بسم الله الرحمن الرحیم

آموزش گام به گام کدنویسی
XMEGA با کامپایلر *IAR*

قسمت اول: آشنایی با محیط برنامه نویسی *IAR*

گردآوری

محمد نحوی

بازبینی و تصحیح

اوژن کی نژاد

اردیبهشت ۹۰

نحوه استفاده از آموزش‌ها:

همانطور که می‌دانید تفاوت بسیاری بین خودآموز و مرجع وجود دارد. حتی اگر بار علمی هر دو یکسان باشد ترتیب مطالب ارائه شده در آنها بسیار متفاوت خواهد بود. علت این است که در خودآموز هدف آموزش خواننده مبتدی است و بنابراین ممکن است حین آموزش از بیان مطالبی که فعلا مورد نیاز نیست و یا ممکن است باعث سردرگمی خواننده شود، اجتناب شود و در جای دیگری با توجه به نیاز، مطالب پیشرفته‌تر گفته شود. این امر باعث از بین رفتن نظم و توالی مطالب ارائه شده در خودآموزها می‌شود. در نقطه مقابل هدف از یک مرجع دسترسی به اطلاعات دقیق، کامل، با رعایت نظم است. بنابراین یک مرجع با توجه به حجم مطالب و جزئیات ارائه شده در هر بخش برای خواننده مبتدی مناسب نیست.

در آموزش‌های پیش‌رو سعی می‌شود شِگِرد میانه اتخاذ شود، بدین معنی که روند ارائه مطالب در قالب خودآموز است ولی مطالب پیشرفته‌تری که ممکن است برای برخی مفید باشد، در کادری نارنجی رنگ از بخش‌های دیگر خودآموز تفکیک شده است تا علاوه بر ارائه مطالب نسبتا کامل‌تر در محل مناسب، از سردرگمی خواننده مبتدی جلوگیری شود.

خلاصه: مطالعه مطالب ارائه شده در کادر نارنجی رنگ برای خواننده مبتدی توصیه نمی‌شود.

شروع کار اینجانب با کامپایلر ***IAR*** از شهریور ۸۸ آغاز شد که با راهنمایی های آقای سپاس‌یار جهت انجام یک پروژه نیمه صنعتی این کامپایلر را انتخاب نمودم. علی رغم مشکلاتی که زمان کار با کامپایلرهایی مانند بسکام، کدویژن، ***AVR GCC*** داشتم، تا کنون هیچ مشکلی هنگام کار با کامپایلر ***IAR*** نداشته‌ام.

کامپایلر ***IAR*** به صورت کاملا حرفه‌ای طراحی شده است بهمین دلیل کتابخانه‌های معمولی که در کامپایلرهایی مانند کدویژن وجود دارد همراه با این کامپایلر نیست. کاربران حرفه‌ای کتابخانه‌های خود را به صورت مجزا جمع‌آوری یا به شخصه کدنویسی می‌کنند در نتیجه نیازی به کتابخانه همراه ندارند.

از طرفی کدنویسی برای کامپایلر ***IAR*** به زبان ***ANSI C***(زبان ***C*** استاندارد) بوده در نتیجه کتابخانه‌های گوناگونی که در این قالب کدنویسی شده باشند بالقوه امکان استفاده در کامپایلر ***IAR*** را دارند، برخلاف کامپایلر کدویژن که برای ساده‌سازی برخی قراردادهای جدیدی را به زبان مورد استفاده افزوده است و استفاده از کتابخانه‌های این کامپایلر در کامپایلرهای دیگر نیاز به زمان زیادی برای تبدیل دارد.

علاوه بر جدید بودن محیط کامپایلر ***IAR***، تفاوت‌هایی که این کامپایلر از منظر کدنویسی به زبان ***C*** با کامپایلر کدویژن دارد، انتخاب و استفاده از این کامپایلر را برای کاربران کدویژن قدری سخت کرده است که در ادامه قصد داریم علاوه بر معرفی محیط کاری کامپایلر***IAR***، به برخی تکنیک‌های کدنویسی نیز اشاره نموده تا از سختی این تغییر قدری کاسته شود.

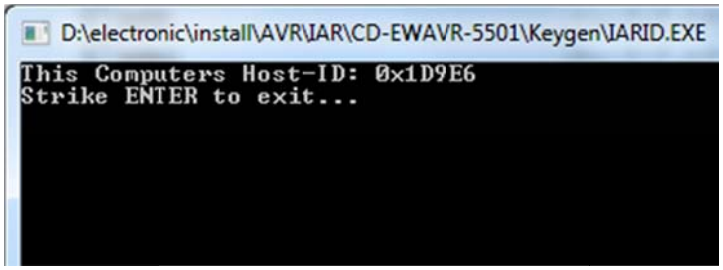
همچنین سعی می شود نحوه افزودن کتابخانه‌های مختلف به پروژه را آموزش داده تا برای بکارگیری ابزار های جانبی مانند ***LED***، صفحه کلید و ... زمان کمتری صرف شود.

