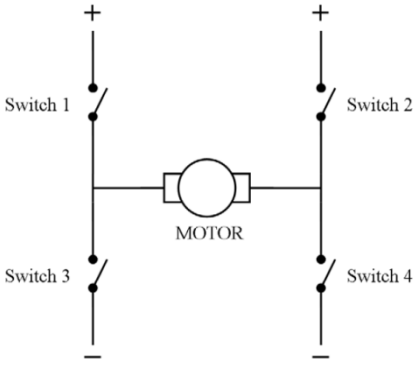


شکل ۱۴-۲ : اتصال باتری به موتور به شکل مستقیم

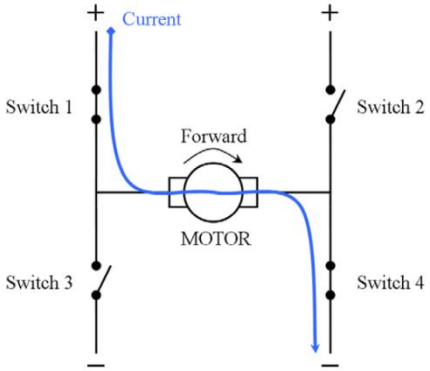


شکل ۱۴-۳ : اتصال باتری به موتور به شکل معکوس

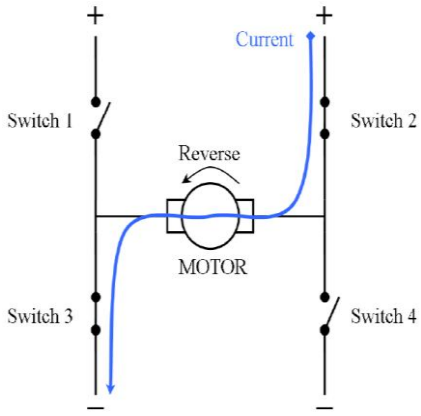
پلH یک مدار بسیار ساده است که به ما اجازه می‌دهد با سیگنال ولتاژ ارسالی از مدار کنترلی خود موتور را در جهت دلخواه بچرخانیم.



با قطع و وصل‌کردن این کلیدها می‌توان جهت چرخش موتور را کنترل‌کرد. اگر کلید ۱و۴ را وصل‌کنیم، موتور رو به جلو حرکت می‌کند:



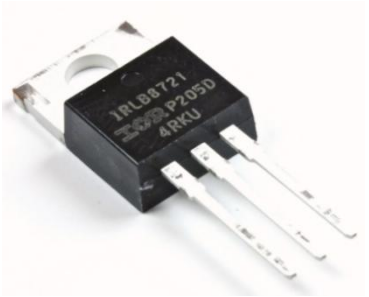
و اگر کلید ۲ و ۳ را وصل کنیم، موتور در جهت معکوس می‌چرخد:



اکنون سوالی که پیش می‌آید این است که چطور می‌توانیم این کلیدها را قطع و وصل نمائیم؟

پاسخ: با استفاده از ترانزیستورهای قدرت.

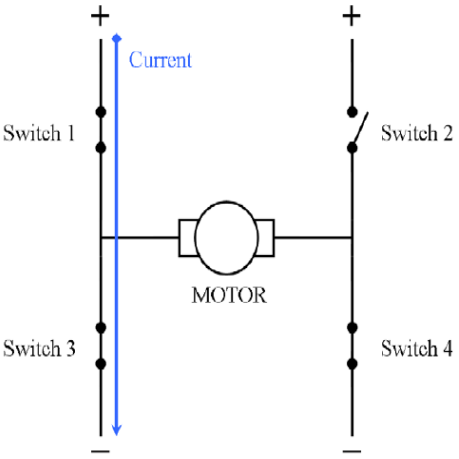
با عملیات سوئیچینگ ترانزیستورها در فصل توان آشنا شدیم. پس برای اینکار از ترانزیستور استفاده می‌کنیم و چون ترانزیستورهای معمولی دارای توان کمتری هستند و جریان قابل تحمل پایینی دارند ، بهترین راه استفاده از ترانزیستور های قدرت می‌باشد.



شکل ۱۴-۷ : ماسفت **IRLB8721**

شکل ۱۴- ۷ مربوط به یک ماسفت قدرت با نام IRLB8721 می‌باشد (یک ماسفت قدرت کانال Nبا ولتاژ و جریان ۳۰V و ۶۰A).

همانطورکه گفته شد در پل H از ماسفت به‌عنوان کلید استفاده می‌کنیم، در این صورت باید مواظب برخی حالت‌های خاص مانند حالت شکل روبرو باشیم:



در این حالت کلید های ۱ و ۳ بسته شده‌اند که باعث می‌شود مثبت و منفی مدار به هم متصل شوند (اتصال کوتاه اتفاق می‌افتد)، در این صورت اگر محافظت‌های لازم صورت نگیرد مدار می‌سوزد.

حال چطور باید این ماسفت‌های قدرت را روشن و خاموش کرد؟ چون این ماسفت‌ها برای روشن‌شدن به ولتاژی در حدود ۸ تا ۱۶ ولت بین گیت و سورس احتیاج دارند و چون ولتاژ خروجی میکروکنترلرها و آی‌سی‌های منطقی هم برابر ۵ ولت می‌باشد، باید به نحوی این ۵ ولت را به ولتاژ بالاتر تبدیل کنیم. همانطورکه در فصل توان گفته شد برای انجام این کار از یک آپ‌امپ استفاده می‌کنیم (البته راه‌حل‌های دیگری هم وجود دارد).

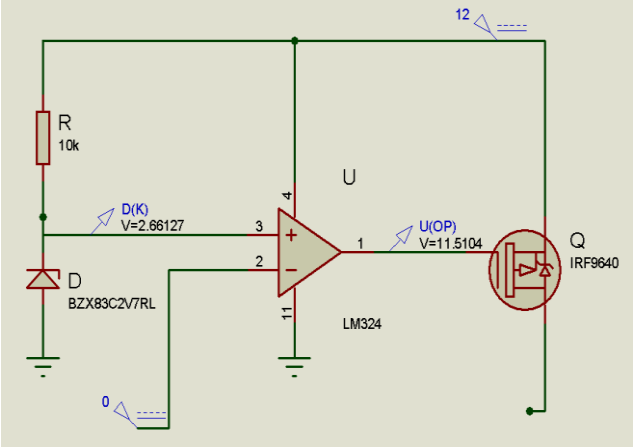
طراحی درایور با مدل پل H

مدار درایور موتور پلH دارای ۴ عدد کلید جهت کنترل بوده که باید در یک حالت کلیدهای ۱ و۴ و در حالت دیگر کلیدهای ۲ و ۳ بسته شوند. برای دو کلید بالایی از دو ماسفت کانال p (P-channel) استفاده می‌کنیم، چرا؟ چون با توجه به اینکه ماسفت‌ها را با تغییر ولتاژ گیت و سورس‌شان می‌توان روشن و خاموش کرد، به همین علت یک پایه باید ولتاژش ثابت باشد (پایه‌ی سورس) تا با تغییر پایه‌ی دیگر (پایه‌ی گیت) ماسفت را خاموش و روشن کنیم. بنابراین ولتاژ سورس را ثابت نگه داشته و با تحریک گیت، ماسفت را روشن و خاموش می‌کنیم. اگر ماسفت بالایی از نوع کانال N می‌بود دراین‌صورت سورس آن به موتور متصل می‌گردید و ولتاژ متغیری پیدا می‌کرد، ولی اگر از نوع کانال P باشد سورس به ولتاژ تغذیه متصل شده و همیشه ثابت باقی می‌ماند.

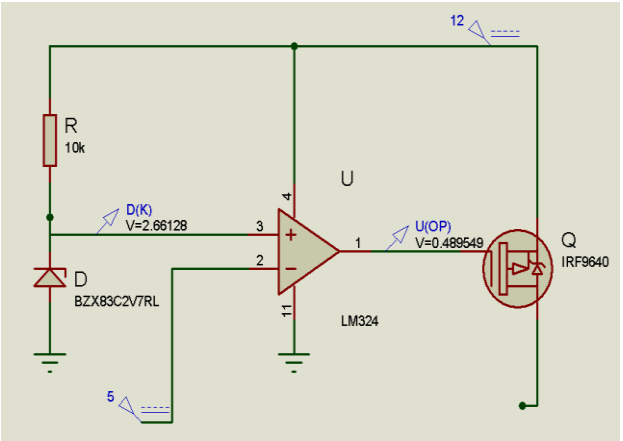
ما برای طراحی درایور پلH از یک ماسفت قدرت کانال P به نام IRF9640 و یک آپ‌امپ به نام LM324 استفاده می‌کنیم (البته راه‌های مختلفی برای طراحی درایور وجود دارد). در فصل توان توضیح دادیم که در آپ‌امپ اگر ولتاژ پایه‌ی مثبت بیشتر از پایه‌ی منفی باشد، ولتاژ خروجی برابر VSS یا همان ولتاژ پایه‌ی ۴ می‌شود و اگر ولتاژ پایه‌ی منفی بیشتر باشد ولتاژ خروجی برابر صفر ولت می‌شود.

در اینجا بخاطر استفاده از ماسفت کانال P، سورس را به ۱۲ ولت وصل کردیم. در این حالت زمانی‌که به گیت ولتاژ ۱۲ ولت برسد، ماسفت خاموش و زمانی‌که ولتاژ گیت صفر ولت بشود، ماسفت روشن‌می‌شود (برای روشن شدن کانال بین درین و سورس باید ولتاژ گیت- سورس بیش از ۸ ولت شود). چون ما می‌خواهیم هر زمان که فرمان ۵ ولت از میکروکنترلر صادر شد، ماسفت روشن شود پس آپ‌امپ را به صورت NOT وصل می‌کنیم یعنی زمانی‌که ۵ ولت روی یکی از پایه‌های میکروکنترلر ایجاد شد، خروجی صفر و زمانی‌که صفر ولت شد، خروجی ۱۲ ولت شود (یعنی هرزمان پایه‌ی میکروکنترلر ۵ ولت یا یک شد، ولتاژ گیت صفر و ماسفت روشن شود) به همین خاطر ولتاژ پایه‌ی مثبت آپ‌امپ را برابر ولتاژ مثبتی بین ۰ و ۵ ولت (برای مثال ۲,۷ ولت) قرار داده و پایه‌ی منفی را به میکروکنترلر می‌زنیم، در این حالت هر وقت میکروکنترلر ولتاژ ۵ ولت

را ایجاد کند به دنبال آن پایه‌ی منفی ۵ ولت شده که در آن صورت از پایه‌ی مثبت کـه ۲,۷ ولـت بود بیشتر شده و در نهایت ولتاژ خروجی صفر ولت گردیده و ماسفت روشن می‌شود. در حالت دیگر یعنی زمانی‌که پایه‌ی میکروکنترلر صفر شود، ولتاژ پایه‌ی منفی صفر شده که در ایـن حالـت ولتـاژ پایه‌ی مثبت ۲,۷ ولت بیشتر از پایه‌ی منفی است، بنـابراین ولتـاژ گیـت ۱۲ولـت شـده و ماسـفت خاموش می‌شود. برای آنکه ولتاژ پایه‌ی مثبت را ۲,۷ ولت قرار بـدهیم از یـک دیـود زنـر اسـتفاده می‌کنیم که برای بایاس‌کردن دیود زنر، سمت کاتد آن را با یک مقاومت به ولتاژ تغذیه و سمت آند را نیز به زمین متصل می‌کنیم که یک جریان اندک از کاتد بـه سـمت آنـد برقـرار شـود و در ایـن شرایط دیود زنر معکوس بایاس شده و ولتاژ دو سر زنر ۲,۷ ولت می‌شود(یعنی ولتاژ کاتد ۲,۷ ولت می‌شود). حال اگر کاتد را به پایه‌ی مثبت آپ‌امپ وصل کنیم، ولتـاژ ۲,۷ ولـت روی پایـه‌ی مثبـت آپ‌امپ می‌افتد.

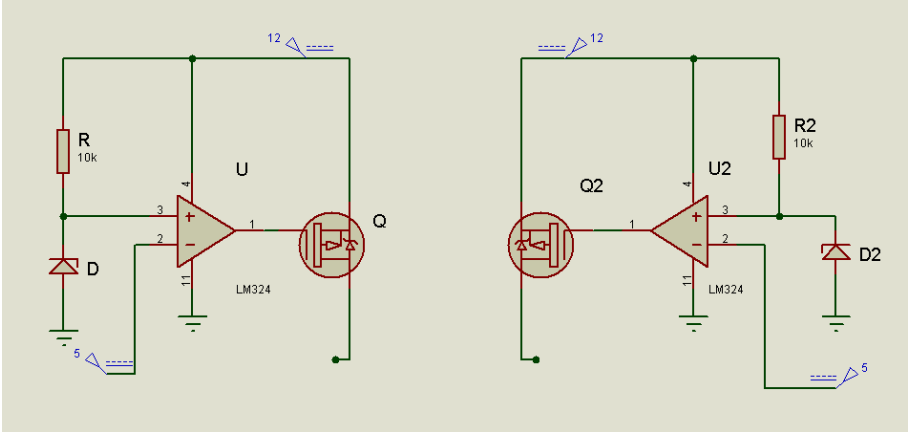


شکل ۱۴-۹ : ولتاژ کنترل صفر ولت و خروجی ۱۲ ولت



شکل ۱۴-۱۰ : ولتاژ کنترل ۱۲ ولت و خروجی صفر ولت

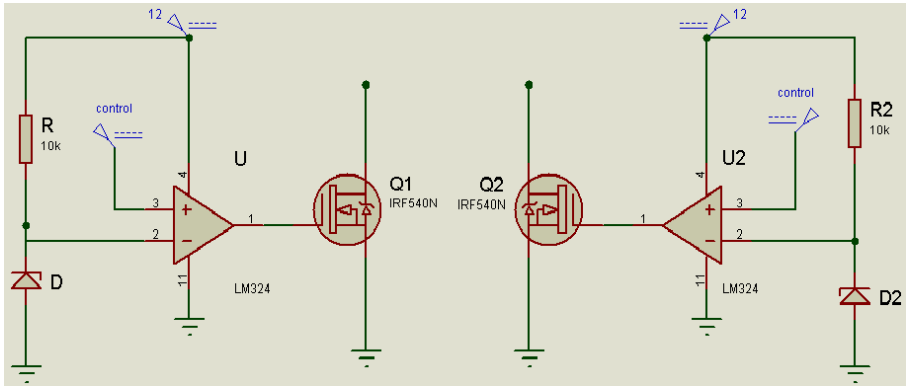
شکل زیر مربوط به هر دو ماسفت بالایی درایور است که قسمت بالای پل H را تشکیل می‌دهد:



شکل ۱۴-۱۱ : دو کلید بالای پل H

حال دو ماسفت پایین را چگونه باید متصل‌کرد؟

بهترین راه این است که برای قسمت پایین درایور از ماسفت کانال N استفاده شود و پایه‌ی سورس آن را به زمین وصل‌کرده و با تحریک گیت، ماسفت را خاموش و روشن کنیم. دراینجا ولتاژ ۲,۷ ولت را به پایه‌ی منفی آپ‌امپ می‌دهیم زیرا نیازی به NOT کردن ولتاژ نیست و زمانی‌که ولتاژ گیت ۱۲ ولت باشد، ماسفت روشن و زمانی‌که ولتاژ گیت صفر باشد، ماسفت خاموش است:

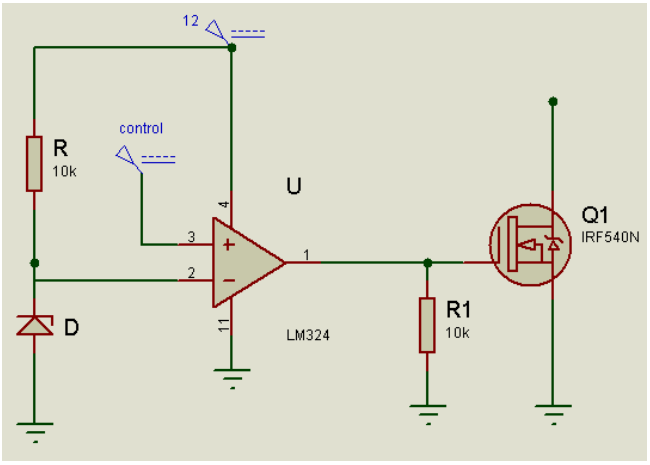


شکل ۱۴-۱۲ : دو کلید پایین پل H

یک نکته‌ی ایمنی : بزرگترین خطر برای ماسفت این است که گیت ماسفت شناور (Float) باشد. فرض‌کنید آپ‌امپ خاموش‌باشد، در این صورت ولتاژ خروجی آپ‌امپ یا همان ولتاژ گیت ماسفت دیگر نه صفر است و نه ۱۲ ولت، چون نه به زمین وصل شده و نه به ۱۲ ولت و هر ولتاژی می‌تواند داشته باشد و این باعث خاموش و روشن شدن آن می‌شود. در این شرایط احتمال سوختن ماسفت بسیار بالا می‌رود.

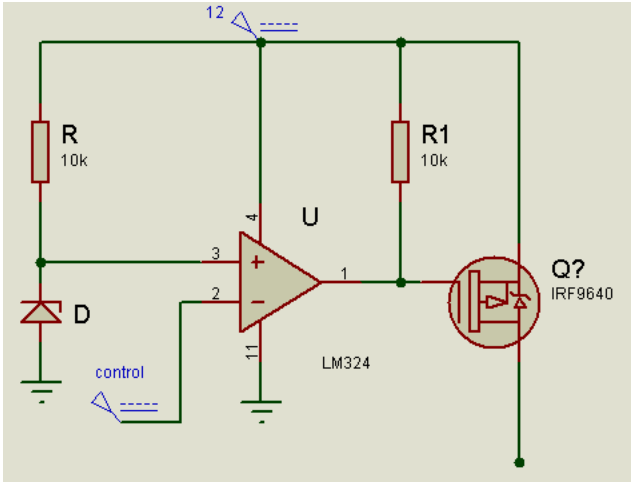
برای حل این مشکل چه باید کرد؟

پاسخ: برای ماسفت‌های کانال N پایه‌ی گیت ماسفت را با یک مقاومت بزرگ به زمین وصل می‌کنیم. در این صورت زمانی‌که آپ‌امپ خاموش است ولتاژ گیت صفر ولت شده (چون وقتی آپ‌امپ خاموش است، هیچ جریانی از آن نمی‌گذرد و با توجه به جریان نداشتن گیت جریان شاخه‌ی مقاومت صفر می‌شود پس افت ولتاژ روی مقاومت هم صفر شده و صفر ولت مستقیماً روی پایه‌ی گیت می‌افتد) و دیگر نمی‌تواند ولتاژ شناور داشته باشد:



شکل ۱۴-۱۳: **Pull-Down** کردن گیت (متصل کردن گیت به زمین با مقاومت)

برای ماسفت‌های کانال P هم گیت را با یک مقاومت به ولتاژ تغذیه وصل می‌کنیم:

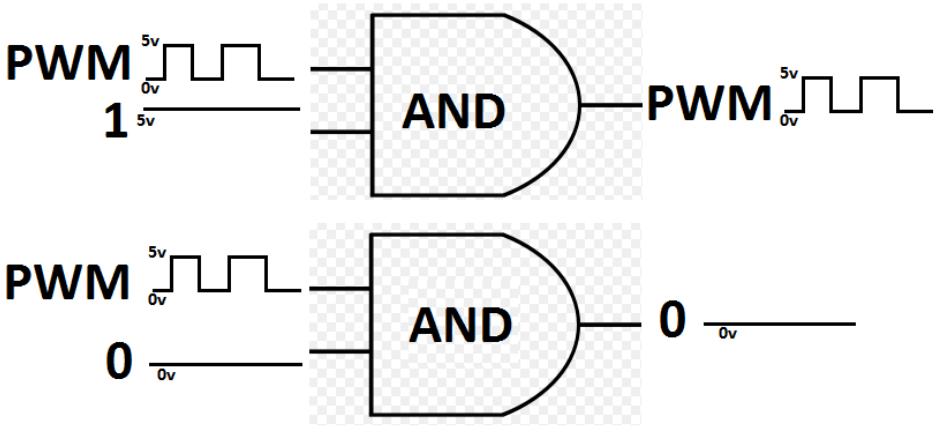


شکل ۱۴-۱۴: **Pull-Up** کردن گیت (متصل کردن گیت به **Vcc** با مقاومت)

تا به الان آموختیم که چهار عدد ماسفت را باید چگونه در مدار بگذاریم.حال PWM تولید شده توسط میکروکنترلر و خروجی‌های میکروکنترلر را باید به کجا وصل‌کنیم؟

پاسخ: اولاً می‌دانیم که خروجی‌های میکروکنترلر عبارت است از یک PWM و دو خروجی دیگر که جهت حرکت را مشخص می‌کنند که بایستی یکی از پایه‌های خروجی یک و دیگری صفر باشد, همچنین می‌دانیم از این چهار ماسفت باید حتماً دو ماسفت روشن و دو ماسفت دیگر خاموش باشند بنابراین بدین روش عمل می‌کنیم که دو خروجی میکروکنترلر را با PWM، And می‌کنیم. برای مثال اگر خروجی‌های میکروکنترلر PORTA.0 و PORTA.1 باشند، PORTA.0 And PWM را به دو ماسفت و PORTA.1 And PWM را نیز به دو ماسفت دیگر متصل می‌کنیم (منظور از وصل کردن به ماسفت‌ها، وصل‌کردن به آپ‌امپ های متصل به ماسفت‌ها است).

در این صورت چه اتفاقی می‌افتد؟ یکی از خروجی‌ها که صفر بود با PWM هم که And بشود باز هم صفر باقی می‌ماند پس به دو ماسفت ولتاژ صفر می‌رسد (یعنی دستور خاموش بودن) و خروجی دیگر که یک بود با PWM که And بشود خود PWM شده و به دو ماسفت دیگر مقدار PWM می‌رسد:



شکل ۱۴-۱۵ : And شدن PWM و ولتاژ کنترلی میکروکنترلر

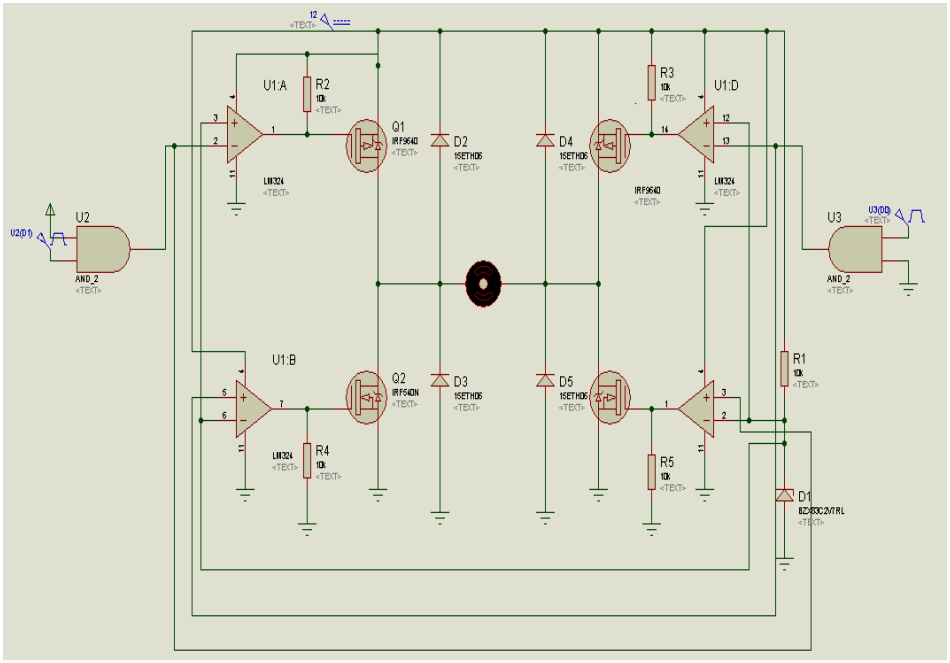
پس سه پایه‌ی خروجی از میکروکنترلر (PORTA.0 و PORTA.1 و PWM) به دو سیگنال تبدیل شدند (یکی PORTA.0 And PWM و یکی PORTA.1 And PWM).

حال این سیگنال ها چگونه به ماسفت‌ها وصل می‌شوند؟

پاسخ: چون کلیدهای یک و چهار با هم و کلیدهای دو و سه نیز با هم هستند یکی از سیگنال‌ها را به ماسفت‌های یک و چهار و آن یکی سیگنال را به ماسفت‌های دو و سه (که یا ماسفت‌های یک و چهار روشن شوند و ماسفت‌های دو و سه خاموش بمانند و یا برعکس) وصل می‌کنیم. پس

ورودی‌های آپامپ‌های یک و چهار به سیگنال یک و ورودی‌های دو و سه به سیگنال دو وصل می‌شوند.

در نهایت در طراحی درایور خود به شکل زیر می‌رسیم:

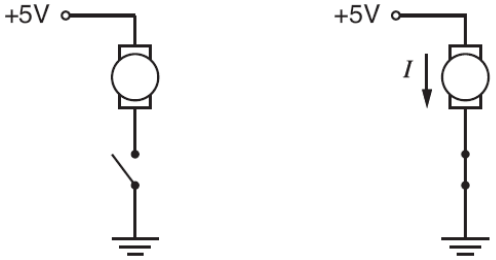


شکل ۱۴-۱۶ : نقشه‌ی کامل درایور موتور پل H

فقط در این شکل برای ایجاد ولتاژ ۲,۷ ولت یک دیود زنر گذاشته‌ایم که ۲,۷ ولت را تولیدکند، سپس آن‌را به چهار پایه‌ای که باید ۲,۷ ولت باشند وصل کرده‌ایم.

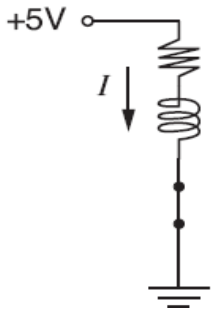
دو آی‌سی دو طرف شکل نیز آی‌سی And هستند که دلیل استفاده از آن را توضیح‌دادیم. در اینجا ما یک جهت فرضی را انتخاب کرده‌ایم و پایه‌ی مثلاً PORTA.0 را صفر(پایه‌ی پایین And سمت راست) و PORTA.1 را یک گذاشته‌ایم (پایه‌ی بالای And سمت چپ).

تنها نکته‌ی باقی‌مانده از شکل بالا چهارعدد دیود اطراف موتور می‌باشد. شکل ساده‌ی روبرو را در نظر بگیرید، فرض‌کنید یک کلید دارید که با خاموش و روشن شدن آن موتور خاموش و روشن می‌شود:



شکل ۱۴-۱۷ : لحظه‌ی باز شدن کلید موتور

یک موتور DC را معمولاً با یک سلف بزرگ و یک مقاومت کوچک مدل می‌کنند:

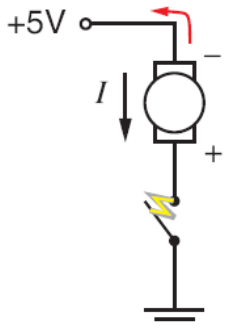


شکل ۱۴-۱۸ : مدل مداری موتور DC

می‌دانید که رابطه‌ی ولتاژ و جریان سلف به شکل زیر است:

$$v = l\frac{di}{dt}$$

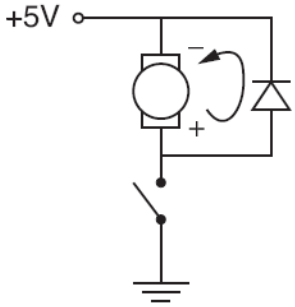
اگر کلید را به یکباره قطع کنیم، $\frac{di}{dt}$ (یعنی تغیرات جریان به زمان) به سمت بی‌نهایت میل‌کرده و یک ولتاژ لحظه‌ای شبیه به جرقه رخ می‌دهد و باعث سوختن ماسفت‌ها می‌شود:



شکل ۱۴-۱۹ : جرقه‌زدن پس از قطع شدن کلید

پس برای اینکه این اتفاق رخ ندهد، یک دیود دو سر موتور می‌گذاریم که در لحظات قطع و وصل کلید، جریان یک دفعه صفر نشود و در یک حلقه چرخیده و مسیر خود را ببندد. این دیود به دیود هرزگرد معروف است:

چون در این درایور، موتور با چهار عدد کلید قطع و وصل می‌شود، باید چهارعدد دیود بگذاریم.



شکل ۱۴-۲۰ : قرار دادن دیود هرزگرد برای جریان

فصل پانزدهم

فیوزبیت‌ها، منابع ریست و پروگرام کردن

در این فصل خواهیم خواند:

۱. منابع ریست(Reset)
۲. فیوزبیت‌های ATmega16
۳. پروگرام کردن میکروکنترلر با نرم‌افزار Progisp

فیوزبیت‌ها، منابع ریست و آموزش پروگرام کردن

در این فصل با منابع ریست و فیوزبیت‌های ATmega16 آشنا می‌شویم و پس از آن نحوه‌ی پروگرام کردن یک میکروکنترلر را خواهیم آموخت تا بتوانیم نتیجه‌ی یک کد نوشته شده در کدویژن را در محیط واقعی مشاهده کنیم.
ابتدا بایستی به این نکته توجه کنیم که فیوزبیت‌های میکروکنترلرهای مختلف خانواده‌ی AVR متفاوت است ولی با یادگیری کار با فیوزبیت‌های یکی از میکروکنترلرهای این خانواده کار با فیوزبیت‌های سایر میکروکنترلرهای این خانواده کار چندان دشواری نخواهد بود.
فیوزبیت‌ها در واقع قسمتی از حافظه‌ی فلش هستند که برای برخی تنظیمات سخت‌افزاری و نرم‌افزاری بکار می‌روند.

در ATmega16 دو فیوزبایت (هر فیوزبایت شامل ۸ فیوزبیت) وجود دارد و همانطور که گفته شد در مدل‌های مختلف AVR می‌تواند این فیوزبیت‌ها متفاوت باشد(برای مثال در ATmega64 سه فیوزبایت وجود دارد).اگر بخواهیم در بیانی ساده فیوزبیت‌ها را توصیف کنیم باید بگوییم با استفاده از فیوزبیت‌ها می‌توان یک سری از امکانات سخت‌افزاری و نرم‌افزاری میکروکنترلر را فعال کرد.
برای مثال اگر بخواهید فرکانس کاری میکروکنترلر را به صورت داخلی یا با استفاده از کریستال خارجی تنظیم کنید بایستی از تنظیمات فیوزبیت‌ها استفاده کنید.در فیوزبیت‌ها نوشته شدن عدد یک به معنی برنامه‌ریزی نشدن (Unprogrammed)و نوشته شدن عدد صفر به معنی برنامه‌ریزی شدن (programmed) می‌باشد.

توجه: یک اشتباه رایج توسط کاربران نوشتن یک در فیوزبیت‌ها برای فعال کردن و صفر برای غیرفعال کردن آن است.

میکروکنترلرهای ATmega16 دارای ۲ فیوزبایت می‌باشند که آنها را در دو جدول ۱۵-۱ و ۱۵-۲ مشاهده می‌کنید:

Fuse High Byte	Bit No.	Description	توضیحات	Default Value
OCDEN	7	Enable OCD	فعال کردن اشکال زدایی از طریق JTAG	1 (unprogrammed, OCD disabled)
JTAGEN	6	Enable JTAG	فعال کردن JTAG	0 (programmed, JTAG enabled)
SPIEN	5	Enable SPI Serial Program and Data Downloading	فعال کردن پروگرام کردن از طریق ارتباط SPI	0 (programmed, SPI prog. enabled)
CKOPT	4	Oscillator options	نحوه ی عملکرد اسیلاتور	1 (unprogrammed)
EESAVE	3	EEPROM memory is preserved through the Chip Erase	حفاظت از حافظه ی eeprom در برابر پاک کردن	1 (unprogrammed, EEPROM not preserved)
BOOTSZ1	2	Select Boot Size	انتخاب اندازه ی حافظه ی Boot loader	0 (programmed)
BOOTSZ0	1	Select Boot Size	انتخاب اندازه ی حافظه ی Boot loader	0 (programmed)
BOOTRST	0	Select reset vector	انتخاب بردار reset	1 (unprogrammed)

جدول ۱۵-۱ : فیوز بایت پر ارزش ATmega16

Fuse Low Byte	Bit No.	Description	توضیحات	Default Value
BODLEVEL	7	Brown-out Detector trigger level	تعیین سطح ولتاژ Brown-out	1 (unprogrammed)
BODEN	6	Brown-out Detector enable	فعال کردن آشکار ساز Brown-out	1 (unprogrammed, BOD disabled)
SUT1	5	Select start-up time	تعیین زمان start-up	1 (unprogrammed)
SUT0	4	Select start-up time	تعیین زمان start-up	0 (programmed)
CKSEL3	3	Select Clock source	انتخاب منبع Clock	0 (programmed)
CKSEL2	2	Select Clock source	انتخاب منبع Clock	0 (programmed)
CKSEL1	1	Select Clock source	انتخاب منبع Clock	0 (programmed)
CKSEL0	0	Select Clock source	انتخاب منبع Clock	1 (unprogrammed)

جدول ۱۵-۲ : فیوزبایت کم‌ارزش ATmega16

در ادامه با کاربرد هر یک از آنها به طور کامل آشنا می‌شویم.

ریست (Reset) میکروکنترلر و منابع ریست

یکی از مهمترین چیزهایی که برای ما بسیار اهمیت دارد کارکرد درست میکروکنترلر در همه‌ی محیط‌ها بخصوص در محیط‌های پرنویز است پس ریست کردن میکروکنترلر در زمان‌هایی که امکان کارکرد نادرست آن وجود دارد بسیار مهم می‌باشد.

زمانیکه یک میکروکنترلر ریست می‌شود به این معنی است که تنظیمات رجیسترهای ورودی/خروجی(I/O) که مربوط به ارتباط میکروکنترلر با سخت‌افزار خارجی هستند به صورت پیش‌فرض قرار می‌گیرند که این تنظیمات پیش‌فرض هر رجیستر در دیتاشیت هر میکروکنترلر آمده است.

با ریست میکروکنترلر برنامه به ابتدای ناحیه‌ی اپلیکیشن و یا به ابتدای ناحیه بوت لودر(Boot Loader) پرش می‌کند و از آنجا برنامه مجدداً اجرا می‌گردد.

ناحیه‌ی اپلیکیشن ناحیه‌ای است که کدهای عادی برنامه در داخل آن نوشته می‌شود و ناحیه بوت لودر به صورت بسیار خلاصه، ناحیه‌ای است که برنامه‌ی نوشته شده در داخل آن از طریق دستور اسمبلی SPM امکان تغییر محتویات حافظه‌ی فلش را دارد. ناحیه‌ی بوت لودر دارای این ویژگی است که با توجه به اینکه امکان تغییر محتویات حافظه‌ی فلش را داراست در بعضی از کاربردها با توجه به کد مورد نظری که بایستی پروگرام شود در این ناحیه بدون نیاز به هرگونه پروگرامری از طریق کد نوشته شده در ناحیه بوت لودر عمل تغییر محتویات فلش انجام می‌شود و دستورالعمل اسمبلی که این امکان رو فراهم می‌کند, دستورالعمل اسمبلی SPM است. در بعضی از میکروکنترلرهای خانواده‌ی AVRدر ناحیه‌ی بوت لودر توسط شرکت سازنده برنامه‌ای از قبل قرار داده می‌شود. برای مثال در سری‌های AT90USB که امکان ارتباط USB در آنها میسر است بوت لودری توسط شرکت ATMEL پروگرام می‌شود که با نرم‌افزار فیلیپس (این نرم‌افزار در سایت ATMELموجود است) امکان پروگرام کردن این نوع میکروکنترلرها از طریق کابل USB وجود دارد. به عبارت ساده‌تر زمانیکه این میکروکنترلر از شرکت سازنده در اختیار مصرف‌کننده قرار می‌گیرد نیازی به پروگرام کردن میکروکنترلر با پروگرامر نیست بلکه با ارتباط USB و با نرم‌افزار بوت لودری که قبلاً نوشته شده و بر روی حافظه‌ی Flash پروگرام شده، از طریق پورت

USB امکان نوشتن در ناحیه لازم در حافظه‌ی فلش وجود دارد(یعنی با استفاده از USB و نرم افزار فیلیپس می‌توانیم این نوع میکروکنترلرها را پروگرام کنیم).

به طور کلی ناحیه‌ی بوت لودر برای تغییر محتویات حافظه‌ی فلش در حین اجرای برنامه مورد استفاده قرار می‌گیرد. یکی از کاربردهای ناحیه بوت لودر زمانی است که بخواهیم بدون اینکه دستگاه پروگرامر را به میکروکنترلر متصل کنیم محتویات فلش آن را تغییر دهیم که در این صورت بایت‌های لازم می‌تواند به طرق مختلف بعنوان مثال از طریق پورت سریال یا سایر روشهای ارتباطی توسط کد نوشته شده برای بوت لودر خوانده شود و در حافظه‌ی فلش نوشته شود و عملاً برنامه‌ی میکروکنترلر توسط ناحیه‌ی بوت لودر در حین اجرا توانایی تغییر و دوباره پروگرام شدن را پیدا می‌کند.

خلاصه مطالب فوق:

اگر مطالب بالا را به طور خلاصه جمع‌بندی کنیم بایستی بگوییم با ریست کردن میکروکنترلر علاوه بر قرار گرفتن رجیسترهای ورودی/خروجی (I/O) به صورت پیش‌فرض (که در دیتاشیت هر میکروکنترلر مشخص است) برنامه به ابتدای ناحیه‌ی اپلیکیشن و یا به ابتدای ناحیه‌ی بوت لودر پرش می‌کند که تعیین‌کننده‌ی این امر که برنامه به کدامیک از دو ناحیه‌ی ذکر شده پرش کند یکی از بیت‌های فیوز High Byte می‌باشد.

فیوزبیت BOOTRST:آدرس ابتدای ناحیه‌ی بوت لودر توسط فیوزبیت‌ها قابل تعیین است که این بیت همان بیت صفر فیوزHigh-Byte است و اگردر داخل این بیت یک نوشته شود (پروگرام نشده) بعد از ریست میکروکنترلر برنامه پرشی به ابتدای ناحیه اپلیکیشن و آدرس صفر انجام می‌دهد و اگرصفر نوشته شود (پروگرام شده) بعد از ریست میکروکنترلر برنامه پرشی به ابتدای ناحیه بوت لودر انجام می‌دهد، مقداردهی این بیت را در جدول زیر مشاهده می‌کنید:

BOOTRST	Reset Address
1	Reset Vector = Application reset (address $0000)
0	Reset Vector = Boot Loader reset

جدول ۱۵-۳ : تنظیم بیت BOOTRST

فیوزبیت‌های BOOTSZ1 و BOOTSZ0:تعیین سایز بوت لودر نیز توسط دو فیوزبیت BOOTSZ1,BOOTSZ0 قابل تعیین هست که در ATmega16 ۴ سایز قابل تعیین است یعنی:۱۲۸کلمه (Word) , ۲۵۶کلمه ,۵۱۲کلمه و ۱۰۲۴کلمه که مطابق جدول زیر تعیین می‌گردد:

BOOTSZ1	BOOTSZ0	Boot Size	Pages	Application Flash Section	Boot Loader Flash Section	End Application section	Boot Reset Address (start Boot Loader Section)
1	1	128 words	2	$0000 - $1F7F	$1F80 - $1FFF	$1F7F	$1F80
1	0	256 words	4	$0000 - $1EFF	$1F00 - $1FFF	$1EFF	$1F00
0	1	512 words	8	$0000 - $1DFF	$1E00 - $1FFF	$1DFF	$1E00
0	0	1024 words	16	$0000 - $1BFF	$1C00 - $1FFF	$1BFF	$1C00

جدول ۱۵-۴ : تنظیم بیت‌های BOOTSZ1,BOOTSZ0

با توجه به آدرس این دو فیوزبیت یعنی BOOTSZ1,BOOTSZ0 همانطور که در ستون آخر جدول بالا نیز مشاهده می‌کنید آدرس شروع ناحیه بوت لودر پس از ریست میکروکنترلر نیز مشخص می‌شود.

منابع ریست

۵ منبع ریست برای ATmega16 وجود دارد که عبارتند از:

۱) **ریست ناشی از کاهش Power-on**: که ناشی از کاهش ولتاژ تغذیه‌ی میکروکنترلر از یک حد مشخص است.
۲) **ریست ناشی از پایه‌ی ریست میکروکنترلر**: که اگر به این پین سطح ولتاژ LOW اعمال شود، میکروکنترلر فرمان ریست را اجرا می‌کند.
۳) **ریست ناشی از Watchdog**: که در فصل تایمر توضیح داده شد.

