دانشگاه اصفهان

دانشکدهٔ فنی و مهندسی

درس: سیستمهای بلادرنگ پیشرفته

موضوع:

پیاده‌سازی یک سیستم‌عامل بلادرنگ برای میکروکنترلرها

استاد درس:

دکتر کمال جمشیدی

علی بهلولی

فروردین ماه 1386

فهرست مطالب

1 چکیده 4
2 مقدمه 5
3 میکروکنترلرهای AVR 7
3-1 ثباتهای AVR 7
3-2 حافظه داده و ثباتهای AVR 8
3-3 خانواده های محصولات AVR 9
3-3-1 Tiny AVR 9
3-3-2 Mega AVR 9
3-3-3 LCD AVR 10
4 نرم‌افزار 10
5 طرح پیشنهادی 11
6 کدهای سیستم‌عامل بلادرنگ 11
6-1 متغیرهای عمومی 12
6-1-1 os_central_table 12
6-1-2 ثابتها 13
6-1-3 نوع داده‌های تعریف شده 14
6-2 Tasks ها 14
6-2-1 مقدمه 14
6-2-2 بلوک کنترل Task (TCB) 14
6-2-3 ایجاد task 16
6-2-4 استک 17
6-2-5 Null task 17
6-2-6 یک نمونه برنامه 17
6-3 زمانبندی taskها 19
6-3-1 Scheduler 19
6-3-2 Dispatcher 19

6-4 ارتباط بین taskها 19
6-4-1 سمافورها 19
6-4-2 Mailboxes 21
6-5 تایمر صفر 23
6-6 سورس برنامه‌ها 24
6-6-1 rtos.h 24
6-6-2 rtos_core.c 26
6-6-3 user_task.c 35
6-6-4 Rtos_semaphore.c 39
6-6-5 rtos_mbox.c 41
6-6-6 user_task.c 42
6-6-7 user_task.h 46
6-6-8 rtos_uart.c 46
7 روش استفاده از سیستم‌عامل 49
8 ضمیمهٔ 1 (مشخصات Atmega32) 50
9 ضمیمهٔ 2 (نحوهٔ کامپایل و اجرا) 52
9-1 سورس فایل rtos.c 56
9-2 پیاده‌سازی سیستم‌عامل روی سخت‌افزار 60
10 مراجع 61

1 چکیده

سیستم عامل بلادرنگ در واقع یک سیستم‌عامل multi task است که برای عملیات بلادرنگ در نظر گرفته شده است و پارامترهای آن به گونه‌ای تنظیم شده‌اند که محدودیت‌های زمانی مورد نیاز را برآورده کند. استفاده از یک سیستم‌عامل بلادرنگ برای پیاده‌سازی سیستم‌های بلادرنگ باعث سادگی و سریع شدن پیاده سازی می‌باشد چون عملا بستری برای پیاده سازی ایجاد شده است و برنامه‌نویس در گیر جزئیات نمی‌گردد. از طرفی دیگر میکروکنترلرهای جدید با داشتن امکانات زیاد و سادگی طراحی مدارهای سخت افزاری مبتنی بر آنها به بستر سخت‌افزاری مناسبی برای سیستم‌های بلادرنگ تبدیل گشته‌اند. در صورتی که برای این میکروکنترلرها یک سیستم‌عامل بلادرنگ نوشته شود آنگاه پیاده‌سازی سیستم‌های بلادرنگ از لحاظ سخت‌افزار و نرم‌افزار با استفاده از میکروکنترلرها بسیار سریع و اقتصادی خواهد بود. در این گزارش به معرفی سیستم‌عامل بلادرنگی پرداخته می‌شود که برای میکروکنترلرهای AVR نوشته شد و با اندکی تغییر می‌توان از آن برای سایر میکروکنترلرها نیز استفاده کرد.

2 مقدمه

سیستم عامل نرم افزاری است که مسئول کنترل و بکار گیری منابع سخت افزاری مانند پردازنده، حافظه، فضای ذخیره‌سازی دیسک و تجهیزات جانبی است. سیستم عامل چهار چوبی را برای اجرای برنامه‌ها و ارتباط بین آن‌ها فراهم می‌کند. سیستم‌عاملها از لحاظ تعداد برنامه‌های در حال اجرا به دو دستهٔ Single Task و multi task تقسیم می‌شوند. در سیستم‌عامل‌های multi task، چندین برنامهٔ آمادهٔ اجرا در حافظه وجود دارند که سیستم‌عامل زمان پردازنده را بین آنها تقسیم می‌کند. قسمتی از سیستم عامل که زمان پردازنده را بین این برنامه‌ها تقسیم می‌کند Scheduler نام دارد. Scheduler بر طبق الگوریتم زمانبندی تعریف شده، یکی از برنامه‌های آماده برای اجرا را انتخاب می‌کند سپس کنترل اجرای دستورات، توسط قسمت دیگری از سیستم عامل به عنوان Dispatcher به این برنامهٔ انتخاب شده منتقل می‌شود.