۱. نحوه نصب و کرک کامپایلر IAR

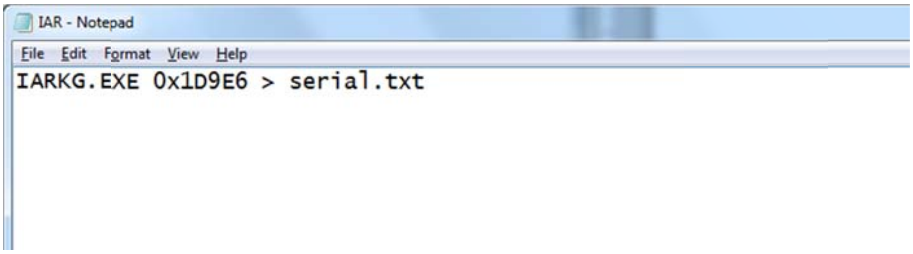
نرم‌افزار را می‌توانید از سایت ***Sonsivri.com*** دانلود و یا از فروشگاه ***http://eshop.eca.ir*** تهیه نمایید.

توضیحاتی که در ادامه آورده شده است برای کرک کردن نرم افزار ***IAR*** است.

در شاخه ***keygen*** سه فایل با نام های (***IAR.bat, IARID.exe, IARKG.exe***) وجود دارد. این سه فایل را بر روی کامپیوتر خود کپی نموده و فایل ***IARID.exe*** را اجرا نمایید. تصویری مانند تصویر زیر ظاهر می شود:



سپس باید بر روی فایل ***IAR.bat*** راست کلیک نموده و گزینه ***edit*** را کلیک نمایید. در پنجره باز شده به جای شماره سریال وارد شده سریالی که در فایل ***IARID.exe*** ظاهر شده است را با **حروف بزرگ** نوشته و فایل را ذخیره نموده و ببندید.

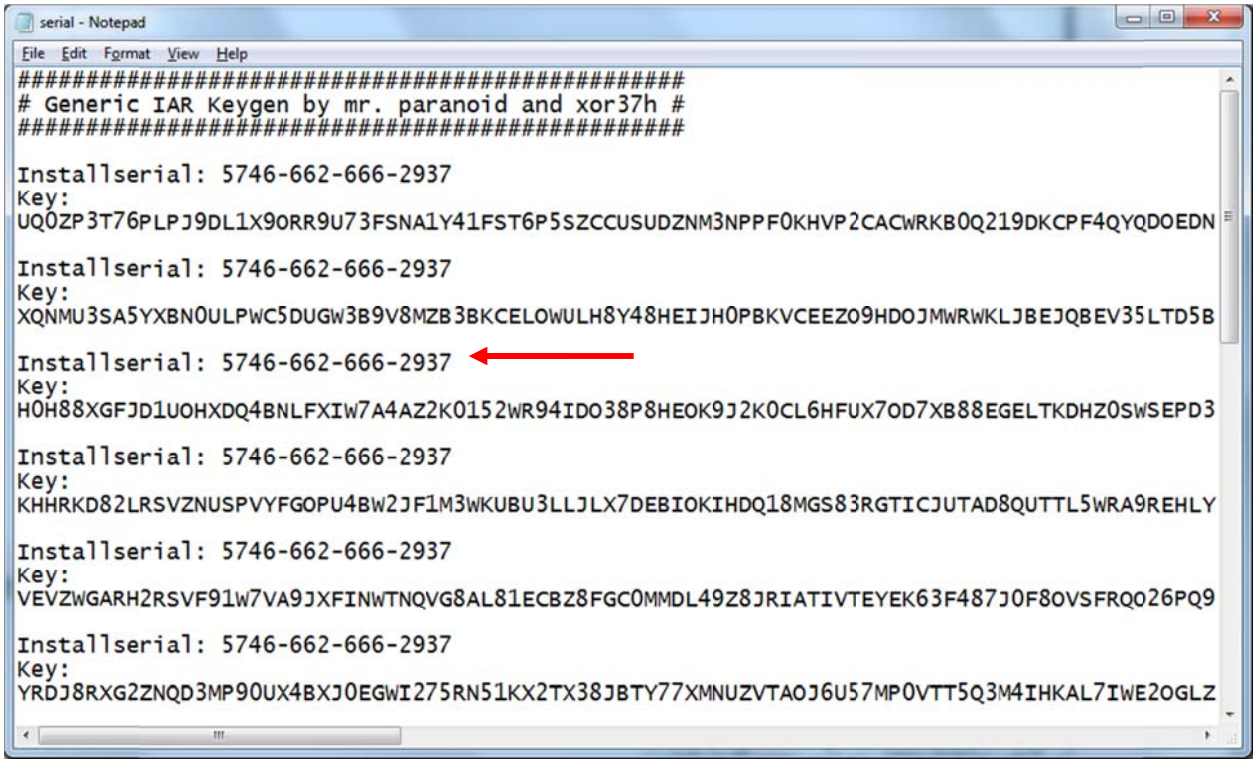


سپس فایل ***IAR.bat*** را اجرا نمایید. پس از اجرای این فایل یک فایل با نام ***serial.txt*** ساخته می شود.