۴) **ریست ناشی از Brown-Out**: که یک شکل دیگری از ریست ولتاژ تغذیه‌ی میکروکنترلر است اما در یک سطح ولتاژ بالاتر.

۵) **ریست ناشی از ارتباط JTAG**: در مورد ارتباط JTAG به طور خلاصه این توضیح لازم است داده شود که در بعضی از شماره‌های AVR امکان ارتباط با پروتکل JTAG وجود دارد که از طریق آن می‌توان هم میکروکنترلر را پروگرام کرد (Erase و دسترسی به بخش های مختلف) و هم امکان Debugging(اشکال‌زدایی) وجود دارد به این معنی که با یک ارتباط سخت‌افزاری و از طریق یک پروگرامر مناسب می‌توانیم از وضعیت رجیسترهای داخلی میکروکنترلر در مراحل مختلف اجرای برنامه مطلع شویم، معمولاً پروگرامرهایی که با پورت JTAG کار می‌کنند پروگرامرهای گران‌قیمتی هستند. در بعضی از شماره‌های AVR همانند ATmega8 پروتکل JTAG وجود ندارد.

Symbol	Parameter	Condition	Min	Typ	Max	Units
V_{POT}	Power-on Reset Threshold Voltage (rising)			1.4	2.3	V
	Power-on Reset Threshold Voltage (falling)			1.3	2.3	V
V_{RST}	$\overline{\text{RESET}}$ Pin Threshold Voltage		0.1 V_{CC}		0.9V_{CC}	V
t_{RST}	Minimum pulse width on $\overline{\text{RESET}}$ Pin				1.5	µs
V_{BOT}	Brown-out Reset Threshold Voltage	BODLEVEL = 1	2.5	2.7	3.2	V
		BODLEVEL = 0	3.6	4.0	4.5	
t_{BOD}	Minimum low voltage period for Brown-out Detection	BODLEVEL = 1		2		µs
		BODLEVEL = 0		2		µs
V_{HYST}	Brown-out Detector hysteresis			50		mV

جدول ۱۵-۵: مشخصه‌های منابع ریست

ریست ناشی از کاهش Power-on

یکی از دلایلی که می‌تواند باعث ریست شدن میکروکنترلر شود کاهش ولتاژ تغذیه‌ی آن می‌باشد که اگر به جدول زیر که توسط شرکت Atmel در دیتاشیت ATmega16 نوشته شده است دقت

کنیم، ریست زمانی اتفاق می‌افتد که ولتاژ تغذیه به مقدار ۲,۳ تا ۱,۳ ولت کاهش پیدا کند، حداقل ولتاژی که مطمئن هستیم در آن میکروکنترلر ریست می‌شود ولتاژ ۱,۳ ولت می‌باشد. برای خارج شدن میکروکنترلر از حالت ریست ولتاژ منبع تغذیه باید حداقل بیشتر از مقدار ۱,۴ ولت و حداکثر تا ۲,۳ ولت افزایش یابد تا مطمئن شویم میکروکنترلر از حالت ریست خارج شده است.

Symbol	Parameter	Condition	Min	Typ	Max	Units
V_{POT}	Power-on Reset Threshold Voltage (rising)			1.4	2.3	V
	Power-on Reset Threshold Voltage (falling)			1.3	2.3	

جدول ۱۵-۶: جدول مربوط به سطح ولتاژهای Power-on Reset

ریست ناشی از Brown-Out

اگر به جدول زیر که فیوزبیت‌های ATmega16 را نشان می‌دهد نگاه کنیم، فیوزبیت‌های شماره ۶ و ۷ از بایت کم ارزش فیوزبیت‌ها مربوط به Brown-out Reset می‌باشند.

Fuse Low Byte	Bit No.	Description	Default Value
BODLEVEL	7	Brown-out Detector trigger level	1 (unprogrammed)
BODEN	6	Brown-out Detector enable	1 (unprogrammed, BOD disabled)

جدول ۱۵-۷ : فیوزبیت‌های BODEN,BODLEVEL

با فعال کردن فیوزبیت شماره ۶ (BODEN) ، Brown-out Reset فعال می‌شود. کار این نوع منبع ریست افزایش سطح ولتاژ ریست می‌باشد به طوریکه اگر بعد از فعال کردن فیوزبیت BODEN فیوزبیت BODLEVEL را در حالت غیرفعال (مقدار ۱) نگه داریم، به جای آنکه میکروکنترلر در ولتاژ تغذیه ۱,۳ ولت ریست شود، در ولتاژ تقریباً ۲,۷ ولت ریست می‌شود که این به معنای حساسیت بیشتر و ریست شدن سریع‌تر است؛ و اگر فیوزبیت BODLEVEL را نیز فعال کنیم (یعنی هم زمان دو فیوزبیت BODEN و BODLEVELرا فعال کنیم) سطح ولتاژ ریست باز

هم بالاتر می‌آید و زمانی که ولتاژ تغذیه به حدود ۴ ولت رسید میکروکنترلر ریست می‌شود. به جدول زیر دقت کنید:

Symbol	Parameter	Condition	Min	Typ	Max	Units
V_{BOT}	Brown-out Reset Threshold Voltage	BODLEVEL = 1	2.5	2.7	3.2	V
		BODLEVEL = 0	3.6	4.0	4.5	
t_{BOD}	Minimum low voltage period for Brown-out Detection	BODLEVEL = 1		2		µs
		BODLEVEL = 0		2		
V_{HYST}	Brown-out Detector hysteresis			50		mV

جدول ۱۵-۸ : جدول مربوط به سطح ولتاژ Brown-out Reset

همانطور که در جدول مشخص است ولتاژ تغذیه باید حداقل به مدت ۲ میکروثانیه در ولتاژ کم (۲,۷ یا ۴ ولت) بماند تا میکروکنترلر ریست شود.

یکی از نکاتی که در انتخاب Brown-out-Reset باید در نظر گرفته شود ولتاژ تغذیه است. اگر ولتاژ تغذیه در حالت عادی برابر ۵ ولت است، مشکلی ایجاد نمی‌شود ولی اگر از میکروکنترلرهای سری A یا L استفاده می‌کنید که توانایی کار در سطوح ولتاژ پایین‌تر مانند ۳,۳ ولت را دارند و ولتاژ تغذیه را نیز برابر ۳,۳ قرار داده‌اید نباید فیوزبیت BODLEVEL را فعال کنید زیرا با فعال کردن این فیوزبیت میکروکنترلر در ولتاژهای کمتر از ۴ ولت ریست می‌شود و از آنجایی که ولتاژ تغذیه را برابر ۳,۳ ولت انتخاب کرده‌اید کارکرد میکروکنترلر دچار اختلال می‌شود. نکته‌ی دیگری که توسط دیتاشیت توصیه شده این است که اگر از EEPROM داخلی میکروکنترلر استفاده می‌کنید حتماً از منبع ریست Brown-out-Reset استفاده کنید زیرا با کم شدن ولتاژ تغذیه کارکرد این حافظه دچار اختلال شده و ممکن است یا پاک شود و یا دستورها را اشتباه اجرا کند.

ریست ناشی از پایه‌ی ریست میکروکنترلر

منبع دیگر ریست ، ریست ناشی از پین ریست می‌باشد که زمانیکه ولتاژ پایه‌ی ریست (پایه شماره ۹ در میکروکنترلرATmega16) در وضعیت Low قرار بگیرد ریست رخ می‌دهد که همانطور که در جدول زیر مشاهده می‌کنید حداقل ولتاژی که لازم است به این پایه اعمال شود تا ریست

رخ‌دهد مقدار 0.1Vcc می‌باشد (یعنی اینکه مقدار ولتاژ پایه ریست به ۰,۱ مقدار منبع تغذیه برسد)و حداقل زمانی که پایه ریست بایستی در این سطح ولتاژ قرار داشته باشد تا ریست رخ دهد مقدار ۱,۵ میکروثانیه ذکر شده است:

Symbol	Parameter	Condition	Min	Typ	Max	Units
V_{RST}	$\overline{\text{RESET}}$ Pin Threshold Voltage		$0.1V_{CC}$		$0.9V_{CC}$	V
t_{RST}	Minimum pulse width on $\overline{\text{RESET}}$ Pin				1.5	µs

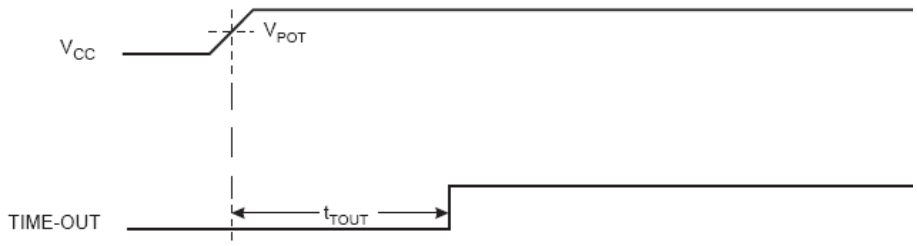
جدول ۱۵-۹ : جدول مربوط به سطح ولتاژ و زمان مورد نیاز ریست پایه‌ی RESET

با افزایش ولتاژ میکروکنترلر بلافاصله از حالت ریست خارج نمی‌شود و یک زمان کوتاهی را سپری می‌کند سپس از حالت ریست خارج شده و به کارکرد عادی خود ادامه می‌دهد، این مدت زمان توسط دو فیوزبیت Start Up time انجام می‌گیرد که مطابق جدول زیر دو بیت SUT1 و SUT2 از جدول فیوز LowByte این وظیفه را بر عهده دارند.

Start Up time به شدت به منابع کلاک میکروکنترلر وابسته است که متناسب با انتخاب منبع کلاک بایستی این دو بیت تنظیم گردد. در دیتاشیت هر نوع میکروکنترلر مقادیر متناسب با هر منبع کلاک توضیح داده شده است(در ادامه با نحوه‌ی تنظیم SUT1 و SUT2 دقیق‌تر آشنا می‌شویم).

SUT1	5	Select start-up time	1 (unprogrammed)
SUT0	4	Select start-up time	0 (programmed)

جدول ۱۵-۱۰: فیوزبیت‌های SUT1,SUT0



شکل ۱۵-۱: زمان خارج شدن از حالت ریست

ریست ناشی از ارتباط JTAG

ریست ناشی از JTAG هم از طریق ارتباط JTAG قابل انجام می‌باشد که این نوع ریست معمولاً توسط پروگرامر انجام می‌گیرد.

فیوزبیت JTAGEN: بیت شماره ۶ از فیوز HighByte مربوط به فعال‌سازی JTAG می‌باشد (برای فعال‌سازی باید این فیوزبیت صفر شود) که به صورت پیش‌فرض این فیوزبیت پروگرام شده و مقدار صفر بر روی آن نوشته شده است.

نکته‌ی کاربردی: فعال بودن JTAG به صورت پیش‌فرض باعث می‌گردد عملاً ۴پین از پورتC(PORTC.2…5) به عنوان ورودی/خروجی (I/O) امکان استفاده نداشته باشند. بسیار پیش می‌آید که کسانی که کار با میکروکنترلرهای AVR را آغاز کرده‌اند در برنامه‌ی خود از آن ۴ پورت C در کد خود بعنوان ورودی/خروجی استفاده می‌کنند ولی در عمل این ۴ پورت به علت صفر بودن بیت فعال‌ساز ارتباط JTAG کار نمی‌کنند، برای اینکه از این ۴ پورت بتوان به صورت ورودی/خروجی (I/O) استفاده کرد باید بوسیله پروگرامر بیت فعال‌ساز JTAG را یک کرد تا غیرفعال شود.

Fuse High Byte	Bit No.	Description	Default Value
JTAGEN	6	Enable JTAG	0 (programmed, JTAG enabled)

جدول ۱۵-۱۱ : فیوزبیت JTAGEN

فیوزبیت OCDEN: یک فیوزبیت مرتبط دیگر به ارتباط JTAG ، فیوزبیت On Chip-Debug Enable (OCDEN) است که برای اینکه عملیات دیباگ (اشکال زدایی) توسط پورت JTAG قابل انجام باشد باید این فیوزبیت پروگرام شود یا بعبارتی بر روی آن صفر نوشته شود (البته شرط آن این است که میکروکنترلر در وضعیت قفل (Lock) نباشد).

Fuse High Byte	Bit No.	Description	Default Value
OCDEN	7	Enable OCD	1 (unprogrammed, OCD disabled)

جدول ۱۵-۱۲: فیوزبیت OCDEN

ریست ناشی از Watchdog Timer

در میکروکنترلر ATmega16 تایمری تحت عنوان Watchdog Timer وجود دارد که کلاک این تایمر مستقل از کلاک CPU می‌باشد و یک کلاک در حدود ۱ مگاهرتز این تایمر را تغذیه می‌کند. علت استفاده از این تایمر این می‌باشد که به هر دلیلی ناشی از نویز یا هر شرایط ناخواسته‌ی دیگری روند طبیعی اجرای خطوط برنامه توسط میکروکنترلر متوقف شد و به اصطلاح میکروکنترلر هنگ کرد این تایمر میکروکنترلر را ریست و میکروکنترلر را از این وضعیت خارج کند. این روش در فصل تایمرها به طور کامل توضیح داده شده است.

فیوزبیت SPIEN: از دیگر فیوزبیت‌های بخش فیوز High-Byte بایستی به فیوزبیت پنجم آن یعنی SPIEN اشاره کنیم که به صورت پیش‌فرض پروگرام شده می‌باشد و این فیوزبیت در هنگام فعال بودن امکان پروگرام کردن سریال میکروکنترلر را از طریق ارتباط SPI فراهم می‌کند. از طریق ارتباط SPI نمی‌توان این فیوزبیت را غیرفعال کرد و برای غیر فعال کردن این فیوزبیت بایستی از پروگرامرهایی که به صورت موازی با میکروکنترلر ارتباط برقرار می‌کنند استفاده کرد(پروگرامر STK200 نمونه‌ای از پروگرامرهای موازی است و همانطور که در فصل SPI خواندیم برقراری ارتباط در پروتکل SPI به صورت سریال می‌باشد).

SPIEN	5	Enable SPI Serial Program and Data Downloading	0 (programmed, SPI prog. enabled)

جدول ۱۵-۱۳: فیوزبیت SPIEN

فیوزبیت EESAVE: اگر بیت شماره ۶ بخش فیوز High Byte یعنی بیت EESAVE پروگرام شود هر بار که حافظه‌ی فلش میکروکنترلر Erase(پاک) می‌شود محتوای حافظه EEPROM میکروکنترلر محفوظ باقی می‌ماند. به صورت کلی اگر این بیت پروگرام نشود با هر بار پاک کردن حافظه‌ی فلش میکروکنترلر محتوای EEPROM میکروکنترلر نیز پاک می‌شود.

EESAVE	3	EEPROM memory is preserved through the Chip Erase	1 (unprogrammed, EEPROM not preserved)

جدول ۱۵-۱۴: فیوزبیت EESAVE

فیوزبیت CKOPT:این فیوزبیت در ارتباط با منابع کلاک معنا پیدا می‌کند و در مواردی باعث افزایش دامنه‌ی نوسان می‌شود. زمانی که از کریستال خارجی به عنوان منبع کلاک استفاده می‌شود باید این فیوزبیت فعال باشد و در زمانی که از منبع کلاک RC داخلی استفاده می‌شود به هیچ عنوان نباید فعال باشد. این فیوزبیت به صورت پیش‌فرض غیرفعال است.

توضیحات مربوط به فیوزبیت‌های بخش فیوز High Byte به اتمام رسید و از بخش فیوز Low Byte ۴ بیت برای منبع کلاک و دو بیت نیز برای Start Up time بایستی معرفی گردد که در ادامه به طور کامل با این فیوزیت‌ها نیز آشنا می‌شویم.

فیوزبیت‌های CKSEL0…CKSEL3 و SUT0،SUT1:در بخش فیوزLow Byte، فیوزبیت‌های Select Clock Source (CKSEL0…3) صفر تا ۳ برای تعیین منبع کلاک و دو فیوزبیت SUT0 و SUT1 برای تعیین START UP TIME باید در مقدار مناسب تنظیم گردد.

با وجود ۴ بیت برای منبع کلاک عملاً ۱۶حالت برای منابع کلاک متصور می‌شود که بسته به مقدار مورد نظر، منبع کلاک مربوطه در آن نوشته شده است.

Table 2. Device Clocking Options Select

Device Clocking Option	CKSEL3..0
External Crystal/Ceramic Resonator	1111 - 1010
External Low-frequency Crystal	1001
External RC Oscillator	1000 - 0101
Calibrated Internal RC Oscillator	0100 - 0001
External Clock	0000

جدول ۱۵-۱۵: انتخاب منبع کلاک با استفاده از فیوزبیت‌های CKSEL3..0

مطابق دیتاشیت اگر کریستال برای منبع کلاک انتخاب شود در جدول بعد ۴ سطر وجود دارد که بسته به وضعیت فیوزبیت Clock-Option(CKOPT) که به صورت یک یا صفر باشد مقادیر CKSEL0 تا CKSEL3 قابل تفسیر است. اگر فیوزبیتClock Option برنامه‌ریزی نشده و به صورت یک باشد و بیت‌های CKSEL مطابق جدول زیرCKSEL3…0 به صورت ۱۰۱ باشد

متناظر با فرکانس‌های ۰,۴ تا ۰,۹ مگاهرتز می‌باشد که همانطور که دیتاشیت آمده برای اتصال کریستال چنین وضعیتی مناسب نیست. برای بقیه حالت‌ها نیز محدوده فرکانسی مطابق جدول زیر تعیین شده است:

Table 4. Crystal Oscillator Operating Modes

CKOPT	CKSEL3..1	Frequency Range (MHz)	Recommended Range for Capacitors C1 and C2 for Use with Crystals (pF)
1	101	0.4 - 0.9	–
1	110	0.9 - 3.0	12 - 22
1	111	3.0 - 8.0	12 - 22
0	101, 110, 111	1.0 ≤	12 - 22

جدول ۱۵-۱۶: تنظیم محدوده فرکانسی کریستال با تنظیم فیوزبیت‌های CKOPT,CKSEL3..0

مطابق جدول فوق همانطور که در سطر آخر مشاهده می‌کنید اگر بخواهیم از کریستال‌های فرکانس بالا استفاده کنیم (مانند کریستال ۱۶ مگاهرتز) بایستی حتماً فیوزبیت CKOPT پروگرام شود.در زیر تصویری از یک کریستال ۱۶ مگاهرتز را مشاهده می‌کنید:

بیت شماره صفر فیوزبیت CKSEL از ۴ فیوزبیت CKSEL3…0(CKSEL0) برای تعیین Start Up time است متناظر با دوبیت SUT1..0 استفاده می‌شود. همانگونه که در جدول بعد نیز مشاهده می‌کنید بسته به این که فیوزبیت CKSEL0 دارای مقدار صفر یا یک باشد، دو وضعیت فیوزبیت‌های Start Up time (SUT) تعیین کننده‌ی مقدار Start Up time است. در مورد Start Up time در جدول بعدی دو ستون وجود دارد که یکی چندین سیکل از کلاک انتخاب شده و ناشی از منبع کلاک انتخاب شده که بعنوان مثال ۲۵۸ سیکل مطابق سطر اول جدول زیر و به همین ترتیب تا حدود ۱۶۰۰۰ کلاک و ستون دیگر به تاخیری که بر حسب

میلی‌ثانیه صورت می‌گیرد اختصاص داده شده که این تاخیر ناشی از تعداد سیکل‌های اسیلاتور Watchdog است:

Table 5. Start-up Times for the Crystal Oscillator Clock Selection

CKSEL0	SUT1..0	Start-up Time from Power-down and Power-save	Additional Delay from Reset (V_{CC} = 5.0V)	Recommended Usage
0	00	258 CK	4.1 ms	Ceramic resonator, fast rising power
0	01	258 CK	65 ms	Ceramic resonator, slowly rising power
0	10	1K CK	–	Ceramic resonator, BOD enabled
0	11	1K CK	4.1 ms	Ceramic resonator, fast rising power
1	00	1K CK	65 ms	Ceramic resonator, slowly rising power
1	01	16K CK	–	Crystal Oscillator, BOD enabled
1	10	16K CK	4.1 ms	Crystal Oscillator, fast rising power
1	11	16K CK	65 ms	Crystal Oscillator, slowly rising power

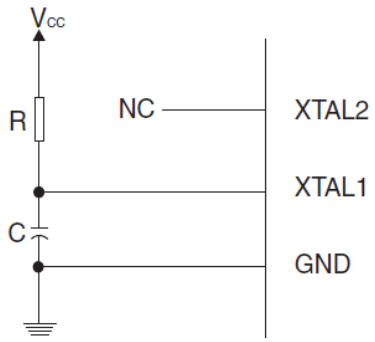
جدول ۱۵-۱۷:تعیین زمان تاخیر Start-up times با تنظیم فیوزبیت‌های CKSEL0,SUT1..0در صورت انتخاب منبع کلاک به صورت کریستال

توجه به این نکته مهم است که به عنوان مثال اگر فیوزبیت‌ها مطابق سطر اول جدول فوق تنظیم گردند بعد از ریست میکروکنترلر مجموع دو زمان ۲۵۶ کلاک منبع کلاک انتخاب شده و ۴,۱ میلی‌ثانیه بعنوان Start Up time(تاخیر پس از رفع ریست برای بازگشت به حالت عادی) در نظر گرفته می‌شود.

گزینه‌ی بعدی یعنی External Low frequency Crystal که یک حالت داردیعنی وضعیت ۱۰۰۱ برای ۴ بیت فیوزبیت CKSEL،تعیین یک منبع کریستالی فرکانس پایین با فرکانس ۳۲۷۴۸ هرتز هست که اگر به جدول ۱۵-۱۶ در کمی قبل‌تر مجدداً نگاه کنیم ملاحظه می‌کنیم برای اسیلاتور کریستالی فرکانس پایین فقط یک وضعیت وجود دارد (وضعیت ۱۰۰۱) در این حالت فیوزبیت‌های SUT تعیین‌کننده‌ی زمان شروع به‌کار میکروکنترلر پس از ریست می‌باشد و اگر

فیوزبیت Clock-Option نیز برنامه‌ریزی شده باشد خازن‌های ۳۶ پیکوفاراد بـه صـورت داخلـی فعال می‌شود و فقط لازم است که کریستال به صورت مستقیم به پین‌هـای XTAL1 و XTAL2 متصل شود.

وضعیت بعدی External RC Oscillator ، وضعیت انتخاب اسیلاتور RC می‌باشد. مقادیر R و C به صورت خارجی مطابق شکل زیر که در دیتاشیت آمده است متصل می‌گردد:



شکل ۱۵-۲ : نحوه‌ی متصل کردن مقاومت و خازن خارجی به پایه‌های XTAL1,2

برای فیوزبیت CKSEL گزینه‌های مختلفی وجود دارد که بسته به رنج و محدوده‌ی فرکانس در جدول زیر ۴ حالت در نظر گرفته شده است:

Table 7. External RC Oscillator Operating Modes

CKSEL3..0	Frequency Range (MHz)
0101	0.1 ≤ 0.9
0110	0.9 - 3.0
0111	3.0 - 8.0
1000	8.0 - 12.0

جدول ۱۵-۱۸:تنظیم فیوزبیت‌های CKSEL3..0 متناسب با فرکانس کاری اسیلاتور RC خارجی

بسته به اینکه کدامیک از محدوده‌های فرکانسی جدول قبل مورد نظر باشد و با توجـه بـه رابطـه‌ی $f = \frac{1}{3RC}$ که در دیتاشیت بعنوان فرکانس تقریبی اسیلاتور ذکر شده است در صورت استفاده از المان‌های R و C به صورت خارجی یکی از این وضعیت‌هـای جـدول بـالا بایسـتی بـرای فیوزبیـت

تنظیم شود و Start Up time هم متناظر با ۴ وضعیتی که دارد در جدول زیر مقادیر مختلفی در نظر گرفته شده است:

Table 8. Start-up Times for the External RC Oscillator Clock Selection

SUT1..0	Start-up Time from Power-down and Power-save	Additional Delay from Reset (V_{CC} = 5.0V)	Recommended Usage
00	18 CK	–	BOD enabled
01	18 CK	4.1 ms	Fast rising power
10	18 CK	65 ms	Slowly rising power
11	6 CK	4.1 ms	Fast rising power or BOD enabled

جدول ۱۵-۱۹:تعیین زمان تاخیر Start-up times با تنظیم فیوزبیت‌های SUT1..0در حالت انتخاب منبع کلاک به صورت اسیلاتور خارجی

بایستی توجه کنیم که در جدول فوق بسته به نوع منبع کلاک انتخابی ستون آخر که تحت عنوان Recommended Usageآمده است بسته به اینکه در چه کاربردی باشیم و از چه گزینه‌ای استفاده کنیم و همچنین بسته به Rise Timeمنبع تغذیه توصیه شده که در هر حالت برای Start Up timeآن وضعیت را در نظر بگیریم. برای مثال وضعیت سطر اول متناظر با فعال بودن Brown-Out Detection(BOD enabled) می‌باشد.

حالت بعدی یعنی Calibrated Internal RC Oscillator انتخاب ۴ فرکانس مختلف بسته به وضعیت فیوزبیت‌ها می‌باشد و استفاده از اسیلاتور داخلی که فرکانس‌های ۱،۲،۴ و ۸ مگاهرتز قابل انتخاب هستند. مقدار پیش‌فرض میکروکنترلر فرکانس یک مگاهرتز می‌باشد که فیوزبیت‌های CKSEL در وضعیت ۰۰۰۱ می‌باشند:

Table 9. Internal Calibrated RC Oscillator Operating Modes

CKSEL3..0	Nominal Frequency (MHz)
0001	1.0
0010	2.0
0011	4.0
0100	8.0

جدول ۱۵-۲۰:تنظیم فیوزبیت‌های CKSEL3..0 متناسب با فرکانس کاری اسیلاتور داخلی

برای Start Up time هم در این حالت مطابق جدول زیر مقادیر نمایش داده شده است:

Table 10. Start-up Times for the Internal Calibrated RC Oscillator Clock Selection

SUT1..0	Start-up Time from Power-down and Power-save	Additional Delay from Reset (V_{CC} = 5.0V)	Recommended Usage
00	6 CK	–	BOD enabled
01	6 CK	4.1 ms	Fast rising power
10	6 CK	65 ms	Slowly rising power
11	Reserved		

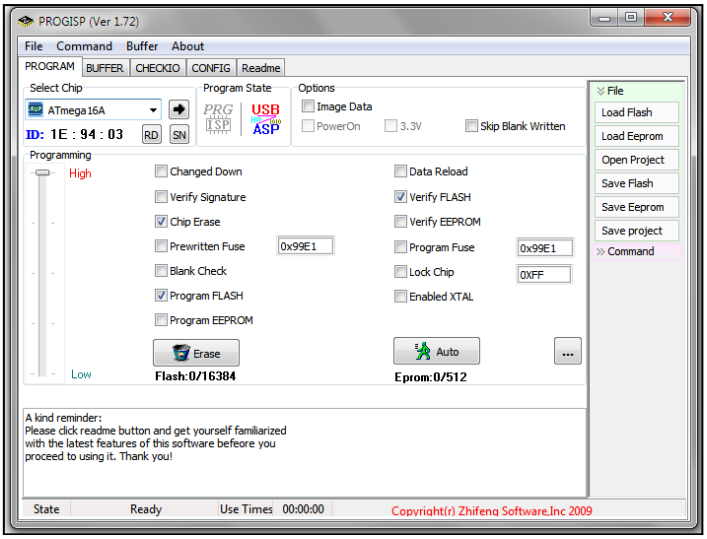
جدول ۱۵-۲۱:تعیین زمان تاخیر Start-up times با تنظیم فیوزبیت‌های SUT1..0در حالت انتخاب منبع کلاک به صورت اسیلاتور داخلی

نکته: در این حالت فیوزبیت Clock Option نبایستی برنامه‌ریزی شود.

با این توضیحات با فیوزبیت‌های میکروکنترلر ATmega16 آشنا شدیم در ادامه با نحوه‌ی پروگرام کردن یک میکروکنترلر به صورت عملی با نرم‌افزار Progisp آشنا می‌شویم.

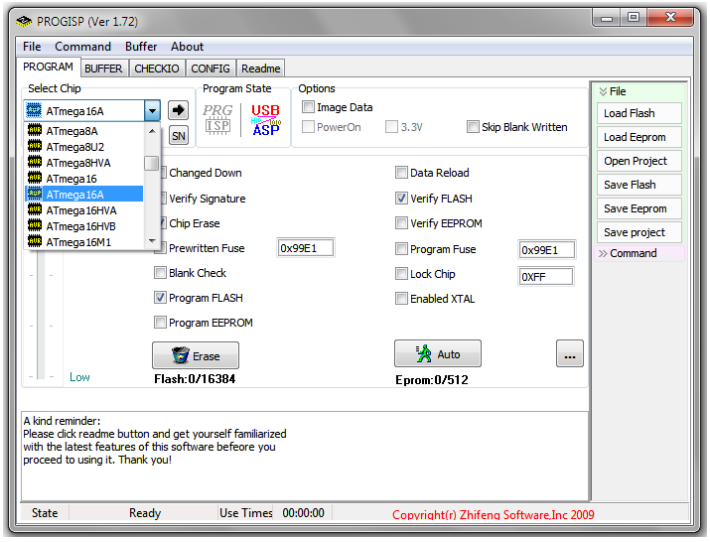
پروگرام کردن با نرم افزار Progisp

با استفاده از این نرم‌افزار و یک پروگرامر می‌توان به سادگی کدهای نوشته شده را بر روی میکروکنترلر پروگرام کرد. این نرم‌افزار در CD ضمیمه آمده است. در شکل زیر محیط این نرم‌افزار را مشاهده می‌کنید:



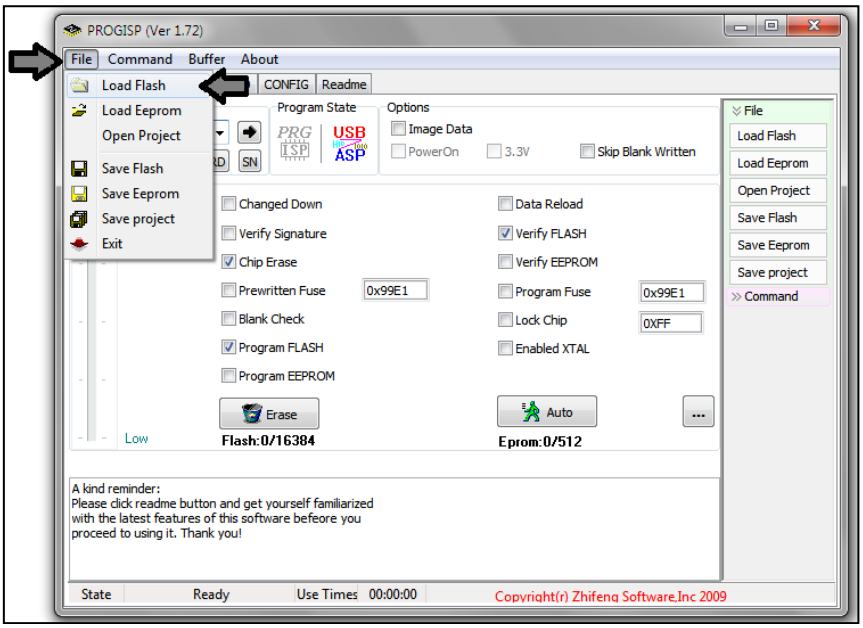
شکل ۱۵-۳ : محیط نرم‌افزار Progisp

نحوه‌ی انتخاب نوع میکروکنترلر برای پروگرام شدن: مطابق شکل زیر می‌توانید از بخش Select Chip نوع میکروکنترلری که می‌خواهید پروگرام کنید را انتخاب نمایید:



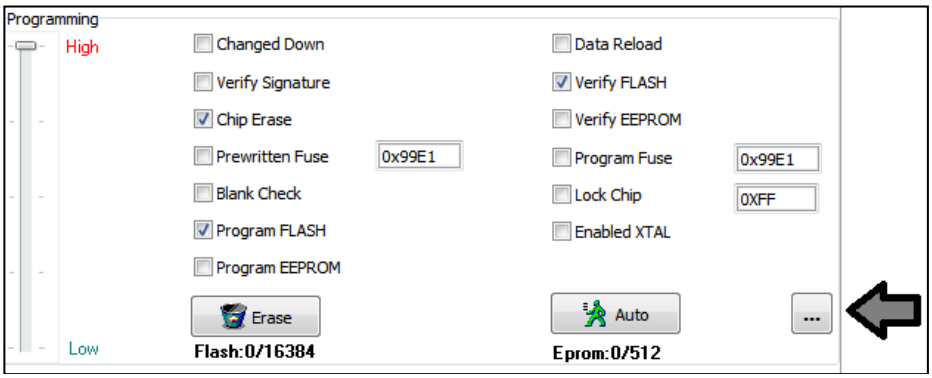
شکل ۱۵-۴ : نحوه‌ی انتخاب نوع Chip

باز کردن کد HEX نوشته شده در نرم‌افزار: مطابق شکل زیر می‌توانید از قسمت File و بخش Load Flashکد خود را از جایی که ذخیره شده است بارگیری کنید:



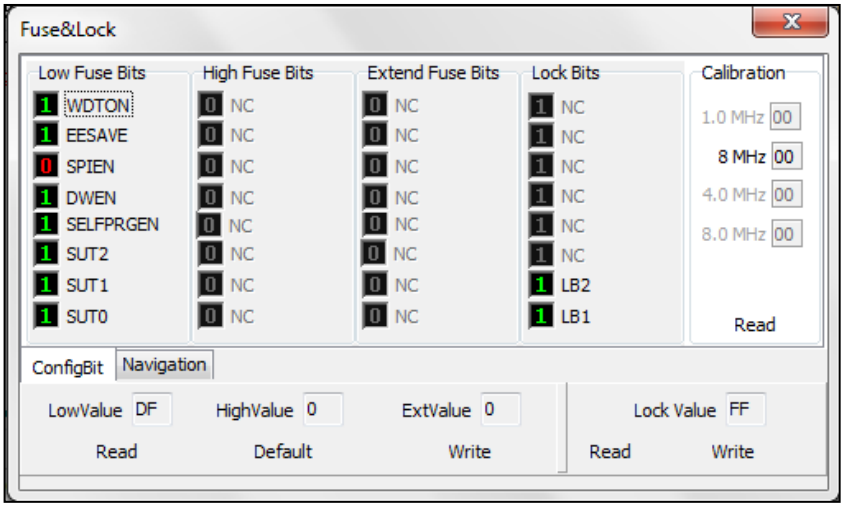
شکل ۱۵-۵ : نحوه‌ی Load کردن کد نوشته شده

نحوه‌ی کار با فیوزبیت‌ها و تنظیم فرکانس کاری: ابتدا مطابق شکل زیر تیک گزینه‌های chip Erase ، program FLASH و verify FLASH را فعال می‌کنیم سپس از بخشی که با فلش مشخص شده است می‌توان تنظیمات فیوزبیت‌ها و فرکانس‌کاری میکروکنترلر را انجام داد:



شکل ۱۵-۶ : نحوه‌ی ورود به بخش کار با فیوزبیت‌ها و تنظیم فرکانس

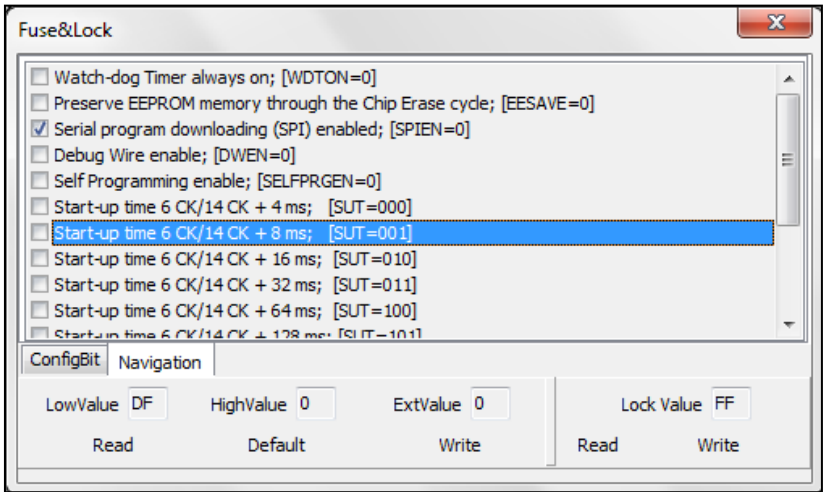
پس از انتخاب آیکون نشان داده شده در شکل قبل پنجره‌ای مانند شکل زیر باز می‌شود که در بخش ConfigBit می‌توان تنظیمات مورد نظر را بر روی فیوزبیت‌ها انجام داد:



شکل ۱۵-۷ : تنظیمات بخش فیوزبیت‌ها

بعد از انجام تنظیمات مورد نظر خود، بر روی گزینه‌ی Write که مطابق شکل در پایین صفحه قرار دارد کلیک می‌کنید تا تنظیمات فیوزبیت‌ها روی میکروکنترلر انجام شود همچنین می‌توانید به

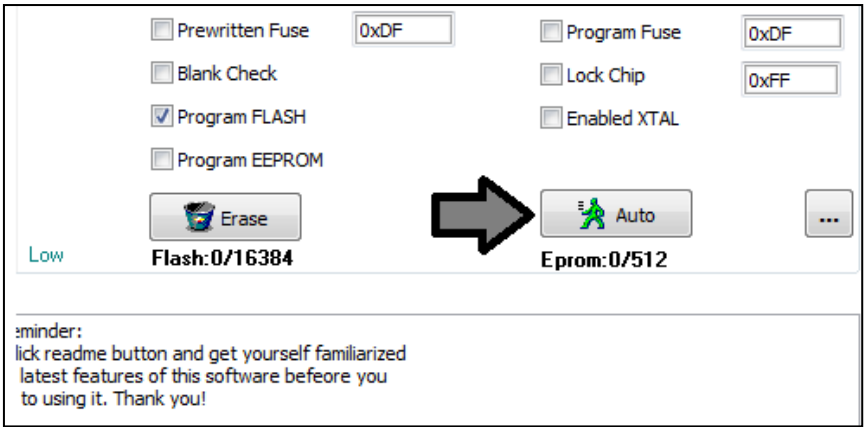
جای انجام تنظیمات به این صورت ، بـر روی تـب Navigation کلیـک کنیـد و تنظیمـات را بـه صورت ساده‌تر انجام دهید:



شکل ۱۵-۸– تنظیم فرکانس کاری میکروکنترلر

پس از انجام تنظیمات گزینه‌ی Write را می‌زنیم.

پس از اینکه تنظیمات بالا تمام شد با زدن گزینه‌ی Auto، میکروکنترلر پروگرام می‌شود و می‌توانید میکروکنترلر را از دستگاه پروگرامر جدا کنید:



شکل ۱۵-۹ : اتمام مراحل پروگرام کردن میکروکنترلر

با انتخاب گزینه‌ی Erase هم می‌توان برنامه‌ای را که بر روی میکروکنترلـر مـی‌باشـد پـاک کـرد. دوباره یادآوری می‌شود که اگر در برنامه از پورت C استفاده کرده‌اید حتماً باید JTAG را غیرفعـال کنید که این کار را می‌توان به سادگی با برداشتن تیک گزینه‌ی Interface Enabled JTAG در تب Navigation انجام داد.

دقت کنید که اگر از پروگرامرهای ارزان قیمت‌تر استفاده می‌کنید که دارای یک جامپر(Jumper) برای انتخاب حالت بینSlow و Fast می‌باشد برای اولین پرگرام هر میکروکنترلر آن را روی حالت Slow قرار دهید.

فصل شانزدهم

پروژه‌ها

در این فصل خواهیم خواند:

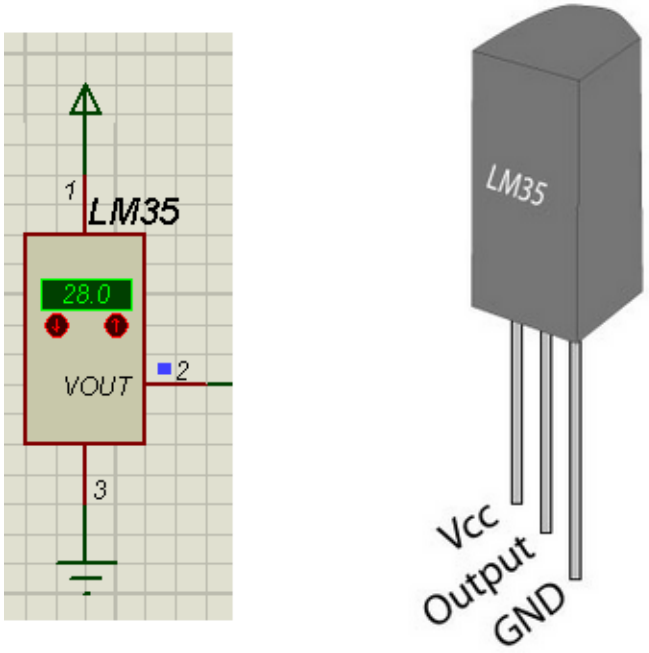
۱. دماسنج به کمک LM35
۲. مسیریاب
۳. موتور همراه با دنده
۴. ساختن زمان سنج
۵. Keypad
۶. تنظیم دما
۷. تولید موج سینوسی با آی‌سی DAC
۸. ساعت و تاریخ میلادی با آی‌سی DS1307
۹. راه‌اندازی سروو موتور
۱۰. راه‌اندازی موتور پله‌ای

پروژه‌ی ۱: دماسنج به کمک سنسور LM35

پیش‌نیاز: ADC و LCD

یکی از سنسورهای تشخیص دما موجود در بازار سنسورLm35 می‌باشد و کار این سنسور این است که تغییرات دمای محیط را به ولتاژ آنالوگ در خروجی خود(Output) تبدیل کند.

