آموزش - rtx قسمت اول

اولین نکته ای که باید بهش اشاره بشه علت بالا بودن قدرت پردازشی هسته های ARM هست. در حالی که در میکروهای آرم سرعت تجهیزات جانبی پایین هست، پس سرعت پردازشی 100 MHzبه چه دردی می خوره؟؟
جواب این است: تو یه پروژه نسبتا حرفه ای و صنعتی، مسئله انجام یکی دو تا وظیفه (task) که نیست. وقتی قرار باشه میکرو تعداد زیادی سنسور و) actuator محرک) رو کنترل کنه، مسئله کارایی خودش رو نشون میده. اینجا به قدرت پردازش بالا نیاز داریم که بتونیم همزمان همه تسک ها رو فعال نگه داریم .پروژه ای رو درنظر بگیرین که قراره از 70 تا سنسور اطلاعات بگیریم پردازش کنیم و به 30 تا actuatorفرمان بدیم. (یه پروژه صنعتی نسبتا ساده (!!میکروهای ARM بخاطر فرکانس بالا می تونن گزینه خوبی باشن. اما نوشتن همچین برنامه ای کار ساده ایه؟ توجه کنید که قرار نیست یه حلقه بنویسیم و شروع کنیم یکی یکی سنسورها رو بخونیم و اون لابه لا توابعی رو صدا کنیم و همه چی به خوبی و خوشی...
هر کدوم شرایط خاص خودشون رو دارن. مسئله زمان بندی تسک ها مطرح میشه .بعضا باید همزمان اجرا بشن و نتایج در یک زمان آماده باشه. ممکنه تسک ها رو هم تاثیر داشته باشن. مسئله وابستگی داده ها هست. اشتراک منابع. مسئله اولویت بندی. خیلی از تسک ها ممکنه مدتی کاری واسه انجام دادن نداشته باشن و یا منتظر یک رویداد باشن. با توجه به منابع محدودی که واسه وقفه داریم عملا نمیشه تو یه حلقه معمولی همچین برنامه نوشت و همه تسک ها رو کنترل کرد .اگرم اینطوری سیستم رو پیدا کنیم، مطمئنا به نصف اون کارایی که میشد برسیم نرسیدیم و این میکرو با قدرت پردازش بالا نتونسته کمک خاصی بهمون بکنه.
اینجاست که پای سیستم عامل (به مخصوص از نوع بی درنگ یا (realtime تو طراحی مون باز میشه. با وجود یک سیستم عامل می تونیم همه این تسک ها رو جدا از هم بدون هیچ نگرانی بنویسیم. سیستم عامل با توجه به طراحی ما، مدیریت اجرای تسک ها رو به عهده می گیره. اجرای همزمان تعداد زیاد تسک بدون ایجاد اختلال در عملکرد سیستم، احتیاج به فرکانس کلاک و کارایی بالا داره و اینجاست که میکروکنترلرهای ARM خودشون رو نشون میدن. ساده بگم مثل اینه که 20 تا mega16رو بذارین کنار هم (با کمی کم لطفی.(!
سیستم عامل مزایای دیگه ای هم داره. امکان تقسیم ساده وظایف در پروژه و انجام پروژه بصورت گروهی .ساده شدن عملیات خطایابی با تست تسک ها. امکان گسترش سیستم در مراحل بعد و خیلی دیگه بیشتر از این مجالش نیست!!

multitaskingیا "عملکرد چند وظیفه ای" یعنی اینکه بتونیم چند کار رو همزمان انجام بدیم. کرنل (kernel) هسته اصلی هر سیستم عامل، این امکان رو برای سیستم عامل فراهم میکنه که چند برنامه جدا از هم (task) رو با هم اجرا کنه multitasking .این امکان رو به ما میده که برنامه ها یا پروژه های پیچیده رو به چند task ساده تبدیل کنیم که مدیریت و تغییر هرکدوم شون خیلی راحتره.
پردازنده های مرسوم که ما می شناسیم،تنها قابلیت اجرای یک دستور العمل رو در هر زمان (سیکل کلاک) دارن. یعنی به ظاهر میشه با خط به خط اجرا کردن دستورات فقط یک تسک در حال اجرا باشه. اما سیستم عامل با جابجایی سریع تسک ها برای پردازنده، به ظاهر نشون میده که همه درحال اجرا هستن.

schedulerیا زمانبند قسمتی از kernel که تصمیم میگیره در هر لحظه از زمان کدوم تسک باید اجرا بشه scheduler .میتونه یه taskرو به حالت معلق در بیاره و درعوض تسک دیگه ای رو اجرا کنه .الگوریتم یه سیستم عامل) multitaskنه (realtimeاینه که به هر تسک میزان زمان پردازش برابر بده و اصطلاحا زمان پردازنده رو عادلانه (fairly)بین تسک ها تقسیم کنه.
این جابجایی تسک ها بدون اختیار هر تسک توسط scheduler انجام میشه Round robin .یکی از الگوریتم های معروف schedulerاین الگوریتم (که شکل قبل داره نمایشش میده (از این قراره که یه قطعه زمانی مشخص تعیین میشه (مثلا 10میلی ثانیه) و به هر تسک به این اندازه زمان داده میشه تا پردازنده دستوراتش رو اجرا کنه .اگه اجرای تسک تموم نشه، schedulerاون تسک رو به حالت معلق در میاره و میره سراغ تسک بعد و به همین اندازه زمان پردازش در اختیارش میذاره. این رویه تکرار میشه و تسک ها بصورت