در فایل ساخته شده سریال سوم که با عبارت:

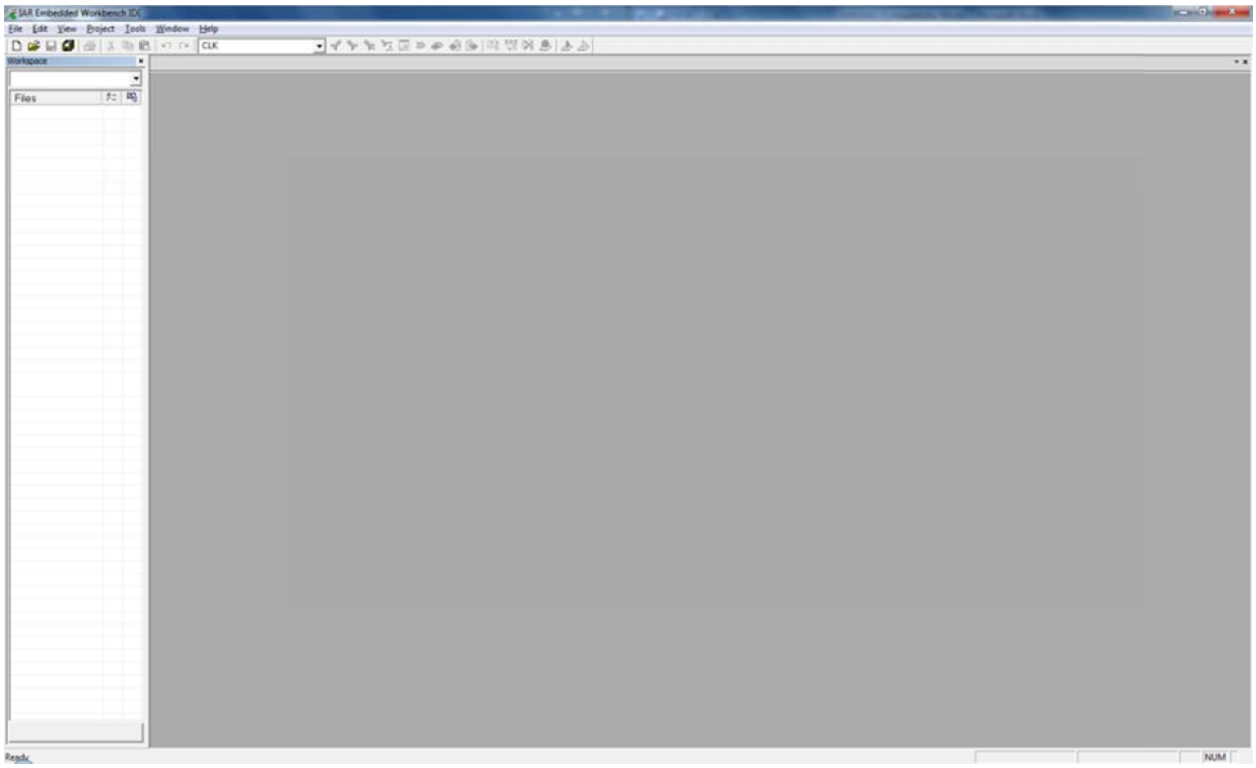
"EWAVR" version "2.25_WIN", no expiration date, exclusive

خاتمه می یابد، سریال ***IAR*** برای ***AVR*** است.

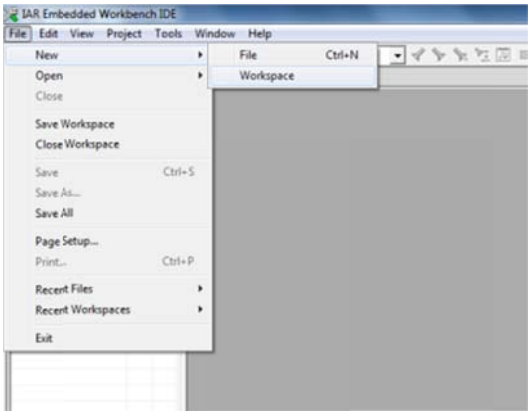


۲. شروع به کار با محیط نرم افزار IAR

نمای ظاهری نرم افزار به صورت زیر است:



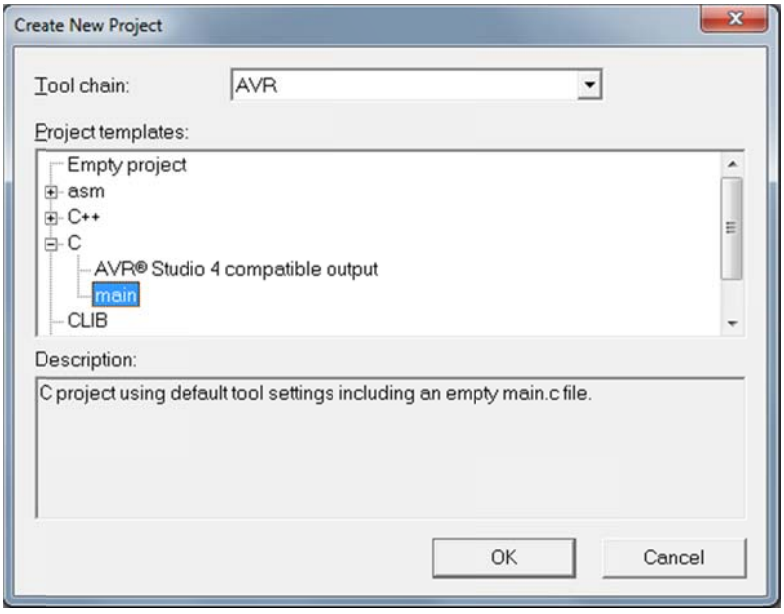
برای شروع ابتدا باید یک ***workspace*** ساخته و سپس یک پروژه را به آن اختصاص دهیم:



برای ساخت پروژه باید مراحل زیر را دنبال کنید:

Project->Create new Projet…

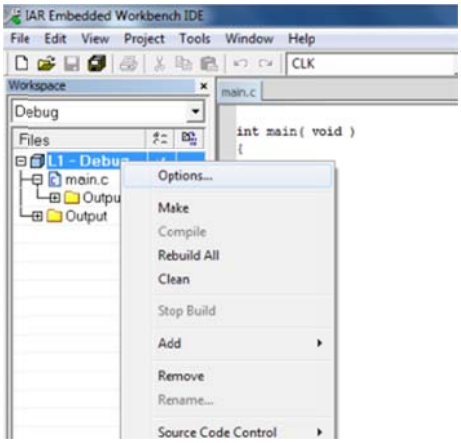
پنجره‌ای مانند زیر باز می شود، مطابق شکل عمل شود:



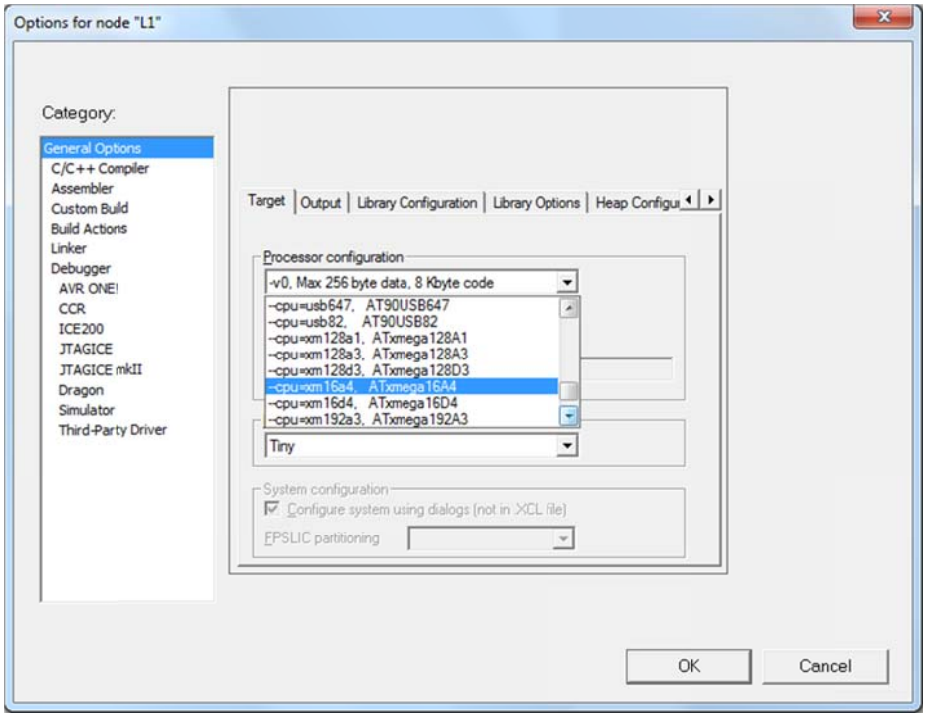
سپس کلید ***OK*** را زده تا پروژه ساخته شود.

اگر مایل به شبیه سازی و پروگرم برنامه از طریق محیط ***AVRStudio*** هستید به جای ***main*** عبارت ***AVRStudio 4 compatible output*** را انتخاب نمایید، فایلی با پسوند ***d90*** ساخته می شود که می توانید آنرا در ***AVRStudio*** استفاده کنید.

برای انجام تنظیمات مربوط به میکرو مورد نظر و ... مانند شکل زیر بر روی ***Option*** کلیک کنید:



در پنجره باز شده مطابق شکل میکرو ***ATXMEGA16A4*** را انتخاب نمایید.



در بخش ***memory model*** اگر نیاز به متغیرهای زیادی در برنامه دارید یا متغیرهایی با ابعاد بزرگ، به جای ***tiny*** عبارت ***small*** را انتخاب کنید(با این میکرو فقط گزینه ***small*** فعال است).