این سنسور دارای سه پایه می‌باشد، در صورتی که سنسور روبروی ما قرار گیرد ترتیب قرار گرفتن پایه ها بدین ترتیب می‌باشد که: اولین پایه سمت چپ، پایه‌ی تغذیه‌ی آی‌سی (VCC) می‌باشد و به ولتاژ ۵ ولت متصل می‌شود. پایه‌ی وسط، ولتاژ خروجی(Vout) است که به میکروکنترلر متصل می‌شود تا میکروکنترلر این ولتاژ خروجی را که به صورت ولتاژ آنالوگ می‌باشد، توسط واحد ADC خود تحلیل کند. پایه‌ی سوم نیز زمین (GND) سنسور است. شمای این آی‌سی را در شکل زیر مشاهده می‌کنید:



شکل ۱۶-۱-۱: سنسور LM35

قطعات مورد نیاز برای طراحی

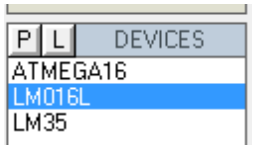
۱) ATmega16

۲)**سنسور** LM35: این سنسورها دارای مقاومت حساس به دما بوده که با تغییرات دما مقدار آن مقاومت تغییر کرده و به دنبال آن ولتاژ خروجی نیز تغییر می‌کند. بنابراین در خروجی خود ولتاژی ایجاد می‌کنند که با توجه به مقدار آن ولتاژ، می‌توان به دمای محیط پی برد.

۳)یک عدد LCD جهت نمایش دما.

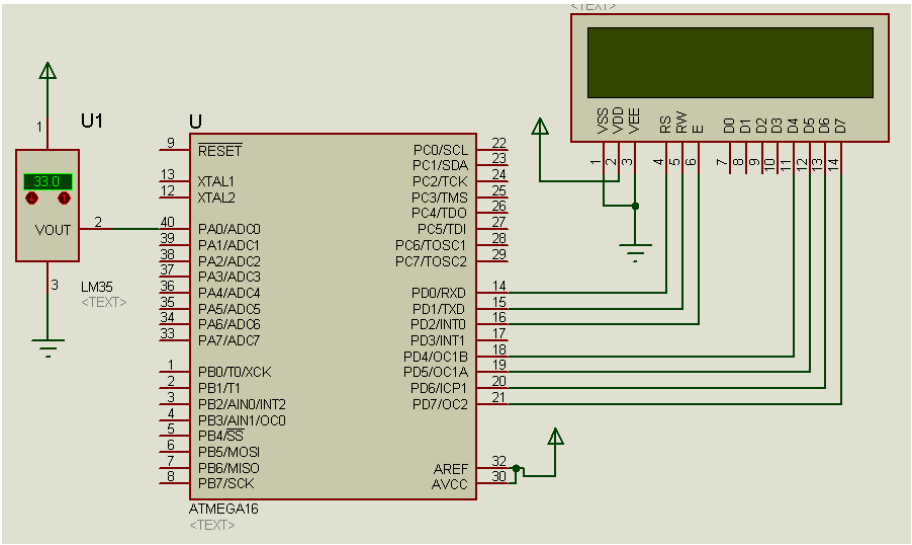
طراحی در پروتئوس

ابتدا قطعات مورد نیاز در پرتئوس را انتخاب می‌کنیم (مطابق شکل مقابل):



شکل ۱۶-۱-۲ : قطعات استفاده شده در شبیه‌سازی

و برای شبیه‌سازی مدار خود را مطابق شکل زیر در پروتئوس می‌بندیم:



شکل ۱۶-۱-۳ : مدار بسته شده در پروتئوس

با توجه به شکل، ولتاژ خروجی آنالوگ LM35 را برای تبدیل به ولتاژ دیجیتال به پایه‌ی ADC0 متصل می‌کنیم.

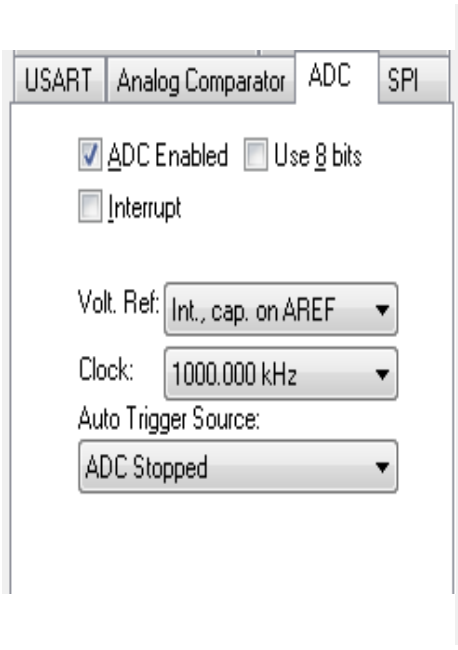
یک عدد LCD را نیز به درگاه D برای نمایش دما متصل می‌کنیم.

یک منبع ۵ولت را نیز به AVCC به منظور تغذیه ADC متصل کرده و به علت اینکه از پایه AREF برای مرجع ADC استفاده می‌کنیم، آن را(AREF) نیز به ۵ ولت متصل می‌کنیم.

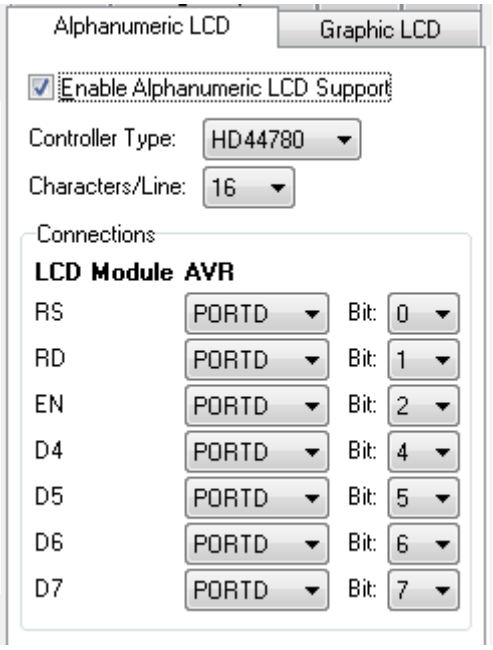
یادآوری: فراموش نکنیم که همیشه باید فرکانس کاری میکروکنترلر را در پروتئوس تنظیم نمائیم.

نوشتن کد در Codevision

پس از انتخاب Atmega16 و تنظیم فرکانس کاری میکروکنترلر بر روی ۸ مگاهرتز، به سراغ سایر تنظیمات می‌رویم.در این بخش تنظیمات درگاه‌ها را تغییری نداده و برای فعال‌کردن واحد ADC و همچنین LCD وارد تب‌های مربوط به هرکدام می‌شویم:



شکل ۱۶-۱-۴ : تنظیمات واحد ADC



شکل ۱۶-۱-۵ : تنظیمات LCD

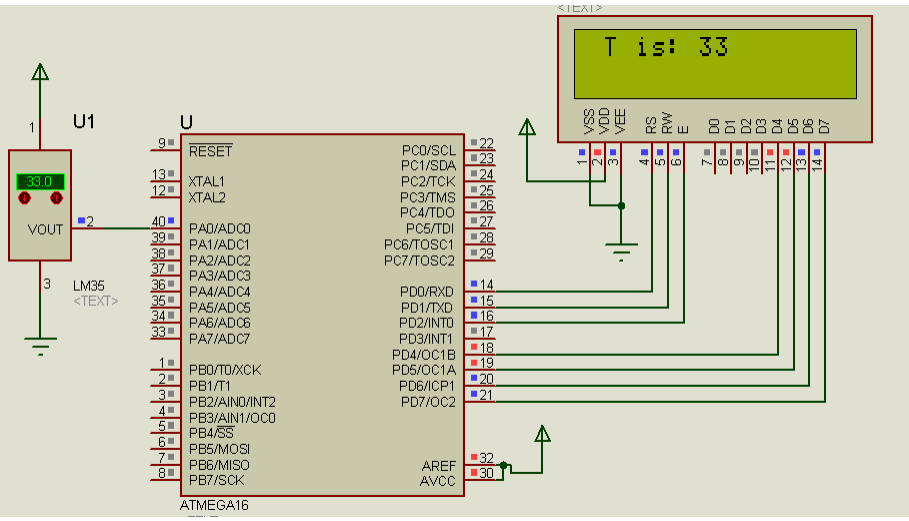
مطابق شکل بالا(۱۶-۱-۵) ولتاژ مرجع (Volt Ref) را در حالت Int cap on AREF قرار می‌دهیم (یعنی از خازن داخلی برای بدست آوردن آن استفاده می‌کنیم).

کد درون while(1):

```
while (1)
      {
      a=read_adc(0);
      lcd_clear();
      sprintf(st," T is: %d",a/4);
      lcd_puts(st);
      delay_ms(300);
      }
```

a یک متغیر از نوع عدد صحیح (int) می‌باشد که مقدار ADC را درون آن ذخیره می‌کنیم و st یک رشته ۳۰عدد کاراکتری(char) می‌باشد که توسط دستور Sprintf مقدار دما را درون آن ذخیره می‌کنیم.

در شکل زیر نتیجه‌ی شبیه‌سازی را زمانی که دمای محیط ۳۳ درجه است مشاهده می‌کنیم(توسط خود سنسور LM35 در محیط شبیه‌سازی می‌توان دمای محیط شبیه‌سازی را تعیین کرد):



شکل ۱۶-۱-۶ : نتیجه‌ی شبیه‌سازی پروژه

پروژه‌ی ۲: مسیریاب

پیش‌نیاز: PIN & PORT و PWM

ربات‌های مسیریاب را به گونه‌های مختلف و برای کاربردهای مختلف می‌سازند. وظیفه‌ی اصلی ربات‌های مسیریاب تعقیب کردن مسیری به رنگ سیاه در زمینه‌ای به رنگ متفاوت سفید است(البته می‌توان رنگ‌هایی دیگر را برای تعقیب مسیریاب تعیین کرد). یکی از کاربردهای عمده‌ی این ربات، حمل و نقل وسایل و کالاهای مختلف در کارخانجات، بیمارستان‌ها، فروشگاه‌ها، کتابخانه‌ها و ... می‌باشد.

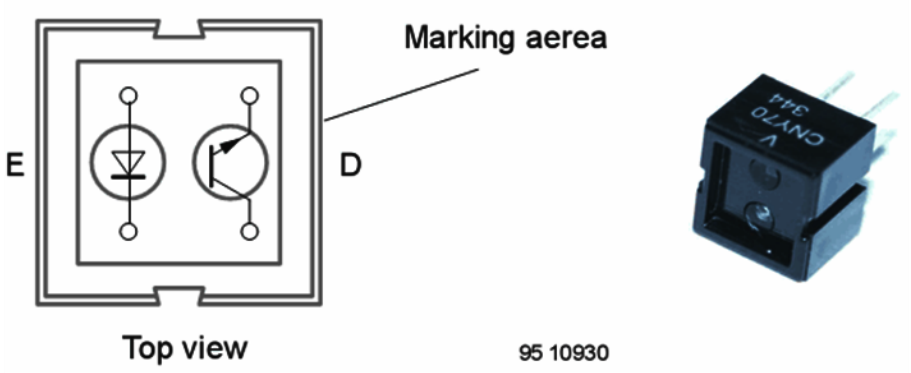
بعنوان مثال در کتابخانه‌ها عملکرد اینگونه ربات‌ها بدین صورت است که بعد ازدادن کد کتاب ،ربات با دنبال کردن مسیری که کد آن توسط متقاضی تعیین شده، به محلی که کتاب در آنجا قرارگرفته است رفته و کتاب را برداشته و به نزد متقاضی کتاب می‌آورد. در بیمارستان‌ها خط‌کشی‌هایی به رنگ‌های مختلف به منظور هدایت ربات‌ها به محل‌های مختلف بیمارستان وجود دارد برای مثال رنگ قرمز ربات را به اتاق جراحی و رنگ آبی ربات را به اتاق زایمان هدایت می‌کند. بیمارانی که توانایی حرکت کردن و جابه‌جا شدن را ندارند و باید از ویلچر استفاده کنند، ویلچیر آنها به گونه‌ای طراحی می‌شود که نقش ربات تعقیب خط را داشته و ویلچر بیمار را از روی مسیر مشخص به محل مورد نظر می‌رساند. مسیریابی که قصد آموزش آن را در این پروژه داریم یک مسیریاب ۵ سنسوره می‌باشد و ربات باید در زمینی به رنگ سفید مسیری به رنگ مشکی را دنبال کند.

قطعات مورد نیاز برای طراحی

۱) **Atmega16**

۲) **درایور موتور یا L298 به همراه دوعدد موتور**

۳) **۵ سنسور CNY70**: این سنسورها درون خود دارای دو سنسور فرستنده و گیرنده‌ی مادون قرمز در کنار هم می‌باشند. سنسورهای مادون قرمز دارای کاربرد وسیعی در زمینه‌ی روباتیک می‌باشند به طور مثال می‌توان برای تشخیص رنگ، تشخیص آتش، تشخیص نور، گرما و .... از آنها استفاده کرد.گیرنده‌های IR(گیرنده مادون قرمز) در واقع نوعی دیود هستند که مقاومت آنها با تغییر میزان اشعه‌ی مادون قرمز در محیط تغییر می‌کند.این سنسورها موج مادون قرمز را فرستاده و بازتاب آن موج را دریافت می‌کنند و در پایه‌ی خروجی خود با توجه به مقدار بازتاب شده که بستگی به رنگ و جنس سطح دارد، ولتاژی را ایجاد می‌کنند که با توجه به مقدار ولتاژ می‌توان به ویژگی‌های آن سطح پی برد. این سنسور را در شکل صفحه بعد مشاهده کنید:



شکل ۱۶-۲- ۱: سنسور CNY70

هر چقدر که سطح تیره‌تر باشد، سنسور ولتاژی نزدیک صفر ولت و هر چقدر سطح روشن‌تر باشد، سنسور ولتاژی نزدیک ۵ولت (چون تغذیه‌ی آن ۵ولت است) را درخروجی خود ایجاد می‌کند.

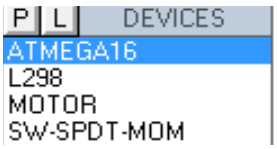
۴)**آی‌سی LM353**: دارای چند آپ‌امپ بوده که برای صفر و یک کردن ولتاژ خروجی سنسورها استفاده می‌شود(تبدیل ولتاژ آنالوگ خروجی سنسورها به ولتاژ دیجیتال یا صفر و یک).در اینجا این آی‌سی طوری بایاس می‌شود تا ولتاژ نزدیک ۵ولت را ۱ و ولتاژ نزدیک صفر ولت را ۰ بدهد.

بنابراین اگر سنسور رنگ سیاه را ببیند صفر و اگر رنگ سفید را ببیند ۱ درخروجی LM353 ظاهر می‌شود که آن را به میکروکنترلر داده و میکروکنترلر نیز موتورها را به حرکت درمی‌آورد.

طراحی در پروتئوس

نکته: با توجه به اینکه در محیط پروتئوس امکان شبیه‌سازی محیطی که ربات در آن قرار دارد را نداریم(یعنی در اینجا نمی‌توان سطحی را قرار داد که ربات با توجه به رنگ سطح روی آن حرکت کند) بنابراین برای شبیه‌سازی این ربات مجبوریم به جای صفر و یکی که LM353 به میکروکنترلر می‌دهد(که حاصل بازتاب موجی است که سنسورها دریافت‌کرده‌اند و به LM353 داده‌اند) از ۵ کلید استفاده کنیم که هر کلید دو حالت صفر و یک را دارد.

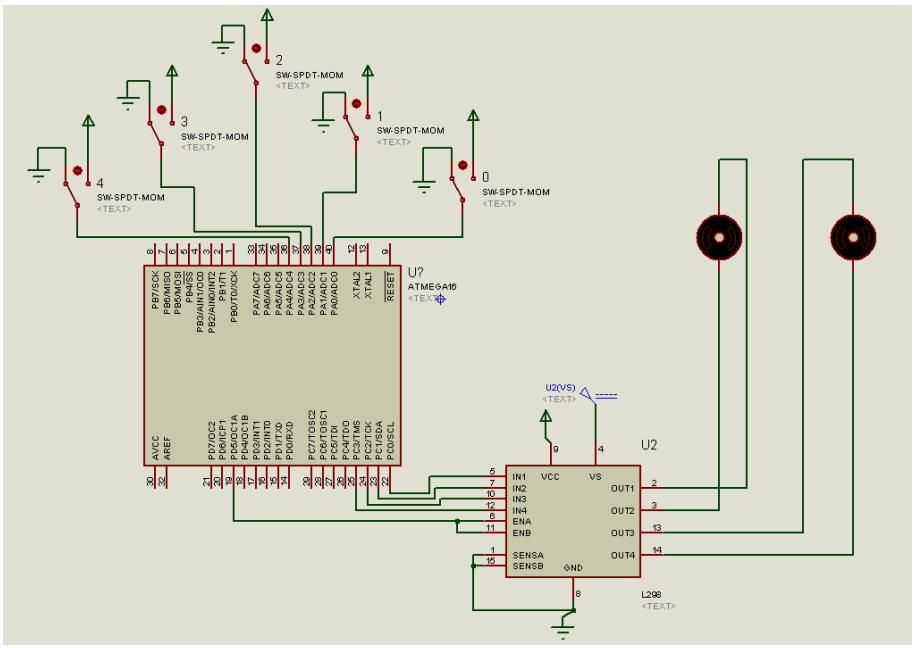
ابتدا قطعات مورد نیاز را مطابق شکل مقابل انتخاب می‌کنیم:



شکل ۱۶-۲-۲ : انتخاب قطعات در پروتئوس

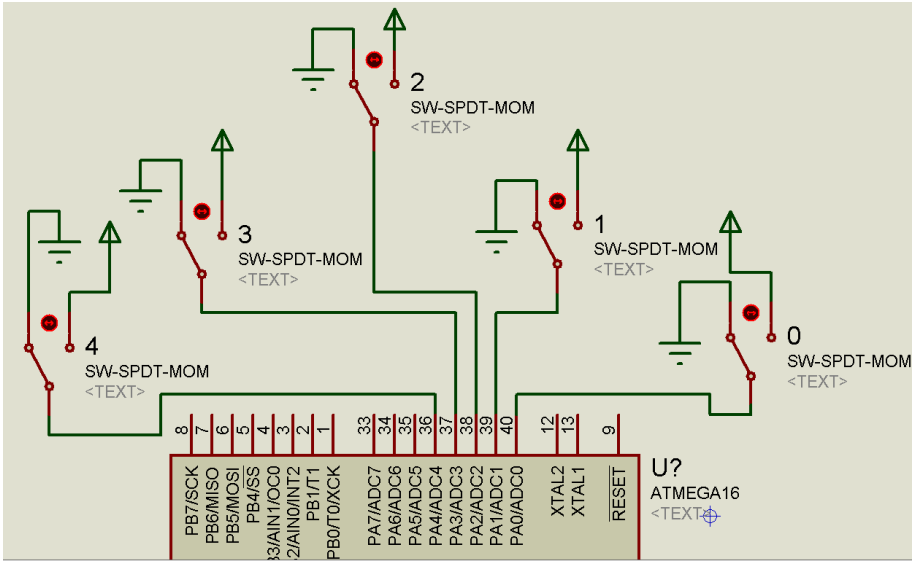
نکته: SW-SPDT-MOM همان کلید دوحالته می‌باشد.

مداری را مطابق شکل زیر با قطعات شکل ۱۶-۲-۲ در پروتئوس می‌بندیم:



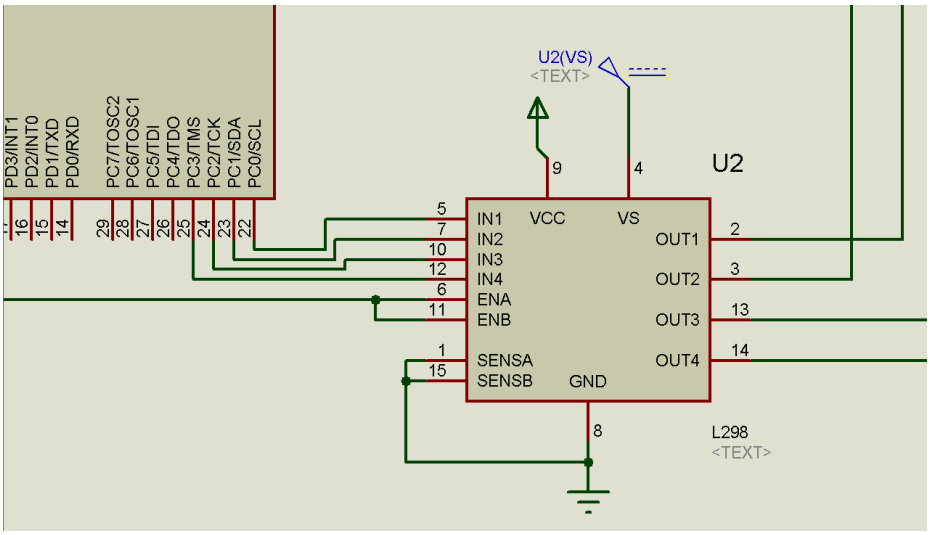
شکل ۱۶-۲-۳ : مدار بسته‌شده در پروتئوس

در این مدار خروجی ۵کلید را همانند شکل زیر به پورت‌های A0,A1,A2,A3,A4 متصل کرده‌ایم:



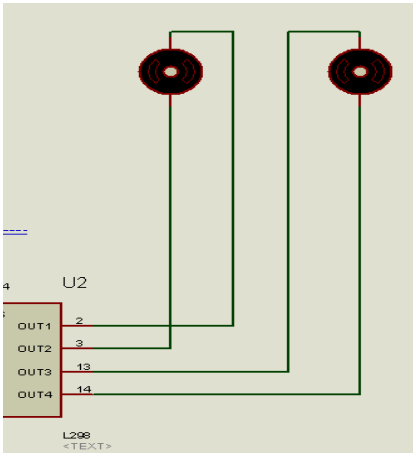
شکل ۱۶-۲-۴ : وصل کردن کلیدها به پین‌های میکروکنترلر

پورت‌های C0,C1,C2,C3 را نیز به همراه OCR1A به L298 متصل کرده‌ایم:



شکل ۱۶-۲-۵ : متصل‌کردن درایور L298 به میکروکنترلر

و خروجی L298 که PWM تقویت‌شده می‌باشد را به موتورها متصل کرده‌ایم:



شکل ۱۶-۲-۶ : متصل‌کردن موتور به درایور L298

محل قرار گرفتن کلیدها نسبت به هم همانند محل قرارگیری سنسورها نسبت به هم روی ربات می‌باشند.

در این حالت اگر کلید ۲ در وضعیت صفر(زمین) و بقیه کلیدها در وضعیت ۱(Power) باشند، نشان‌دهنده‌ی این است که ربات روی مسیر سیاه حرکت می‌کند، پس PINA.2 میکروکنترلر صفر و بقیه PINهای میکروکنترلر، ۱ را دریافت می‌کنند و میکروکنترلر با توجه به این داده‌ها موتورها را طوری به حرکت درمی‌آورد که به مسیر خودش روی خط سیاه ادامه‌دهد.

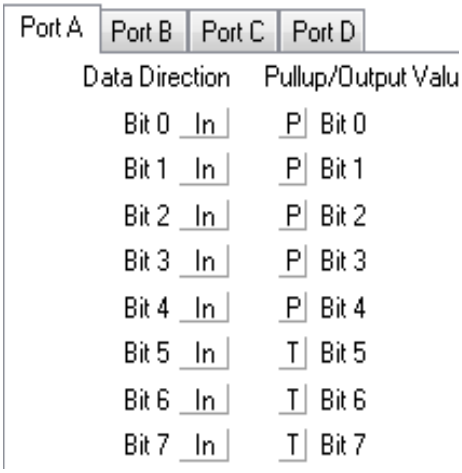
اگر کلید۱ در وضعیت صفر(زمین) و بقیه کلیدها در وضعیت ۱(Power) باشند، در این حالت مسیر سیاه‌رنگ با زاویه‌ای در حدود ۴۵درجه به راست منحرف‌شده است، پس باید ربات به سمت راست حرکت کند که میکروکنترلر دستور این کار را با خاموش کردن موتور راست و روشن کردن موتور سمت چپ با PWM کمتر از ماکزیمم انجام می‌دهد.

اگر کلید شماره صفر در وضعیت صفر(زمین) و بقیه کلیدها در وضعیت ۱(Power) باشند، در این حالت مسیر سیاه‌رنگ با زاویه ای در حدود ۹۰درجه به راست منحرف‌شده‌است، پس باید ربات به سمت راست اما با شدتی بیشتر از حالت قبل حرکت‌کند که میکروکنترلر دستور این کار را با خاموش کردن موتور راست و روشن کردن موتور سمت چپ با PWM ماکزیمم انجام می‌دهد(البته برای اینکه ربات با سرعت بیشتری به سمت راست بچرخد می‌توان موتور چپ را در جهت موافق و موتور راست را در جهت مخالف چرخاند).

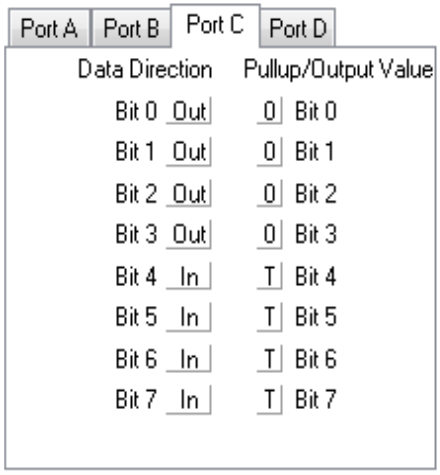
دو وضعیت بعد نیز همانند دو وضعیت قبل هستند، با این تفاوت که موتور باید به سمت چپ حرکت کند و بدین صورت انجام می‌شود که موتور سمت چپ خاموش شده و به موتور سمت راست PWM داده می‌شود.

نوشتن کد در Codevision

پس از انتخاب Atmega16 و تنظیم فرکانس کاری ۸ مگاهرتز تنظیمات درگاه‌ها را مطابق شکل‌های زیر انجام می‌دهیم:



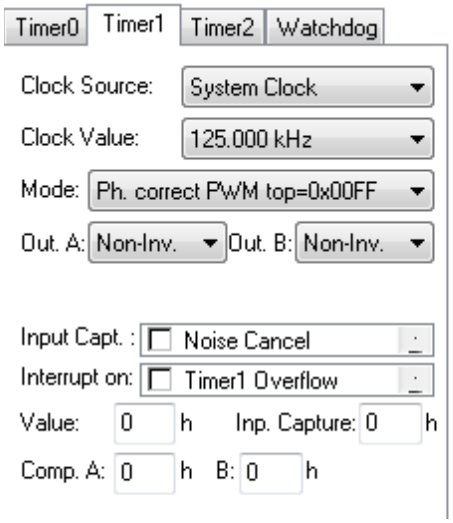
شکل ۱۶-۲-۷ : تنظیمات درگاه A



شکل ۱۶-۲-۸ : تنظیمات درگاه C

به درگاه‌های B و D هم نیازی نداریم.

سپس برای تنظیم PWM به سراغ فعال کردنTimer1 برای استفاده از پایه‌ی OCR1A میکروکنترلر می‌رویم:



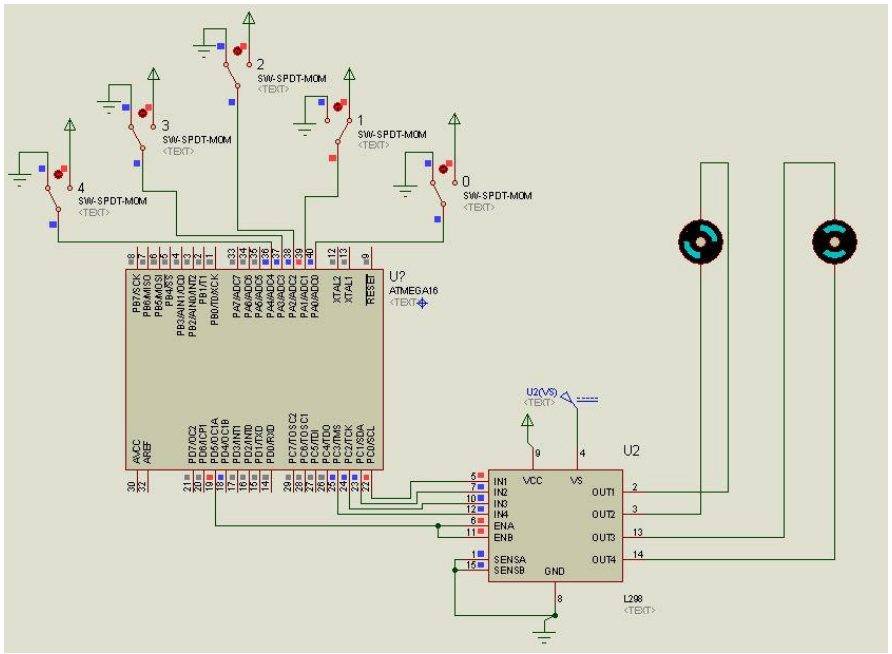
شکل ۱۶-۲-۹: تنظیمات TIMER1 برای فعال‌کردن PWM

کد درون while(1):

```
while (1)
      {
      if(PINA.2==1){
      PORTC.0=1;   // LEFT MOTOR
      PORTC.1=0;   //LEFT MOTOR
      PORTC.2=1;   // RIGHT MOTOR
      PORTC.3=0;   //RIGHT MOTOR
      OCR1A=255;
      }
      if(PINA.1==1){
      PORTC.0=1;   // LEFT MOTOR
      PORTC.1=0; // LEFT MOTOR
      PORTC.2=0;   //RIGHT MOTOR
      PORTC.3=0;   //RIGHT MOTOR
      OCR1A=128;
      }
      if(PINA.0==1){
      PORTC.0=1;
      PORTC.1=0;
      PORTC.2=0;
      PORTC.3=0;
      OCR1A=255;
      }
```

```
if(PINA.3==1){
PORTC.0=0;
PORTC.1=0;
PORTC.2=1;
PORTC.3=0;
OCR1A=128;
}
if(PINA.4==1){
PORTC.0=0;
PORTC.1=0;
PORTC.2=1;
PORTC.3=0;
OCR1A=255;
}
if(PINA.4==0&&PINA.3==0&&PINA.2==0&&PINA.1==0&&PINA.0==0){
PORTC.0=0;
PORTC.1=0;
PORTC.2=0;
PORTC.3=0;
}
}
```

و در انتها نتیجه‌ی شبیه‌سازی کد نوشته شده را در شکل زیر مشاهده می‌کنیم:



شکل ۱۶-۲-۱۰ : نتیجه‌ی شبیه‌سازی

پروژه‌ی۳: موتور همراه با دنده

پیش‌نیاز: PIN & PORT و PWM

در این پروژه قصد طراحی مداری را داریم که شامل یک موتور به همراه دو کلید می‌باشد که یک کلید برای عوض کردن دنده‌ی موتور و دیگری برای معکوس کردن جهت چرخش موتور استفاده می‌شود، همچنین شماره‌ی دنده‌ی موتور همزمان روی یک Seven Segment نمایش داده شود(حداکثر دنده ۷ می‌باشد).

قطعات مورد نیاز برای طراحی

۱)Atmega16

۲)یک عدد موتور

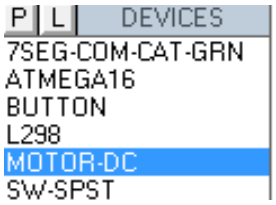
۳)L298 برای درایو کردن موتور

۴)7segment

۵)دو عدد کلید

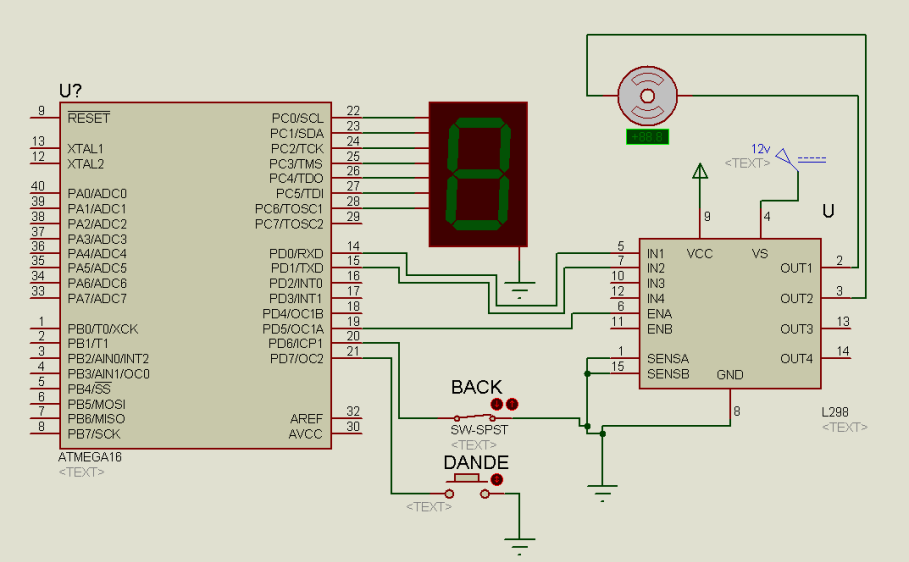
طراحی در پروتئوس

ابتدا قطعات مورد نیاز در پرتئوس را مطابق شکل مقابل انتخاب می‌کنیم:



شکل ۱۶-۳-۱ : قطعات استفاده شده در شبیه‌سازی

برای شبیه‌سازی مدار خود را مطابق شکل زیر در پروتئوس می‌بندیم:



شکل ۱۶-۳-۲ : مدار بسته شده در پروتئوس

کلید BACK را از نوع SW-SPST و کلید دنده را از نوع BUTTON انتخاب کرده‌ایم. نحوه فعال‌سازی Seven Segment همانند LCD نیست، بلکه Seven Segment از ۷ خط ورودی تشکیل شده است و هر خط آن با ۰ و ۱ کردن پایه‌ی مخصوص به آن خط روشن می‌شود.

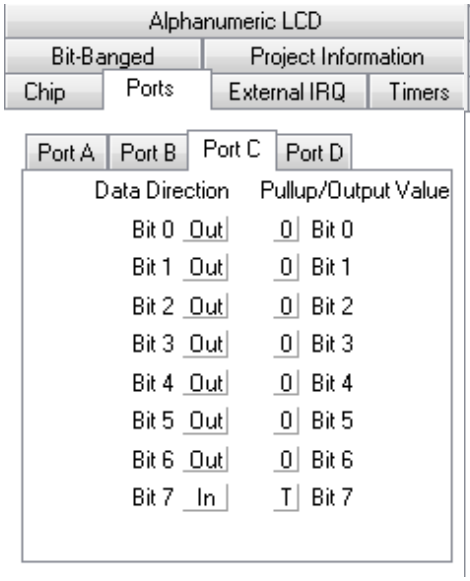
برای اتصال Seven Segment به میکروکنترلر از درگاه C استفاده کرده‌ایم. کلیدها را نیز به PINهای PIND.6 و PIND.7 متصل نموده‌ایم. از تایمر یک هم (OCR1A) برای تولید PWM استفاده کرده‌ایم و همچنین ۰ و ۱ برای تعیین جهت چرخش موتور را با پایه‌های PORTD.0 و PORTD.1 کنترل می‌کنیم.

توجه: PWM را به کمک L298 تقویت کرده و به دو سر موتور وصل کرده‌ایم.

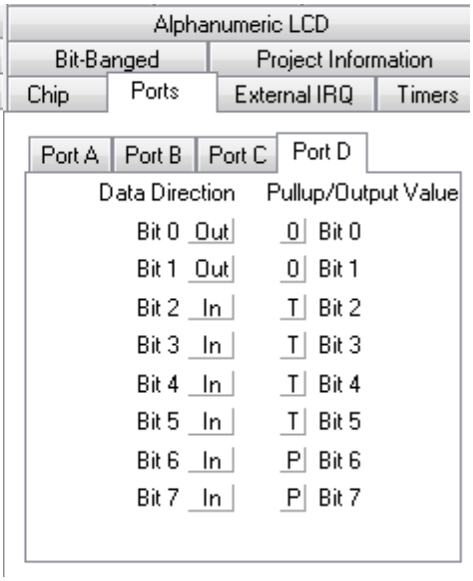
یادآوری: فراموش نکنیم که همیشه باید فرکانس کاری میکروکنترلر را در پروتئوس تنظیم نمائیم.

نوشتن کد در Codevision:

پس از انتخاب ATmega16 و فرکانس ۸ مگاهرتز،تنظیمات درگاه‌های A و B را تغییری نداده و به سراغ تنظیمات درگاه‌های C و D می‌رویم:

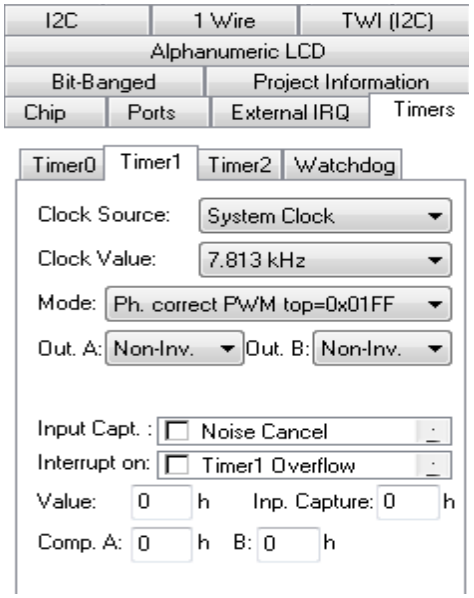


شکل ۱۶-۳-۳: تنظیمات درگاه C



شکل ۱۶-۳-۴ : تنظیمات درگاه D

سپس تولید PWM با استفاده از تایمر ۱:



شکل ۱۶-۳-۵ : تنظیمات تایمر ۱ برای تولید PWM

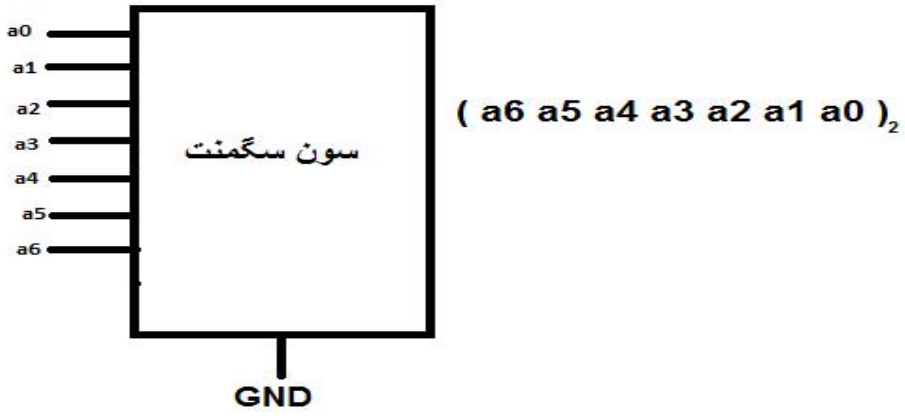
پس از تنظیمات مربوط به کدویزارد، در بخش کدنویسی ابتدا متغیرهای مورد نیاز خود را تعریف می‌کنیم:

```
#include <mega16.h>
#include <delay.h>
// Declare your global variables here
char lcd[8]={0x3F,0x06,0x5B,0x4F,0x66,0x6D,0x7D,0x07};
char m=0;
void main(void)
{
```

در اینجا lcd[8] آرایه‌ای حاوی ۸ کاراکتر می‌باشدکه هر کدام شامل یک وضعیت از صفر تا ۷، Seven Segment می‌باشند(یعنی اولین خانه‌ی آرایه وضعیت صفر و به همین ترتیب آخرین خانه‌ی آرایه وضعیت ۷ را دارا می‌باشد).