چرخشی زمان پردازش از پردازنده میگیرن...
غیر از این، هر تسک هم میتونه خودش رو به حالت معلق در بیاره. فرض کنید یه taskبخواد تاخیر ایجاد کنه. مثلا هر 100 میلی ثانیه اجرا بشه. تو این زمان تاخیر، خود تسک اعلام میکنه که عملیاتی برای پردازش نداره و زمان پردازش پردازنده در اختیار تسک دیگه ای قرار میگیره. حالت دیگه اینه که taskبخواد با یکی از منابع یا امکانات جانبی (peripheral) پردازنده (مثل پورت سریال (کار کنه، که اگه توسط تسک دیگه ای اشغال باشه، باید خودش رو مسدود کنه و منتظر بمونه تا اون منبع آزاد بشه. حالت های دیگه ای هم هست که یه taskخودش رو مسدود میکنه. مثلا منتظر یه وقفه خارجی یا دریافت پیام از تسک دیگه ای باشه و...
تصویر زیر رو ببینید

در زمان 1 تسک 1 در حال اجراست
در زمان 2 کرنل زمان اجرا رو از تسک 1 میگیره و در عوض اجرای تسک 2 رو ادامه میده (زمان3(
تسک 2 در زمان اجرای خودش (زمان4) یکی از امکانات جانبی (peripheral) رو در احتیار خودش میگیره که با اون کار کنه.
در زمان 5 کرنل زمان پردازش تسک2 رو میگیره و تسک 3 رو ادامه میده(زمان6 .(توجه کنید که تسک هایی که به حالت معلق در اومدن، کارشون تموم نشده.
تسک 3می خواد با همون peripheral تسک 2 کار کنه. اما چون این peripheral در اختیار تسک 2 هست، تسک 3 خودش رو مسدود میکنه (زمان 7). بنابراین دوباره تسک 1 ادامه داده میشه (زمان8) و این روند تکرار میشه تا جایی که تسک 2 دست از سر اون peripheralبرداره (زمان9.(
زمان بعدی که تسک 3 اجرا بشه)زمان10) اون peripheral مشغول نیست و تسک 3 میتونه باهاش کار کنه. همونطور که می بینید تسک 3 به اندازه تسک های دیگه زمان اجرا از کرنل گرفته...

RTOS : Real Time Operating System

سیستم های realtime برای پاسخ به رویدادهایی طراحی میشن که این رویدادها برای دریافت پاسخ، اصطلاحا یک ضرب العجل (deadline)دارن. یعنی تو زمان مشخصی حتما باید بهشون پاسخ داده بشه .برای این سیستم ها زمان انجام عمل مورد نظر دقیقا قابل محاسبه هست و نباید از بازه ای که محاسبه میشه تجاوز کنه.
بعد از multitasking خصوصیت بعدی یک سیستم عامل، realtimeبودن. اساس کار schedulerسیستم عامل realtime مشابه multitaskingاما با هدف متفاوت. هدف schedulerبلادرنگ اینه که deadlineرو درنظر بگیره و تا قبل از اتمام فرصت مورد نظر برای یک تسک، عملیات تسک رو کامل کنه.
اینکار چطور ممکن میشه؟
خیلی ساده! برنامه نویس باید برای هر تسک یک اولویت (Priority) تعیین کنه scheduler .همیشه زمان پردازش رو به تسک با اولویت بالاتر میده و تا وقتی تسک با اولویت بالاتر آماده برای اجرا باشه، تسک اولویت پایین تر هیچ زمان پردازشی نمیگیره. در صورتی هم که چند تسک با اولویت برابر آماده اجرا باشن، زمان پردازش بینشون تقسیم میشه.
وقتی چندتا تسک با اولویت برابر تعریف میکنیم. زمان پردازش بینشون تقسیم میشه و سرعت اجرای هر کدوم نسبت به زمانی که به تنهایی اجرا بشن کم میشه .فرض کنید تسکی داریم که 500 میلی ثانیه طول میکشه اجرا بشه (به تنهایی) و تو پروژه ای که داریم هم حتما باید 500 میلی ثانیه باشه نه بیشتر .بنابراین به این تسک نسبت به بقیه تسک ها یک اولویت بالاتر میدیم و اینطوری مطمئنیم که در زمان اجرای این تسک، هیچ پردازش دیگه ای نداریم و تو همون 500 میلی ثانیه انجام میشه .