در زبانه ***system*** دو بخش خیلی مهم وجود دارد:

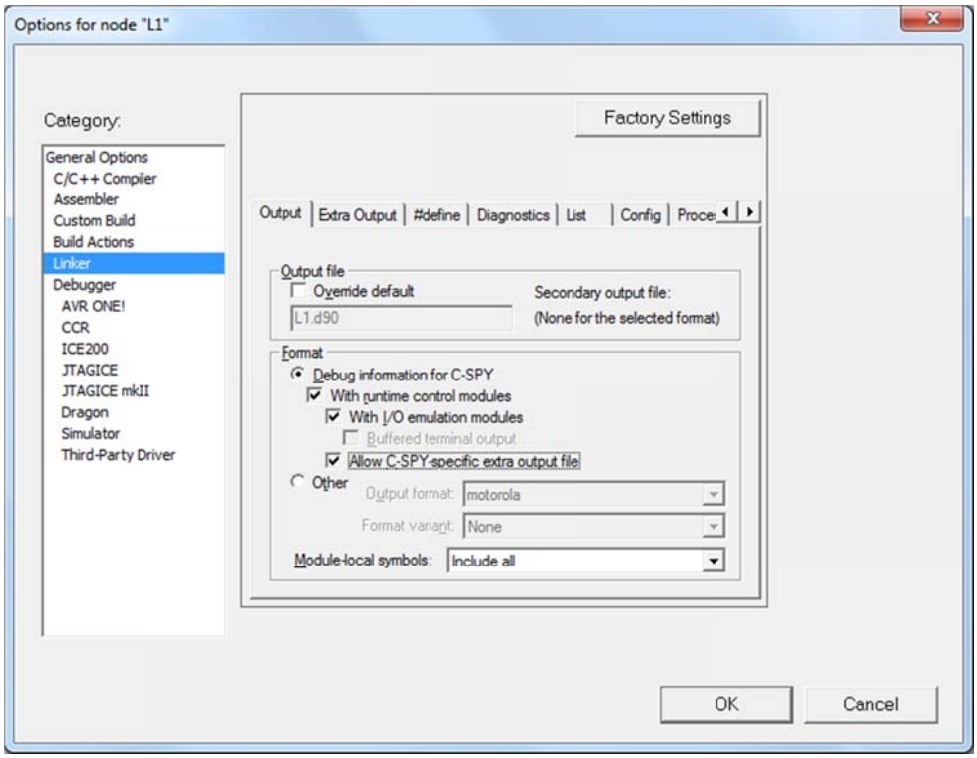
- ***Data stack(CSTACK)*** که تعیین کننده میزان حافظه برای ذخیره متغیرهای محلی، آرگومان‌های تابع و ذخیره رجیسترها به هنگام اجرای فرآیند وقفه است.
- ***Return Address stack(RSTACK)*** که تعیین کننده میزان حافظه برای ذخیره آدرس برگشتی توابع است. میزان حافظه در این بخش به صورت تعداد سطوح برگشتی تعیین می‌شود.

نکته بسیار مهم در خصوص ***CSTACK*** این است که اگر مثلا متغیرهای سراسری بزرگی مثل یک آرایه ۵۱۲ بایتی در برنامه تعریف کرده‌اید باید مراقب بود که داده‌های متغیرهای محلی با داده‌های سراسری تداخل نداشته باشد در غیر این صورت حالت‌های ناخواسته در برنامه ایجاد می شود.

در همین بخش با انتخاب گزینه ***Enable bit definitions ....*** می‌توان به تعاریف بیتی دسترسی داشت.

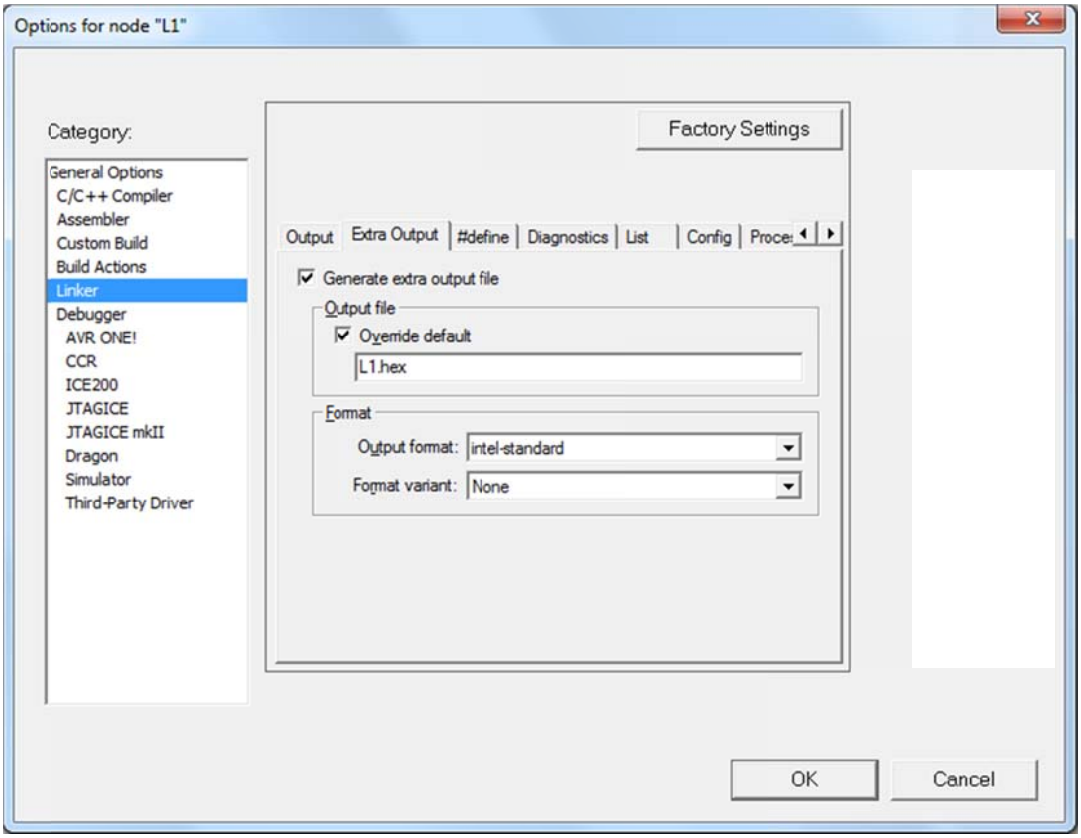
در همین دسته بندی در زبانه ***heap Configuration*** می‌توانید میزان حافظه مورد نیاز احتمالی برای توابعی مانند ***malloc*** و ***calloc*** را تعیین کنید.