هر کدام از درایه‌های این آرایه نمایانگر یک وضعیت Seven Segment می‌باشد بدین شکل که هر درایه ۷ بیتی می‌باشد که برای مثال با توجه به شکل زیر:

0x3f ➔0111111(عبارت سمت چپ در مبنای هگزادسیمال و سمت راست در مبنای باینری می‌باشد) نشان‌دهنده‌ی عدد صفر در سگمنت و 0x4f➔1001111 نشان‌دهنده‌ی عدد ۳ در سگمنت می‌باشد.



شکل ۱۶-۳-۷ : شمای یک سون سگمنت

برای صفر و یک کردن پایه‌های Seven Segment از پورت‌های میکروکنترلر کمک می‌گیریم(در اینجا از PORTC).پس برای اینکه سگمنت عدد صفر را نمایش دهد بایستی PORTC=0X3F باشد(یعنی PORTC=00111111). بیت آخر درگاه c یعنی پایه هشتم چون مورد استفاده قرار نمی‌گیرد همیشه صفر باقی می‌ماند.

یادآوری: در عبارت PORTC=00111111 ، PORTC.0=1 و PORTC.1=1 و PORTC.2=1 و ... و PORTC.6=0 و PORTC.7=0 می‌باشد.

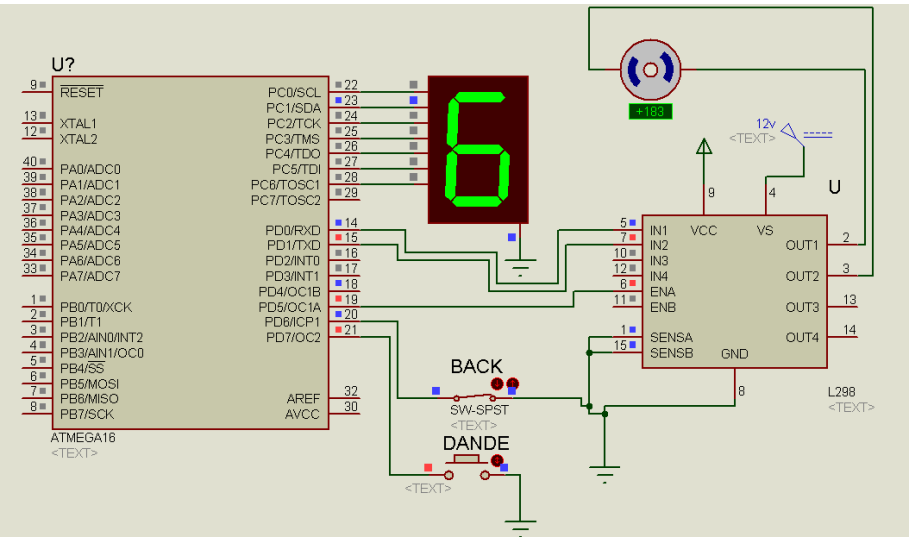
```
PORTC=lcd[0]; //hamoon adad sefr
while (1)
      {
      if(PIND.7==0)
      {
        m++;
        if(m==8) m=0;
        PORTC=lcd[m];

      }
      if(PIND.6==0)
      {
        PORTD.0=0;
        PORTD.1=1;
      }
      else
      {
        PORTD.0=1;
        PORTD.1=0;
      }
      OCR1A=73*m;
      delay_ms(150);
      }
```

با توجه به توضیحات گفته شده کد آن به شکل مقابل می‌باشد:

دستور قبل از While(1) یعنی PORTC=lcd[0] به این دلیل نوشته شده است که پس از روشن شدن ابتدا عدد صفر را نمایش دهد.

دستور ;delay_ms(150) برای ما یک تاخیر به اندازه ۱۵۰ میلی ثانیه فراهم می‌کند و علت استفاده از آن برای این است که سرعت انجام دستورالعمل‌ها بالاست و اگر این دستور را نمی‌گذاشتیم با زدن یک بار کلید دنده، شرط if چندین بار تکرار میشد و موتور چند دنده عوض می‌کرد(البته برای جلوگیری از این مشکل میتوان راه‌حل‌های موثرتری را نیز مورد استفاده قرار داد مثل استفاده از وقفه‌های خارجی، while و...).

و در انتها نتیجه‌ی شبیه‌سازی کد نوشته شده را در شکل زیر مشاهده می‌کنیم:



شکل ۱۶-۳-۷ : نتیجه‌ی شبیه‌سازی پروژه

پروژه‌ی۴: ساختن زمان‌سنج(TIMER)

پیش‌نیاز:LCD و وقفه‌ی تایمر

در این پروژه قصد داریم زمان‌سنجی بسازیم که با راه انداختن مدار شروع به محاسبه‌ی زمان کند و زمان را به صورت hour:min:sec بر روی LCD نمایش دهد.در این پروژه برای ساختن زمان‌سنج از وقفه‌ی تایمر استفاده می‌کنیم.

قطعات مورد نیاز برای طراحی

۱)ATmega16

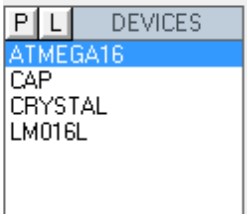
۲)Lcd16x2 برای نمایش زمان

۳)کریستال خارجی RTC 32.768khz

۴) دوعدد خازن 30pF

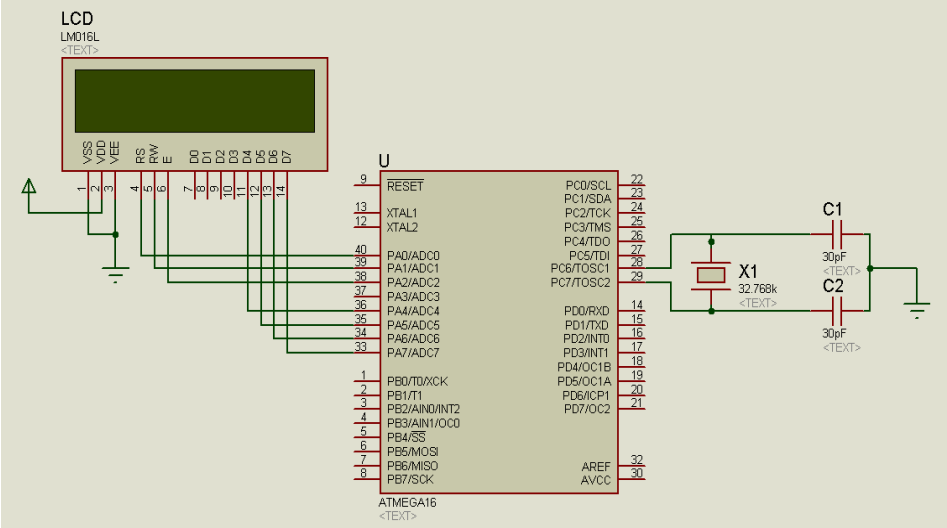
طراحی در پروتئوس

ابتدا قطعات مورد نیاز در پرتئوس را همانند شکل زیر انتخاب می‌کنیم:



شکل ۱۶-۴-۱ : قطعات استفاده شده در شبیه‌سازی

برای شبیه‌سازی، مدار خود را مطابق شکل زیر در پروتئوس می‌بندیم:



شکل ۱۶-۴-۲ : مدار بسته شده در پروتئوس

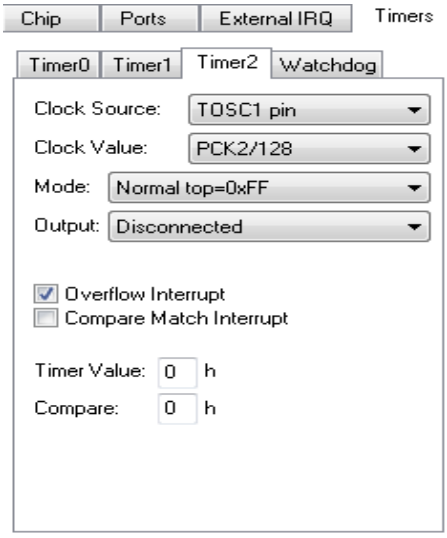
در این مدار کریستال را به پایه‌های TOSC 1 و TOSC 2 میکروکنترلر و LCD را به درگاه A میکروکنترلر متصل می‌کنیم.

ضمنا دو عدد خازن را همانند شکل به دوسر کریستال می‌بندیم(برای کاهش نویز فرکانس ایجاد شده توسط کریستال).

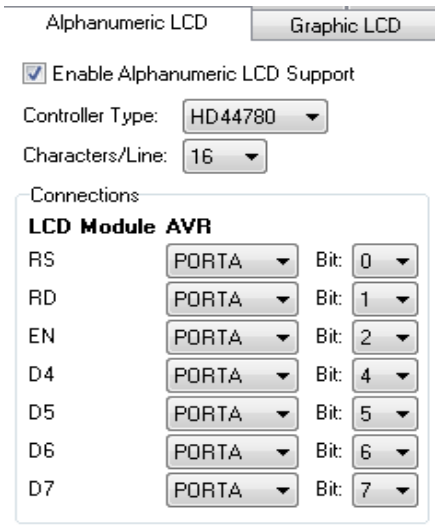
یادآوری: فراموش نکنیم که همیشه باید فرکانس کاری میکروکنترلر را در پروتئوس تنظیم نمائیم.

نوشتن کد در Codevision

پس از انتخاب قطعه (chip) ATmega16 و تنظیم فرکانس کاری میکروکنترلر بر روی ۸ مگاهرتز به سراغ فعال کردن LCD و وقفه‌ی تایمر می‌رویم(به تنظیمات درگاه‌های میکروکنترلر کاری نداریم):



شکل ۱۶-۴-۴ : فعال کردن وقفه‌ی تایمر۲



شکل ۱۶-۴-۳ : تنظیمات LCD

چون ما قصد داریم زمان ۱ ثانیه را ایجاد کنیم، می‌توانیم فرکانس تایمر را از کریستال خارجی تامین کنیم(البته میتوان از فرکانس داخلی میکرو نیز استفاده کرد). بنابراین Clock Source را در حالت TOSC1 PIN قرار داده و Clock Value را مطابق شکل در حالت تقسیم بر ۱۲۸ انتخاب می‌کنیم که در این حالت چون فرکانس کریستال ۳۲۷۶۸ هرتز می‌باشد و این فرکانس بر ۱۲۸ تقسیم می‌شود، بنابراین فرکانس وقفه برابر۲۵۶ هرتز می‌شود و چون Mode را بر روی 0xFF تنظیم نموده‌ایم، وقفه پس از ۲۵۶ کلاک با فرکانس ۲۵۶ هرتز سرریز می‌شود که در این صورت مدت زمان سرریز شدن ۱ثانیه خواهد شد.

بخش کدنویسی در محیط کدویژن

ابتدا متغیرهای مربوطه را بالاتر از وقفه به شکل زیر تعریف می‌کنیم:

```
int second=0, minute=0,hour=0;
char time[15];
```

سپس کد خود را به شکل زیر می‌نویسیم که این کد مربوط به درون تابع وقفه می‌باشد:

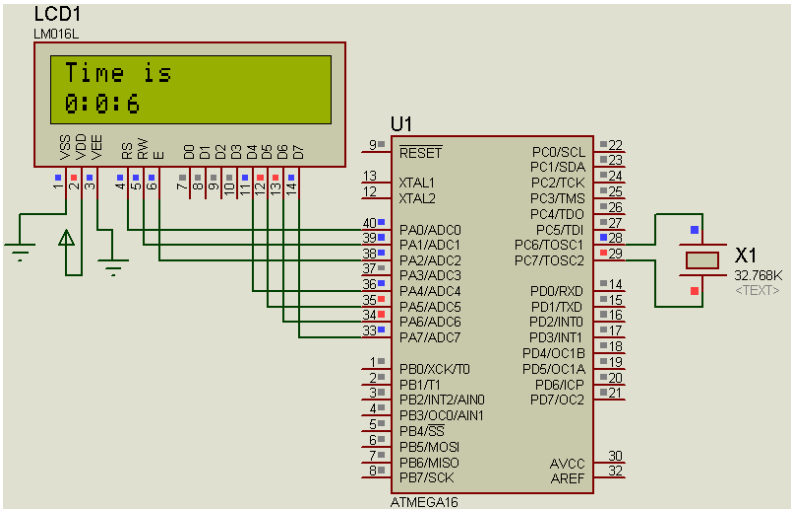
```
interrupt [TIM2_OVF] void timer2_ovf_isr(void)
{
 if(second==59)
        {
                second=0;
                if(minute==59)
                {
                        minute=0;
                        if(hour==24)
                                hour=0;
                        else
                                hour++;
                }
                else
                        minute++;
        }
        else
                second++;

        sprintf(time,"Time is\n%d:%d:%d",hour, minute, second);
        lcd_clear();
        lcd_puts(time);
}
```

به کمک دستور Sprintf می‌توان عبارتی را که در آن یک متغیر یا حتی چندین متغیر به همراه چندین کاراکتر می‌باشد را به یک رشته تبدیل کرد که در اینجا به رشته‌ی time تبدیل نموده‌ایم.

توجه: "\n" یک کاراکتر کنترلی برای رفتن به خط بعد می‌باشد.

ضمناً در این پروژه چون فقط از وقفه‌ی تایمر و LCD استفاده می‌شود همه‌ی کد در داخل تابع وقفه نوشته شده است و درون حلقه‌ی while(1) کدی نوشته نمی‌شود.

در انتها نتیجه‌ی شبیه‌سازی کد نوشته شده را در شکل زیر مشاهده می‌کنیم:



شکل ۱۶-۴-۵ : نتیجه‌ی شبیه‌سازی پروژه

پروژه‌ی۵: صفحه‌کلید(KEYPAD)

پیش‌نیاز: PIN & PORT و LCD

در این پروژه قصد داریم تا اتصال صفحه کلید ماتریسی به میکروکنترلر را آموزش دهیم(نمایش اعداد بین ۰ تا ۹ روی یک LCD به کمک میکروکنترلر و KEYPAD).

فرض کنید می‌خواهیم از ۱۶عدد کلید برای اتصال به پایه‌های میکروکنترلر استفاده کنیم، در این حالت اگر قرار باشد هر کلید را به یک پایه‌ی میکروکنترلر متصل کنیم ۱۶ پایه از میکروکنترلر اشغال می‌شود، با این روش اگر بخواهیم از ۳۲ عدد کلید استفاده کنیم، کل پایه‌های ATmega16 اشغال می‌شود. در این حالت برای حل مشکل از روش ماتریس استفاده می‌شود بدین شکل که مثلاً برای ۱۶ کلید ۴ سطر و ۴ستون در نظر گرفته می‌شود که در این حالت سطر یک و ستون یک همان کلید اول، سطر یک و ستون دو همان کلید دوم و... می‌باشد، در اینصورت برای ۱۶کلید ۸ پایه(۴ سطر+۴ ستون) اشغال می‌شود.

در این پروژه چون می‌خواهیم از ۱۶کلید استفاده کنیم پس نیاز است که کلیدها را به صورت ۴ سطر و ۴ ستون ببندیم که برای این کار۴ سطر را به ۴ پایه از میکروکنترلر وصل می‌کنیم و آنها را به صورت خروجی تعریف می‌کنیم(PORT) و همچنین ۴ ستون را به ۴ پایه‌ی دیگر از میکروکنترلر وصل کرده و آنها را به صورت ورودی(PIN) تعریف می‌کنیم.

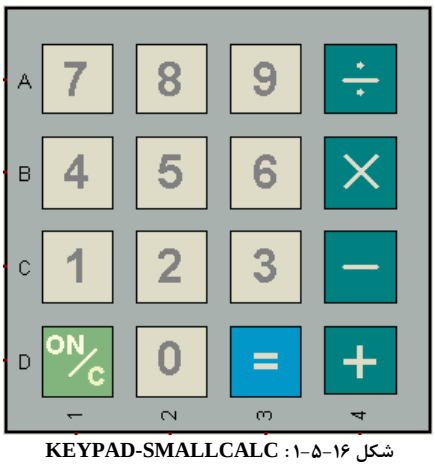
قطعات مورد نیاز برای طراحی:

۱) ATmega 16

۲) LCD 16X2

۳) KEYPAD که در اینجا ما از KEYPAD-SMALLCALC استفاده کرده‌ایم.

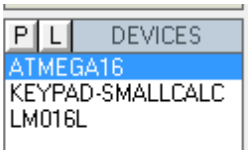
در شکل زیر یک KEYPAD-SMALLCALC را مشاهده می‌کنید:



شکل ۱۶-۵-۱ : KEYPAD-SMALLCALC

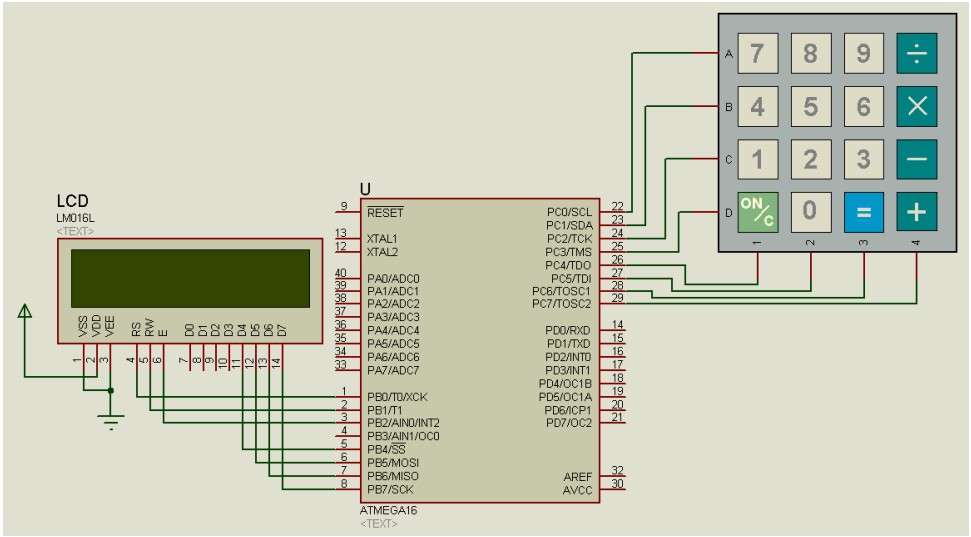
طراحی در پروتئوس

ابتدا قطعات مورد نیاز در پروتئوس را همانند شکل زیر انتخاب می‌کنیم:



شکل ۱۶-۵-۲ : قطعات مورد نیاز در شبیه‌سازی

و برای انجام شبیه‌سازی مداری را مطابق شکل زیر می‌بندیم:



شکل ۱۶-۵-۳ : مدار بسته شده در پروتئوس

همانند شکل LCD را به درگاه B متصل می‌کنیم.

سطرهای A و B و C و D مربوط به KEYPAD را به ترتیب به PORTC.3،PORTC.2،PORTC.1،PORTC.0 متصل می‌کنیم.

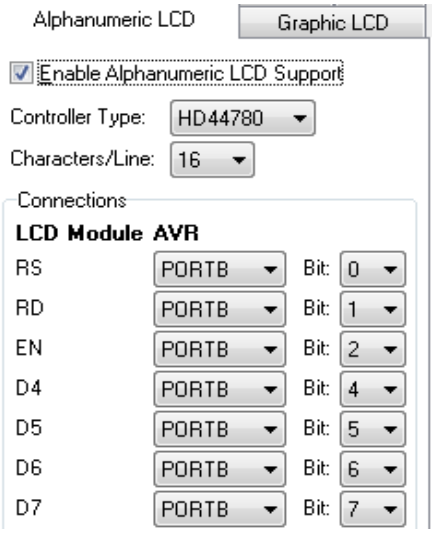
ستون های۱ ، ۲ ، ۳و ۴ مربوط به KEYPAD را به ترتیب به PINC.6،PINC.5،PINC.4 و PINC.7 متصل می‌کنیم(پس سطرها را باید به پورت‌ها و ستون‌ها را نیز باید به پین‌ها وصل نمود).

یادآوری: فراموش نکنیم که همیشه باید فرکانس کاری میکروکنترلر را در پروتئوس تنظیم نمائیم.

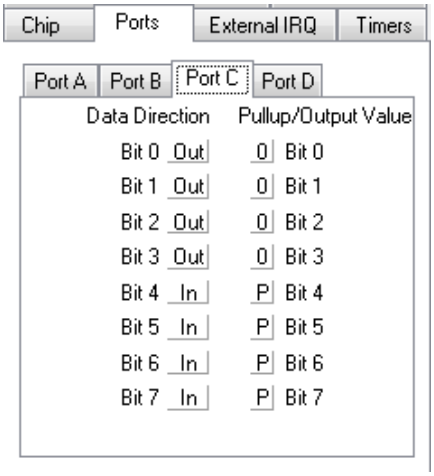
نوشتن کد در Codevision

پس از انتخاب ATmega16 و تنظیم فرکانس کاری میکروکنترلر بر روی ۸ مگاهرتز به سراغ تنظیم درگاه‌ها می‌رویم.

تنظیمات درگاه A و B و D را تغییری نداده و درگاه C و LCD را تنظیم می‌کنیم:



شکل ۱۶-۵-۵ : تنظیمات LCD



شکل ۱۶-۵-۴ : تنظیمات درگاه C

در بخش کدنویسی در محیط کدویژن ابتدا کتابخانه stdio.h را به خاطر استفاده از دستور sprint به کد خود اضافه می‌کنیم:

```
#include <mega16.h>
#include <stdio.h>
```

سپس ثابت‌های زیر را تعریف می‌کنیم:

```
#define column1 PINC.4
#define column2 PINC.5
#define column3 PINC.6
#define column4 PINC.7
```

در این تعریف PINC.4،PINC.5،PINC.6و PINC.7 را به اسم جدید column1 و column2 و column3 و column4 تغییر نام می‌دهیم که در این حالت اگر در if یا جایی از برنامه بخواهیم PINC.4 را نام ببریم بجای آن column1 را نام می‌بریم (این نامگذاری برای فهم بهتر استفاده شده است، تا به‌جای PINC.4 بگوییم ستون یک).

متغیرهای زیر را نیز تعریف می‌کنیم:

```
int v[4]={0XFE,0XFD,0XFB,0XF7};
int num[16]={7,8,9,0,4,5,6,0,1,2,3,0,0,0,0,0};
int number;
char st[1];
int row,c;
```

در آرایه‌ی v چهار مقدار ذخیره می‌شود که این چهار مقدار چهار وضعیت از درگاه C می‌باشد برای مثال 0XFE که مقدار اول آرایه می‌باشد به معنی عدد باینری 11111110 است،که در این حالت همه‌ی پایه‌های درگاه C ، به‌جز پایه‌ی شماره‌ی صفر، یک هستند پس در وضعیت 0XFE تنها PORTC.0 صفر می‌باشد. در وضعیت‌های بعدی نیز به ترتیب PORTC.1(سطر دوم) و PORTC.2(سطر سوم) و PORTC.3(سطر چهارم) صفر(خاموش) می‌باشند.

از آرایه‌ی num برای برگرداندن عدد متناظر با کلید فشار داده شده استفاده می‌شود که ما بجای علامت‌های ضرب و تقسیم و... و دکمه‌ی روشن KEYPAD عدد صفر را قرار داده‌ایم.

کد داخل While(1):

```
while (1)
      {
       for (row=0; row<4; row++){
        c=4;
        PORTC=v[row];
        if(column1==0) c=0;
        if(column2==0) c=1;
        if(column3==0) c=2;
        if(column4==0) c=3;

        if (c!=4){
         number=num[(row*4)+c];
         while(column1==0);
         while(column2==0);
         while(column3==0);
         while(column4==0);
         sprintf(st,"%d",number);
         lcd_puts(st);
        }
       }
      }
```

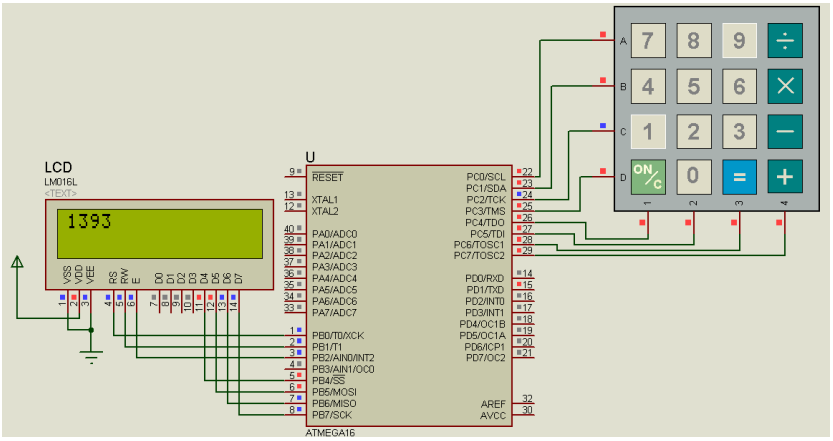
در حلقه‌ی for متغیر row وجود دارد که بیانگر شماره‌ی سطر می‌باشد. این متغیر در این حلقه از ۰ تا ۳ تغییر می‌کند. در ابتدای حلقه مقدار ۴ در متغیر c ریخته می‌شود (مقدار پیش‌فرض ۴ بیانگر این است که هنوز هیچ کلیدی زده نشده است).

در حلقه‌ی اول که مقدار row صفر می‌باشد با دیدن دستور PORTC=v[0](که row=0) خانه‌ی شماره‌ی صفر آرایه‌ی v(که شامل وضعیتی از درگاه c می‌باشد که سطر اول صفر و بقیه‌ی سطرها یک)را در داخل PORTC می‌ریزد، بنابراین بجز سطر اول که صفر می‌شود بقیه‌ی سطرها یک می‌شوند.در چهار شرط if بعد از این دستور اگر کلیدی زده شود با توجه به اینکه این کلید

در کدام ستون زده شده است در متغیر C شماره‌ی کلید ستون فشرده شده ریخته می‌شود. شرط آخر نیز زمانی برقرار است که کلیدی زده شده باشد. اگر کلیدی زده شود، شماره‌ی سطر آن کلید در متغیر row و شماره‌ی ستون آن کلید در متغیر column ذخیره شده و در اولین دستور در شرط آخر مقدار آن کلید که قبلاً در آرایه‌ای به نام num ذخیره کرده بودیم درون متغیر number ریخته می‌شود.

چهار حلقه‌ی while بعد از این دستور نیز بخاطر آن است که تا وقتی کاربر دستش را از روی کلید برنداشته در این حلقه بماند. بعد از آن نیز بخاطر استفاده از دستور lcd_puts(st) و اینکه عبارت داخل پرانتز آن باید یک کاراکتر باشد(منظور st می‌باشد)، ابتدا با دستور sprintf عدد را به رشته تبدیل می‌کنیم و سپس توسط lcd نمایش می‌دهیم.

در انتها نتیجه‌ی شبیه‌سازی کد نوشته شده را در شکل زیر مشاهده می‌کنیم:



شکل ۱۶-۵-۶ : نتیجه‌ی شبیه‌سازی پروژه

پروژه‌ی۶: تنظیم دما

پیش‌نیاز: PIN & PORT،ADC،USART،LCD و تایمر

فرض کنید می‌خواهیم برای یک کارخانه سیستم تنظیم دما بسازیم، بدین شکل که هرثانیه یک‌بار دمای محیط را چک کرده و بر روی یک LCD نمایش دهد و اگر دما بیشتر از ۳۰ درجه شد چراغ خطر و فن را روشن کرده و جمله‌ی DANGER>Temperaturegreater than30degrees را نیز علاوه بر اینکه بر روی LCD نمایش دهد به کمک ارتباط بی‌سیم به یک گیرنده بفرستد.

قطعات مورد نیاز برای طراحی:

۱)ATmega16

۲)LCD 40x2

۳)LM35 (همان سنسور دماسنج می‌باشد)

۴)FAN

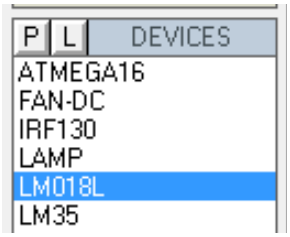
۵)IRF160: یک نوع ماسفت قدرت است که در اینجا برای روشن کردن FAN مورد استفاده قرار می‌گیرد(جهت سوئیچ‌زنی).

۶)LAMP

۷)یک ماژول فرستنده و یک ماژول گیرنده

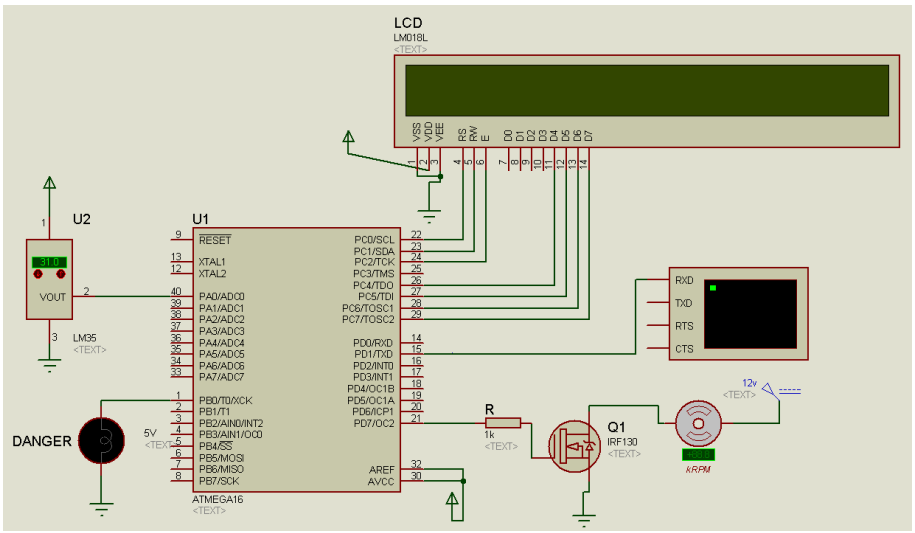
طراحی در پروتئوس

ابتدا قطعات مورد نیاز در پروتئوس را همانند شکل زیر انتخاب می‌کنیم:



شکل ۱۶-۶-۱ : قطعات استفاده شده در شبیه‌سازی

برای شبیه‌سازی مدار خود را مطابق شکل زیر در پروتئوس می‌بندیم:



شکل ۱۶-۶-۲ : مدار بسته شده در پروتئوس

همانند شکل LCD را به درگاه C میکروکنترلر متصل می‌کنیم.

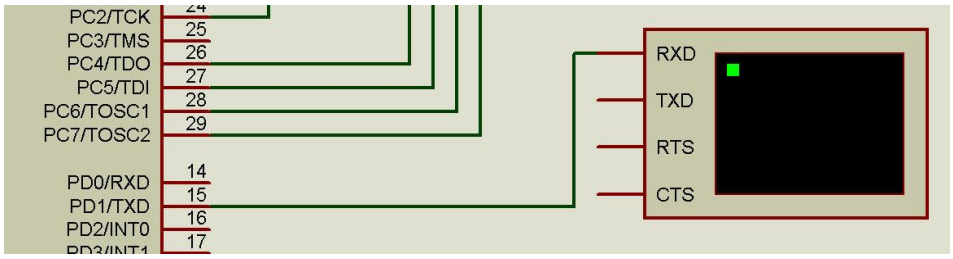
برای راه‌اندازی FAN نیاز به استفاده از PWM نیست و از یک ولتاژ ثابت(۱۲ولت) که به پایه‌ی مثبت آن وصل می‌شود برای راه‌اندازی FAN استفاده می‌کنیم(زیرا نیازی به کنترل سرعت در راه‌اندازی FAN نیست) و همچنین برای تامین توان مورد نیاز FAN از یک ماسفت بعنوان کلید استفاده می‌کنیم که با فرمان پایه‌ی میکروکنترلر سر منفی FAN به زمین متصل شود و FAN روشن گردد(مطابق توضیحات فصل توان).

لامپ خطر را نیز به PORTB.0 وصل نموده‌ایم که در هنگام بروز خطر روشن می‌شود.

پایه‌ی خروجی LM35 را نیز برای اندازه‌گیری دما به ADC0 وصل می‌کنیم.

چون در پروتئوس ماژول‌های فرستنده و گیرنده نداریم پس از VIRTUALTERMINAL استفاده می‌کنیم . این ابزار را می‌توان از نوار ابزار قسمت (INSTRUMENTS) پروتئوس انتخاب کرد که از آن برای آزمایش ارتباط سریال استفاده می‌شود. پایه‌های TXD و RXD مربوط به VIRTUALTERMINAL را به ترتیب به پایه‌های RXD میکروکنترلر زمانی که قرار است میکروکنترلر داده‌ای را دریافت کند و TXD آن زمانی که قرار است میکروکنترلر داده‌ای را ارسال کند متصل می‌کنند. اطلاعاتی که قرار است از طرف VIRTUALTERMINAL ارسال شود(TXD) و یا اینکه توسط این وسیله دریافت شود(RXD) در پنجره‌ای که در هنگام شبیه سازی باز می‌شود نمایش داده می‌شود.

در اینجا چون قصد داریم میکروکنترلر داده‌ای را بفرستد وگیرنده داده‌ی ارسال شده توسط میکروکنترلر را دریافت کند بایستی پایه‌ی TXD (فرستنده) میکروکنترلر را به پایه‌ی RXD (گیرنده) VIRTUALTERMINAL متصل کنیم، مطابق شکل زیر:



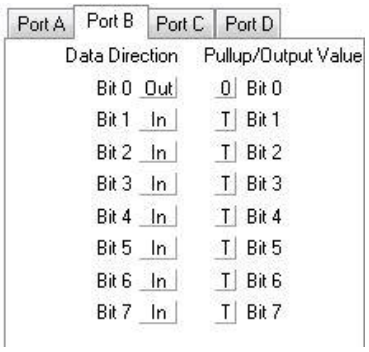
شکل ۱۶-۶-۳ : نحوه‌ی اتصال VIRTUALTERMINAL به میکروکنترلر

در کارهای عملی باید ماژول فرستنده را به پایه‌ی TXD میکروکنترلر وصل کرد تا این ماژول داده را دریافت کرده و از طریق بیسیم ارسال کند و گیرنده را نیز باید به پایه‌ی RXD وسیله‌ای که قصد داریم داده را با آن دریافت کنیم(مثلاً موبایل، کامپیوتر و...) متصل کنیم.

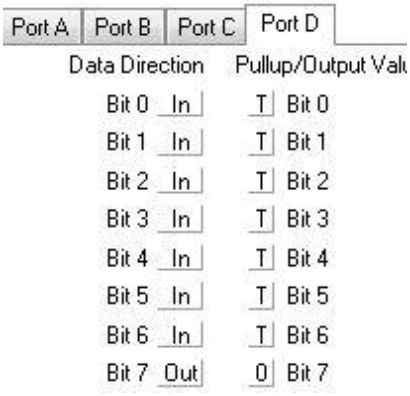
نوشتن کد در Codevision

پس از انتخاب ATmega16 و تنظیم فرکانس کاری ۸ مگاهرتز، به سراغ تنظیمات درگاه‌ها می‌رویم.

تنظیمات درگاه A و C را تغییر نداده و تنظیمات درگاه B و D را مطابق شکل‌های زیر انجام می‌دهیم:

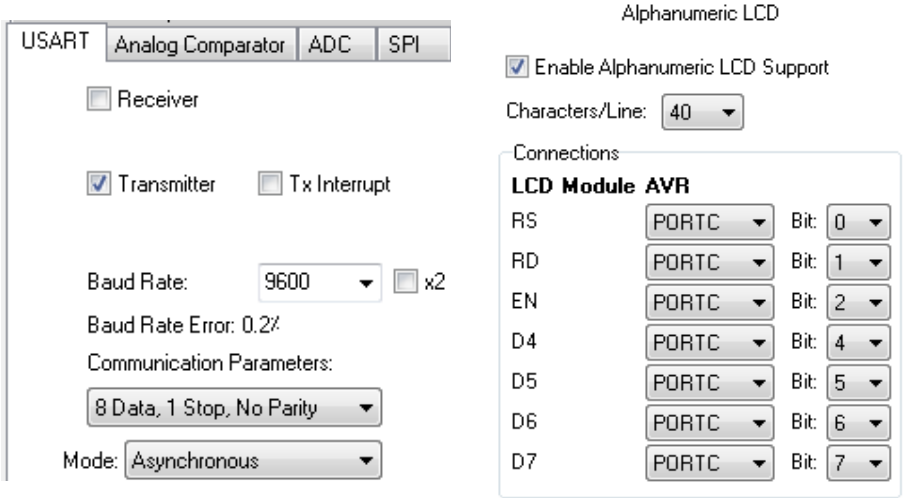


شکل ۱۶-۶-۴ : تنظیمات درگاه B



شکل ۱۶-۶-۵ : تنظیمات درگاه D

تنظیمات مربوط به LCD کاراکتری و واحد USART را نیز انجام می‌دهیم:

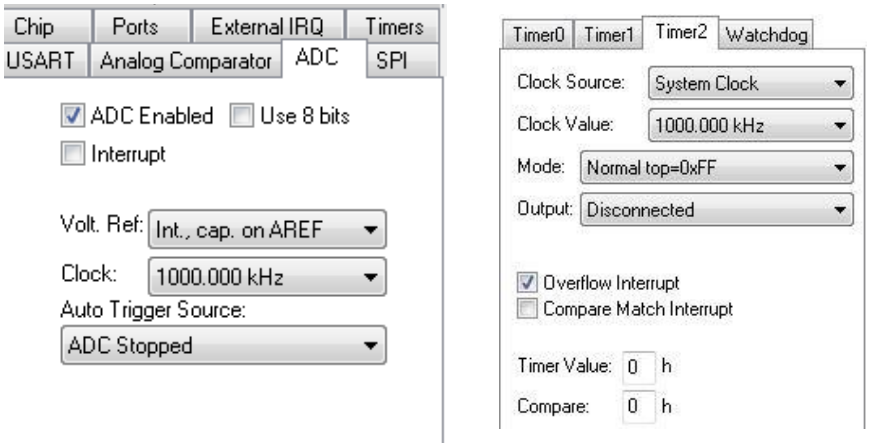


شکل ۱۶-۶-۶ : تنظیمات **LCD**

شکل ۱۶-۶-۷ : تنظیمات واحد **USART**

چون قصد داریم هر ۱ ثانیه یکبار میکروکنترلر کاری را انجام دهد،از وقفه در تایمر استفاده می‌کنیم. بنابراین باید وقفه‌ای را بسازیم که هر ۱ ثانیه یکبار سرریز شود. در اینجا از تایمر ۲ میکروکنترلر استفاده می‌کنیم و فرکانس آن‌را ۱مگاهرتز قرار می‌دهیم و مد کاری آن را بر روی Normal top=0xFF می‌گذاریم. در اینصورت برای ایجاد زمان یک ثانیه: TCNT را برابر عدد 199 یا 0xC7 قرار می‌دهیم تا هر وقفه با توجه به فرکانس تایمر ۰.۲ میلی‌ثانیه طول بکشد، پس برای ایجاد ۱ ثانیه باید ۵۰۰۰ بار وقفه انجام شود.