آموزش - rtx قسمت دوم

از سال 2005 کهARM ، شرکت keil رو خرید، تمرکز نرم افزارهای keil رو محصولات ARM شد.
تعدادی از محصولات: keil
) uVision IDEمیکرو ویژن) محیط توسعه و دیباگر و شبیه ساز برای محصولات مبتنی بر پردازنده های آرم هست.
ARM Compilation Toolsشامل کامپایلر و اسمبلر و لینکر. (معمولا با(uVision
RL-ARMکتابخونه های) realtime سیستم عاملRTX ، CAN، TCP/IP، FLASHو(..
MDK-ARMمجموعه همه موارد بالا میشه
برای اینکه بتونین از RTX و کلا کتابخونه بلادرنگ کیل (RL-ARM)استفاده کنین باید لیسانس MDK-ARM Professionalرو داشته باشید که کرک رو من پیوست کردم(اگه آنتی ویروستون گفت ویروسه بزنین تو دهنش غیرفعالش کنید :دی من خودم الان اس ات فول آپدیت ویروس نمی شناستش ولی رو یه سیستم دیگه همین اس ات آپدیت نشده میگه ویروسه(!!
اون طور که من فهمیدم فک کنم از نسخه 4.16 به بعد دیگه کلا RL-ARM با MDK-ARM Professional یکی شد

RTXدر دو ورژن پیاده سازی شده:
ورژنARM7/ARM9
ورژنCortex M
Cortex Mها امکاناتی برای RTOS درنظر گرفتن. بخاطر همین سیستم عامل با عملکرد بهتری دارن. اما از نظر نحوه کارکرد برای ما تقریبا فرقی نمیکنه.
برای راه اندازی RTX در یک پروژه مراحل زیر رو دنبال کنید:

پروژه جدید ایجاد کنید. از منوی) Project – Options باید قبلش target پروژه رو انتخاب کنید) یا رو پنجره Project رو target راست کلیک و options رو انتخاب کنید. تو پنجره باز شده در تب target قسمت operating system ، گزینه RTX Kernel رو انخاب کنید.
فایل پیکربندی (config) سیستم عامل رو با توجه به میکروکنترلر از پوشه Keil/ARM/Startupتو پوشه پروژه کپی و به پروژه keilاضافه کنید. مثلا برای سری LPC23xx فایل RTX_Conf_LPC23xx.c در پوشه Keil/ARM/Startup/Philips .بعدا در مورد تنظیمات این فایل توضیح میدم.
تنظیمات مربوط به وقفه های نرم افزاری تو فایل اسمبلی startup باید تغییر کنه. (همون فایل اسمبلی که خود keil به پروژه اضافه میکنه. مثلا LPC23xx.s برای فیلیپس و.(...
خط SWI_Handler B SWI_Handler رو تو این فایل حذف کنید(اولش ; بذارید (و بجاش این دستور رو اضافه کنید:
IMPORT SWI_Handler
اول این دستور باید به tab یا فاصله بزنید. خلاصه قسمتی از کد که اینطوریه:
کد:

```
Undef_Handler   B       Undef_Handler
SWI_Handler     B       SWI_Handler
PAbt_Handler    B       PAbt_Handler
DAbt_Handler    B       DAbt_Handler
IRQ_Handler     B       IRQ_Handler
FIQ_Handler     B       FIQ_Handler
```

باید به این تغییر داده بشه(کد با ; کامنت شده: (

کد:

```
                IMPORT  SWI_Handler
Undef_Handler   B       Undef_Handler
;SWI_Handler    B       SWI_Handler
PAbt_Handler    B       PAbt_Handler
DAbt_Handler    B       DAbt_Handler
IRQ_Handler     B       IRQ_Handler
FIQ_Handler     B       FIQ_Handler
```

علت اینکار: توی آرم های 7 و 9 سیستم برای صدا کردن RTOS از دستورات وقفه نرم افزاری (SWI)استفاده می کنه. وقتی startup اولیه توسط کیل به پروژه اضافه میشه بردار این وقفه ها میره توی یه حلقه بسته واسه اینکار واسه سیستم عامل مشکل ایجاد میکنه.
این به این علته که یه بخشی از RTX توی مد)privileged supervisor دسترسی بالا) اجرا میشه و با SWI کار میکنه واسه همین ما باید اون خط رو حذف کنیم و خط گفته شده رو جایگزین کنیم .**البته این کار برای هسته های Cortex نیازی نیست.**
ضمنا برای استفاده از سیستم عامل باید فایل rtl.h رو هم تو فایل main پروژه include کنید.
این یه مثال ساده برای:LPC2378

کد:

```
#include <LPC23xx.h>
#include <RTL.h>

__task void task1(void);
__task void task2(void);

int main()
{
   IO0DIR = 0x00000003;

   os_sys_init(task1);

   while (1) {}  // ino nezarinam farghi nemikone chon asan barname be inja
nemirese !
   return 0;
}

__task void task1(void)
{
   int i = 0;
   os_tsk_create(task2, 1);
   while (1)
   {
      os_dly_wait(50);
      if (i == 0)
      {
         IO0SET = 0x00000001;
         i = 1;
      }
      else
      {
         IO0CLR = 0x00000001;
```

```
         i = 0;
      }
   }
}

__task void task2(void)
{
   int i = 0;
   while (1)
   {
      os_dly_wait(25);
      if (i == 0)
      {
         IO0SET = 0x00000002;
         i = 1;
      }
      else
      {
         IO0CLR = 0x00000002;
         i = 0;
      }
   }
}
```

تو این برنامه دو تا تسک تعریف شده. تو هر تسک یکی از پایه های میکرو بطور جدا (با تاخیر متفاوت) روشن و خاموش میشه.

آموزش - rtx قسمت سوم

تنظیمات فایل کانفیگ:

فایل config که به پروژه اضافه کردین رو تو محیط uVision باز کنید. پایین ادیتور روی Configuration Wizard کلیک کنید. اینطوری می تونین بدون درگیر شدن با کدها، تنظیمات رو اتجام بدین.