در بخش دسته‌بندی بر روی گزینه ***Linker*** کلیک نمایید و مطابق شکل گزینه ***Allow C-spy-specific extra output file*** را انتخاب نمایید.



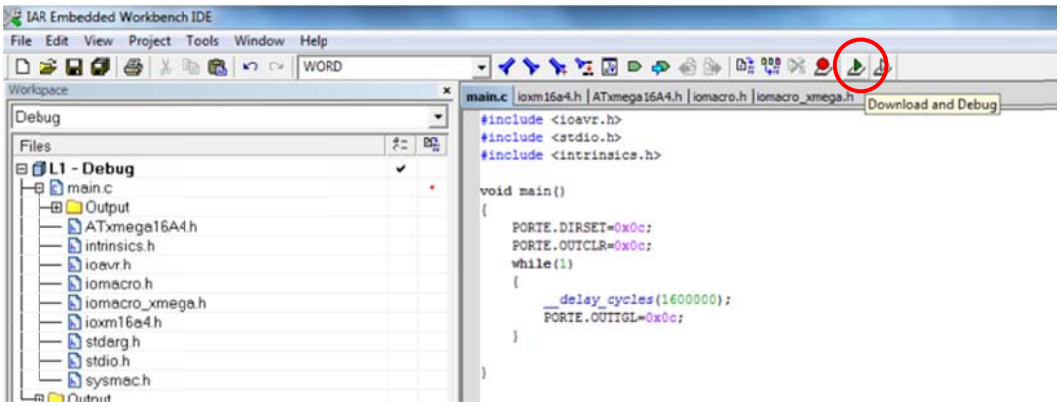
حال به زبانه ***Extra Output*** رفته و گزینه ***Generate extra output file*** را انتخاب نمایید. سپس گزینه ***override default*** را انتخاب نموده و پسوند فایل را به ***.hex*** تغییر دهید.

در بخش ***format*** گزیته ***intel-standard*** یا گزینه ***intel-extended*** را انتخاب نمایید.



کلید ***Ok*** را زده و وارد صفحه اصلی برنامه شوید.

برای کامپایلر برنامه از کلید زیر یا ***F7*** استفاده کنید:



فایل هگز برنامه در شاخه‌ای که پروژه قرار دارد در آدرس زیر ذخیره می شود:

...\Debug\Exe\L1.hex

۳. شروع کد نویسی با کامپایلر *IAR*

کتابخانه های معمولی که برای هر پروژه اغلب لازم هستند شامل موارد زیر است:

```
#include <ioavr.h>
#include <stdio.h>
#include <intrinsics.h>
```

با اضافه نمودن کتابخانه ***ioavr.h*** با توجه به تنظیمات قبلی انجام شده در خصوص میکرو مورد استفاده در پروژه، کتابخانه مربوط به میکرو به پروژه افزوده می شود.

با افزودن کتابخانه ***intrinsics.h*** نیز امکان استفاده از توابع ذاتی میکرو فعال می شود.

```
__enable_interrupt();
```

تابع بالا برای فعال نمودن وقفه سراسری استفاده می شود.

```
__disable_interrupt();
```

تابع فوق وقفه سراسری را غیر فعال می کند.

```
__delay_cycles(100);
```

تابع فوق به اندازه ۱۰۰ سیکل میکرو تاخیر ایجاد می کند. به عنوان مثال اگر اسیلاتور داخلی ۲ مگاهرتز استفاده شده باشد، عبارت فوق تاخیری معادل ۵۰ میکرو ثانیه است.

جهت کسب اطلاعات بیشتر در این خصوص به راهنمای ***IAR*** با عنوان ***IAR C/C++ compiler reference guide*** مراجعه نمایید.

۴. چند نکته برنامه‌نویسی

به دلیل اینکه در آینده احتیاج به استفاده از کتابخانه‌های آماده خواهیم داشت و برای درک بهتر این کتابخانه‌ها، برخی روش‌های برنامه‌نویسی که معمولا در کتابخانه‌هایی از این دست استفاده می شود، اشاره می‌شود. دانستن این تکنیک‌ها کمک زیادی به درک عملکرد کتابخانه می‌کند.