فعال کردن وقفه‌ی تایمر۲ و واحد ADC میکرو:



شکل ۱۶-۶-۸ : تنظیمات **Timer2** میکروکنترلر

شکل ۱۶-۶-۹ : تنظیمات واحد **ADC**

در بخش کدنویسی خود در محیط کدویژن زمانی که از واحدهای LCD و USART استفاده می‌کنیم باید یک کتابخانه‌ی دیگر یعنی کتابخانه‌ی stdio.h را نیز به ابتدای برنامه‌ی خود اضافه کنیم، چرا که بعضی از توابع کار با این واحدها در این کتابخانه موجود می‌باشد:

```
#include <mega16.h>
#include <stdio.h>
```

سپس متغیرهای مورد نیاز خود را تعریف می‌کنیم:

```
#include <stdio.h>
int n=0,temp=0;
char st[80];
```

برای تنظیم وقفه‌ی تایمر نیز مطابق کد زیر عمل می‌کنیم:

```
// Timer/Counter 2 initialization
// Clock source: System Clock
// Clock value: 1000.000 kHz
// Mode: Normal top=0xFF
// OC2 output: Disconnected
ASSR=0x00;
TCCR2=0x02;
TCNT2=0xC7;
OCR2=0x00;
```

حال کد مربوط به تابع وقفه را می‌نویسیم:

```
interrupt [TIM2_OVF] void timer2_ovf_isr(void)
{

   n++;
   if(n==5000)
   {
    n=0;
    temp=temp/4; // chon adc 4 barabar dama ast
    PORTB.0=0;
    PORTD.7=0;
    lcd_clear();
    sprintf(st,"Temperature is %d degrees",temp);
    lcd_puts(st);
    if(temp>30)
    {
        PORTB.0=1; // DANGER LAMP
        PORTD.7=1; // FAN
        printf("DANGER>Temperature greater than 30 degrees");
        lcd_clear();
        lcd_puts("DANGER>Temperature greater than 30 degrees");
    }
    }
    TCNT2=0xC7;
}
```

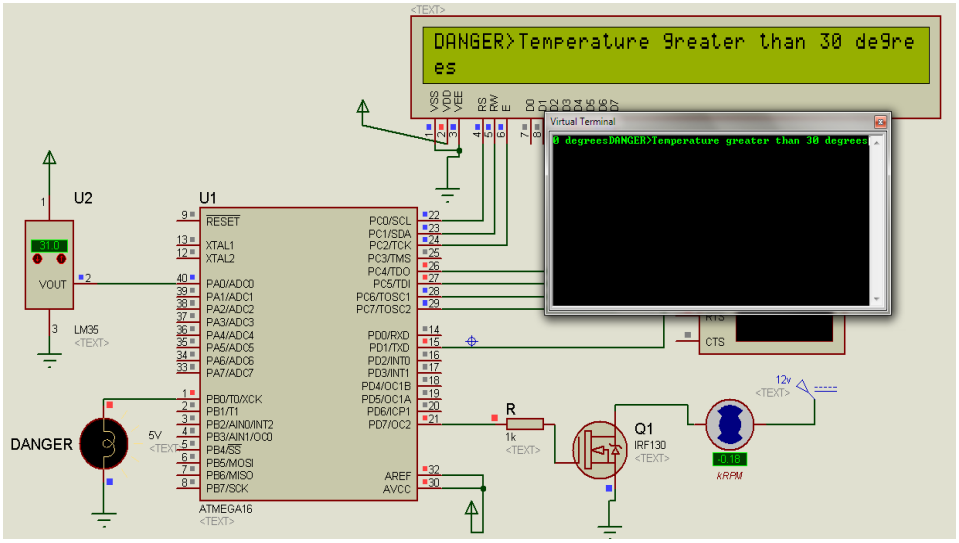
یادآوری: علت اینکه temp بر ۴ تقسیم شده است به خاطر این است که ADC در این تنظیمات ۴ برابر دما را می‌خواند.

کد داخل While(1):

```
while (1)
      {
          temp=read_adc(0);

      }
```

نکته: علت اینکه دستور read_adc(0) را در داخل while(1) نوشتیم به خاطر این است که دستور مربوط به خواندن پایه‌های ADC (read_adc()) را نمی‌توان در داخل وقفه نوشت چرا که خود دستور read_adc() یک وقفه می‌باشد و میکروکنترلرهای ۸ بیتی AVR دارای وقفه‌های تو در تو نمی‌باشند و با نوشتن این دستور داخل یک وقفه احتمال اشتباه عملکرد برنامه بوجود می‌آید بنابراین آن را در While(1) می‌نویسیم.

و در انتها نتیجه‌ی شبیه‌سازی کد نوشته شده را در شکل زیر مشاهده می‌کنیم:



شکل ۱۶-۶-۱۰ : نتیجه‌ی شبیه‌سازی پروژه

پروژه‌ی۷: تولید موج سینوسی به کمک آی‌سیDAC

پیش‌نیاز :PIN & PORT و وقفه

در میکروکنترلر ما با سطح‌های منطقی(۰ و ۱) یعنی ولتاژهای دیجیتال کارمی‌کنیم حتی PWM نیز به صورت پله‌ای با دو سطح ۰ و ۱ ایجادمی‌شود. در این پروژه ما قصدداریم از صفر و یک‌های میکروکنترلر استفاده‌کرده و یک ولتاژ آنالوگ سینوسی بسازیم. بدین‌منظور ما از یک آی‌سی به نام DAC(DIGITAL ANALOG CONVERTER) استفاده می‌کنیم و مقادیر دیجیتال(باینری) را به آنالوگ تبدیل می‌کنیم.

نکته:آی‌سی‌های DAC برعکس کار ADC میکروکنترلر را انجام می‌دهند.

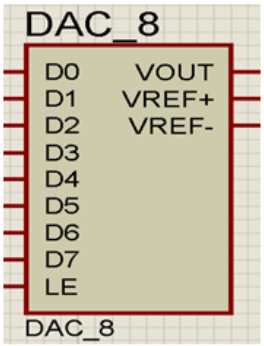
قطعات مورد نیاز برای طراحی

۱)ATmega16

۲)اسیلوسکوپ

۳)آی‌سیDAC ۸بیتی: استفاده از مبدلهای دیجیتال به آنالوگ و آنالوگ به دیجیتال امروزه به خصوص در مدارات میکروکنترلر و میکروپروسسور افزایش چشم‌گیری داشته‌است. البته در میکروکنترلرهای سری AVR بسته به نوع آن‌ها دارای چندین مبدل انالوگ به دیجیتال می‌باشند، ولی فعلاً از مبدل دیجیتال به آنالوگ در آن‌ها استفاده نشده است، که برای استفاده از این نوع مبدلها باید از آی‌سی‌های مربوط به آن استفاده کرد.این آی‌سی‌ها، آی‌سی‌های DAC می‌باشند. آی‌سی DAC که در این پروژه استفاده‌کرده‌ایم نوع ۸بیتی آن بوده که یعنی ورودی آن ۸بیتی می‌باشد(چون قرار است ورودی آن یک درگاه ATmega16 باشد که ۸بیتی است).

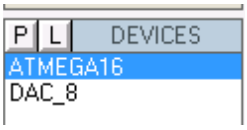
شکل ۱۶-۷-۱ : آی‌سی DAC



شکل ۱۶-۷-۲ : آی‌سی DAC مورد استفاده در پرتئوس

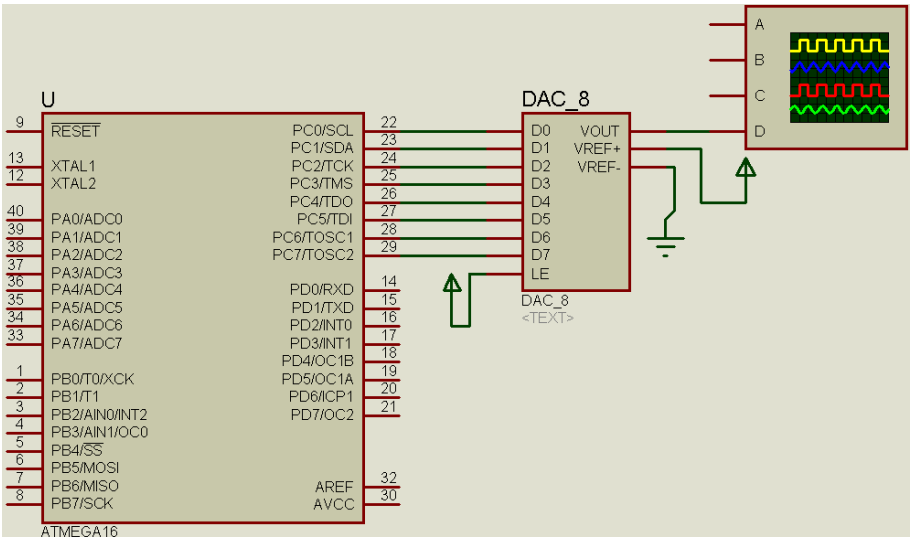
طراحی در پروتئوس

ابتدا قطعات موردنیاز در پروتئوس را انتخاب می‌کنیم:



شکل ۱۶-۷-۳ : انتخاب قطعات در پروتئوس

مداری را مطابق شکل زیر می‌بندیم:



شکل ۱۶-۷-۴ : مدار بسته‌شده در پروتئوس

همان‌طور که در شکل بالا مشخص است DAC را به درگاه C متصل نموده‌ایم(پایه‌ی D0 را به C0 و به همین ترتیب تا D7 را به C7 متصل نموده‌ایم)، که دراین‌صورت این ۸ پایه درگاه C یک عدد ۸بیتی باینری را می‌سازد که بیت پرارزش آن D7 (C7) و بیت کم‌ارزش آن D0(C0) می‌باشد.

DAC_8 را همانند شکل بایاس می‌کنیم.که +Vref و LE آن را به power و -Vref آن را به Gnd(زمین) می‌زنیم. همچنین Vout(ولتاژ خروجی) را به یک اسیلوسکوپ جهت نمایش ولتاژ خروجی می‌زنیم.

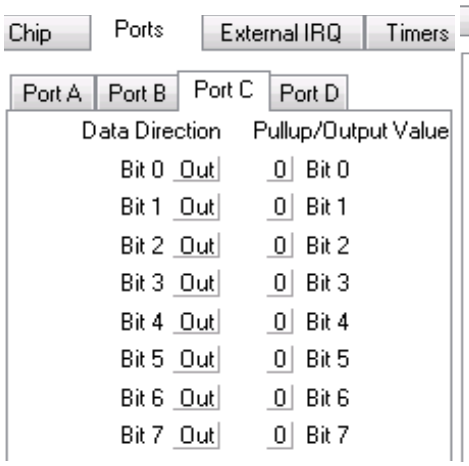
نکته: اسیلوسکوپ(oscilloscope) را از جعبه ابزار( ) انتخاب می‌کنیم.

یادآوری: فراموش نکنیم که همیشه باید فرکانس کاری میکروکنترلر را در پروتئوس تنظیم نمائیم.

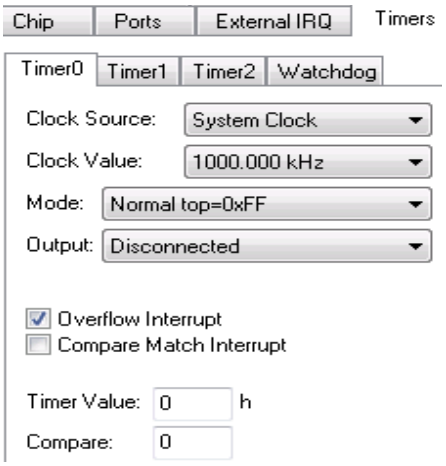
نوشتن کد در Codevision

پس از انتخاب ATmega16 و فرکانس کاری ۸مگاهرتز به سراغ تنظیم درگاه‌ها می‌رویم. تنظیمات درگاه A، B و D را تغییری نمی‌دهیم.

تنظیم درگاه C و فعال کردن وقفه تایمر صفر:



شکل ۱۶-۷-۵ : تنظیمات درگاه C



شکل ۱۶-۷-۶ : تنظیمات وقفه‌ی تایمر صفر

علت استفاده از تایمر به این خاطر است که می‌خواهیم یک سری دستور با یک دوره‌ی زمانی ثابت انجام شوند.

در بخش کدنویسی کد را به صورت زیر می‌نویسیم و در ابتدا متغیرهای زیر را تعریف می‌کنیم:

```
unsigned char sin[]={  //sine wave
   0x80,0x83,0x86,0x89,0x8c,0x8f,0x92,0x95,0x98,0x9c,0x9f,0xa2,0xa5,0xa8,0xab,0xae,
   0xb0,0xb3,0xb6,0xb9,0xbc,0xbf,0xc1,0xc4,0xc7,0xc9,0xcc,0xce,0xd1,0xd3,0xd5,0xd8,
   0xda,0xdc,0xde,0xe0,0xe2,0xe4,0xe6,0xe8,0xea,0xec,0xed,0xef,0xf0,0xf2,0xf3,0xf5,
   0xf6,0xf7,0xf8,0xf9,0xfa,0xfb,0xfc,0xfc,0xfd,0xfe,0xfe,0xff,0xff,0xff,0xff,0xff,
   0xff,0xff,0xff,0xff,0xff,0xff,0xfe,0xfe,0xfd,0xfc,0xfc,0xfb,0xfa,0xf9,0xf8,0xf7,
   0xf6,0xf5,0xf3,0xf2,0xf0,0xef,0xed,0xec,0xea,0xe8,0xe6,0xe4,0xe2,0xe0,0xde,0xdc,
   0xda,0xd8,0xd5,0xd3,0xd1,0xce,0xcc,0xc9,0xc7,0xc4,0xc1,0xbf,0xbc,0xb9,0xb6,0xb3,
   0xb0,0xae,0xab,0xa8,0xa5,0xa2,0x9f,0x9c,0x98,0x95,0x92,0x8f,0x8c,0x89,0x86,0x83,
   0x80,0x7c,0x79,0x76,0x73,0x70,0x6d,0x6a,0x67,0x63,0x60,0x5d,0x5a,0x57,0x54,0x51,
   0x4f,0x4c,0x49,0x46,0x43,0x40,0x3e,0x3b,0x38,0x36,0x33,0x31,0x2e,0x2c,0x2a,0x27,
   0x25,0x23,0x21,0x1f,0x1d,0x1b,0x19,0x17,0x15,0x13,0x12,0x10,0x0f,0x0d,0x0c,0x0a,
   0x09,0x08,0x07,0x06,0x05,0x04,0x03,0x03,0x02,0x01,0x01,0x00,0x00,0x00,0x00,0x00,
   0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x01,0x02,0x03,0x03,0x04,0x05,0x06,0x07,0x08,
   0x09,0x0a,0x0c,0x0d,0x0f,0x10,0x12,0x13,0x15,0x17,0x19,0x1b,0x1d,0x1f,0x21,0x23,
   0x25,0x27,0x2a,0x2c,0x2e,0x31,0x33,0x36,0x38,0x3b,0x3e,0x40,0x43,0x46,0x49,0x4c,
   0x4f,0x51,0x54,0x57,0x5a,0x5d,0x60,0x63,0x67,0x6a,0x6d,0x70,0x73,0x76,0x79,0x7c
};
int n=0;
```

sin یک آرایه‌ی ۲۵۶تایی بوده که شامل ۲۵۶وضعیت مختلف از درگاه C می‌باشد(چون هر درگاه ۸ بیت دارد، پس هر وضعیت آن درگاه یک عدد ۸بیتی‌است که دراین‌صورت ۲۵۶ وضعیت مختلف دارد).

برای مثال عدد هگزادسیمال 0X80 که عدد باینری 10000000 می‌باشد، به معنای این است که پورت ۰ تا ۶ آن درگاه صفر(صفر ولت) و پورت ۷ آن ۱(۵ولت)می‌باشد.

اینکه این ۲۵۶ وضعیت چه معنایی می‌دهند در ادامه توضیح‌خواهیم‌داد.

متغیر n نیزبه عنوان شمارشگر آرایه می‌باشد.

کد داخل وقفه:

می‌خواهیم یک موج سینوسی تولیدکنیم. کد را برای یک دوره‌تناوب آن(که شامل ۳۶۰درجه میباشد)مینویسیم و با تکرار این کد در داخل وقفه، موج شکل می‌گیرد.

برای ایجاد این موج سینوسی ولتاژ صفر را به عنوان پیک منفی و ولتاژ ۵ولت را به عنوان پیک مثبت این موج درنظرمی‌گیریم(پس صفر موج سینوسی ۲.۵ولت است).

چون ورودیDAC ۸بیت می‌باشد(از نوع ۸بیتی آن استفاده‌کردیم)، پس ورودی آن دارای ۲۵۶حالت مختلف می‌باشد.برای اینکه بتوانیم از ۰ تا ۳۵۹ درجه‌ی موج را توسط این ۲۵۶حالت نشان‌دهیم، باید زاویه‌ی ما از صفردرجه به فاصله ۱.۴۱درجه (360/256 = 1.41) تا ۳۵۹ درجه تغییرکند. بدین صورت می‌توانیم موج سینوسی را در یک دوره به صورت گسسته بسازیم که اگر مثلاً از ۱۰بیت یا بیشتر به جای ۸بیت استفاده می‌کردیم، موج سینوسی پیوسته‌تر می‌بود. حال به سراغ دامنه‌ی موج در این زاویه‌ها می‌رویم(که با فاصله ۱.۴۱ درجه تغییرمیکنند). برای بدست-آوردن دامنه از فرمول زیر استفاده‌می‌کنیم:

دامنه = 128 + 127 * SIN(teta)

که teta از ۰ تا ۳۵۹ به فاصله ی ۱.۴۱ درجه تغییر می‌کند. برای مثال در لحظه‌ی اول که teta صفر درجه است در فرمول بالا دامنه ۱۲۸ می‌شود که بیانگر ۲.۵ولت یعنی نقطه‌ی صفر موج سینوسی است(چون ما از ۸بیت استفاده‌کردیم در این صورت ماکزیمم عددی که می‌توان با این ۸بیت نشان‌داد ۲۵۵ می‌باشد که بیانگر ماکزیمم ولتاژ یعنی ۵ولت می‌باشد همچنین مینیمم عدد با این ۸بیت صفرمی‌باشد که بیانگر صفر ولت است). پس دامنه‌ی لحظه‌ی اول ۱۲۸ است که در مبنای ۱۶، 0X80 می‌شود و درایه‌ی اول آرایه sin تعریف‌شده‌است.

در لحظه‌ی بعدی teta ۱.۴۱ درجه می‌شود، دراین‌صورت دامنه حدوداً برابر ۱۳۱ شده که در مبنای ۱۶ برابر 0x83 بوده و درایه‌ی دوم آرایه‌ی sin تعریف شده را تشکیل‌می‌دهد. بقیه‌ی درایه‌های این آرایه نیز به همین صورت به‌دست‌آمده‌اند تا یک دوره‌ی موج سینوسی را بسازند.

علت اینکه اعداد دامنه را مستقیماً استفاده کرده‌ایم و از فرمول ذکرشده استفاده نکرده‌ایم، به خاطر این است که تاخیر محاسبه sin را حذف‌کنیم تا دستورات به سرعت پشت سرهم انجام شوند و موج حاصل پیوسته‌تر شود.

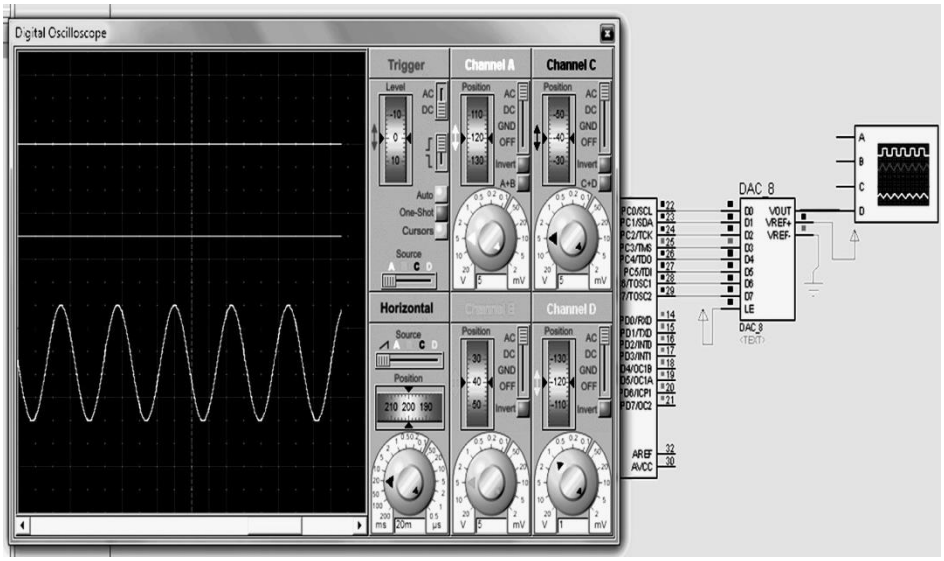
بنابراین کد داخل وقفه به شکل زیر می‌باشد:

```
interrupt [TIM0_OVF] void timer0_ovf_isr(void)
{
PORTC=sin[n];
n++;
if(n==256) n=0;
}
```

در داخل while(1) نیز هیچ کدی نوشته نمی‌شود.

نکته: با این روش دوره زمانی موج حاصل برابر ۲۵۶*(زمان سرریز وقفه) خواهد شد که با تغییر زمان سرریز وقفه میتوان دوره زمانی موج حاصل را کنترل کرد(فرکانس نیز از رابطه $1/T$ که T دوره می‌باشد به‌دست می‌آید).

و در انتها نتیجه‌ی شبیه‌سازی کد نوشته شده را در شکل زیر مشاهده می‌کنیم:



شکل ۱۶-۷-۸ : نتیجه‌ی شبیه‌سازی

پروژه‌ی۸: ساعت و تاریخ میلادی به کمک آی‌سی DS1307

پیش‌نیاز: PORT&، PIN، ADC و I2C

در این پروژه قصد داریم به کمک ارتباط I2C، بین میکروکنترلر ATmega16 و آی‌سی DS1307 ارتباط برقرار کنیم و با نوشتن برنامه بر روی میکروکنترلر با استفاده از کتابخانه‌ی DS1307.H، ساعت و تاریخ میلادی را از روی این آی‌سی خوانده و بر روی یک LCD نمایش دهیم، همچنین با اضافه کردن ۴ کلید امکان ویرایش زمان و تاریخ را نیز فراهم کنیم.

قطعات مورد نیاز برای طراحی

۱) **ATmega 16**

۲) **LCD 16X2**

۳) **۴ عدد کلید** (BUTTON)

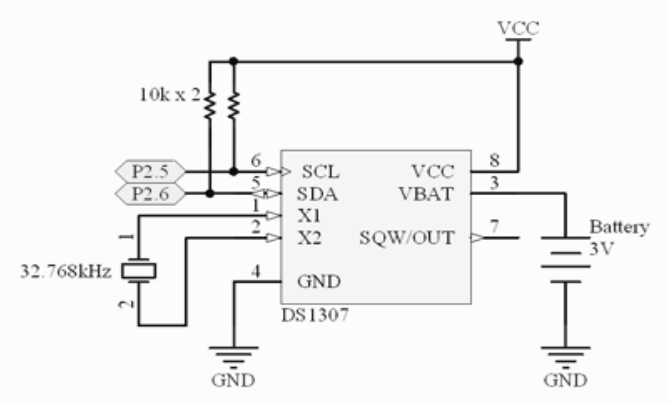
۴) **آی‌سی DS1307**: آی‌سی ساعت و تاریخ می‌باشد که با وصل کردن یک کریستال ساعت(۳۲.۷۶۸ کیلوهرتز) به پایه‌ی X1 و X2 آن و متصل نمودن یک باتری ۳ولت(به منزله‌ی باتری کمکی) به پایه‌ی Vbat و GND شروع به کار می‌کند و ساعت و تاریخ را در حافظه‌ی خود ذخیره می‌کند. این آی‌سی اطلاعات را به کمک ارتباط I2C در اختیارکاربر قرار می‌دهد. بنابراین باید آن را به وسیله‌ی ارتباط I2C به میکروکنترلر متصل نمود. ضمناً می‌توان ساعت و تاریخ مورد نظر خود را روی حافظه‌ی آن نوشت(همانند زمانی که شما یک ساعت می‌خرید و ساعت و تاریخ آن را اصلاح می‌کنید). در شکل زیر بسته‌بندی DIP آی‌سی DS1307 را مشاهده می‌کنید:

شکل ۱۶-۸-۱: DS1307

البته به‌غیر از وصل باتری کمکی (Vbat) به آی‌سی هنگامی که می‌خواهیم از این آی‌سی اطلاعات را بخوانیم و یا بر روی آن بنویسیم باید ولتاژ ۵ ولت را به پایه VCC آن متصل کنیم. مزیت استفاده از این آی‌سی این است که با قطع تغذیه‌ی میکروکنترلر و تغذیه‌ی آی‌سی، ساعت عقب نمی‌ماند و دقیق به کار خود ادامه می‌دهد.

توجه: تا وقتی باتری کمکی(Vbat) به این آی‌سی متصل است این آی‌سی کار می‌کند و محاسبه‌ی زمان را انجام می‌دهد و با قطع آن محاسبه‌ی زمان نیز متوقف می‌شود.

نحوه‌ی بستن آی‌سی DS1307 در مدار مطابق شکل زیر می‌باشد. به دو مقاومت ۱۰کیلو اهمی که یک سر آنها به پایه‌های SCL و SDA و سر دیگرشان به ۵ ولت(VCC) وصل شده است، دقت کنید همانطور که از فصل ارتباط I2C بخاطر داریم این مقاومت‌ها، مقاومت‌های بالاکشنده (Pull-Up) نام دارند و برای این است که در ارتباط I2C سطح منطقی خط را در حالت بیکاری (Idle) در سطح منطقی ۱ یا High نگه دارند:



شکل ۱۶-۸-۲ : نحوه‌ی بستن آی‌سی DS1307

قطعات استفاده شده در مدار فوق:

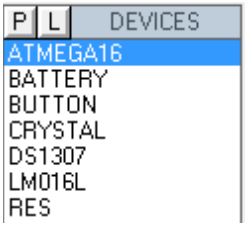
۱)کریستال 32.768 KHZ (برای استفاده در DS1307)

۲)دو عدد مقاومت ۱۰ کیلو اهم(مقاومت‌های بالاکشنده(Pull-Up) برای بالا نگه داشتن سطح منطقی خط ارتباطی I2C)

۳)باتری ۳ ولت

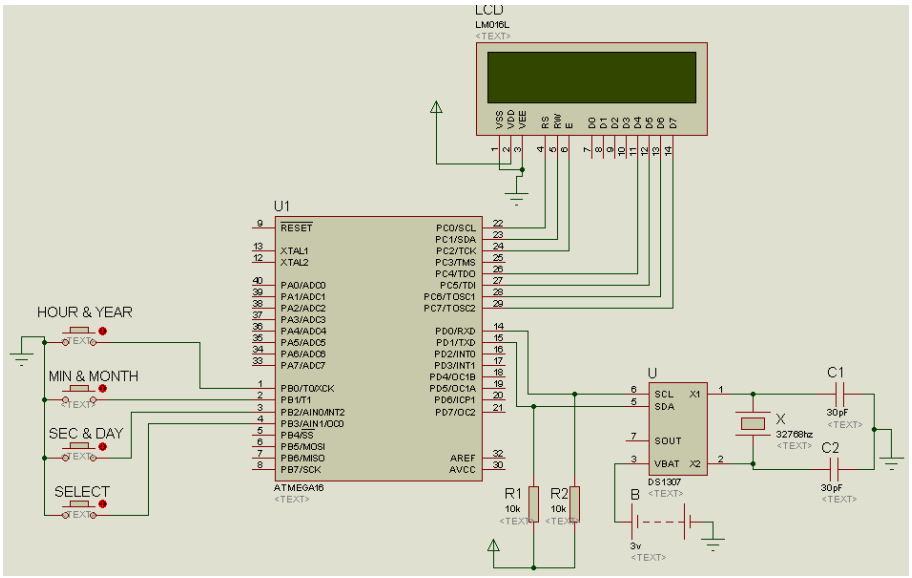
طراحی در پروتئوس

ابتدا قطعات مورد نیاز در پرتئوس را همانند شکل زیر انتخاب می‌کنیم:



شکل ۱۶-۸-۳ : قطعات استفاده شده در شبیه‌سازی

برای شبیه‌سازی مدار خود را مطابق شکل شماره‌ی زیر در پروتئوس می‌بندیم:

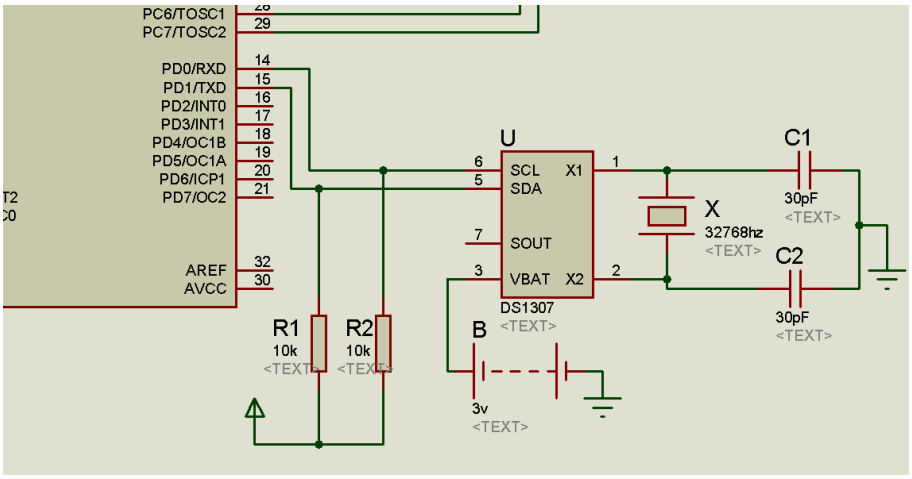


شکل ۱۶-۸-۴ : مدار بسته شده در پروتئوس

همانطور که در مدار بسته شده مشاهده می‌کنید، ۴عدد کلید وجود دارد که با زدن کلید SELECT (پایین‌ترین کلید) وارد تنظیمات ساعت شده و می‌توان با زدن ۳ کلید دیگر زمان و یا تاریخ را تغییر داد.

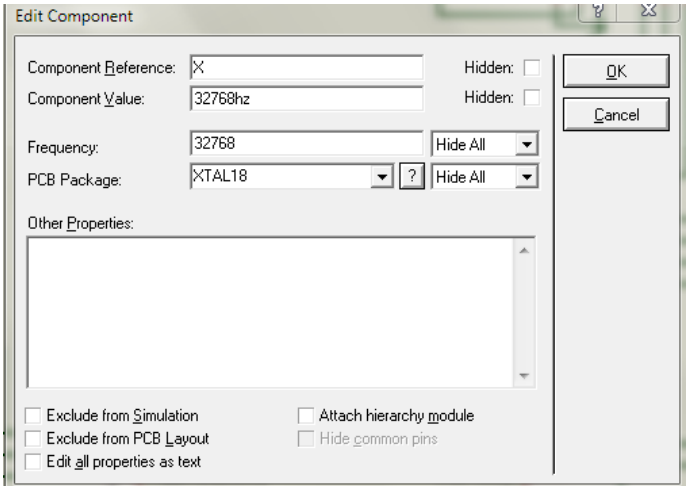
همانند شکلLCD به درگاه C متصل شده است و در ارتباط I2C دو پایه‌یD0 و D1 میکروکنترلر به ترتیب نقش SCL و SDA را دارند.

نحوه‌ی بستن DS1307 به میکروکنترلر نیز مطابق شکل زیر می‌باشد:



شکل ۱۶-۸-۵ : نحوه‌ی اتصال DS1307 به میکروکنترلر ATmega16

برای تغییر فرکانس کریستال با دو بار کلیک روی کریستال وارد تنظیمات آن شده و مقدار آن را تغییر می‌دهیم(مطابق شکل ۱۶-۸-۶):



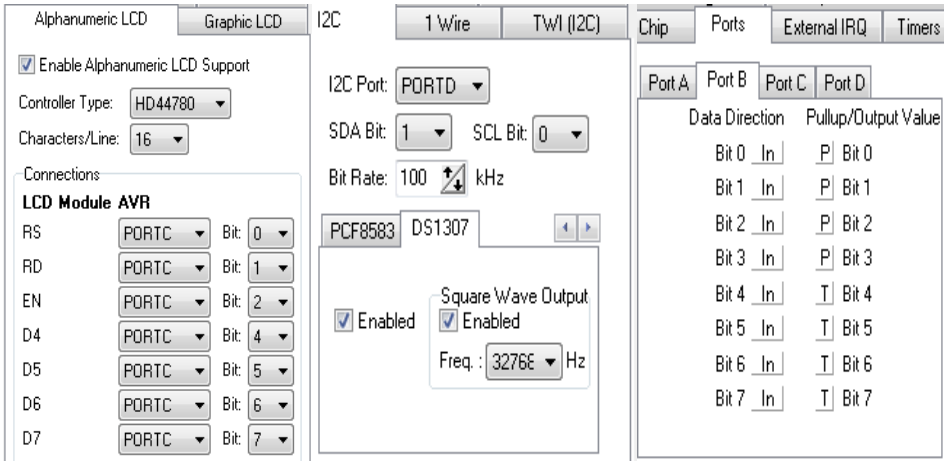
شکل ۱۶-۸-۶ : تنظیمات فرکانس کریستال در پروتئوس

ضمنا دو عدد خازن را همانند شکل به دوسر کریستال می‌بندیم(برای کاهش نویز فرکانس ایجاد شده توسط کریستال).

نوشتن کد در Codevision

پس از انتخاب ATmega16 و تنظیم فرکانس کاری میکروکنترلر بر روی ۱ مگاهرتز، به سراغ تنظیمات درگاه‌ها می‌رویم.

تنظیمات درگاه‌های A، C و D را تغییر نداده و درگاه B ، واحد I2C و LCD را تنظیم می‌نمائیم:



شکل ۱۶-۸-۷ : تنظیمات درگاه B
شکل ۱۶-۸-۸ : تنظیمات واحد I2C
شکل ۱۶-۸-۹ : تنظیمات LCD

همانطور که قبلاً هم گفته شد SCL را بر روی PORTD.0 و SDA را روی PORTD.1 تنظیم می‌کنیم.

```
#include <mega16.h>
#include <stdio.h>
#include <delay.h>
```

در بخش کدنویسی در محیط کدویژن در ابتدای کد خود کتابخانه‌های روبرو را اضافه می‌کنیم.کتابخانه delay.h به خاطر استفاده از دستور()delay_ms می‌باشد و کتابخانه‌ی stdio.h نیز بخاطر استفاده از دستور sprintf می‌باشد.

سپس متغیرهای زیر را تعریف می‌کنیم:

```
// Declare your global variables here
unsigned char h,m,s,year,month,day,hour,min,sec,year_new,month_new,day_new;
int select=0;
char st[32];

void main(void)
{
```

چون از دستور rtc_get و rtc_set استفاده کردیم پس باید متغیرهایی که به عنوان آرگومان در این توابع مورد استفاده قرار می‌گیرند unsigned char باشند.

کدداخل While(1):

```
while (1)
      {
       if(select!=0)
           {
           if(PINB.0==0)    //IF4
               {
               delay_ms(100);
               if(select==1)hour++;
               if(select==2)year_new++;
               if (hour>23)hour=0;
               if (year_new>99)year_new=90;
               }
           if(PINB.1==0)   //IF3
               {
               delay_ms(100);
               if(select==1)min++;
               if(select==2)month_new++;
               if (min>59)min=0;
               if (month_new>12)month_new=1;
               }
           if(PINB.2==0) // IF2
               {
               delay_ms(100);
               if(select==1)sec++;
               if(select==2)day_new++;
               if (sec>59)sec=0;
               if (day_new>31)day_new=0;
               }
           lcd_gotoxy(0,0);
           sprintf(st,"Time:%d:%d:%02d\nDate:%d/%d/%d\n",hour,min,sec,year_new,month_new,day_new);
           lcd_puts(st);
           }
```

```
        if(PINB.3==0) // IF 1
            {
            delay_ms(100);
            lcd_clear();
            select++;
            if(select==1)
                {
                hour=h;
                min=m;
                sec=s;
                lcd_gotoxy(14,0);
                lcd_puts("<");
                }

            if(select==2)
                {
                year_new=year;
                month_new=month;
                day_new=day;
                lcd_gotoxy(14,1);
                lcd_puts("<");
                }
            if(select>2)
                {
                rtc_set_time(hour,min,sec);
                rtc_set_date(day_new,month_new,year_new);
                select=0;
                }
            }
    if(select==0)
        {
        rtc_get_time(&h,&m,&s);
        rtc_get_date(&day,&month,&year);
        lcd_gotoxy(0,0);
        sprintf(st,"Time:%d:%d:%02d\nDate:%d/%d/%d\n",h,m,s,year,month,day);
        lcd_puts(st);
        }

    }
}
```

علت استفاده از delay_ms(100) در این کد به این دلیل است که زمانی‌که دست کاربر روی کلید است برنامه چندین بار حلقه‌ی while را تکرار نکند و به اندازه ۱۰۰میلی ثانیه منتظر بماند.
متغیر SELECT نیز برای شمارش تعداد دفعات زدن کلید SELECT می‌باشد، بنابراین زمانیکه مدار روشن می‌شود، مقدار متغیرSELECT صفر می‌باشد و در نتیجه کد وارد آخرین دستور شرطیif می‌شود و به کمک دستورهای
rtc_get_time(&h,&m,&s) و rtc_get_date(&day,&month,&year)
آی‌سی DS1307 ساعت و دقیقه و ثانیه را از روی سیستم کامپیوتر کاربر خوانده(البته در شبیه‌سازی اینگونه است و در واقعیت از خود آی‌سی این زمان محاسبه می‌شود) و در متغیرهای h وm وs ذخیره می‌کند. همچنین روز و ماه و سال را نیز از روی سیستم کاربر خوانده و در متغیرهای day وmonth وyear ذخیره می‌کند سپس این زمان و تاریخ را روی lcd نمایش

می‌دهد. کاراکترهای کنترلی \n و %d02 که در دستور sprint مشاهده می‌کنیم به ترتیب بخاطر رفتن به خط بعد و تا دو رقم نشان دادن متغیر صحیح می‌باشد(در اینجا به این علت می‌باشد که صدم ثانیه نشان داده نشود).

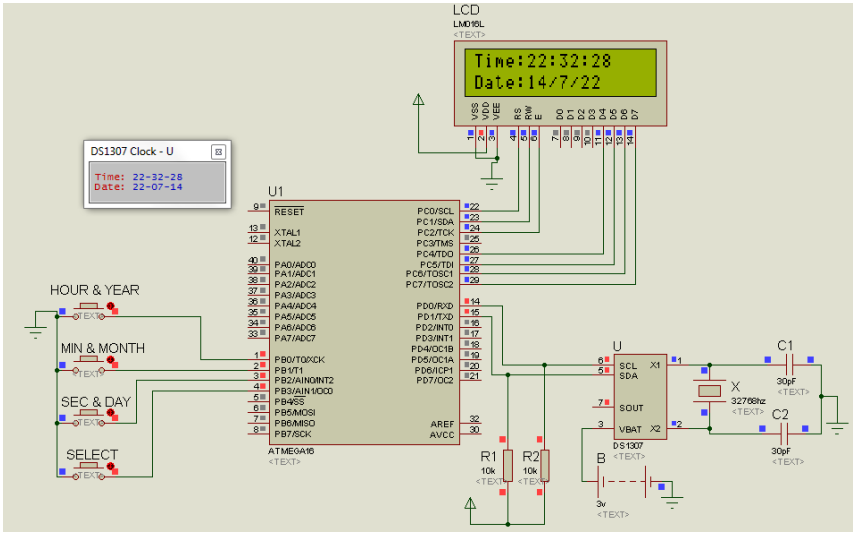
حال با زدن کلید SELECT برنامه وارد تنظیمات ساعت می‌شود (ابتدا تنظیم زمان بعد از دوباره فشار دادن کلید وارد تنظیم تاریخ و با فشار مجدد کلید موجب ذخیره تغییرات شود):

با زدن کلید SELECT کد وارد دستور شرطی if1 می‌شود، که دراین دستور شرطی اگر کلید SELECT یکبار زده شده باشد(چون متغیر SELECT ابتدا صفر است وقتی وارد این if می‌شود یک می‌شود) به تغییر زمان می‌پردازد و در این لحظه اگر کلید SEC را بزنیم ثانیه را تغییر می‌دهد (وارد دستور شرطی if2 می‌شود) و اگر کلید MIN را بزنیم دقیقه را تغییر می‌دهد (وارد دستور شرطی if3 می‌شود) و اگر کلید HOUR را بزنیم ساعت را تغییر می‌دهد (وارد دستور شرطی if4 می‌شود).

اگر بخواهیم تاریخ را تنظیم کنیم کلید SELECT را یکبار دیگر می‌زنیم که دوباره وارد دستور شرطی if1 می‌شود و متغیر SELECT را برابر دو می‌کند و ادامه‌ی تنظیمات در این حالت نیز مشخص است(که همانند حالت تنظیم زمان می‌باشد برای مثال اگر کلید DAY رابزنیم روز را تغییر می‌دهد و...).

پس از تغییر زمان و تاریخ کلید SELECT را مجدداً زده تا کد وارد دستور شرطی if1 شود،در این حالت مقدار متغیر SELECT سه می‌شود و چون این مقدار بیشتر از دو است این زمان و تاریخ در آی‌سی DS1307 به عنوان زمان و تاریخ جدید وارد می‌شود.

و در انتها نتیجه‌ی شبیه‌سازی کد نوشته شده را در شکل زیر مشاهده می‌کنیم:



شکل ۱۶-۸-۱۰ : نتیجه‌ی شبیه‌سازی پروژه

پروژه‌ی۹: راه اندازی سروو موتور(SERVO MOTOR)

پیش‌نیاز: تایمر

سروو موتورها یکی از انواع موتورها می‌باشند که قابلیت چرخش دقیق در زاویای بین ۰ تا ۳۶۰ درجه را دارا می‌باشند البته میزان چرخش در سروو موتورها متفاوت است و بستگی به مُدل آن‌ها دارد. سروو موتورها اکثراً در ربات‌ها و جاهایی که نیاز به چرخش دقیق و کنترل زاویه وجود دارد استفاده می‌شوند، برای مثال می‌توان از آنها برای چرخش دوربین و کنترل دقیق زاویه‌ی دوربین استفاده کرد. در شکل زیر تصویر یک سروو موتور را مشاهده می‌کنید:

شکل ۱۶-۹-۱: سروو موتور

سروو موتوری که ما قصد بررسی آن را در این پروژه داریم دارای سه پایه می‌باشد که دو پایه‌ی آن VCC و GND و پایه‌ی دیگر آن CTRL است که پایه‌ی کنترلی این نوع موتور می‌باشد. برای چرخش این موتورها نیاز است یک پالس بر روی پایه CTRL اعمال شود.