در بخش Task configuration ، جلوی Number of concurrent running tasks باید تعداد تسک ها مشخص بشه.
قسمت Task Stack size مربوط به حافظه استک اختصاص داده شده به هر تسک هست و هر چه برنامه بزرگتر شه باید این مقدارو بیشتر کنید(با توجه به پروژتون و در واقع باید تست کنید بهینه ترین حالت رو پیدا کنید <-- اگه کمتر از نیاز برنامه باشه RTX هنگ می کنه) اگر در مورد خود Stack سوالی هست بپرسین پیشتر توضیح بدم.
در بخش Tick Timer Configuration تنظیمات زمانی انجام میشه. کرنل نیاز به یک سخت افزار تایمر داره که دائم طی بازه زمانی مشخصی توسط اون تایمر صدا زده بشه تا مدیریت تسک ها و اعمال سیستم عامل رو انجام بده. تایمر با وقفه دادن مکرر در فاصله زمانی مشخصی که تعیین می کنیم، اصطلاحا Tick رو ایجاد میکنه و با هر Tick تابع مدیریت سیستم صدا زده میشه. زمان مشخص شده برای Tickخیلی مهمه. تمام توابع سیستم عاملی که به نوعی با زمان در ارتباط هستن، با ضریبی از زمان Tick کار میکنن.
در قسمت Hardware timer می تونین یکی از تایمرها رو انتخاب کنید. جلوی Timer clock valueباید فرکانس کلاکی که به تایمر وارد میشه رو تعیین کنید Timer tick value .مدت زمانTick رو تعیین میکنه. مثلا 10000 میکرو ثانیه (10 میلی ثانیه.(
در بخش System configuration اگه گزینه Round Robin task switching انتخاب شده باشه، طبق الگوریتم Round

Robin، تسک ها بصورت چرخشی سوییچ میشن .جلوی Round Robin timout باید مشخص کنید که به هر تسک به اندازه چند Tickفرصت پردازش داده بشه. مثلا اگه 5 باشه، و زمان هر Tick هم 10میلی ثانیه باشه، به هر تسک در هر بار 50 میلی ثانیه فرصت پردازش داده خواهد شد. این همون قطعه زمانی که تو پست مربوط به multitask و scheduler گفتم. همونطور که می بینید schedulerیکی از همون قسمت هایی که با Tick کار میکنه.

هرچقدر زمان Tick کمتر باشه، سیستم عامل با دقت و سرعت بیشتری کار میکنه .اما به همون اندازه که زمان Tick رو کم می کنید، توابع سیستم عاملی بیشتر صدا زده میشن، میزان بیشتری از زمان پردازش رو سیستم عامل میگیره. تو هلپ کیل پیشنهاد کرده زمان Tick بین 1 تا 100 میلی ثانیه باشه .بستگی به سخت افزار و میکرویی که استفاده میکنین داره...

اگه Round Robin رو غیر فعال کنین، جابجایی تسک ها بصورت خودکار (در اون بازه زمانی مشخص) انجام نمیشه و فقط دستورات مدیریتی که تو تسک ها می نویسین روند اجرا رو مشخص میکنه.

آموزش - rtx قسمت چهارم

از این قسمت وارد برنامه نویسی می شیم

با یه مثال شروع می کنم:

کد:

```
#include <LPC23xx.h>
#include <RTL.h>

OS_TID   tsk_ID1, tsk_ID2;

__task void task_init(void);
__task void task1(void);
__task void task2(void);

void blink1(void);
void blink2(void);

int main()
{
   os_sys_init(task_init);

   while(1) {}
   return 0;
}

__task void task_init(void)
{
   //initialization ports for led
   IO0DIR = 0x00000003;

   tsk_ID1 = os_tsk_create(task1, 1);   // priority:1
   tsk_ID2 = os_tsk_create(task2, 1);   // priority:1

   os_tsk_delete_self();    // necessary
}
```

```
__task void task1(void)
{
   int i = 0;
   while(1)
   {
      for (i = 0; i < 0x7ffff; i++); //process

      blink1(); // roshan ya khamush kardan led1
   }
}

__task void task2(void)
{
   int i = 0;
   while(1)
   {
      for (i = 0; i < 0x8ffff; i++); //process

      blink2(); // roshan ya khamush kardan led2
   }
}
```

برای تعریف تسک، از تابعی با مقدار برگشتی از نوع voidو مشخصه task__ استفاده می کنیم. هر تسک باید به شکل تابعی با حلقه بی پابان باشه.