۴-۱. خوانایی برنامه

در برنامه‌نویسی حرفه‌ای برای افزایش خوانایی برنامه‌ها به جای استفاده مستقیم از عنوان پورت یا شماره بیت آن، از معادل آنها استفاده می‌شود.

مثلا فرض کنید ***LED*** قرمز رنگی به بیت صفر پورت ***B*** میکرو متصل شده است که با ۱ شدن پورت روشن و با صفر شدن آن خاموش می‌شود. برای خاموش و روشن کردن این ***LED*** می توان از دستور های زیر استفاده نمود:

```
PORTB|=0x01; // LED on
PORTB&=0xfe//  LED off
```

برای خوانایی بهتر می تواند کد را به صورت زیر نوشت:

```
PORTB|=0x01;//LED on
PORTB&=~(0x01);// LED off
```

عبارت ***(0x01)~*** به معنی ***NOT*** بیتی است و معادل ***(0xfe)*** است ولی نوشتن آن به این صورت خوانایی برنامه را افزایش می دهد.

باز هم بهتر:

```
#define EnRedLED() PORTB|=0x01
#define DiRedLED() PORTB&=~(0x01)
EnRedLED();//LED on
DiRedLED();// LED off
```

باز هم بهتر:

```
#define RedLED 0 // bit position
#define EnRedLED() PORTB|=(0x01<<RedLED)
#define DiRedLED() PORTB&=~(0x01<<RedLED)
EnRedLED();//LED on
DiRedLED();// LED off
```

a<<b به معنی شیفت بیتی یک بایت است. در اینجا بیت‌های بایت ***a*** به تعداد ***b*** به سمت چپ شیفت پیدا می‌کنند مثلا:

0x01<<5 ➔ 0x20

باز هم بهتر:

```
#define RedLED 0
#define LEDPORT PORTB
#define EnRedLED() LEDPORT |=(0x01<<RedLED)
#define DiRedLED() LEDPORT &=~(0x01<<RedLED)
EnRedLED();//LED on
DiRedLED();// LED off
```

تفاوت این کدها در خوانایی بیشتر و امکان اعمال تغییرات راحت‌تر در آینده است.

مثلا در کد آخر اگر پورت ***LED*** و یا بیت مربوط به آن تغییر کند فقط دو خط برنامه تغییر خواهد کرد که می توان تنظیماتی از این دست را در هدر فایل معینی در اختیار کاربر قرار داد.

گاهی پیچیدگی‌هایی که از نظر نوشتاری در کدهای حرفه ای وجود دارد از این دست بوده و هدف از آن افزایش خوانایی و آسان سازی اعمال تغییرات در آینده است.

۴-۲. تعریف نوع جدید

یک برنامه‌نویس حرفه‌ای همواره سعی می‌کند کد خود را طوری بنویسد که اگر در آینده سخت‌افزار تغییر کرد تا حد امکان با کمترین تغییراتی بتواند برنامه را برای سخت افزار جدید احیا کند.

یکی از مواردی که این مسئله به وضوح خود را نشان می دهد در اندازه متغیرهاست. مثلا در یک بستر سخت افزاری ممکن است ***int*** ۳۲ بیتی باشد در دیگری ***int*** ۱۶ بیتی باشد. اگر برنامه‌نویس با فرض اینکه ***int*** ۱۶ بیتی است برنامه را نوشته باشد اگر این نرم افزار در بستری استفاده شود که ***int*** در آن ۳۲ بیتی است، برنامه به درستی کار نخواهد کرد. برای رفع این مشکل از قابلیت تعریف نوع جدید استفاده می-شود:

typedef int INT;(فرض کنید در این حالت ***int*** ۱۶ بیتی است

حال در کل برنامه متغیرهای ***int*** به صورت زیر تعریف می شوند:

```
INT x,y,f;
```

حال اگر برنامه در بستری اجرا شود که ***int*** در آن ۳۲ بیتی است برای تبدیل تمامی متغیرهای***int*** تعریف شده به متغیرهای ۱۶ بیتی فقط نوع تعریف شده را به صورت زیر تغییر می دهیم:

typedef short int INT;(فرض کنید در این حالت ***int*** ۳۲ بیتی است

در این حالت فقط با یک تغییر کوچک در یک خط برنامه تمامی متغیر های ***int*** تعریف شده در برنامه ۱۶ بیتی خواهند شد.(کلمه کلیدی ***short*** قبل از ***int*** به معنی متغیر ***int*** ۱۶ بیتی است)

نکته

قرارداد نانوشته‌ای بین برنامه‌نویسان مرسوم است که انتهای نوع‌های جدیدی که می سازند، ***"_t"*** را قرار می‌دهد و بدین طریق به خواننده می‌رساند که این یک نوع جدید تعریف شده است(***t*** از ***typedef*** گرفته شده است).

مثال:

```
typedef unsigned long register32_t;
```

۴-۳. قابلیت‌های *define*(تعریف ماکرو*Macro*)

کلمه کلیدی ***define*** علاوه بر تعریف ثوابت کاربردهای فراوان دیگری نیز دارد که اغلب در کدنویسی حرفه‌ای از آنها استفاده می شود. در اینجا به برخی از آنها اشاره می شود:

به کاربرد ***define*** و **#** در این مثال توجه کنید:

(این مثال از کتاب ***McGraw-Hill - C - The Complete Reference, 4th Ed*** است.)