این پالس باید دارای ویژگی‌های زیر باشد:

فرکانس خاص: که باید دارای فرکانسی بین ۵۰ تا ۱۰۰ هرتز باشد(یعنی دوره‌ی زمانی ۲۰ تا ۱۰ میلی‌ثانیه).

عرض پالس مثبت خاص: اگر فرکانس را ۵۰ هرتز انتخاب کنیم یعنی دوره‌ی زمانی ۲۰ میلی‌ثانیه را برای موج انتخاب کنیم بایستی عرض پالس مثبت (میزان یک بودن موج در یک دوره) بین یک تا ۲ میلی‌ثانیه باشد به طوریکه اگراین عرض پالس مثبت (میزان یک بودن پالس در یک دوره) یک میلی‌ثانیه باشد، موتور کاملاً به سمت چپ و هنگامیکه ۲ میلی‌ثانیه باشد موتور کاملاً به سمت راست می‌چرخد و در حالتیکه ۱.۵میلی‌ثانیه باشد نیز موتور در حالت وسط قفل می‌شود(البته همه‌ی اینها زمانی است که فرکانس را ۵۰ هرتز انتخاب کنیم). اما ممکن است در عمل چنین نباشد یا حتی نوع سروو متفاوت باشد(سروها دارای انواع مختلفی هستند) که در چنین حالت‌هایی می‌توان از دیتاشیت یا آزمون و خطا برای بدست‌آوردن عرض پالس مناسب استفاده کرد.

در این پروژه قصد داریم یک سروو موتور را از زاویه منفی ۹۰ درجه تا مثبت ۹۰ درجه با فواصل ۴۵ درجه با یک تاخیر ۱ ثانیه‌ای بچرخانیم.

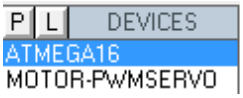
قطعات مورد نیاز برای طراحی

۱) ATmega16

۲) یک عدد سروو موتور: سروو موتورها از سه بخش موتورDC ، جعبه دنده و مدار کنترل زاویه موتور تشکیل می‌شوند که تمامی این موارد در یک Package قرار دارند. در مورد نحوه‌ی کار یک سروو موتور باید گفت عموماً این موتورها دارای سه سیم می‌باشند که ۲ تا از آنها تغذیه موتور و سیم دیگر فرمان را دریافت می‌کند.(البته نوع ۵ سیمه هم وجود دارد که فرق آن با نوع سه سیمه این است که تغذیه مدار کنترل باید از بیرون متصل گردد).

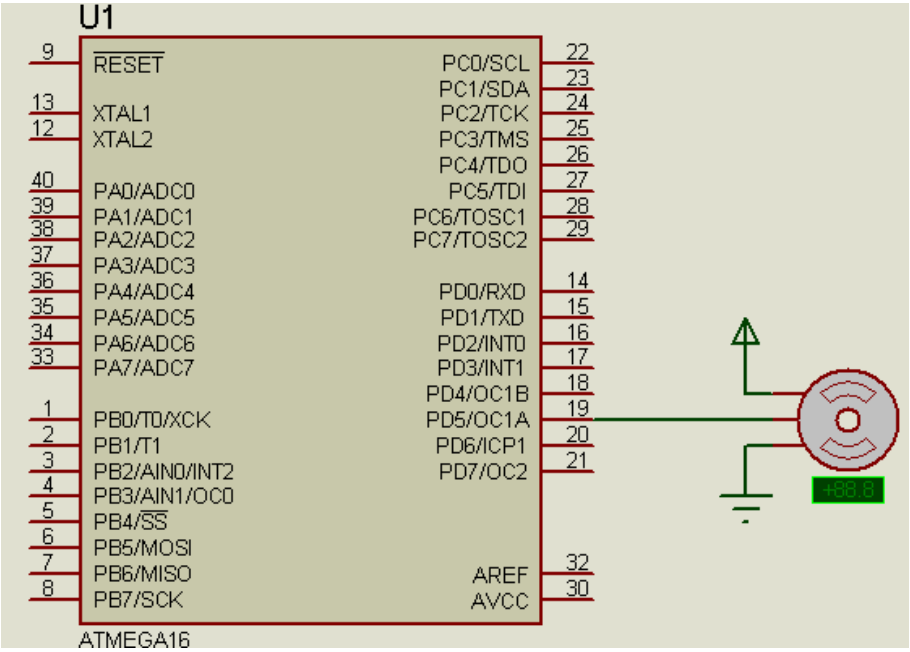
طراحی در پروتئوس

ابتدا قطعات مورد نیاز در پروتئوس را مطابق شکل زیر انتخاب می‌کنیم:



شکل ۱۶-۹-۲ : قطعات استفاده شده در شبیه‌سازی

و برای شبیه‌سازی مدار خود را مطابق شکل زیر در پروتئوس می‌بندیم:



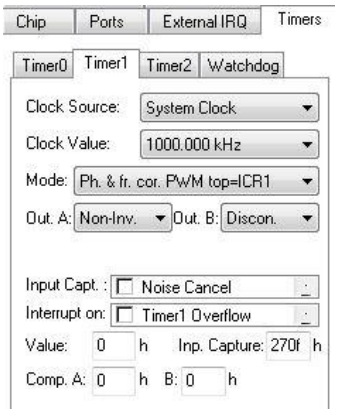
شکل ۱۶-۹-۳ : مدار بسته شده در پروتئوس

سروو موتور را همانند شکل به میکروکنترلر متصل می‌کنیم و پایه‌ی وسط آن‌را که پایه‌ی کنترل آن می‌باشد به OCR1A برای تولید پالس وصل می‌کنیم. ضمناً پایه‌ی بالایی VCC و پایه‌ی پایین آن نیز GND می‌باشد.

نوشتن کد در Codevision

پس از انتخاب ATmega16 و تنظیم فرکانس کاری میکروکنترلر بر روی ۸ مگاهرتز به سراغ تنظیمات تایمر می‌رویم(در این پروژه ما از تایمر شماره یک استفاده کردیم).

برای کنترل سروو موتور قصد داریم موجی را بسازیم که دارای دوره‌ی ۲۰میلی‌ثانیه (۵۰هرتز) باشد پس می‌توان از مُدهای NORMAL،CTC یا PWM برای درست کردن این موج استفاده کرد ولی چون ما می‌خواهیم میزان زمان یک بودن آن موج را (که قرار است بین ۱ تا ۲ میلی‌ثانیه باشد) تغییر بدهیم پس باید از مُد PWM استفاده کنیم که این کار را می‌توان با تغییر مقدار OCR انجام داد.



شکل ۱۶-۹-۴ : تنظیمات تایمر۱

برای ایجاد دوره‌ی ۲۰میلی‌ثانیه می‌توانیم Clock Value را برابر ۱مگاهرتز قرار دهیم که در این حالت هر کلاک تایمر ۱ میکروثانیه طول می‌کشد. در این پروژه از مُد Ph. & fr. Cor. PWM top=ICR1 استفاده می‌کنیم. همانطور که می‌دانید در این مُد مقدار تایمر به صورت مثلثی تا مقدار حداکثر (ICR1) افزایش می‌یابد و سپس از حداکثر با همان شیب به صفر بازمی‌گردد، پس اگر بخواهیم دوره‌ی پالس برابر ۲۰میلی‌ثانیه باشد باید از ۰ تا حداکثر را در ۱۰میلی‌ثانیه طی کند(از حداکثر تا صفر را نیز در ۱۰میلی‌ثانیه دوم دوره طی می‌کند)، بنابراین مقدار حداکثر(ICR1) بدست می‌آید: 10ms/1µs= 10000

یعنی بایستی ۱۰۰۰۰ کلاک ۱میکروثانیه‌ای از ۰ تا ۹۹۹۹ بخورد تا ICR1 برابر ۹۹۹۹ یا 0X270F شود پس باید در برنامه مقدار آن را برابر 0X270F قرار دهیم.

می‌خواهیم موج خروجی را از پایه‌ی OCR1A دریافت کنیم پس Out A را در حالت Non-Inv قرار می‌دهیم:

چون در برنامه از تاخیر یا همان delay استفاده می‌کنیم باید کتابخانه‌ی delay.h رادر ابتدای برنامه تعریف کنیم:

```
#include <mega16.h>
#include <delay.h>
```

کد درون while(1):

همانطور که قبلا گفته شد اگر در این سروو موتور از موجی با دوره‌ی ۲۰میلی‌ثانیه استفاده کنیم و در صورتی که از مُد FAST PWM برای ایجاد موج استفاده کرده باشیم برای زاویه‌ی ۹۰ درجه بایستی میزان یک بودن موج ۲میلی‌ثانیه باشد(چون این مُد شامل یک شیب است)، یعنی اگر قرار باشد ماکزیم PWM،9999(ICR1) باشد با یک تناسب ساده مقدار PWM برای چنین موج حدود ۱۰۰۰ خواهد بود:

$$\frac{9999}{20ms} = \frac{PWM}{2ms}$$

اما اگر از مُد اصلاح فاز یا همان ph correct pwm استفاده کنیم (که در این پروژه از مُدی که شامل این اصلاح می‌باشد استفاده کردیم) در این صورت چون این مُد شامل دو شیب می‌باشد که هر شیب ۱۰ میلی‌ثانیه بوده(در این حالت که ما دوره را ۲۰میلی‌ثانیه درنظرگرفتیم) و اینکه مقدار pwm برای تنها یک شیب است بنابراین برای زاویه‌ی ۹۰ درجه میزان ۱ بودن موج بایستی ۱ میلی‌ثانیه باشد(۱میلی‌ثانیه برای شیب افزایش PWM و ۱میلی‌ثانیه دیگر نیز برای شیب کاهش PWM). مقدار PWM نیز همانند بالا با یک تناسب ساده بدست می‌آید:

$$\frac{9999}{10ms} = \frac{PWM}{1ms}$$

که برابر همان ۱۰۰۰ می‌باشد.

پس اگر از مُد ph correct pwm استفاده کنیم زمانی موتور کاملاً به سمت چپ می‌رود که عرض پالس مثبت(میزان یک بودن پالس در یک دوره) ۰.۵ میلی‌ثانیه باشد و هنگامی موتور کاملاً به سمت راست می‌چرخد که این عرض ۱میلی‌ثانیه باشد و در حالتی‌که این عرض ۰.۷۵ میلی‌ثانیه باشد موتور در حالت وسط قفل می‌شود. با توجه به این عرض پالس‌ها و تناسب بالا مقدار PWM یا OCR1A برای زوایای ۹۰- و ۰ نیز به ترتیب برابر ۵۰۰ و ۷۵۰ خواهد بود.

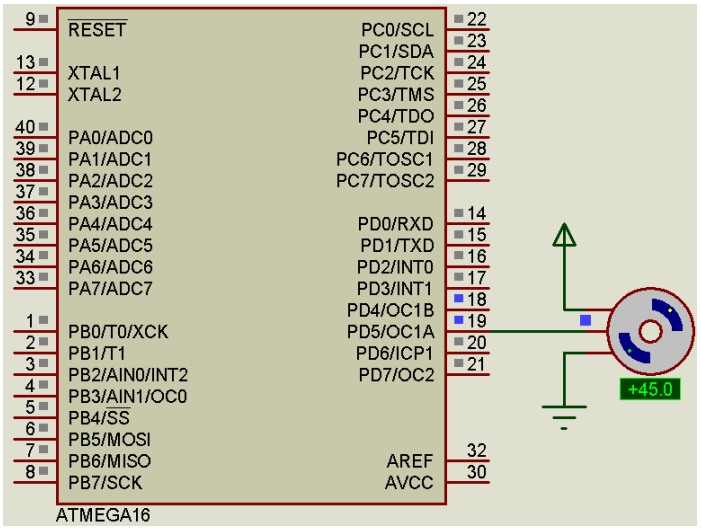
اما اگر از مُد fast pwm استفاده کنیم عرض پالس مثبت‌ها دو برابر حالت phase correct pwm خواهد شد.

کد نوشته شده:

```
while (1)
      {
      OCR1A=500;
      delay_ms(1000);
      OCR1A=625;
      delay_ms(1000);
      OCR1A=750;
      delay_ms(1000);
      OCR1A=875;
      delay_ms(1000);
      OCR1A=1000;
      delay_ms(1000);
      }
}
```

باتوجه به اینکه اختلاف زاویه‌ی ۹۰درجه معادل اختلاف OCR ۲۵۰ شده است پس اختلاف زاویه‌ی ۴۵ درجه نیز معادل اختلاف OCR ۱۲۵ خواهد بود و بدین ترتیب OCR1A برای زوایای ۴۵- و ۴۵ درجه برابر ۶۲۵ و ۸۷۵ خواهند شد.

و در انتها نتیجه‌ی شبیه‌سازی کد نوشته شده را در شکل زیر مشاهده می‌کنیم:



شکل ۱۶-۹-۵ : نتیجه‌ی شبیه‌سازی پروژه

پروژه‌ی ۱۰: راه اندازی موتور پله‌ای (STEPPER MOTOR)

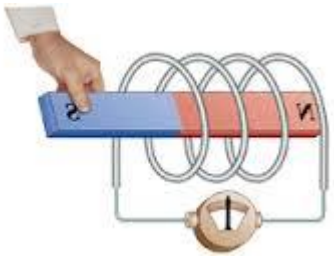
پیش‌نیاز: PIN & PORT

در این پروژه قصد داریم یک موتور پله‌ای دوقطبی را به کمک دو کلید یکی برای حرکت ساعتگرد و دیگری برای حرکت پادساعتگرد به چرخش درآوریم.

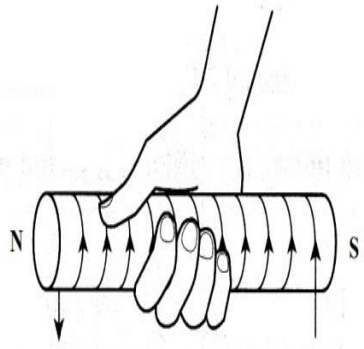
مفاهیم اولیه‌ی طرز کار موتور پله‌ای

ماشین الکتریکی ابزاری است برای تبدیل انرژی الکتریکی به مکانیکی و یا بالعکس. موتورهای الکتریکی، نوعی از ماشین‌های الکتریکی هستند که انرژی الکتریکی را به مکانیکی تبدیل می‌کنند. موتورها این تبادل انرژی را به کمک میدان مغناطیسی انجام می‌دهند بدین شکل که انرژی الکتریکی را به انرژی مغناطیسی تبدیل کرده و این انرژی در میدان مغناطیسی ذخیره می‌شود و سپس این انرژی را به کمک قوانین فیزیکی به انرژی مکانیکی تبدیل می‌کنند.

برای اینکه بفهمیم چگونه می‌توان انرژی الکتریکی را به انرژی مغناطیسی تبدیل کرد کافیست طرز ساخت یک آهنربای الکتریکی را به یاد بیاوریم بدین ترتیب که دور یک میخ(فلز) سیمی را می‌پیچیدیم و دو سر آن سیم را به باطری وصل می‌کردیم و در نهایت میخ خاصیت آهنربایی پیدا می‌کرد، این بدین معنی است که با پیچیدن یک سیم حاوی جریان به دور یک هسته(فلز) می‌توان یک میدان مغناطیسی ایجاد کرد که جهت این میدان به کمک قانون دست راست تعیین می‌شود:

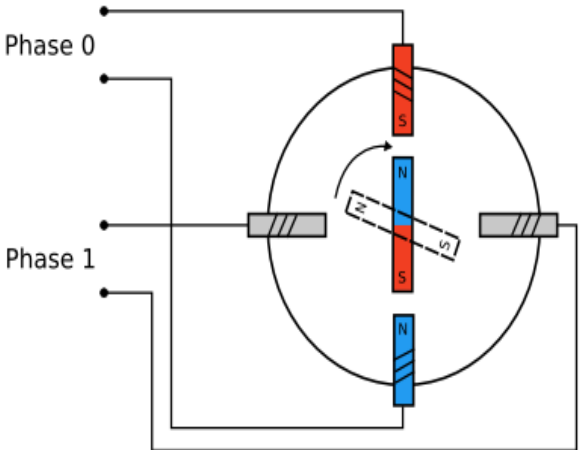


شکل ۱۶-۱۰-۱-ب : ایجاد جریان القایی



شکل ۱۶-۱۰-۱-الف : قانون دست راست برای تعیین جهت میدان مغناطیسی

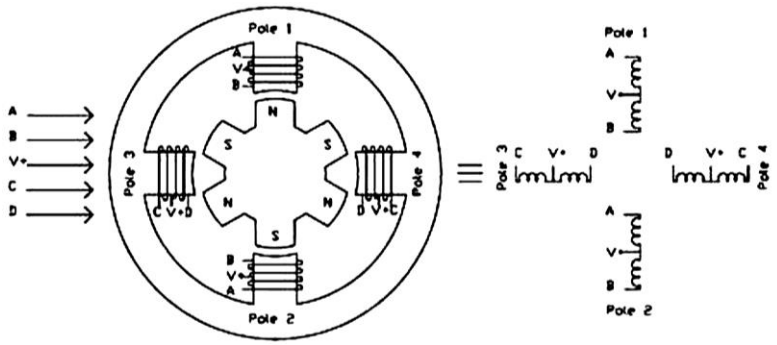
یکی از راه‌ها برای تبدیل انرژی مغناطیسی به الکتریکی نیز این است که یک روتور آهنربایی را در داخل این میدان مغناطیسی ایجاد شده به کمک سیم‌پیچ قرار دهیم. برای مثال در شکل زیر ولتاژ فاز صفر(phase 0) دو سر سیم‌پیچ پیچیده شده دور هسته‌های بالا و پایین و ولتاژ فاز یک(phase 1) دو سر سیم‌پیچ پیچیده شده دور هسته‌های چپ و راست قرار گرفته اند.



شکل ۱۶-۱۰-۲ : ایجاد میدان مغناطیسی دوار

همچنین یک روتور آهنربایی نیز در مرکز این موتور قرار گرفته است. همانند شکل بالا اگر phase 0 را به ۰و۱(منبع) و phase 1 را به زمین(۰ و۰) بزنیم بخاطر سیم‌پیچ بالا و پایین یک میدان مغناطیسی در همان جهتی که در شکل نشان داده شده ایجاد می‌شود. در این حالت اگر روتور آهنربایی نیز در همان وضعیت نشان داده شده در شکل، یعنی در حالت افقی باشد به خاطر نیروی جاذبه بین قطب‌های همنام روتور در جهت عقربه‌های ساعت می‌چرخد تا اینکه به حالت عمودی برسد که در آن حالت متوقف می‌شود(۹۰ درجه در جهت عقربه ساعت می‌چرخد). برای اینکه بخواهیم ۹۰ درجه دیگر در جهت عقربه‌های ساعت بچرخد باید phase 0 را به زمین و phase 1 را به ولتاژ(صفر و یک) بزنیم که باتوجه به حالت قبل در این حالت روتور آهنربایی از حالت عمودی به حالت افقی تغییر وضعیت می‌دهد(۹۰ درجه در جهت عقربه ساعت می‌چرخد). در این صورت زاویه‌ی گام ۹۰ درجه می‌باشد.

حال اگر این سیستم را شبیه به شکل زیر بسازیم که روتور دارای ۶ قطب است. در این صورت زاویه‌ی گام ۳۰ درجه خواهد بود. بنابراین با تغییر شکل روتور آهنربایی و سیم پیچ‌ها می‌توان گام زاویه را تغییر داد:



شکل ۱۶-۱۰-۳ : استپر با روتور ۶ قطب

موتور پله‌ای(استپر موتور)

در موتور پله‌ای یا استپر موتور الگوریتم چرخیدن موتور به صورت توضیحات فوق می‌باشد. اگر شفت این موتورها را با دست بچرخانید احساس می‌کنید که موتور به صورت گسسته و پله پله حرکت خواهد کرد و حرکت پیوسته و یکنواختی ندارد. زمانیکه سر سیم‌هایی را که از این موتور خارج شده است بگیرید و شفت این موتور را بچرخانید ولتاژ القا شده را احساس خواهید کرد. در موتورهای بزرگ پله‌ای این ولتاژ دارای مقدار بزرگی می‌باشد که برای محافظت مدارهای راه‌انداز در برابر این ولتاژ القایی از دیود استفاده می‌کنند.

موتورهای استپر دارای دقت حرکت بسیار زیادی می‌باشند که بستگی به زاویه گام(Stepangle) آنها دارد و زمانی مورد استفاده قرار می‌گیرند که ما نیاز به دقت حرکت بالایی داشته باشیم. این موتورها تقریباً مشابه موتورهای DC هستند که حرکت دورانی دارند اما با حرکتی دقیق‌تر و حساب شده‌تر. کاربرد اصلی این موتورها در جاهایی است که به کنترل دور دقیق نیاز است. یکی از مهمترین ویژگی‌های این موتورها این است که به صورت لحظه‌ای قابلیت توقف دارند، بدون اینکه لختی دورانی باعث چرخیدن چند پله‌ی اضافی گردد. همچنین این موتورها به صورت دیجیتال قابل کنترل هستند(که در ادامه بیشتر آشنا می‌شویم) به همین خاطر می‌توان به راحتی آنها را با گیت‌های منطقی و یا میکروکنترلرها کنترل کرد. اما این نوع موتورها دارای معایبی هم می‌باشند از

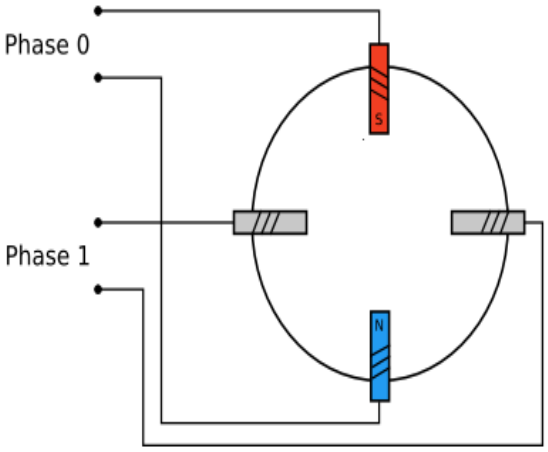
جمله اینکه حداکثر سرعت کاری و قدرت این موتورها کم است و دارای قیمت بالایی هم می‌باشند. به خاطر مزایای گفته شده از آنها در کنترل ربات‌های حساس، پرینترها، وسایل صنعتی و پزشکی، دیسک درایوها و ... استفاده می‌شود.

نکته: گام زاویه‌ای(Stepangle) هر موتور پله‌ای در برگه‌ی مشخصات فنی مربوط(datasheet) به آن موتور موجود می‌باشد.

همانطور که گفته شد استپرها دارای انواع مختلفی از لحاظ زاویه‌ی پله(زاویه گام) می‌باشند که هر چه این زاویه‌ی پله کمتر باشد دقت این نوع موتور بیشتر می‌باشد. برای مثال اگر زاویه پله ۱.۸ درجه باشد در این صورت موتور باید ۲۰۰پله و اگر زاویه پله ۱۵ درجه باشد موتور باید ۲۴ پله (24*15 =360) بچرخد تا یک دور کامل بزند.

این موتورها عموماً دارای چهار قطب می‌باشند که سیم‌پیچ‌ها برروی این چهار قطب قرار می‌گیرند و با ارسال بیت‌های ۰ و ۱ به این سیم‌پیچ‌ها می‌توان میدان مغناطیسی ایجاد کرد.

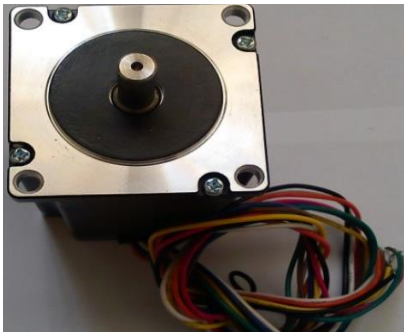
در شکل زیر صفر و یک را به سیم‌پیچ بالا و پایین موتور متصل کرده‌ایم. میدان ایجاد شده به وسیله‌ی این سیم‌پیچ‌ها باعث به حرکت در آمدن روتور می‌شود البته باید این سیم‌پیچ‌ها را به صورت متوالی صفر و یک کرد در غیر این صورت موتور منظم نخواهد چرخید. هر بار که این سیم‌پیچ‌ها دارای ولتاژ شوند روتور با توجه به زاویه‌ی پله‌ی آن یک پله می‌چرخد. اینکه روتور ساعتگرد می‌چرخد یا پادساعتگرد بستگی به ترتیب ولتاژدهی سیم‌پیچ‌ها دارد.



شکل ۱۶-۱۰-۴ : شمای کلی تحریک موتور پله‌ای

توجه: می‌توان برای افزایش دقت استپری که در اختیار داریم فاصله بین دو پله را به نصف و یا به چند قسمت تقسیم کرد(این کار را به کمک آی‌سی‌های 3955 یا LMD18245 انجام می‌دهند).

همانند شکل از داخل استپر موتور تعدادی سیم رنگی بیرون آمده است که هر کدام از این سیم‌ها به یک سر از سیم‌پیچ‌های گفته شده متصل است و یک سیم نیز بین تمامی سیم‌پیچ‌ها مشترک است. حال با فهم این مطالب به سراغ به حرکت درآوردن یک موتور پله‌ای می‌رویم.



شکل ۱۶-۱۰-۵ : تصویری از یک استپر موتور

استپر موتورها را می‌توان به دوصورت ساعتگرد و پادساعتگرد چرخاند و می‌توان آنها را به دو صورت پله‌ای و نیم پله (برای افزایش دقت) حرکت داد. بیشترین مقدار گشتاور مورد نیاز برای به چرخش درآوردن این موتورها، در مواقع شروع و توقف می‌باشد. در این مواقع نیز برای افزایش گشتاور از الگوریتمی خاص برای عوض کردن ۰و ۱ها استفاده می‌کنند. اما عوض کردن این ۰و ۱ های سیم‌پیچ‌ها برای هر یک از این نوع چرخش‌ها چگونه است؟ پاسخ: الگوریتم و ترتیب تغییر ۰و۱های سیم‌پیچ‌ها را به سه دسته تقسیم می‌کنیم(ما حالتی را بررسی می‌کنیم که موتور پله‌ای دارای چهار عدد سیم‌پیچ باشد) :

۱)**پله‌ای یا تمام پله**(Full stepping): در این حالت ابتدا سیم‌پیچ ۱ را تحریک می‌کنیم و بقیه سیم‌پیچ‌ها را بدون تحریک می‌گذاریم سپس اگر بخواهیم ساعتگرد بچرخد سیم‌پیچ ۲ را تحریک می‌کنیم و بقیه سیم‌پیچ‌ها را بدون تحریک می‌گذاریم ولی اگر بخواهیم پادساعتگرد بچرخد سیم‌پیچ ۴ را تحریک می‌کنیم و بقیه را بدون تحریک می‌گذاریم و در ادامه نیز برای حالت ساعتگرد سیم‌پیچ ۳و سپس ۴، برای حالت پادساعتگرد سیم‌پیچ ۳و سپس۲ را تحریک می‌کنیم تا در این حالت یک دور کامل بزنیم.

شماره	A	B	C	D
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	1	0	0	0
6	0	1	0	0
7	0	0	1	0
8	0	0	0	1

شکل ۱۶-۱۰-۶ : چرخش ساعتگرد به روش Full stepping

۲)**تامین گشتاور بیشتر**: بدین صورت که ابتدا سیم‌پیچ‌های ۱ و ۴ را تحریک و بقیه بدون تحریک سپس سیم‌پیچ‌های ۱و۲ تحریک و بقیه بدون تحریک سپس سیم‌پیچ‌های ۲و۳ تحریک و بقیه بدون تحریک سپس سیم‌پیچ‌های ۳و۴ تحریک و بقیه بدون تحریک قرار می‌دهیم که در این حالت به صورت ساعتگرد می‌چرخد برای چرخیدن پادساعتگرد باید همانند حالت قبل سیم‌پیچ‌ها را با ترتیب برعکس گفته شده تحریک کرد. برای چرخش ساعتگرد با گشتاور بیشتر مانند شکل بالا(شکل ۱۶-۱۰-۷) عمل می‌کنیم.

شماره	A	B	C	D
1	1	0	0	1
2	1	1	0	0
3	0	1	1	0
4	0	0	1	1
5	1	0	0	1
6	1	1	0	0
7	0	1	1	0
8	0	0	1	1

شکل ۱۶-۱۰-۷ : چرخش ساعتگرد با گشتاوری بیشتر

۳)**نیم‌پله‌ای** (Half stepping): زمانی مورد استفاده قرار می‌گیرد که بخواهیم از استپر موتوری که داریم با دقت بیشتری استفاده کنیم که در این صورت زاویه پله نصف می‌شود. ترتیب تحریک سیم‌پیچ‌ها برای چرخش ساعتگرد همانند شکل مقابل می‌باشد.

شماره	A	B	C	D
1	1	0	0	0
2	1	1	0	0
3	0	1	0	0
4	0	1	1	0
5	0	0	1	0
6	0	0	1	1
7	0	0	0	1
8	1	0	0	1
9	1	0	0	0
10	1	1	0	0
11	0	1	0	0
12	0	1	1	0
13	0	0	1	0
14	0	0	1	1
15	0	0	0	1
16	1	0	0	1

نکته: جدول‌های نشان داده شده برای حالت ساعتگرد می‌باشند، برای چرخش به صورت پادساعتگرد می‌توان از همان جداول ولی با ترتیب از پایین به بالا استفاده کرد.

قطعات مورد نیاز برای طراحی

۱)**ATmega16**

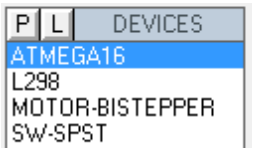
۲)**STEPPER MOTORBIPOLAR**

۳)**L298**: برای تامین توان مورد نیاز جهت چرخش موتور به علت ناکافی بودن توان خروجی میکروکنترلر.

۴)**دو عدد کلید**

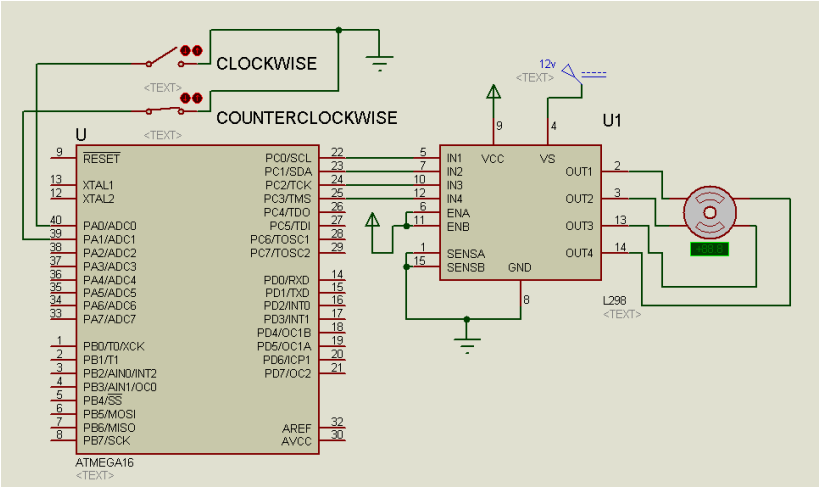
طراحی در پروتئوس

ابتدا قطعات موردنیاز در پرتئوس را همانند شکل مقابل انتخاب می‌کنیم:



شکل ۱۶-۱۰-۹ : قطعات استفاده شده در شبیه‌سازی

و برای شبیه‌سازی مدار خود را مطابق شکل زیر در پروتئوس می‌بندیم:



شکل ۱۶-۱۰-۱۰ : مدار بسته شده در پروتئوس

همانند شکل پایه‌های L298،IN1,IN2,IN3,IN4 را به پورت‌های PORTC.0 PORTC.1 , PORTC.2 , PORTC.3 متصل نموده‌ایم.

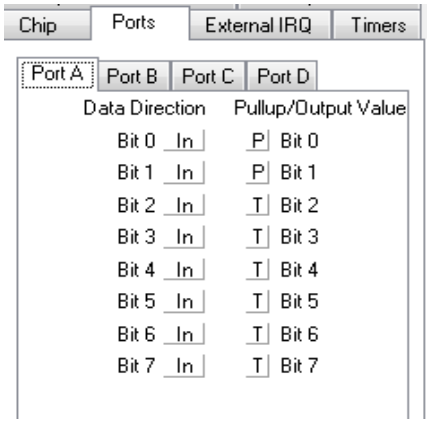
همانطورکه گفته شد برای راه‌اندازی موتور پله‌ای نیاز به PWM نیست، بنابراین از ۰ و ۱ خروجی میکروکنترلر استفاده‌کرده و به علت ناکافی بودن ولتاژ خروجی میکروکنترلر از L298 استفاده می‌کنیم. بدین ترتیب پایه‌های Enable(ENA و ENB) L298 را به ۵ولت متصل‌کرده و۴ خروجی L298 را به ۴سر موتور وصل می‌کنیم.

دو عدد کلید را نیز به پین‌های PINA.0,PINA.1 وصل می‌کنیم.

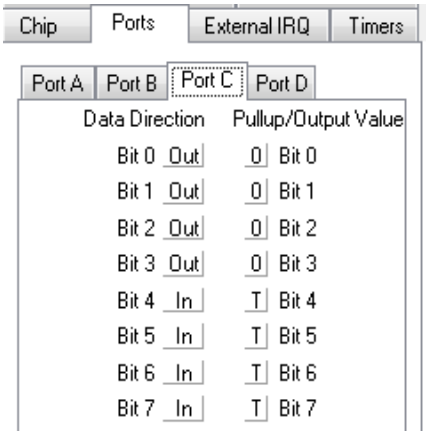
نوشتن کد در Codevision

پس از انتخاب ATmega16 و تنظیم فرکانس کاری میکروکنترلر بر روی ۸مگاهرتز به سراغ تنظیم درگاه‌ها می‌رویم.

تنظیمات درگاه‌های B وD را تغییر نمی دهیم و به سراغ تنظیم درگاه‌های A و C می‌رویم:



شکل ۱۶-۱۰-۱۱ : تنظیمات درگاه A



شکل ۱۶-۱۰-۱۲ : تنظیمات درگاه C

تذکر: برای راه‌اندازی موتور پله‌ای از پالس یا موج استفاده نمی‌کنیم بلکه همانگونه که گفته شد از پورت‌های میکروکنترلر برای ۰و۱ کردن سیم‌پیچ‌های میدان استپر استفاده می‌کنیم که باعث راه افتادن موتور می‌شود.

کد درون while(1):

```
while (1)
      {
      if(PINA.0==0)
        {
        PORTC=1;
        PORTC=2;
        PORTC=4;
        PORTC=8;
        }
      if(PINA.1==0)
        {
        PORTC=8;
        PORTC=4;
        PORTC=2;
        PORTC=1;
        }
      if(PINA.0==1 && PINA.1==1) PORTC=0;
      }
}
```

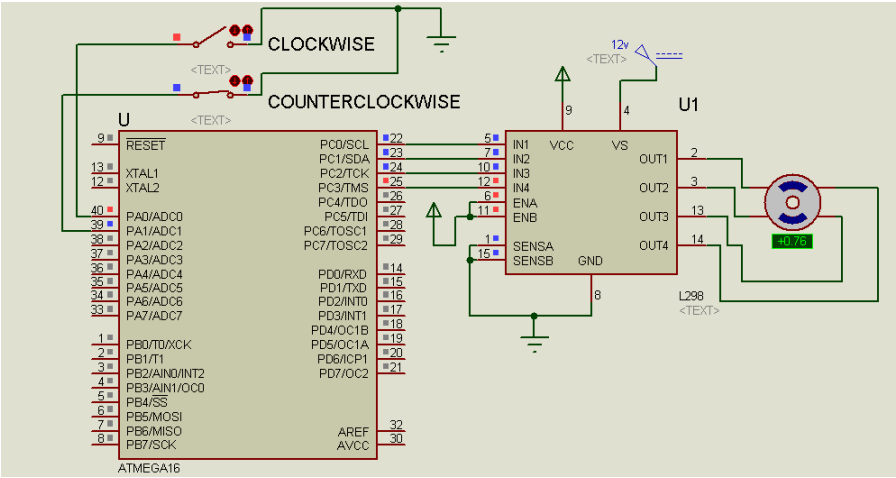
که PORTC=1،PORTC=2،PORTC=4 و PORTC=8 به ترتیب بیانگر اعداد باینری 00000001،00000010،00000100 و 00001000 برای پورت‌های C می‌باشد که برای انجام جدول زیر می‌باشد:

شماره	A	B	C	D
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

جدول ۱۶-۱-۱

در کد فوق شرط (if) اول برای چرخیدن در جهت عقربه‌های ساعت و شرط دوم برای چرخیدن در خلاف جهت عقربه‌های ساعت می‌باشد. شرط سوم هم برای حالت توقف در همان زاویه می‌باشد.

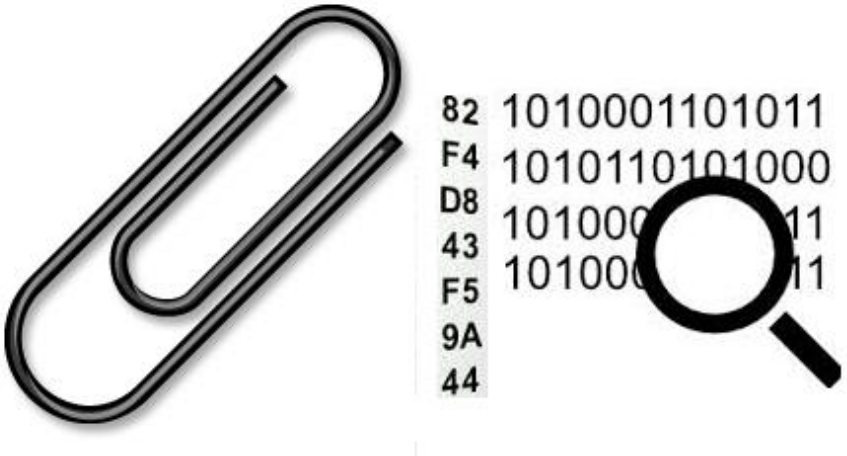
در انتها نتیجه‌ی شبیه‌سازی کد نوشته شده را در شکل زیر مشاهده می‌کنیم:



شکل ۱۶-۱۰-۱۳ : نتیجه‌ی شبیه‌سازی پروژه

پیوست:

آموزش اعداد باینری، دسیمال و هگزادسیمال



پیوست (آموزش اعداد باینری, دسیمال و هگزادسیمال)

اعدادی که بشر از آن برای محاسبات خود بهره می‌جوید اعداد در مبنای ۱۰ هستند چرا که راحت‌ترین روش برای محاسبات عددی می‌باشند و الگوی گرفته‌شده هم از ۱۰ انگشت دست می‌باشد ولی کامپیوترها برای محاسبات خود به اعدادی نیاز دارند که اعمال حسابی را با دقت بالا و با کمترین حجم اشغال شده برای هر عدد انجام دهند برای همین در کامپیوتر ها از مبنای دو یا اعداد باینری استفاده می‌شود که شامل تنها دو رقم ۰ و ۱ می‌باشند. در این نحوه‌ی نمایش به راحتی می توان یک سطح ولتاژ را به عدد صفر (برای مثل مقدار ولتاژ صفر ولت) و سطح ولتاژ دیگری را به یک (برای مثال ۵ ولت) نسبت داد و بدین ترتیب کامپیوترها می‌توانند با محیط پیرامون خود از طریق همین سطح ولتاژها ارتباط برقرار کنند. نوع دیگری از نحوه‌ی نمایش که می‌توان به واسطه‌ی آن اعداد را نمایش داد و در کدنویسی میکروکنترلرها هم بسیار پرکاربرد می‌باشد اعداد مبنای ۱۶ یا هگزادسیمال هستند (به اختصار با HEX نمایش می‌دهند)که بیشتر کار با رجیسترها در کدنویسی با این اعداد انجام می‌گیرند. در جدول زیر اعداد در مبنای ۲ و ۱۰ و ۱۶ نشان داده شده‌اند:

مبنای ۱۶ (hexadecimal)	مبنای ۲ (binary)	مبنای ۱۰ (decimal)
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

نحوه‌ی تبدیل مبناهای دیگر به مبنای ۱۰: برای تبدیل سایر مبناها به مبنای ۱۰ مطابق زیر عمل می‌کنیم:

$$a_n r^n + a_{n-1} r^{n-1} + \ldots a_2 r^2 + a_1 r + a_0 + a_{-1} r^{-1} + a_{-2} r^{-2} + a_{-m} r^{-m}$$

که ضرایب a_j بین ۰ تا r-۱ هستند .