کد:

```
__task void task1(void);
__task void task1(void)
                        {
   //initialization
   while(1){
      //task process
   }
}
```

تابع os_sys_initسیستم عامل رو پس از انجام مقداردهی های اولیه، فعال میکنه .این تابع یک آرگومان ورودی برای مشخص کردن اولین تسک برای اجرا داره. یعنی با اجرای این تابع، سیستم عامل فقط یک تسک رو میشناسه. این تسک بطور خودکار اولویت 1 (کمترین اولویت) رو میگیره. دستور بعد از os_sys_initاجرا نمیشه. درواقع بعد از این تابع، فقط تسک ها هستن که اجرا میشن و برنامه به خط بعد از os_sys_init برنمی گرده. پس برای معرفی بقیه تسک ها به سیستم عامل، باید تو تسک اول اقدام کنیم. معرفی تسک ها توسط تابع os_tsk_createانجام میشه. آرگومان اول تابع، نام تسک و آرگومان دوم اولویت که میتونه بین 1تا 254 باشه. اعداد بزرگتر اولویت بیشتری دارن .اولویت 0 و 255 رزرو شده هستن و نباید استفاده کنیم. خروجی این تابع یک شماره مشخصه از نوع OS_TID برای تسک ایجاد شده که می تونیم ذخیره کنیم .برای بعضی از توابع سیستم عامل باید از شماره تسک برای مشخص کردن تسک استفاده کنیم. بد نیست یه نگاهی به پروتوتایپ این دو تابع بندازید:

کد:

```
void os_sys_init (void (*task)(void) );  /* Task to start */
```

```
OS_TID os_tsk_create (
    void (*task)(void),    /* Task to create */
    U8    priority );      /* Task priority (1-254) */
```

معمولا تسک اول برای مقداردهی اولیه سیستم و معرفی تسک ها برنامه نویسی میشه .همونطور که می بینید تسکی به نام task_init برای این کار تعریف کردم .اگه دقت کنید این تسک به شکل حلقه بی نهایت نیست و کارش تموم میشه و به پایان تابع میرسه. برای همچین تسک هایی و هر تسکی که عملیاتش تحت شرایطی تموم میشه حتما باید در پایان از تابع os_tsk_delete_self استفاده کنید تا سیستم عامل این تسک رو حذف کنه. اگه این کارو نکنید سیستم هنگ میکنه.
تابع دیگه ای هست به نام . os_tsk_delete این تابع می تونه با گرفتن شماره مشخصه هرتسک (از نوع OS_TID) اونو از سیستم حذف کنه (همین یک پارامتر رو داره). یعنی میشه از تو یه تسک، تسک دیگه ای رو حذف کرد .ID هر تسک، خروجی تابع create اونه.
تو کدی که می بینید tsk_ID1 و tsk_ID2 شماره مشخصه task1 و task2 هستن.

آموزش - rtx قسمت پنجم

خب برنامه پست قبل رو ادامه میدیم. برین تو فایل config و تیک گزینه round robin task switchingرو حذف کنید. پروژه رو کامپایل کنید و نتیجه رو ببینین.
فقط تسک اول اجرا میشه (led) !دوم چشمک نمیزنه .(سیستم عامل دیگه به شکل round robin عمل نمیکنه. می دونیم که با round robin بعد از هر قطعه زمانی مشخص شده سوییچ تسک انجام میشه. و چون این قطعه زمانی خیلی کوتاه هست، سیستم به ظاهر همه تسک ها رو موازی و همزمان اجرا میکنه.
بدون round robin باید چیکار کرد؟
Cooperative Multitasking
میشه "مشارکتی" ترجمه کرد.
این همون چیزیه که سیستم عامل های اولیه به این شکل عمل میکردن. روند سوییچ تسک ها توسط خود تسک ها مشخص میشه. یعنی خود تسک باید اعلام کنه که فرصت رو در اختیار تسک دیگه قرار بده. دیگه تو برنامه ای که نوشتیم وسط اون حلقه forکه مثلا عملیات پردازش رو شبیه سازی میکرد، سوییچ تسک انجام نمیشه .باید بین عملیات تسک به سیستم عامل اعلام کنید زمان پردازش رو در اختیار تسک دیگه ای قرار بده. این کار توسط دستور os_tsk_pass انجام میشه .من هر کدوم از اون حلقه های for رو به چند قسمت تقسیم کردم و وسط اونا به سیستم عامل اعلام کردم که بره سراغ تسک بعد.
کد:

```
__task void task1(void)
{
   int i = 0;
   while(1){
      for (i = 0; i < 0x4ffff; i++);   //first portion of process
      os_tsk_pass();
      for (i = 0; i < 0x4ffff; i++);   //second portion of process
      os_tsk_pass();
      for (i = 0; i < 0x4ffff; i++);   //third portion of process
      os_tsk_pass();
      for (i = 0; i < 0x4ffff; i++);   //fourth portion of process
```

```
      blink1();
   }
}

__task void task2(void)
{
   int i = 0;
   while(1){
      for (i = 0; i < 0x5ffff; i++);   //first portion of process
      os_tsk_pass();
      for (i = 0; i < 0x5ffff; i++);   //second portion of process
      os_tsk_pass();
      for (i = 0; i < 0x5ffff; i++);   //third portion of process
      os_tsk_pass();
      for (i = 0; i < 0x5ffff; i++);   //fourth portion of process

      blink2();
   }
}
```

ما همیشه از round robin استفاده می کنیم. اما در این حالت هم میشه این تابع رو صدا کرد و سوییچ تسک انجام داد (قبل از اینکه قطعه زمانی تسک تموم بشه). اگه این تابع رو صدا بزنین و تسک بعدی اولویت کمتری داشته باشه (یعنی تسک هم اولویت با تسک جاری آماده اجرا نباشه)، سوییچ تسک انجام نمیشه. این تابع همیشه بین تسک های هم اولویت سوییچ میکنه. خب اگه تسک بعد اولویت بیشتری داشته باشه چی؟؟!