The # operator, which is generally called the *stringize* operator, turns the argument it precedes into a quoted string. For example, consider this program:

```
#include <stdio.h>

#define mkstr(s) # s

int main(void)
{
  printf(mkstr(I like C));

  return 0;
}
```

The preprocessor turns the line

```
printf(mkstr(I like C));
```

into

```
printf("I like C");
```

به کاربرد ***define*** و ***##*** در مثال زیر توجه کنید: (منبع همان)

The ## operator, called the *pasting* operator, concatenates two tokens. For example:

```
#include <stdio.h>

#define concat(a, b) a ## b

int main(void)
{
  int xy = 10;

  printf("%d", concat(x, y));

  return 0;
}
```

The preprocessor transforms

```
printf("%d", concat(x, y));
```

into

```
printf("%d", xy);
```

توضیح اینکه در مثال بالا هر حرفی جای ***a*** و ***b*** قرار گیرد معادل خواهد بود با ***ab*** که در مثال پایین عبارت ***concat(x,y)*** معادل با ***xy*** است.(با دقت مثال را نگاه کنید)

یکی از کاربردهای قابلیت بالا برای دستیابی بایت بالا و پایین یک متغیر ۱۶ بیتی یا ۲ بایتی است که در ***XMEGA*** از آن استفاده شده است.

نکته: اگر هنگام تعریف یک ***define*** احتیاج بود بقیه عبارت در خطوط پایین نوشته شود باید در انتهای هر خط از **** استفاده شود.(به مثال پایین توجه کنید)

به عنوان نمونه:

```
typedef volatile unsigned char register8_t;
typedef volatile unsigned short register16_t;
#define WORDREGISTER(regname)       \
        union { \
                register16_t regname; \
                struct { \
                        register8_t regname ## L; \
                        register8_t regname ## H; \
                }; \
```

```
}
```

در اینجا یک نوع جدید ۱۶ بیتی یا ۲ بایتی تعریف شده است که قابلیت دسترسی به هر کدام از دو بایت با نامی که برای متغیری که از این نوع تعریف شده است، وجود دارد.

برای توضیح باید گفت که در واقع در اینجا نوع جدیدی تعریف شده است به نام ***WORDREGISTER*** که برای تعریف یک متغیر از این نوع به صورت زیر عمل می شود:

```
WORDREGISTER(x);
xL=0xaa;
xH=0x55
```

در اینجا متغیر ***x*** از نوع ***WORDREGISTER*** تعریف شده و بایت پایین(با پسوند ***L*** که به انتهای نام افزوده شده است) آن با مقدار ***0xaa*** و بایت باارزش آن(با پسوند ***H*** که به انتهای نام افزوده شده است) با مقدار ***0x55*** مقدار دهی شده است.

توضیح برنامه اینکه با قرار دادن ***regname*** به صورت زیر در پرانتز

```
WORDREGISTER(regname)
```

به کامپایلر می فهمانیم که به جای ***regname*** هر نام دیگری می تواند قرار داشته باشد.

با دستور ***register16_t regname;*** کامپایلر یک متغیر دو بایتی یا ۱۶ بیتی می سازد. پس از آن یک ***struct*** به صورت زیر تعریف می کند:

```
struct { \
        register8_t regname ## L; \
        register8_t regname ## H; \
}; \
```

یعنی دو متغیر یک بایتی پشت سر هم (به عبارت ***## L*** و ***## H*** توجه کنید)

حال هر دو نوع متغیر تعریف شده یعنی هم ***register16_t*** و هم ***struct*** را در یک ***union*** قرار داده است.

همانطور که می دانید ***union*** به صورتی که از آن استفاده شده است یک فضای مشترک ۲ بایتی را به هر دو نوع متغیر تعریف شده اختصاص می دهد(***register16_t*** و ***struct***) که بنابراین دستورات زیر معادل هم هستند:

```
x=0x55aa;  ➔ xH=0x55; xL=0xaa;
```

ممکن است سوالی پیش بیاید که چه فرقی است بین دستورات زیر:

```
register16_t x;
```

و

```
WORDREGISTER(x);
```

باید گفت که از نظر کامپایلر هیچ تفاوتی ندارند ولی دستور دوم قابلیت دستیابی به بایت بالا و پایین با

قرار دادن ***L*** یا ***H*** در انتهای نام متغیر را فرآهم می کند که کمک شایانی به خوانایی برنامه می کند.

در مثال زیر برای دستیابی به بایت با ارزش بالاتر یا پائین تر باید به صورت زیر عمل کرد:

```
register16_t x;
x&=0xff00;
x|=0x00aa;
x&=0x00ff;
x|=0x5500;
```

که نتیجه این دستورات معادل است با:

```
WORDREGISTER(x);
xL=0xaa;
xH=0x55;
```

همانطور که ملاحظه می شود، خوانایی و سادگی کد به طور مشهودی افزایش یافته است.