مثال: تبدیل مبناهای زیر را به مبنای ده انجام دهید (0x نماد اعداد هگزادسیمال و 0b نماد اعداد باینری هست):

0xFF=255

0xBC=188

0xA1= 161

0x1F= 31

0b00010000=16

0b10100010=162

0b00111001=57

0xCA=202

0b00110011=51

به عنوان نمونه از مثال بالا نحوه‌ی تبدیل اعداد 0xBC و 0b10100010 به مبنای ده را در زیر مشاهده می‌کنید:

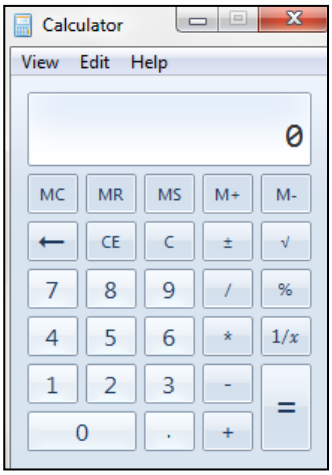
$$0xBC = 12*16^0 + 11*16^1 = 188$$

(11 12)

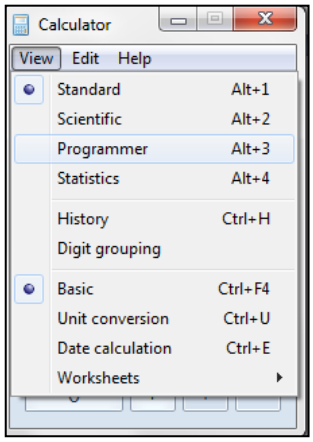
$$0b10100010 = 0*2^0+1*2^1+0*2^2+0*2^3+0*2^4+1*2^5+0*2^6+1*2^7$$

محاسبه و تبدیل اعداد باینری , هگزادسیمال و دسیمال با استفاده از ماشین حساب کـامپیوتر: راه کاربردی‌تر برای محاسبه و تبدیل این اعداد استفاده از ماشین حساب کامپیوتر اسـت کـه نحـوه‌ی استفاده از آن را در ادامه خواهیم آموخت.

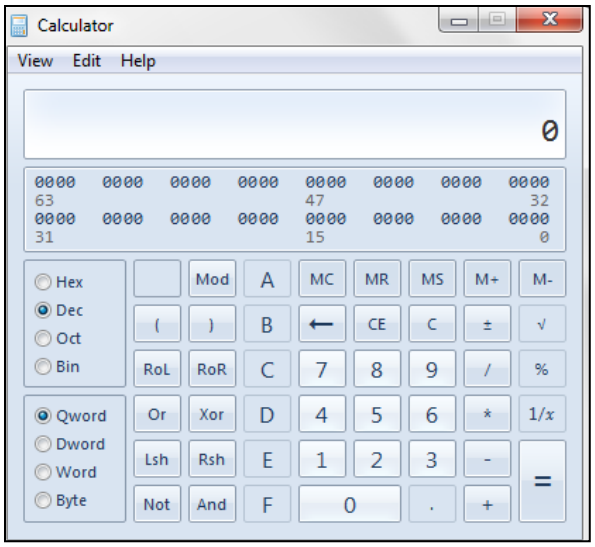
در کامپیوترها ماشین‌حسابی تحت عنوان calculator وجود دارد, که تصویر آن را در زیر مشاهده می‌کنید:



برای اینکه بتوان از این ماشین حساب در حالتی استفاده کرد که توانایی تبـدیل مبناهـا را داشـته باشد, آن را در مُد programmer قرار می‌دهیم که نحوه‌ی این مُد کاری را در شکل زیر مشاهده می‌کنید: (کلیک بر روی گزینه‌ی View و انتخاب گزینه‌ی programmer)

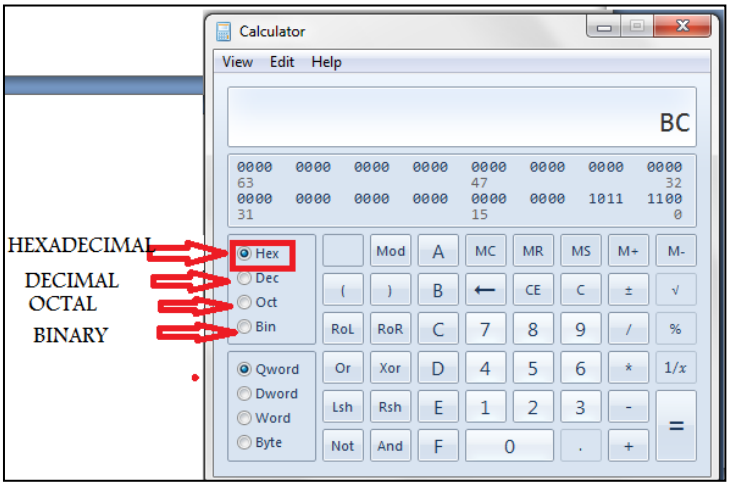


حال صفحه‌ی ماشین حساب به شکل زیر تبدیل می‌شود:

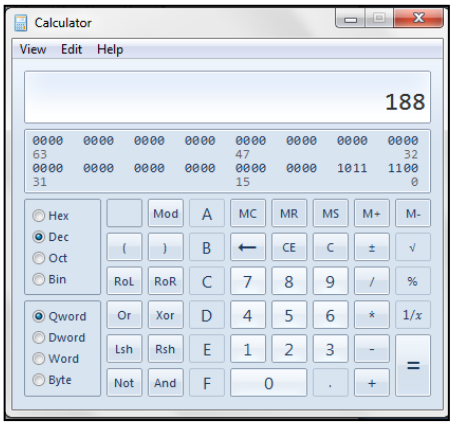


مثال ۱: عدد هگزادسیمال 0xBC را با استفاده ماشین حساب به مبنای ده تبدیل کنید:

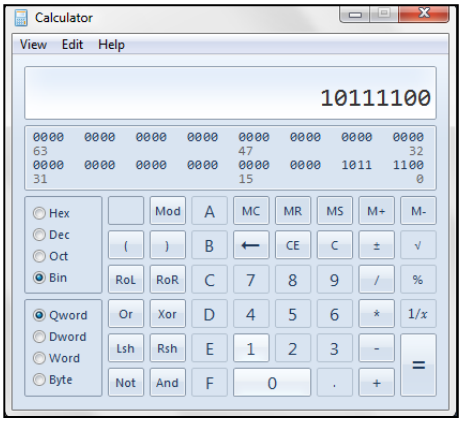
مطابق شکل زیر ابتدا مُد اعداد هگزادسیمال را انتخاب می کنیم و عدد خود را وارد می‌کنیم:



حال مبنا را بر روی اعداد دسیمال قرار می‌دهیم تا عدد وارد شده به مبنای ۱۰ تبدیل گردد که نتیجه مطابق شکل زیر برابر ۱۸۸ می‌شود:

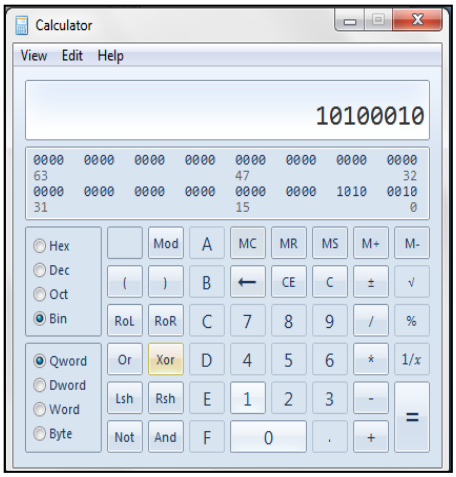


اگر بخواهیم عدد وارد شده را در مبنای ۲ مشاهده کنیم، عدد را در مُد مبنای ۲ (bin) قرار می‌دهیم:

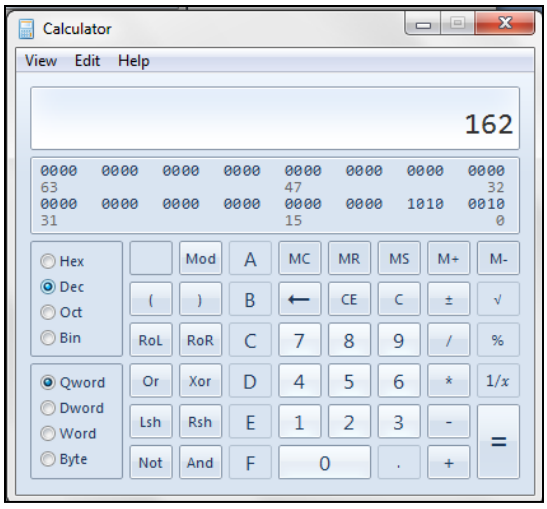


مثال۲: عدد 0b10100010 را در مبنای ۱۰ و ۱۶ محاسبه کنید:

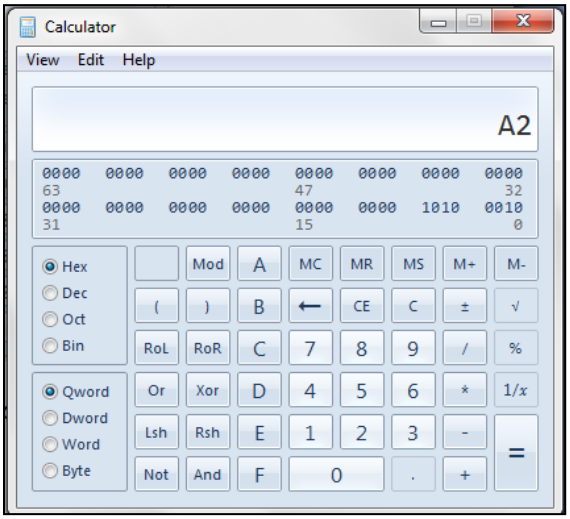
ابتدا عدد را در مبنای ۲ (bin)وارد می کنیم:



مشاهده عدد در مبنای ۱۰ (Decimal):



و در نهایت مشاهده عدد در مبنای۱۶ (Hex):



منابع:

[1]"AVR Datasheet",Atmel Corporation 1998-2006

[2]"Atmega16 Datasheet",Atmel company,2009

[3]Barnet Cox and o.cull,"Embedded C Programming and the Atmel AVR"

[4] John Morton,"AVR – An Introductory Course",Newnes ,1998

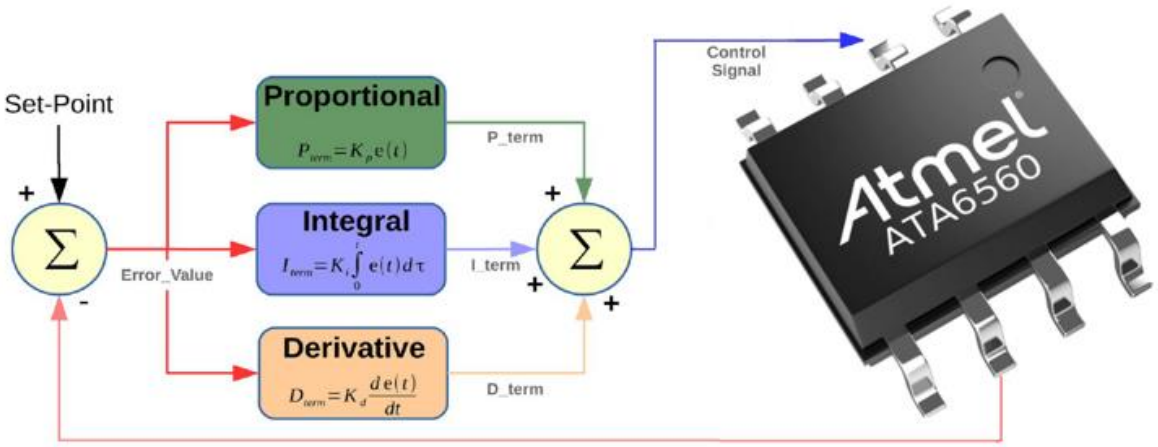
[5]Steven F.Barrett,Daniel J.Pack,"Atmel AVR microcontroller Primer,2008

[6]"mega16/32 AVR Boot.Loader",Progressive Resources LCC,October 22, 2003

[7]D.Gadre,"The AVR Microcontroller,Teta Mcgraw Hill Education Private Limited,2003

پیوست تکمیلی کتاب-بخش ۱

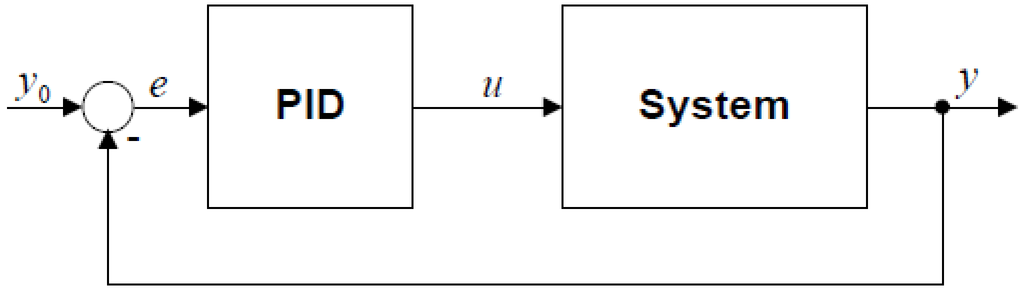
پیاده سازی pid در میکروکنترلر AVR



پیاده سازی pid در میکروکنترلر avr

ابتدا مزایای استفاده از pid را شرح می‌دهیم سپس به پیاده سازی pid در avr می‌پردازیم:

تصویر زیر شکل شماتیک یک کنترلر pid را نشان می‌دهد

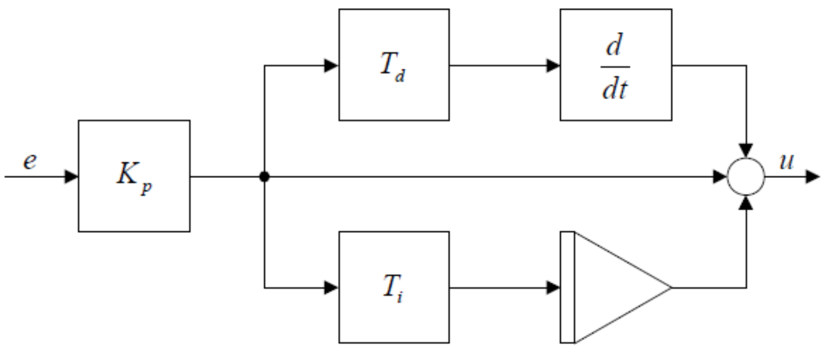


کنترلر pid مقدار اندازه گیری شده (y) را با مقدار مرجع (□□□□□□□□□□□□□□□□□□□□□□□□) که ما آن را تعیین می‌کنیم (y_0) مقایسه می‌کند مقدار اختلاف این دو مقدار را error یا خطا می‌نامیم، این مقدار خطا (e) وارد تابع pid می‌شود و pid مقدار u را محاسبه می‌کند که این مقدار ورودی سیستم اصلی ما می‌شود. (برای مثال اگر سیستم ما سیستم کنترل حرارت محیط باشد خروجی y مقدار دمای اندازه گیری شده‌ی محیط است، مقدار y_0 مرجع دمای دلخواه ما است (مثلا ۲۵ درجه) و اختلاف این دو دما مقدار e را می‌سازد، حال تابع pid با توجه به این اختلاف به سیستم دستور تولید سرما یا گرمای مورد نیاز را می‌دهد تا در نهایت دمای محیط به ۲۵ درجه برسد)

ایده‌ی اصلی pid به این صورت است که حالت فعلی سیستم توسط یک سنسور خوانده شود سپس از مقدار مرجع (مقدار حالت دلخواه) کم شود و مقدار خطا (error) مشخص شود، سپس مقدار خروجی pid از سه طریق تناسب (p)، انتگرال (i) و مشتق (d) محاسبه شود. در تناسب؛ مقدار خروجی متناسب است با مقدار خطا یعنی $p = k_p * error$ ، در قسمت انتگرال خروجی متناسب است با مجموع خطاها $i = k_i * \int error$ که البته در سیستم‌های گسسته انتگرال به مجموع تبدیل می‌شود $i = k_i * \sum error$ و در قسمت مشتق، خروجی متناسب است با مقدار مشتق خطا که در سیستم‌های گسسته مشتق به اختلاف تبدیل می‌شود:

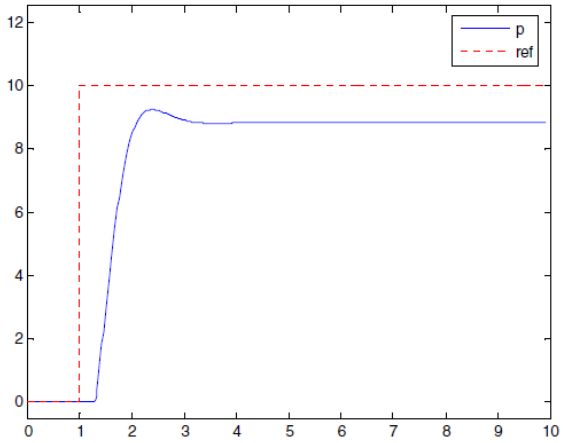
$$d = k_d * (error - error_{pre})$$

شکل زیر بلوک دیاگرام این کنترلر را نمایش می‌دهد:



قسمت تناسبی (p)

این کنترلر نمی‌تواند به تنهایی کنترلر خوبی باشد و معمولا با خطای حالت دائمی‌همراه است؛ تصویر زیر نمودار پاسخ پله‌ی سیستم را با کنترلر p نشان می‌دهد:

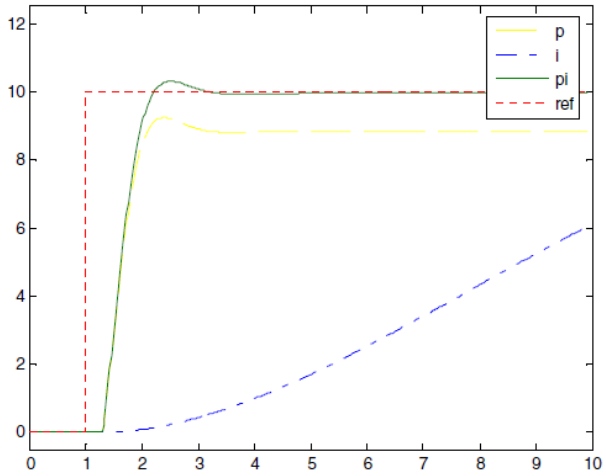


همانطور که در تصویر مشخص شده است نمودار قرمز رنگ پاسخ مرجع است که باید به آن برسیم و نمودار آبی رنگ پاسخ این سیستم با کنترلر p؛ خطای حالت دائمی‌در شکل فوق کاملا مشخص است.

قسمت انتگرال (i)

این کنترلر پاسخ حالت نهایی را بهبود می‌بخشد ولی پاسخ حالت گذرا را کمی بدتر می‌کند

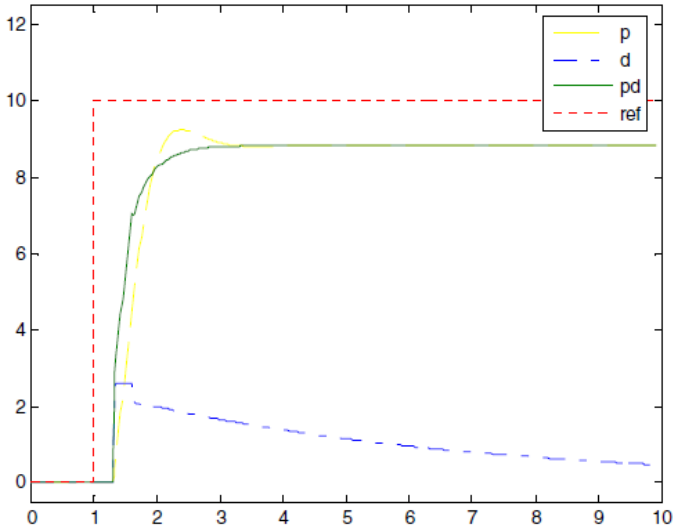
تصویر زیر را نگاه کنید:



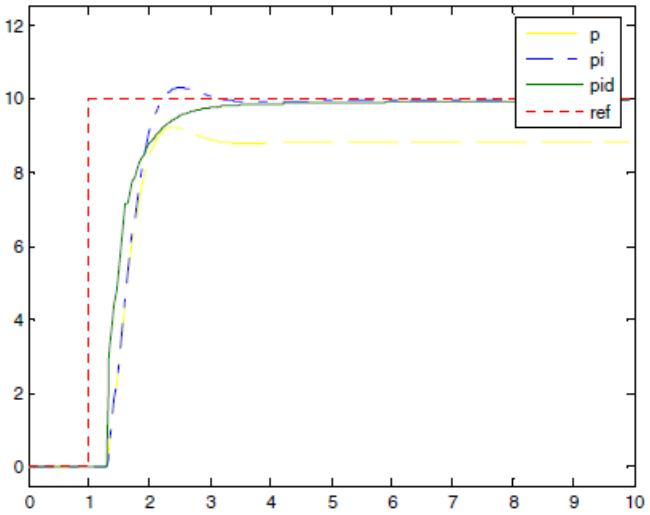
همانطور که مشخص است کنترلر i به تنهایی پاسخ حالت گذرای خوبی ندارد و اگر به صورت pi باشد پاسخ مناسب تری دارد.

کنترلر مشتق (D)

این کنترلر به تنهایی استفاده نمی‌شود و به صورت pd یا pid استفاده می‌شود؛ d می‌تواند پاسخ حالت گذرا را بهبود ببخشد ولی مقدار بزرگ d می‌تواند باعث ناپایداری سیستم شود:



در نمودار زیر کنترلرهای مختلف مقایسه شده اند:



حال به استفاده از pid در avr می‌پردازیم:

برای اینکار ابتدا کتابخانه‌ی pid.h را در مسیر پروژه کپی می‌کنیم و آن را در برنامه‌ی اصلی اضافه می‌کنیم:

```
#include <mega16.h>

#include <delay.h>

#include "pid.h"
```

معرفی کتابخانه‌ی pid.h

این کتابخانه را با کدویژن باز می‌کنیم:

Notes | pidboost.c * | pid.h

```
19  #ifndef PID_H
20  #define PID_H
21
22  #define SCALING_FACTOR  128
23
24  /*! \brief PID Status
25   *
26   * Setpoints and data used by the PID control algorithm
27   */
28  typedef struct PID_DATA{
29    //! Last process value, used to find derivative of process value.
30    float lastProcessValue;
31    //! Summation of errors, used for integrate calculations
32    float sumError;
33    //! The Proportional tuning constant, multiplied with SCALING_FACTOR
34    float P_Factor;
35    //! The Integral tuning constant, multiplied with SCALING_FACTOR
36    float I_Factor;
37    //! The Derivative tuning constant, multiplied with SCALING_FACTOR
38    float D_Factor;
39    //! Maximum allowed error, avoid overflow
40    float maxError;
41    //! Maximum allowed sumerror, avoid overflow
42    float maxSumError;
43  } pidData_t;
```

همانطور که مشاهده می‌کنید در ابتدا یک ثابت با نام SCALING_FACTOR تعریف شده است؛ این ثابت در ضرایب pid ضرب شده و باعث افزایش دقت می‌شود و در آخر کار هنگام برگرداندن خروجی pid، خروجی بر این ثابت تقسیم می‌شود.

در ادامه‌ی این کتابخانه ثابتی به عنوان مقدار حداکثر تعریف می‌شود:

```
48    */
49  // Maximum value of variables
50  #define MAX_INT          (140*SCALING_FACTOR)
51  #define MAX_LONG         (280*SCALING_FACTOR)
52  #define MAX_I_TERM       (MAX_LONG / 2)
```

اگر به کدهای کتابخانه به دقت نگاه کنیم ثابت MAX_INT مقدار حداکثر قسمت P و همچنین مقدار حداکثر خروجی PID را تعیین می‌کند و ثابت MAX_I_TERM مقدار حداکثر قسمت I را تعیین می‌کند. این مقادیر حداکثر با توجه به سیستم انتخاب می‌شوند برای مثال همانطور که در تصویر مشخص است ما مقادیر حداکثر را برابر ۱۴۰ برای ضریب P و مقدار حداکثر (۲۸۰/۲) را برای ضریب I انتخاب کردیم؛ به این دلیل که دی سیستم ما خروجی PID سیستم به عنوان دیوتی سایکل یک مبدل استفاده می‌شود که در مقدار ۲۰۰ برابر دیوتی سایکل ۱۰۰ درصد و مقدار ۰ برابر دیوتی سایکل صفر در صد می‌باشد و به سیستم ما نباید دیوتی سایکلی بیش از ۷۰ درصد اعمال شود، لذا MAX_INT را برابر ۱۴۰ قرار دادیم.

تنها جایی که باید متناسب با سیستم و برنامه‌ی خود در این کتابخانه تغییر ایجاد کنیم همین انتخاب مقدار حداکثر است.

برای درک بیشتر این کتابخانه می‌توانید دستورات درون آن را مورد مطالعه قرار دهید. حال وارد کد برنامه‌ی اصلی خود می‌شویم و ضرایب p و i و d را مشخص می‌کنیم:

برای مثال ما از کنترلر pi با ضرایب زیر استفاده کرده‌ایم:

```
#include <mega16.h>

#include <delay.h>

#include "pid.h"

#define K_P     0.012256
#define K_I     73.72
#define K_D     0.00
```

سپس متغیرهای زیر را که در برنامه مورد استفاده قرار می‌گیرند تعریف می‌کنیم:

```
struct PID_DATA pidData;
int gFlags=0;
int referenceValue, measurementValue;
int out;
int a=0;
```

این متغیرها شامل:

- یک ساختمان به نام PID_DATA است که در طول برنامه و توسط کتابخانه‌ی pid.h مقداردهی می‌شود و از مقادیر آن استفاده می‌شود.
- یک متغیر به نام gFlags که به عنوان پرچم pid استفاده می‌شود؛ یعنی زمانی که این متغیر یک باشد تابع pid فراخوانی می‌شود؛ این متغیر در طول برنامه توسط تایمر صفر در زمان‌های ثابت ،که بسته به سیستم ما آن را تعیین می‌کنیم، صفر و یک می‌شود.
- دو متغیر با نام‌های referenceValue و measurementValue که اولی با مقدار مرجع و دومی‌با مقدار اندازه گیری شده خروجی سیستم مقدار دهی می‌شوند.
- یک متغیر به نام out که خروجی تابع pid را در آن میریزیم و از آن استفاده می‌کنیم.
- متغیر a را به خاطر ایجاد زمان‌های دلخواه در تایمر صفر تعریف کردیم.

حال تابع راه اندازی pid را با توجه به کتابخانه‌ی pid.h می‌نویسیم:

```
void Init(void)
{

pid_Init(K_P * SCALING_FACTOR, K_I * SCALING_FACTOR , K_D *
SCALING_FACTOR ,&pidData);
}
```

تابع pid باید در زمان‌های مشخصی اجرا شود، پس باید تایمر میکروکنترلر را فعال کرده و در زمان‌های مشخص تابع pid را فراخوانی کنیم، برای مثال ما تایمر صفر را فعال کرده‌ایم، به صورتی که هر ۲۵۵ میکروثانیه وارد تابع وقفه شود، و درون تابع وقفه کدی نوشته‌ایم که که بعد از پنج بار که وارد تابع وقفه شد پرچم تابع pid را فعال کند (یعنی ۵*۲۵۵ میکروثانیه یک بار pid انجام شود)

```
// Timer 0 output compare interrupt service routine
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{

a++;
if(a==5) {
```

```
gFlags=1;
a=0;
}

}
```

قسمت adc را نیز فعال می‌کنیم که بتوانیم مقدار خروجی سیستم را بخوانیم.

ادامه‌ی کد را به صورت زیر می‌نویسیم:

```
Init();
while (1)
{

{ if(gFlags)

referenceValue = 512 ;
measurementValue = read_adc(0);

out = pid_Controller(referenceValue, measurementValue,
&pidData);
if(out<0)
out=0;
OCR1A=out;
gFlags =0;
}
}
```

توسط دستورات فوق ابتدا تابع ()Init را می‌نویسیم که مقادیر ضرایب pid را در کتابخانه‌ی pid.h مقدار دهی می‌کند.

سپس توسط دستور if اگر پرچم pid فعال بود (یعنی در زمان‌های مشخص) ابتدا مقدار مرجع را مقدار دهی می‌کنیم (در اینجا چون مرجع ما ۲,۵ ولت بود مقدار ۵۱۲ را انتخاب کردیم؛ چون قرار است این مرجع با ورودی adc مقایسه شود آن را به دیجیتال نوشتیم)

سپس مقدار خروجی سیستم توسط تابع adc خوانده می‌شود،

سپس توسط تابع pid_controller مقدار مرجع، مقدار خوانده شده توسط adc و ساختمان pidData که در ابتدا تعریف کرده بودیم را به کتابخانه pid.h می‌فرستیم، که خروجی آن را که خروجی تابع pid می‌باشد را در متغیر out که خروجی ما می‌باشد می‌ریزیم.

ما در این برنامه چون قصد کنترل دیوتی سایکل را داشتیم مقدار خروجی را به عنوان OCR1A که مشخص کننده‌ی دیوتی سایکل خروجی PWM تایمر یک می‌باشد قرار داده‌ایم.

کد برنامه‌ی اصلی نوشته شده به صورت زیر می‌باشد:

```
#include <mega16.h>
#include <delay.h>
#include "pid.h"

#define K_P 0.012256
#define K_I 23.72
#define K_D 0.00
```

```
struct PID_DATA pidData;
int gFlags=0;
int referenceValue, measurementValue;
int out;
int a=0;
void Init(void)
{

pid_Init(K_P * SCALING_FACTOR, K_I * SCALING_FACTOR , K_D *
SCALING_FACTOR , &pidData);
}


// Timer 0 output compare interrupt service routine
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{

a++;
if(a==5) {
gFlags=1;
a=0;
}

}

#define ADC_VREF_TYPE 0x40

// Read the AD conversion result
unsigned int read_adc(unsigned char adc_input)
{
ADMUX=adc_input | (ADC_VREF_TYPE & 0xff);
// Delay needed for the stabilization of the ADC input voltage
delay_us(10);
// Start the AD conversion
ADCSRA|=0x40;
// Wait for the AD conversion to complete
while ((ADCSRA & 0x10)==0);
ADCSRA|=0x10;
return ADCW;
}

// Declare your global variables here

void main(void)
{


// Port D initialization
// Func7=In Func6=In Func5=Out Func4=In Func3=In Func2=In Func1=In
Func0=In
// State7=T State6=T State5=0 State4=T State3=T State2=T State1=T State0=T
PORTD=0x00;
DDRD=0x20;

// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: 1000.000 kHz
// Mode: CTC top=OCR0
// OC0 output: Disconnected
```

```
                                                    TCCR0=0x0A;
                                                    TCNT0=0x00;
                    OCR0=0xFF;        ////////////////////////////255 micro sec

                                            // Timer/Counter 1 initialization
                                                // Clock source: System Clock
                                                 // Clock value: 8000.000 kHz
                                        // Mode: Ph. & fr. cor. PWM top=ICR1
                                                    // OC1A output: Inverted
                                                     // OC1B output: Discon.
                                                      // Noise Canceler: Off
                                                // Input Capture on Falling Edge
                                               // Timer1 Overflow Interrupt: Off
                                                 // Input Capture Interrupt: Off
                                               // Compare A Match Interrupt: Off
                                               // Compare B Match Interrupt: Off
                                                                    TCCR1A=0x80;
                                                                    TCCR1B=0x11;
                                                                    TCNT1H=0x00;
                                                                    TCNT1L=0x00;
                                                                     ICR1H=0x00;
200 = 20kHz      top OCR1A = ICR1L=0xC8;                 ////////////////PWM fr
                                                                    OCR1AH=0x00;
                                                                    OCR1AL=0x00;
                                                                    OCR1BH=0x00;
                                                                    OCR1BL=0x00;


                            // Timer(s)/Counter(s) Interrupt(s) initialization
                                                                    TIMSK=0x02;


                                                // Analog Comparator initialization
                                                    // Analog Comparator: Off
                    // Analog Comparator Input Capture by Timer/Counter 1: Off
                                                                     ACSR=0x80;
                                                                    SFIOR=0x00;

                                                             // ADC initialization
                                                // ADC Clock frequency: 1000.000 kHz
                                                  // ADC Voltage Reference: AVCC pin
                                            // ADC Auto Trigger Source: ADC Stopped
                                                        ADMUX=ADC_VREF_TYPE & 0xff;
                                                                    ADCSRA=0x83;

                                                    // Global enable interrupts
                                                                    #asm("sei")


                                                                      Init();
                                                                    while (1)
                                                                            {

                                                          {                 if(gFlags)

                                                                referenceValue = 512 ;
                                                      measurementValue = read_adc(0);
```

```
                    out = pid_Controller(referenceValue, measurementValue,
                                                          &pidData);
                                                                if(out<0)
                                                                   out=0;
                                                        OCR1A=out;


                                                                gFlags =0;
                                                                  }
                                                                          }
                                                                          }
```

و تابع کتابخانه به صورت زیر می‌باشد:

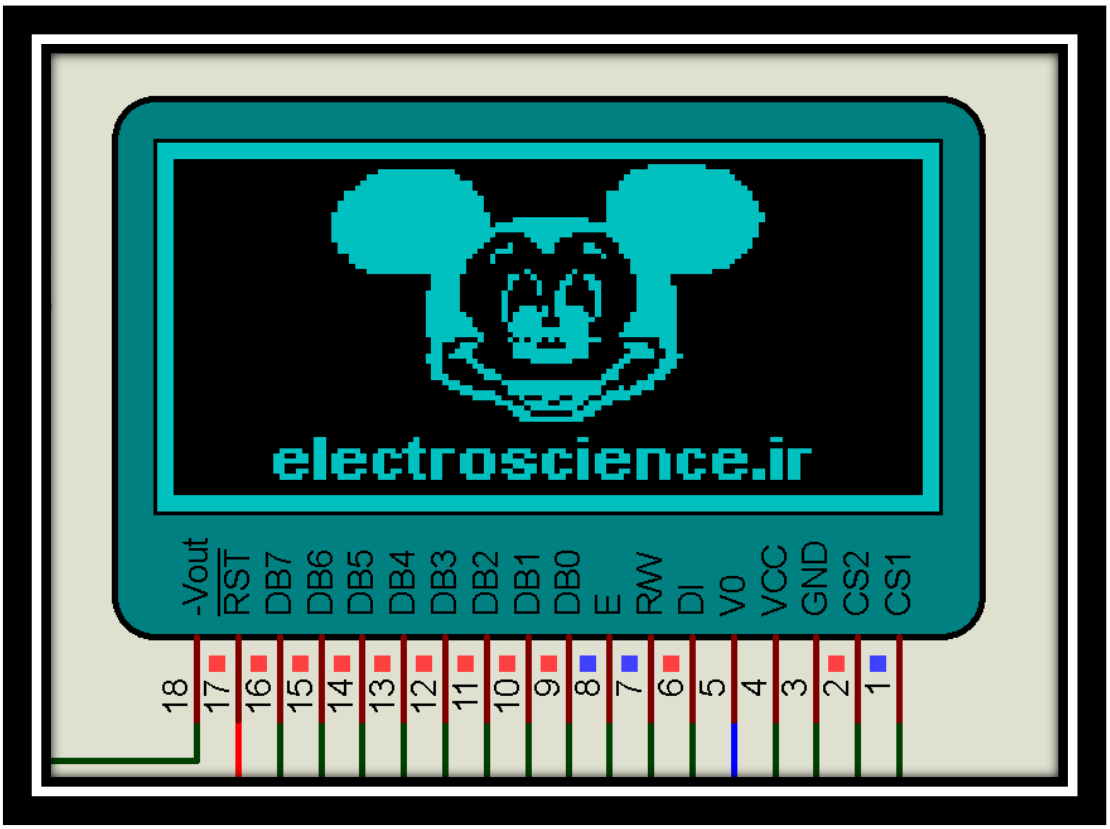
```
#ifndef PID_H
#define PID_H

#define SCALING_FACTOR  128

/*! \brief PID Status
 *
 * Setpoints and data used by the PID control algorithm
 */
typedef struct PID_DATA{
  //! Last process value, used to find derivative of process value.
  float lastProcessValue;
  //! Summation of errors, used for integrate calculations
  float sumError;
  //! The Proportional tuning constant, multiplied with SCALING_FACTOR
  float P_Factor;
  //! The Integral tuning constant, multiplied with SCALING_FACTOR
  float I_Factor;
  //! The Derivative tuning constant, multiplied with SCALING_FACTOR
  float D_Factor;
  //! Maximum allowed error, avoid overflow
  float maxError;
  //! Maximum allowed sumerror, avoid overflow
  float maxSumError;
} pidData_t;

/*! \brief Maximum values
 *
 * Needed to avoid sign/overflow problems
 */
// Maximum value of variables
#define MAX_INT         (140*SCALING_FACTOR)
#define MAX_LONG        (280*SCALING_FACTOR)
#define MAX_I_TERM      (MAX_LONG / 2)

// Boolean values
#define FALSE           0
#define TRUE            1

//void pid_Init(int16_t p_factor, int16_t i_factor, int16_t d_factor,
struct PID_DATA *pid);
//int16_t pid_Controller(int16_t setPoint, int16_t processValue, struct
PID_DATA *pid_st);
//void pid_Reset_Integrator(pidData_t *pid_st);
```

```
void pid_Init(float p_factor, float i_factor, float d_factor, struct
PID_DATA *pid)
// Set up PID controller parameters
{
  // Start values for PID controller
  pid->sumError = 0;
  pid->lastProcessValue = 0;
  // Tuning constants for PID loop
  pid->P_Factor = p_factor;
  pid->I_Factor = i_factor;
  pid->D_Factor = d_factor;
  // Limits to avoid overflow
  pid->maxError = MAX_INT / (pid->P_Factor + 1);
  pid->maxSumError = MAX_I_TERM / (pid->I_Factor + 1);
}


/*! \brief PID control algorithm.
 *
 *  Calculates output from setpoint, process value and PID status.
 *
 *  \param setPoint  Desired value.
 *  \param processValue  Measured value.
 *  \param pid_st  PID status struct.
 */
int pid_Controller(int setPoint, int processValue, struct PID_DATA *pid_st)
{
  int ret;
  float error, p_term, i_term,temp,d_term ;

  error = setPoint - processValue;
  error=error/200.0;
  // Calculate Pterm and limit error overflow
  if (error > pid_st->maxError){
    p_term = MAX_INT;
  }
  else if (error < -pid_st->maxError){
    p_term = -MAX_INT;
  }
  else{
    p_term = pid_st->P_Factor * error;
  }

  // Calculate Iterm and limit integral runaway
  temp = pid_st->sumError + error;
  if(temp > pid_st->maxSumError){
    i_term = MAX_I_TERM;
    pid_st->sumError = pid_st->maxSumError;
  }
  else if(temp < -pid_st->maxSumError){
    i_term = -MAX_I_TERM;
    pid_st->sumError = -pid_st->maxSumError;
  }
  else{
    pid_st->sumError = temp;
    i_term = pid_st->I_Factor * pid_st->sumError;
  }

  // Calculate Dterm
```

```
  d_term = pid_st->D_Factor * (pid_st->lastProcessValue - processValue);

  pid_st->lastProcessValue = processValue;

  ret = (p_term + i_term + d_term) / SCALING_FACTOR;

  //ret=ret/200;
  if(ret > (MAX_INT/SCALING_FACTOR)){
    ret = (MAX_INT/SCALING_FACTOR);
  }
  else if(ret < -(MAX_INT/SCALING_FACTOR)){
    ret = -(MAX_INT/SCALING_FACTOR);
  }

  return(ret);
}

/*! \brief Resets the integrator.
 *
 *  Calling this function will reset the integrator in the PID regulator.
 */
void pid_Reset_Integrator(pidData_t *pid_st)
{
  pid_st->sumError = 0;
}


#endif
```

پیوست تکمیلی کتاب-بخش ۲

کار با ال سی دی گرافیکی



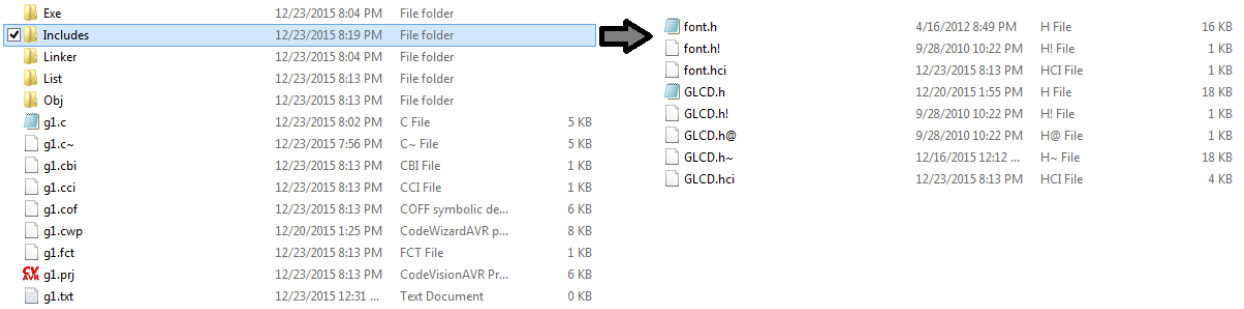
کار با ال‌سی‌دی های گرافیکی (GLCD)

برای کار با GLCD می‌توانیم از کتابخانه‌های موجود استفاده کنیم و با استفاده از آنها دستورات مورد نظر یا اشکال و نوشته‌های مورد نظر خود را در ال‌سی‌دی نمایش دهیم.