این سیستم عامل یه سیستم عامل بلا درنگه .(realtime) یعنی هیچ وقت امکان نداره تسک با اولویت بیشتری آماده اجرا باشه و تسک با اولویت کمتری درحال اجرا باشه.

Preemptive multitasking

روش cooperative دیگه جواب گوی سیستم های پیچیده و مخصوصا realtime نبود .این شد که سیستم عامل ها preemptive شدن .یعنی در حالت عادی فقط منتظر دستوری مثل os_tsk_pass نمیشن. و همینطور در حالت round robin منتظر اینتراپت تایمر schedulerنمیشن (که بعد از اون قطعه زمانی سوییچ تسک انجام بشه). در روش Preemptive سیستم عامل به رویدادها پاسخ میده. یعنی هروقت رویدادی اتفاق افتاد که تسک با اولویت بالاتری آماده اجرا شد، بی معطلی سوییچ انجام میشه و اون تسک اجرا میشه .البته این فقط برای تسک های اولویت بالاتر نیست. تسک هایی که با اولویت برابر هم که تو صف اجرای تسک ها هستن اگه بعد از رویدادی آماده اجرا بشن ممکنه درست بعد از تسک جاری زمان پردازش بگیرن. توجه کنید که preemptive کامل شده .cooperative یعنی در این حالت هم os_tsk_pass و round robin عمل میکنن. این سیستم عامل از Preemptiveتبعیت میکنه.

آموزش - rtx قسمت ششم

این پست در مورد توابع تاخیر سیستم عامله. موقع استفاده از سیستم عامل باید از این توابع برای ایجاد تاخیر ایجاد کنین. این بحث در ادامه بحث روند های اجرای تسک ها و .preemptive multitasking البته نمیشه گفت توابع تاخیر دقیقا به سیستم عامل های preemptiveمربوط میشه.

برای اینکه دقیقا زمان بندی رو ببینین باید برنامه رو پروگرم کنین و در عمل ببینین. در این مورد شبیه ساز اصلا دقیق عمل نمیکنه...

تابع os_dly_waitسیستم عامل، یک ورودی داره که تعداد tick برای ایجاد یه تاخیر رو مشخص می کنه. با مثال توضیح بدم راحتره.
برنامه زیر رو ببینین:
کد:

```
#include <LPC23xx.h>
#include <RTL.h>

OS_TID   tsk_ID1, tsk_ID2, tsk_ID3;

__task void task_init(void);
__task void task1(void);
__task void task2(void);
__task void task3(void);

int main()
{
   os_sys_init(task_init);

   while(1) {}
   return 0;
}

__task void task_init(void)
{
   /*
    * initialization ports for led
    */

   tsk_ID1 = os_tsk_create(task1, 1);   // priority:1
   tsk_ID2 = os_tsk_create(task2, 1);   // priority:1
   tsk_ID3 = os_tsk_create(task3, 2);   // priority:2

   os_tsk_delete_self();      // necessary
}

__task void task1(void)
{
   int i = 0;
   while (1)
   {
      for (i = 0; i < 0x5ffff; i++);   //first portion of process
      for (i = 0; i < 0x5ffff; i++);   //second portion of process

      // Blink //
   }
}

__task void task2(void)
{
   int i = 0;
   while (1)
   {
      for (i = 0; i < 0x6ffff; i++);   //first portion of process
      for (i = 0; i < 0x6ffff; i++);   //second portion of process
```

```
      // Blink //
   }
}

__task void task3(void)
{
   while (1)
   {
      os_dly_wait(50);
      // Blink //
   }
}
```

Task3رو اضافه کردم که اولویتش 2 هست. یعنی از task1و task2 اولویت بیشتری داره. تو این تسک فقط عمل چشمک زدن انجام میشه. با شروع به کار سیستم عامل، task3که بالاترین اولویت رو داره اجرا میشه. در ابتدای این تسک تابع os_dly_wait با ورودی 50 فراخوانی میشه. یعنی می خوام به اندازه 50 تیک تو اجرای این تسک تاخیر ایجاد کنم. با توجه به اینکه زمان هر تیک رو 10 میلی ثانیه مشخص کردم، این تاخیر میشه 500 میلی ثانیه. خب بعد از اجرای این تابع، سیستم عامل این تسک رو به حالت معلق در میاره تا زمان تاخیرش تموم بشه و برگرده تسک رو ادامه بده. نکته مهم اینه که تو این زمان وقت پردازنده تلف نمیشه. سیستم عامل میره تسک های با اولویت بیشتر رو اجرا میکنه. یعنی تو این زمان task1 و task2 که اولویت برابری دارن بطور همزمان اجرا میشن. به محض اینکه زمان تاخیر برای task3 تموم شد، ادامه این تسک بدون مزاحمت هیچ تسک دیگه ای اجرا میشه. دستور بعدش اجرا میشه و دوباره به wait میرسه... اگه دقت کنین عمل چشمک زدن توسط task3 خیلی منظم انجام میشه. این بخاطر اینه که اولویت بالاتری داره و به محض اینکه زمان تاخیرش به اتمام برسه، فورا اجرا میشه.

task3رو تغییر بدین:

کد:

```
__task void task3(void)
{
   int i = 0;
   while (1)
   {
      os_dly_wait(50);
      for (i = 0; i < 0x9ffff; i++);   //process
      // Blink //
   }
}
```

واضحه که بخاطر حلقه for زمان تاخیر بیشتر میشه .فرض کنید می خوایم تسکی تعریف کنیم که به طور متناوب طی زمان دقیقی اجرا بشه. تو بحث های کنترلی همچین مواردی زیاد پیش میاد که عملی در زمان های دقیقی اجرا بشه. همین task3با وجود حلقه .for می خوایم هربار که حلقه تسک (while)دوباره می خواد اجرا بشه 500 میلی ثانیه از شروع اجرای دفعه قبل زمان گذشته باشه .یعنی عملی می خوایم که هر 500 میلی ثانیه اجرا بشه (صرف نظر از اینکه اجرای خودش چقدر طول میکشه). ما نمی دونیم پردازش تو تسک)حلقه (forچقدر طول میکشه. بستگی به شرایط داره و موقعیت داره. ممکنه یه بار 200 میلی ثانیه طول بکشه یه بار دیگه 100 میلی ثانیه. اگه مطمئن بودیم که مثلا 100 میلی ثانیه طول میکشه، تاخیر اول تسک رو 40 (400 میلی ثانیه (تعریف میکردیم .توجه کنید که تاخیر این حلقه for رو میشه حساب کرد. اما اینو من همینطوری نوشتم. در عمل کد واقعی که برای تسک می نویسین با توجه به شرطها و ورودی ها ممکنه به شکل های مختلف اجرا بشه. ضمنا اگه تسک های هم اولویت هم داشته باشیم دیگه بدتر ...

خب راه حل:
توی سیستم عامل ها (مخصوصا realtime)روش بسیار مهمی دیگه ای برای ایجاد تاخیر وجود داره. در این روش هر بار که به تابع تاخیر رسیده بشه، مدت زمان گذشته شده از فراخوانی بار قبل تابع محاسبه میشه و از تاخیر جدید کم میشه. مثلا اگه زمان تناوب رو 500 میلی ثانیه تعریف کرده باشیم و 100 میلی ثانیه از تاخیر بار قبل گذشته باشه، با رسیدن به تابع تاخیر، 400 میلی ثانیه تاخیر ایجاد میشه.
کد تسک باید اینطوری نوشته بشه:
کد:

```
__task void task3(void)
{
   int i = 0;
   os_itv_set(50);
   while (1)
   {
      os_itv_wait();
      for (i = 0; i < 0x9ffff; i++);   //process
      // Blink //
   }
}
```

تابع os_itv_set زمان تناوب رو مشخص میکنه و os_itv_wait هم تاخیر تناوبی رو ایجاد میکنه.
حالا اگه اجرای تسک بیشتر از زمان تاخیر اجرای مجدد تسک طول بکشه چه اتفاقی میوفته؟؟
اولا که برنامه نباید اینطوری نوشته شه!!! منطقی نیست دیگه!!
اما خب اگه اشکالی پیش بیاد و اینطوری بشه، تسک دوباره اجرا نمیشه .تابع تسک فقط یک بار اجرا میشه. برنامه طبق روال عادی اجرا میشه و بعد از اینکه به تابع تاخیر رسید، هیچ تاخیری ایجاد نمیشه .مثل یه دستوری عادی ازش رد میشه.

آموزش - rtx قسمت هفتم

رویدادها

این پست به تعیین رویدادها و اینکه چطوری تسک ها می تونن اجرای هم رو کنترل کنند، اختصاص داره(preemptive) .
طبق معمول با مثال توضیح میدم. فرض کنید تسکی داریم که برای اجرا شدنش باید چندتا تسک دیگه کامل یا تا حدودی اجرا شده باشن. مثلا به داده های نیاز داره که توسط چندتا تسک دیگه تامین میشه. یا اینکه زمینه اجرای این تسک رو، چندتا تسک دیگه آماده کنن. به دلیل اینکه ممکنه شرایط مختلفی پیش بیاد ما نمی تونیم دقیق مشخص کنیم که اون تسک ها کی به مرحله ای میرسن که زمینه برای اجرای تسک مورد نظر آماده بشه.
هر تسک یک پرچم 16 (flag) بیتی برای تعیین رویدادها داره. هر بیت یک رویداد. فرض کنید task3 به مرحله ای از اجرا که رسید که باید منتظر 2 تا رویداد بمونه بعد ادامه بده. این تسک با اشاره به 2 تا از بیت های پرچمش به سیستم عامل اعلام میکنه که منتظره این دو پرچم فعال بشن و بعد ادامه بده. این 2 تا رویداد توسط دو تسک دیگه به نام های task1 و task2 تامین میشن. پس تو هر کدوم از تسک های 1 و 2 باید در قسمت مورد نظر این دو پرچم فعال شده و رویداد خودشون رو اعلام کنن. به محض اینکه هر دو پرچم فعال شدن، task3روند اجرای خودش رو ادامه میده.
کد:

```
#include <LPC23xx.h>
```

```
#include <RTL.h>

OS_TID   tsk_ID1, tsk_ID2, tsk_ID3;

__task void task_init(void);
__task void task1(void);
__task void task2(void);
__task void task3(void);

int main()
{
   os_sys_init(task_init);

   while(1) {}
   return 0;
}

__task void task_init(void)
{
   /*
    * initialization ports for led
   **/

   tsk_ID1 = os_tsk_create(task1, 1);   //priority:1
   tsk_ID2 = os_tsk_create(task2, 1);   //priority:1
   tsk_ID3 = os_tsk_create(task3, 2);   //priority:2

   os_tsk_delete_self();      //necessary
}

__task void task1(void)
{
   int i = 0;
   while(1)
   {
      for (i = 0; i < 0x5ffff; i++);   //process
      os_evt_set(0x1, tsk_ID3);
   }
}

__task void task2(void)
{
   int i = 0;
   while(1)
   {
      for (i = 0; i < 0x6ffff; i++);   //process
      os_evt_set(0x2, tsk_ID3);
   }
}


__task void task3(void)
{
   while(1)
   {
      os_evt_wait_and(0x3, 0xffff);   //wait for bit 1 and 2 forever
```

```
      // Blink //
   }
}
```