برای اینکار ابتدا در پروژه‌ی خود کتابخانه‌ی زیر را اضافه می‌کنیم :

```
#include "Includes/GLCD.h"
```

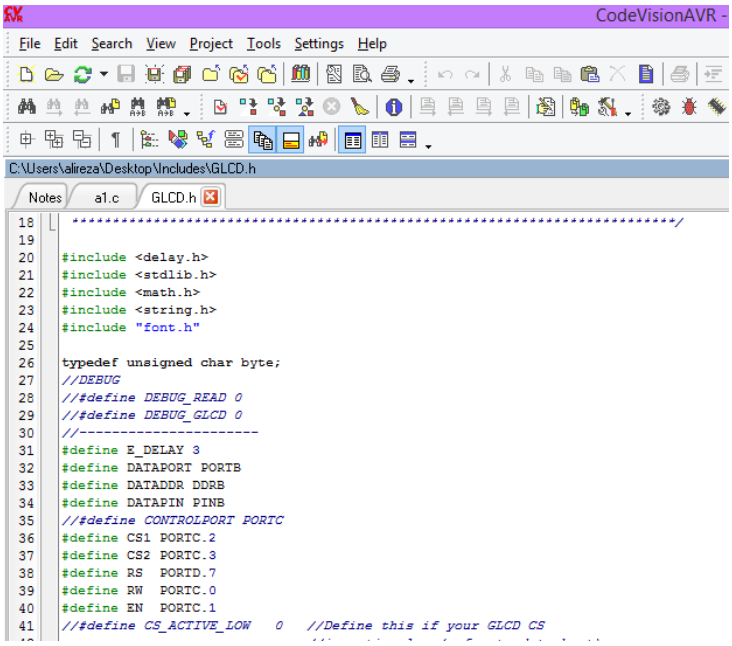
و فایل مربوط به کتابخانه‌های (GLCD.h و font.h)را در پوشه‌ی Includes در مسیر پروژه قرار می‌دهیم:



یا به عبارتی فولدر Includes را که در سایت برای دانلود گذاشته‌ایم در مسیر پروژه کپی کنید.

(www.electroscience.ir)

حال برای تنظیم پایه‌های ال سی دی گرافیکی، فایل GLCD.h را از طریق کدویژن باز می‌کنیم.



برای کار با این ال‌سی‌دی باید پورت دیتا و پایه‌های کنترلی را مشخص کنیم، همانطور که از نام آنها مشخص است پایه‌های کنترلی دستورات را (نظیر رفتن به مختصات خاص، پاک کردن صفحه‌ی نمایش و ...) به ال‌سی‌دی ارسال می‌کنند و پایه‌های دیتا، اطلاعاتی که باید در ال‌سی‌دی نمایش داده شود را مشخص می‌کنند.

حال توسط سه ماکرو زیر پورت دیتا را مشخص می‌کنیم:

#define DATAPORT PORTB

#define DATADDR DDRB

#define DATAPIN PINB

در دستورات فوق PORTB به عنوان پورت دیتا انتخاب شده است.

توسط ماکروهای زیر نیز پایه‌های کنترلی ال‌سی‌دی را مشخص می‌کنیم:

define CS1 PORTC.2#

define CS2 PORTC.3#

define RS PORTD.7#

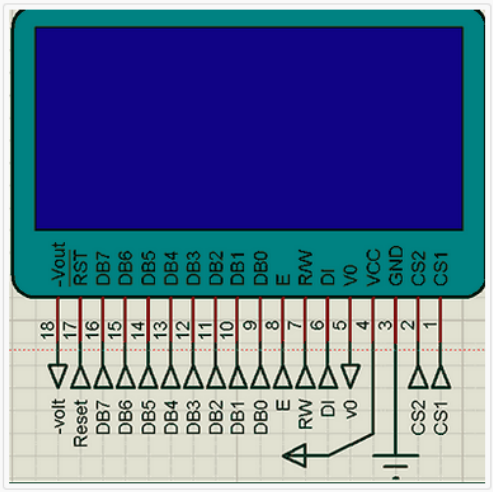
define RW PORTC.0#

define EN PORTC.1#

```
19
20  #include <delay.h>
21  #include <stdlib.h>
22  #include <math.h>
23  #include <string.h>
24  #include "font.h"
25
26  typedef unsigned char byte;
27  //DEBUG
28  //#define DEBUG_READ 0
29  //#define DEBUG_GLCD 0
30  //----------------------
31  #define E_DELAY 3
32  #define DATAPORT PORTB
33  #define DATADDR DDRB
34  #define DATAPIN PINB
35  //#define CONTROLPORT PORTC
36  #define CS1 PORTC.2
37  #define CS2 PORTC.3
38  #define RS  PORTD.7
39  #define RW  PORTC.0
40  #define EN  PORTC.1
41  //#define CS_ACTIVE_LOW   0   //Define this if your GLCD CS
42                                //is active low (refer to datasheet)
```

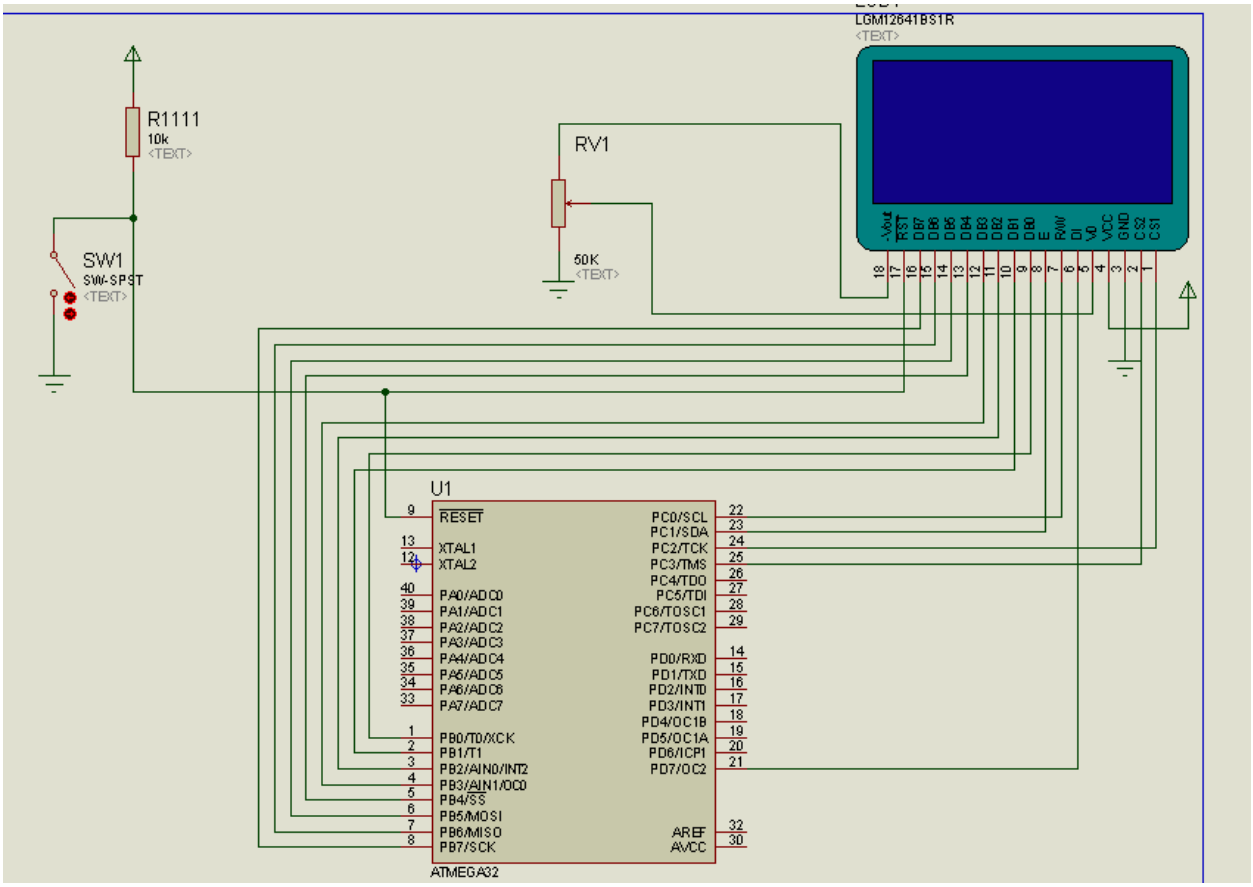
پایه‌های ال سی دی به صورت جدول زیر است:

توضیح پایه های LCD گرافیکی



VSS	به زمین(قطب – منبع تغزیه) وصل میشه
VDD	به ۵ ولت وصل میشود
VO	درخشندگی LCD – مانند شکل بالا بسته شود
D/I یا RS	Table
R/W	اگه ۰ بشه یعنی نوشتن و اگه ۱ بشه یعنی خوندن
E	پایه فعال ساز پذیرش اطلاعات
DB0 تا DB7	پایه های انتقال اطلاعات
CS1 و CS2	انتخاب تراشه اول و دوم
RES	برا RESET کردن این قطعه میباشد.
VEE	Table
A و K	قطب مثبت و منفی مربوط به روشنایی LCD

حال می‌توانیم برای شبیه سازی در نرم افزار پروتئوس، ال سی دی گرافیکی ۶۴*۱۲۸ را با نام (LGM12641BS1R) پیدا کنیم و طبق تنظیمات فوق پایه‌های آن را متصل کنیم:



توسط پتانسیومتر بین پایه‌های vo و vout (vee)- کنتراست صفحه را تنظیم می‌کنیم. فقط به این نکته توجه کنید که پایه‌ی DI در ال سی دی همان پایه‌ی RS می‌باشد.

دستورات GLCD

حال برای نمایش بر روی ال سی دی کافیست از دستورات جدول زیر استفاده کنیم:

توابع	توضیحات	پارامترها
glcd_on()	روشن کردن GLCD	نیازی ندارد
glcd_off()	خاموش کردن GLCD	نیازی ندارد
set_start_line(unsigned char line)	مشخص کردن خط بالایی بر روی صفحه نمایش	line : شماره سطر مورد نظر از بالا Range: 0-63
goto_col(unsigned int x)	رفتن به ستون دلخواه	**: x** شماره ستون مورد نظر Range: 0-127
goto_row(unsigned int y)	رفتن به سطر دلخواه	**: y** شماره سطر دلخواه (Range: 0-7)
goto_xy(unsigned int x, unsigned ,int y)	رفتن به سطر و ستون دلخواه	**:x** مختصات ستون **: y** مختصات سطر
glcd_write(unsigned char b)	نوشتن‌یک بایت دلخواه در مختصات فعلی	**: b** یک بایت دیتایی که قصد دارید بر روی مختصات فعلی بنویسید
glcd_clrln(unsigned char ln)	پاک کردن سطر دلخواه	شماره سطر دلخواهی که می‌خواهید پاک شود (Range: 0-7)
glcd_clear()	پاک کردن صفحه نمایش	نیازی ندارد
glcd_read(unsigned char column)	خواندن بایت مختصات فعلی	**: column** شماره ستون فعلی [2]
point_at(unsigned int x, unsigned int y, byte color)	اضافه کردن نقطه در مختصات دلخواه	**: x** شماره ستون **: y** شماره سطر **: color** ۰=نقطه سفید ۱=نقطه سیاه
h_line(unsigned int x,unsigned int y, byte l,byte s,byte c)	کشیدن‌یک خط افقی	**: x** شماره ستونی که خط از آنجا شروع می‌شود **: y** شماره سطری که خط از آنجا شروع میشود **: l** طول خط **: s** فاصله بین نقطه‌های خط: ۰=خط متصل ۱=خط نقطه چین خط تیره **: c** ۰=نقاط سفید ۱=نقاط سیاه

v_line(**unsigned int x,unsigned int y, signed int l,byte s,byte c)**	کشیدن یک خط عمودی	**: x** شماره ستونی که خط از آنجا شروع می‌شود **: y** شماره سطری که خط از آنجا شروع می‌شود **: l** طول خط **: s** فاصله بین نقطه‌های خط: ۰=خط متصل ۱=خط نقطه چین خط تیره **: c** ۰=نقاط سفید ۱=نقاط سیاه
line(unsigned int x1,unsigned int y1, unsigned int x2,unsigned int y2, byte s,byte c)	کشیدن یک خط دلخواه (با هر شیبی)	**: x1** شماره ستونی که خط از آنجا شروع می‌شود **: y1** شماره سطری که خط از آنجا شروع می‌شود **: x2** شماره ستونی که خط در آنجا تمام می‌شود **: y2** شماره سطری که خط در آنجا تمام می‌شود **: s** فاصله بین نقطه‌های خط: ۰=خط متصل ۱=خط نقطه چین خط تیره **: c** ۰=نقاط سفید ۱=نقاط سیاه
rectangle(**unsigned int x1,unsigned int y1, unsigned int x2,unsigned int y2,** **byte s,byte c)**	کشیدن یک مستطیل(یا مربع)	**: x1** مختصات x نقطه سمت چپ بالای مستطیل **: y1** مختصات y نقطه سمت چپ بالای مستطیل **: x2** مختصات x نقطه سمت راست پایین مستطیل **: y2** مختصات y نقطه سمت راست پایین مستطیل **: s** فاصله بین نقطه‌های خط: ۰=خط متصل ۱=خط نقطه چین خط تیره **: c** ۰=نقاط سفید ۱=نقاط سیاه
cuboid(**unsigned int x11,unsigned int y11,** **unsigned int x12,unsigned int y12,** **unsigned int x21,unsigned int y21,**	کشیدن یک مکعب با مشخص کردن ۲	**: x11** مختصات x نقطه چپ بالایی سطح اول **: y11** مختصات y نقطه چپ بالایی سطح اول

unsigned int x22,unsigned int y22, byte s,byte c)	سطح (مستطیل‌یا مربع)	**: x12** مختصاتx نقطه راست پایینی سطح اول **: y12** مختصاتy نقطه راست پایینی سطح اول **: X21** مختصاتx نقطه چپ بالایی سطح دوم **: Y21** مختصاتy نقطه چپ بالایی سطح دوم **: x22** مختصات x نقطه راست پایینی سطح دوم **: y22** مختصات y نقطه راست پایینی سطح دوم **: s** فاصله بین نقطه‌های خط: ۰=خط متصل ۱=خط نقطه چین خط تیره **: c** ۰=نقاط سفید ۱=نقاط سیاه
h_parallelogram(unsigned int x1,unsigned int y1, unsigned int x2,unsigned int y2, byte l,byte s,byte c)	کشیدن‌یک متوازی الاضلاع با مشخص کردن سطح بالایی و پایینی افقی	**: x1** مختصات Xنقطه چپ بالایی **: y1** مختصات Yنقطه چپ بالایی **: x2** مختصات Xنقطه راست پایینی **: y2** مختصات Yنقطه راست پایینی **: l** طول خطوط افقی (بالایی‌یا پایینی) **: s** فاصله بین نقطه‌های خط: ۰=خط متصل ۱=خط نقطه چین خط تیره **: c** ۰=نقاط سفید ۱=نقاط سیاه
v_parallelogram(unsigned int x1,unsigned int y1, unsigned int x2,unsigned int y2, byte l,byte s,byte c)	کشیدن متوازی الاضلاع با مشخص کردن سطوح عمودی سمت چپ و راست	**: x1** مختصات Xنقطه چپ بالایی **: y1** مختصات Yنقطه چپ بالایی **: x2** مختصات Xنقطه راست پایینی **: y2**

		مختصات yنقطه راست پایینی **: l** طول خطوط عمودی (راست‌یا چپ) **: s** فاصله بین نقطه‌های خط: ۰=خط متصل ۱=خط نقطه چین خط تیره **: c** ۰=نقاط سفید ۱=نقاط سیاه
h_parallelepiped(**unsigned int x11,unsigned int y11,** **unsigned int x12,unsigned int y12,** **byte l1,** **unsigned int x21,unsigned int y21,** **unsigned int x22,unsigned int y22,** **byte l2,** **byte s,byte c)**		**: x11** مختصات xنقطه چپ بالایی سطح اول **: y11** مختصات yنقطه چپ بالایی سطح اول **: x12** مختصات xنقطه راست پایینی سطح اول **: y12** مختصات yنقطه راست پایینی سطح اول
		: s فاصله بین نقطه‌های خط: ۰=خط متصل ۱=خط نقطه چین خط تیره **: c** ۰=نقاط سفید ۱=نقاط سیاه **:l1** عرض خط افقی سطح اول **: l2** عرض خط افقی سطح دوم
v_parallelepiped(**unsigned int x11,unsigned int y11,** **unsigned int x12,unsigned int y12, byte l1,** **unsigned int x21,unsigned int y21,** **unsigned int x22,unsigned int y22, byte l2,**	کشیدن متوازی السطوح بر اساس دو سطح عمودی (Seev_parallelogram)	**: x11** مختصات xنقطه چپ بالایی سطح اول **: y11** مختصات yنقطه چپ بالایی سطح اول **: x12**

byte s,byte c)		مختصات Xنقطه راست پایینی سطح اول **: y12** مختصات Yنقطه راست پایینی سطح اول
		: X21 مختصات Xنقطه چپ بالایی سطح دوم **: Y21** مختصات Yنقطه چپ بالایی سطح دوم **: x22** مختصات Xنقطه راست پایینی سطح دوم **: y22** مختصات Yنقطه راست پایینی سطح دوم
		: s فاصله بین نقطه‌های خط: ۰=خط متصل ۱=خط نقطه چین خط تیره **: c** ۰=نقاط سفید ۱=نقاط سیاه **:l1** طول خط عمودی سطح اول **: l2** طول خط عمودی سطح دوم
circle(**unsigned int x0,unsigned int y0,** **unsigned int r,byte s,byte c)**	کشیدن‌یک دایره	**: x0** مختصات Xنقطه وسط دایره **: y0** مختصات Yنقطه وسط دایره r : شعاع دایره **: s** فاصله بین نقطه‌های خط: ۰=خط متصل ۱=خط نقطه چین خط تیره **: c** ۰=نقاط سفید ۱=نقاط سیاه
glcd_putchar(byte c,int x,int y,byte l, byte sz)	نوشتن‌یک کارکتر بر روی مختصات دلخواه بر اساس سایز SZ	**: c** کارکتری که قرار است تایپ شود **: x** ستونی که می‌خواهید نوشته شروع به تایپ کند (یک کاراکتر هشت ستون را اشغال می‌کند)

		: y شماره سطر مورد نظر برای نوشتن **: l** زبان کاکتر 0=انگلیسی 1=فارسی‌یا عربی **: sz** سایز فونت عددی بین ۱ تا ۷
glcd_puts(**byte *c,int x,int y,unsigned char l,byte sz,signed char space)**	نوشتن‌یک رشته (که ذخیره شده در حافظه‌ی فلش) بر روی نمایشگر	**: c** یک اشاره گر به فرم رشته که قرار است روی نمایشگر نوشته شود **: x** شماره ستونی که قرار است نوشته از آنجا شروع شود .)یک کاراکتر۸ ستون اشغال میکند(**: y** شماره سطر مورد نظر **: l** زبان نوشته 0=انگلیسی 1=فارسی‌یا عربی **: sz** سایز نوشته ازیک تا هفت **: space** انگلیسی :قابلیت فاصله گذاری بین حروف فارسی و عربی: قابلیت فاصله گذاری بین کلمات
bmp_disp(flash byte *bmp, **unsigned int x1,unsigned int y1,** **unsigned int x2,unsigned int y2)**	نمایش‌یک تصویر BMP با استفاده از آرایه ساخته شده درحافظه فلش	**: bmp** یک اشاره گر به آرایه ای که تصویر در آن ذخیره شده است **: x1** مختصات Xنقطه سمت چپ بالای مکانی که می‌خواهید تصویر در آن نمایش‌یابد **: y1** مختصات Yنقطه سمت چپ بالای مکانی که می‌خواهید تصویر در آن نمایش‌یابد **: x2** مختصات Xنقطه سمت راست پایین مکانی که می‌خواهید تصویر در آن نمایش‌یابد **: y2** مختصات Yنقطه سمت راست پایین مکانی که می‌خواهید تصویر در آن نمایش‌یابد

به عنوان مثال برنامه‌ی زیر را در نظر بگیرید:

ابتدا بدون هیچ تنظیم خاصی در کدویزارد یک پروژه‌ی جدید ایجاد کرده سپس پوشه‌ی Includes در مسیر پروژه قرار می‌دهیم.

حال کتابخانه‌ی GLCD.h را به آن اضافه می‌کنیم:

```
13  Comments:
14
15
16  Chip type               : ATmega32
17  Program type            : Application
18  AVR Core Clock frequency: 8.000000 MHz
19  Memory model            : Small
20  External RAM size       : 0
21  Data Stack size         : 512
22  *****************************************************/
23
24  #include <mega32.h>
25  #include <stdio.h>
26
27  #include "Includes/GLCD.h"
28
29
30  // Declare your global variables here
31
32  void main(void)
33  {
34
```

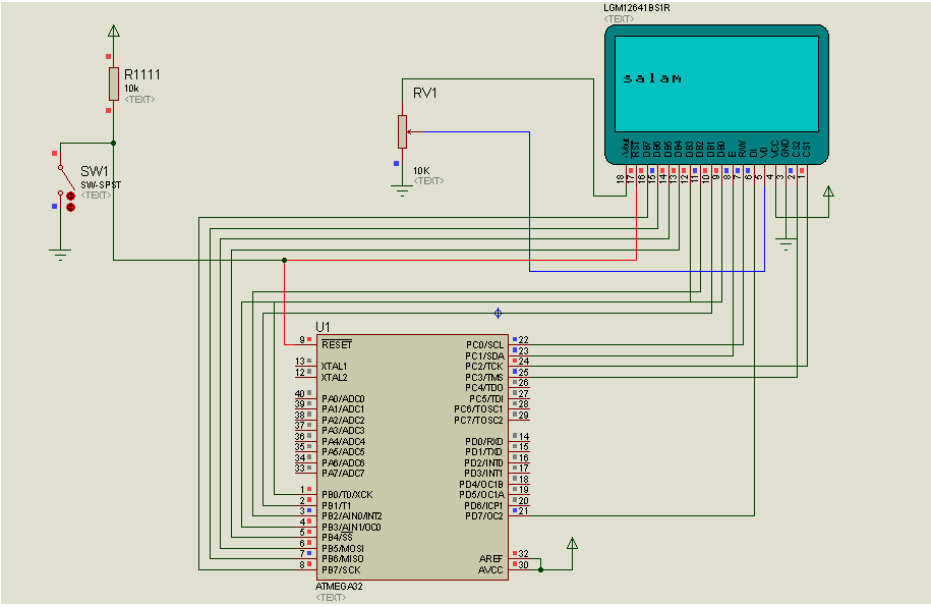
کتابخانه‌ی GLCD.h را توسط کدویژن باز می‌کنیم و پورت دیتا و پورت کنترل را انتخاب می‌کنیم:

```
22  #include <math.h>
23  #include <string.h>
24  #include "font.h"
25
26  typedef unsigned char byte;
27  //DEBUG
28  //#define DEBUG_READ 0
29  //#define DEBUG_GLCD 0
30  //----------------------
31  #define E_DELAY 3
32  #define DATAPORT PORTB
33  #define DATADDR DDRB
34  #define DATAPIN PINB
35  //#define CONTROLPORT PORTC
36  #define CS1 PORTC.2
37  #define CS2 PORTC.3
38  #define RS  PORTD.7
39  #define RW  PORTC.0
40  #define EN  PORTC.1
41  //#define CS_ACTIVE_LOW   0   //Define this if your GLCD CS
42                                //is active low (refer to datasheet)
43  #pragma used+
44
45  //--------Arabic----------
46  static int prevLet = 193;
47  static byte st = 0;
```

حال به برنامه‌ی خود باز می‌گردیم و دستور زیر را می‌نویسیم:

```
157 // E - PORTC Bit 1
158 // RD /WR - PORTC Bit 0
159 // RS - PORTD Bit 7
160 // /RST - PORTD Bit 3
161 // CS1 - PORTC Bit 2
162 // CS2 - PORTC Bit 3
163
164 // Specify the current font for displaying text
165 // No function is used for reading
166 // image data from external memory
167 glcd_on();
168 while (1)
169       {
170
171       glcd_puts("salam",2,3,0,1,0);
172
173
174       }
175 }
```

در نرم افزار پروتئوس این برنامه را اجرا می‌کنیم:

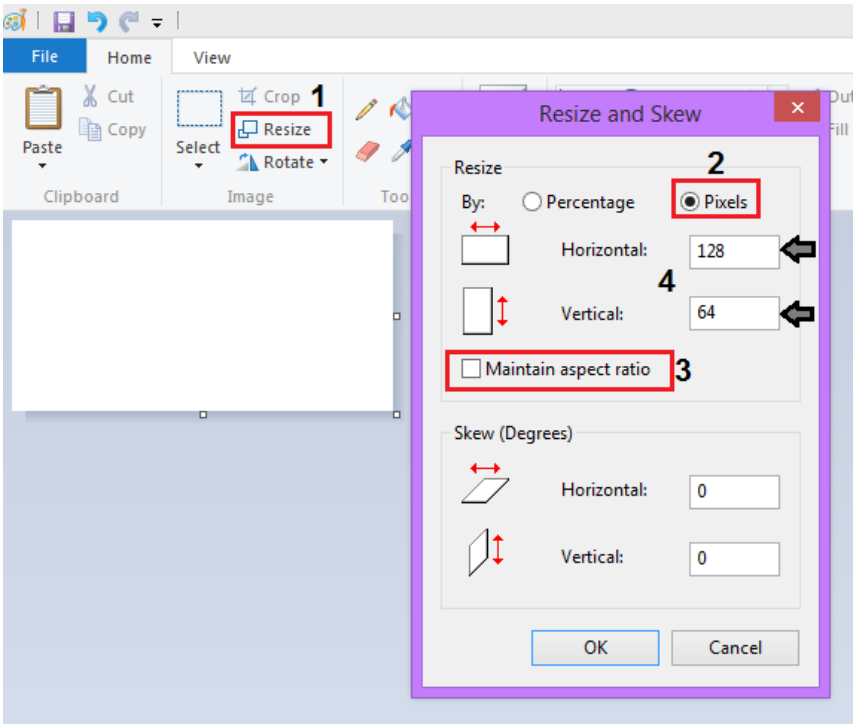


به همین ترتیب با توجه به جدول دستورات می‌توانیم هر نوشته‌یا شکل دلخواه را بر روی ال سی دی نمایش دهیم.

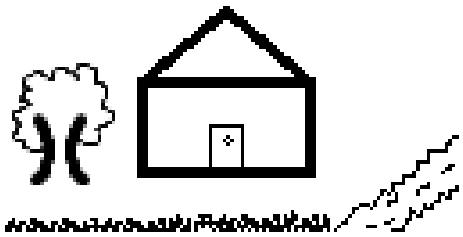
نمایش تصویر دلخواه در ال سی دی

برای اینکار کافیست نرم افزار paint را باز کنیم و ابعاد صفحه را متناسب با اندازه‌ی ال سی دی انتخاب کنیم.

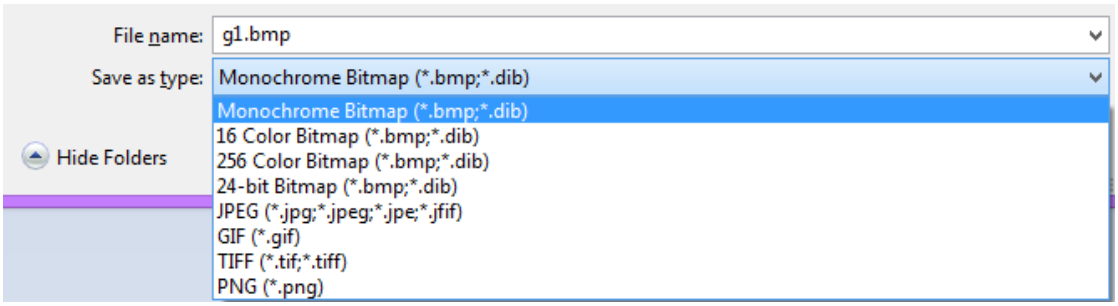
برای اینکار مراحل زیر را انجام دهید:



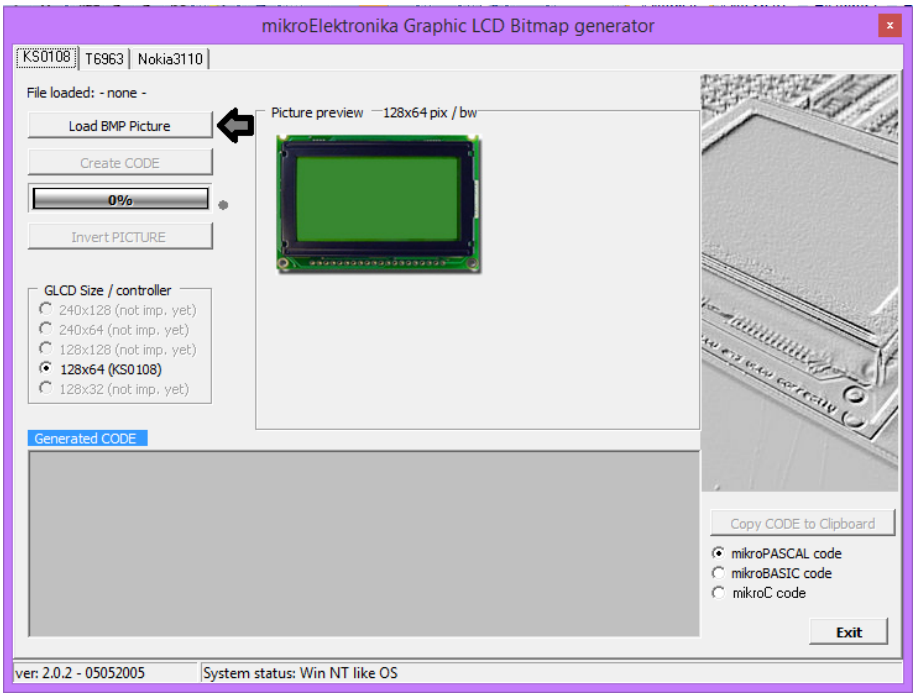
حال شکل مورد نظر خودمان را رسم می‌کنیم :



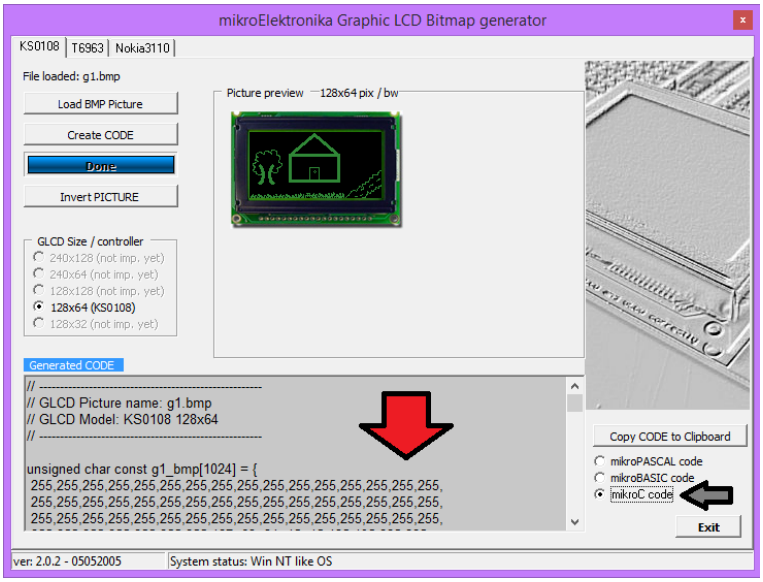
نکته: اگر تصویری آماده داشتیم می‌توانیم آن را در paint باز کنیم و سپس اندازه‌ی آن را همانند مراحل بالا ۶۴*۱۲۸ قرار دهیم. حال تصویر را با فورمت bmp ذخیره می‌کنیم:



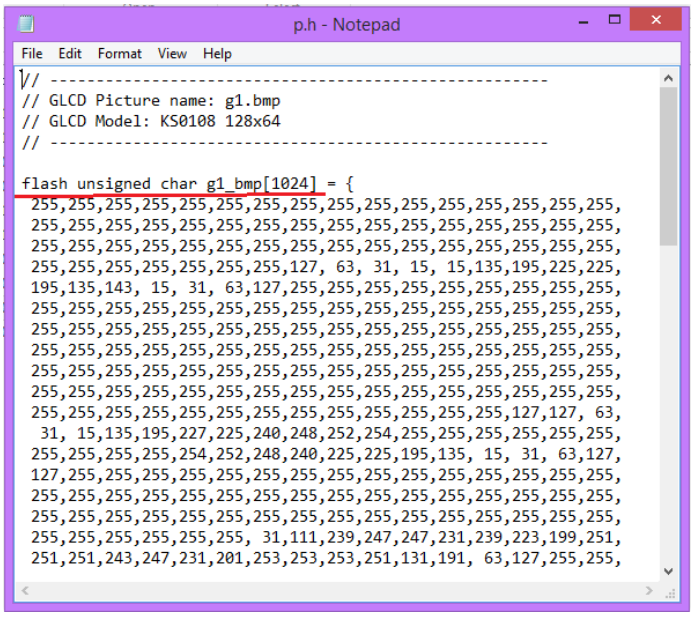
سپس نرم افزار glcd_editor را باز می‌کنیم و تصویر را لود می‌کنیم:



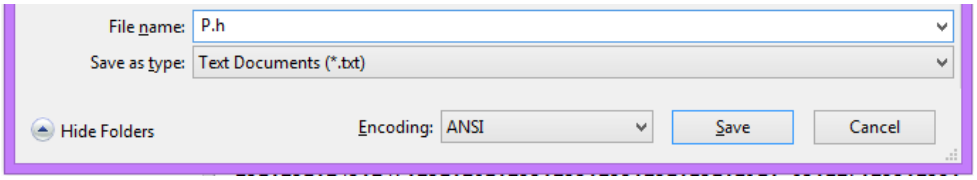
بعد از لود شدن عکس روی گزینه‌ی mikroC code کلیک می‌کنیم:



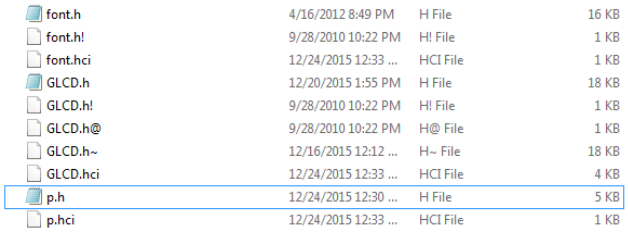
کد فوق را درون‌یک note pad کپی می‌کنیم و نوع متغیر را به flash unsigned char تغییر می‌دهیم و نام آن را نیز نام دلخواه می‌گذاریم:



همانطور که مشخص است آرایه را از نوع flash و به صورت unsigned char تعریف کردیم، نام آن را نیز به صورت دلخواه g1_bmp گذاشتیم(که هر نام دیگری می‌توانستیم بگذاریم). حال آن را با فورمت h. ذخیره می‌کنیم:



این فایل را در فولدر Include در مسیر پروژه ذخیره می‌کنیم.



font.h	4/16/2012 8:49 PM	H File	16 KB
font.h!	9/28/2010 10:22 PM	H! File	1 KB
font.hci	12/24/2015 12:33 ...	HCI File	1 KB
GLCD.h	12/20/2015 1:55 PM	H File	18 KB
GLCD.h!	9/28/2010 10:22 PM	H! File	1 KB
GLCD.h@	9/28/2010 10:22 PM	H@ File	1 KB
GLCD.h~	12/16/2015 12:12 ...	H~ File	18 KB
GLCD.hci	12/24/2015 12:33 ...	HCI File	4 KB
p.h	12/24/2015 12:30 ...	H File	5 KB
p.hci	12/24/2015 12:33 ...	HCI File	1 KB

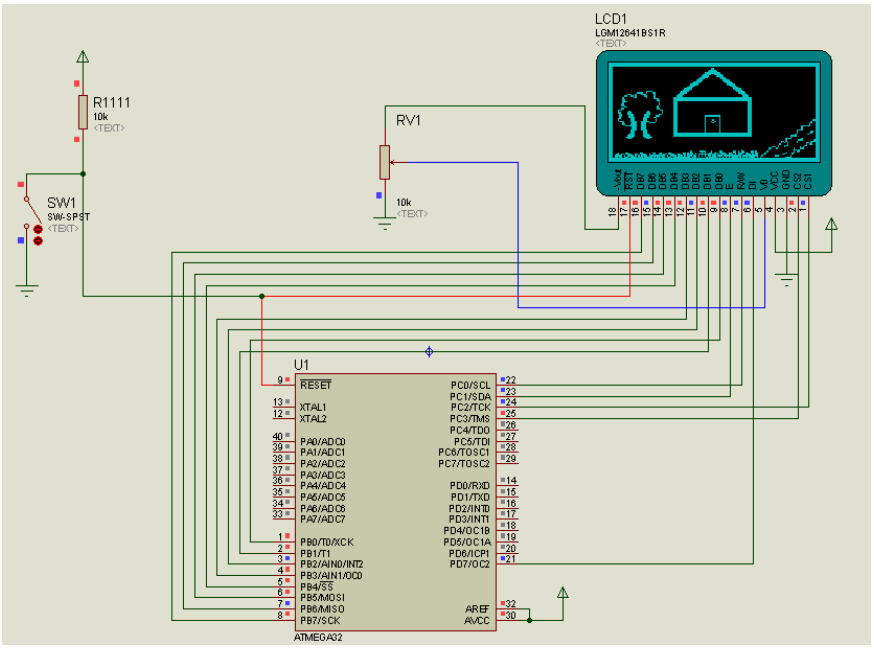
سپس در برنامه‌ی خود آن را اضافه می‌کنیم:

```
16  Chip type               : ATmega32
17  Program type            : Application
18  AVR Core Clock frequency: 8.000000 MHz
19  Memory model            : Small
20  External RAM size       : 0
21  Data Stack size         : 512
22  *****************************************************/
23
24  #include <mega32.h>
25  #include <stdio.h>
26
27  #include "Includes/GLCD.h"
28  #include "Includes/p.h"
29
30  // Declare your global variables here
31
32  void main(void)
33  {
34
```

سپس کد نمایش تصویر را می‌نویسیم، دقت کنید که نام آرایه را g1_bmp تعریف کردیم.

```
161 // CS1 - PORTC Bit 2
162 // CS2 - PORTC Bit 3
163
164 // Specify the current font for displaying text
165 // No function is used for reading
166 // image data from external memory
167 glcd_on();
168 while (1)
169       {
170
171
172       bmp_disp(g1_bmp,0,0,127,7);
173
174       }
175 }
176
```

در کد فوق دو عدد اول (۰و۰) مختصات بالا سمت چپ تصویر و دو عدد بعد (۱۲۷و۷) مختصات پایین سمت چپ تصویر هستند. حال این کد را در پروتئوس شبیه سازی می‌کنیم:


LCD1
LGM12641BS1R
<TEXT>
R1111
10k
<TEXT>
RV1
10k
<TEXT>
SW1
SW-SPST
<TEXT>
U1
RESET
XTAL1
XTAL2
PA0/ADC0
PA1/ADC1
PA2/ADC2
PA3/ADC3
PA4/ADC4
PA5/ADC5
PA6/ADC6
PA7/ADC7
PB0/T0/XCK
PB1/T1
PB2/AIN0/INT2
PB3/AIN1/OC0
PB4/SS
PB5/MOSI
PB6/MISO
PB7/SCK
PC0/SCL
PC1/SDA
PC2/TCK
PC3/TMS
PC4/TDO
PC5/TDI
PC6/TOSC1
PC7/TOSC2
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1
PD4/OC1B
PD5/OC1A
PD6/ICP1
PD7/OC2
AREF
AVCC
ATMEGA32