در ابتدای اجرای خودش توسط تابع os_evt_wait_and اعلام میکنه که منتظر بیت شماره 0 و 1 پرچم میمونه. پارامتر اول الگوی بیت های مورد نظر تابع رو مشخص میکنه (حداکثر 16 بیت). 0 x3یعنی دو بیت اول. پارامتر دوم timeout رو مشخص میکنه. یعنی اندازه زمانی که تسک منتظر اتفاق افتادن این رویدادها باشه. بعد از گذشت این زمان، بدون توجه به رویدادها تسک ادامه پیدا میکنه .این پارامتر هم 16 بیتی به غیر از 0 .xffffمقدار 0 xffffیعنی بی نهایت)تابع اونقدر منتظر میمونه تا رویدادها صورت بگیرن). خب بعد از اجرای این تابع، تسک به حالت معلق درمیاد task1 .و task2ادامه داده میشن تا هر کدوم به دستور os_evt_set برسن. اینجا همونجایی که توسط این تسک ها اعلام رویداد میشه. پارامتر دوم این تابع شماره ID تسکی که می خوایم بیت های پرچمش رو یک کنیم. پارامتر اول بیت های مورد نظر برای فعال شدن رو مشخص میکنه .همونطور می بینید task1 بیت شماره 0و task2 بیت شماره 1 از پرچم task3 رو فعال میکنه. بعد از فعال شدن هر دو بیت، task3دوباره فعال میشه، تابع wait تسک3 همه بیت های پرچم رو صفر میکنه (برای دفعه بعد) و از تابع خارج میشه تا تسک ادامه پیدا کنه.
تابع wait رویداد یک مقدار برگشتی هم داره (برای وقتی که timeout غیر از 0xffffباشه) اگه مقدار برگشتی OS_R_EVT باشه یعنی با فعال شدن پرچم ها)رویداد) از تابع گذشته شده و اگه OS_R_TMO باشه یعنی زمان timeout تموم و از تابع خارج شده. با توجه به این مقدار برگشتی میشه تصمیم گیری مناسب کرد.

تابع مشابه دیگه ای به نام os_evt_wait_or وجود داره که فقط منتظر یکی از بیت های مشخص شده پرچم میمونه. یعنی لازم نیست همه بیت های مشخص شده برای پرچم یک بشن تا از تابع گذشته بشه. فقط یکی از بیت ها کافیه. برای اینکه بفهمیم کدوم یکی از بیت ها (کدوم رویداد (باعث فعال شدن تسک شده، از تابع os_evt_getاستفاده میکنیم. خروجی این تابع مقدار 16 بیتی که بیت رویداد مورد نظر رو یک کرده.

یه نکته مهم اینکه تو توابع وقفه) irq نه (fiq از تابع isr_evt_set برای فعال کردن پرچم ها استفاده میشه (نه تابع(os_evt_set

می خوام نقش این توابع تعیین رویداد رو بیشتر درک کنید
فکر کتید اگه این توابع نبودن باید چیکار میکردین؟؟
...
...
میشه چندتا متغیر سراسری تعریف کرد. مقدارشون صفر بشه. تو تابعی که قراره waitبخوره، این متغیر ها رو تو حلقه چک میکنیم. تا موقعی که صفر هستن از حلقه گذشته نشه. مقدار این متغیرها هم تو تسک هایی که رویداد رو تولید میکنن یک میشه...
خب بنظرتون اشکالش چیه؟
اگه task3 که می خواد منتظر رویداد باشه، اولویتش از تسک های دیگه بیشتر باشه (مثل مثال بالا)، تا موقعی که داره دستوراتش (حلقه چک کردن متغیر (اجرا میشه، اصلا اجازه اجرا شدن تسک های دیگه رو نمیده!! (یعنی برنامه همین جا قفل میشه(
اگه این تسک با تسک های دیگه هم اولویت باشه، اجازه اجرا به اونا هم داده میشه. اما خب زمان پردازش به task3 هم داده میشه. یعنی پردازش بیخود. تلف کردن وقت پردازنده .در حالی که اگه از تابع تعیین رویداد سیستم عامل استفاده کنیم، دیگه سراغ task3نمیاد(تسک به حالت معلق درمیاد) تا وقتی که رویدادها صورت بگیرن.