الگوریتمهای زمانبندی که توسط Scheduler استفاده می‌شود به دو دستهٔ Preemptive و NonPreemptive تقسیم می‌شوند. در روش زمانبندی NonPreemptive زمانی که کنترل به یک برنامه داده شد، نمی‌توان کنترل اجرای برنامه را از آن گرفت تا اینکه برنامه به اتمام برسد و کنترل را به سیستم عامل برگرداند. در روش زمانبندی Preemptive سیستم‌عامل می‌تواند کنترل را از یک برنامهٔ در حال اجرا بگیرد و به برنامهٔ دیگر دهد. این روش چون سیستم‌عامل قادر است کنترل اجرای برنامه را موقتا قطع و برنامهٔ دیگری را اجرا کند بنابراین Scheduler می‌تواند الگوریتمهای زمانبندی الویت دار و round robinرا پیاده‌سازی کند.

سیستم عامل باید محیطی را برای ارتباط بین برنامه‌ها فراهم کند تا برنامه‌ها بتوانند با یکدیگر تبادل داده انجام دهند این کار معمولا با استفاده از PIPE یا MailBox انجام می‌گیرد. همچنین همزمانی مورد نیاز بین آنها را با استفاده از سیگنال‌ها و سمافورها فراهم می‌کند.

سیستم عامل بلادرنگ(RTOS[1]) سیستم‌عاملی است که قادر به اجرای برنامه‌های تعیین شده در بازهٔ زمانی مشخصی باشد و در صورتی که این بازهٔ زمانی رعایت نشود، سیستم عامل در انجام وظیفهٔ خود شکست خورده است. با توجه به نکتهٔ فوق سیستم‌عامل بلادرنگ اولا باید multi task باشد، ثانیا Preemptive باشد تا بتواند مکانیزم اولویت بندی را پیاده‌سازی کند. هدف اصلی در این نوع سیستمها برآورده کردن محدودیتهای زمانی می‌باشد. با توجه به اینکه کاربرد سیستم‌عامل های بلادرنگ در صنعت می‌باشد، نیازی به تهیهٔ رابط کاربر خیلی پیچیده نمی‌باشند و خیلی از آنها رابط کاربر خیلی ساده‌ای در حد چند سوئیچ و نمایشگر ساده می‌باشند. خیلی از سیستم‌عامل‌های بلادرنگ به قسمت مدیریت حافظه نیازی ندارند، چون

[1] Real Time Operating System

Taskهای مورد نیاز برای اجرا از همان ابتدا مشخص هستند و نیازی به پیاده سازی این قسمت ندارند. در جدول 1 تعدادی از سیستم عامل های بلادرنگ متن باز و اختصاصی ارائه شده‌اند.

جدول 1: **لیستی از سیستم‌عاملهای بلادرنگ**

OpenSorces RTOS	Proprietary RTOS
ECos	Windows CE
µCOS	Ardence RTX
EROS	ChorusOS
CapROS	DMERT
Coyotos	DNIX
Fiasco (L4clone)	Symbian OS
FreeRTOS	embOS (Segger)
C Executive	HP-1000/RTE
scmRTOS	Intime
MenuetOS	ITRON
Nut/OS	LynxOS
Phoenix-RTOS	MERT
Prex	MicroC/OS-II
RTAI	µnOS
RTEMS	MQX RTOS
RTLinux	Nucleus
Solaris	OSE
SHaRK	OSEK/VDX
TRON Project	OSEKtime
TUD:OS	Phar Lap ETS
Xenomai	PikeOS
Marte OS	QNX

در این گزارش به نحوهٔ پیاده سازی یک سیستم‌عامل بلادرنگ برای میکروکنترلرها پرداخته می‌شود. این سیستم‌عامل به زبان C نوشته شده است و برای کمپایل آن از نرم‌افزار CodeVisionAvr 1.23.8C standard استفاده شده است.

روند گزارش به این صورت است که در قسمت‌های بعدی به ترتیب، ابتدا تراشه‌های AVR معرفی می‌شود سپس با توجه به خصوصیات آنها و نیازهای مورد نیاز به ارائهٔ طرح پیشنهادی برای سیستم عامل مزبور پرداخته شده و در نهایت لیست برنامه‌ها به همراه توضیحات در مورد آنها آورده می‌شود. این گزارش دارای دو ضمیمه می‌باشد که در ضمیمهٔ 1، مشخصات میکرو کنترلر Atmega32 که به عنوان سخت‌افزار سیستم‌عامل

در نظر گرفته شده است آورده می‌شود و در ضمیمهٔ 2 نحوهٔ کامپایل و پراگرام کردن برد آزمایشی به همراه تصاویری از آن و همچنین کد برنامهٔ C نوشته شده که در کامپیوتر PC اجرا می گردد ارائه گردیده است.

3 میکروکنترلرهای AVR

تکنولوژی AVR برای اولین بار در سال 1997 توسط شرکت Atmel ارائه شد و بعد از آن جزء تولیدات محبوب این شرکت قرار گرفت. مزیت اصلی این تکنولوژی داشتن هسته RISC همراه با تعداد زیادی ثبات کاری یا Working Register است.این ثباتها به ALU مرتبط هستند و توسط آنها می توان تعداد زیادی ریز دستورالعمل را در مدت زمان یک پالس ساعت اجرا کرد به عبارتی دیگر اجرای هر دستورالعمل یک پالس ساعت لازم دارد در حالیکه اجرای این ریز دستورالعملها در میکروکنترلرهای دیگر در تعداد زیادتری از پالس ساعت اجرا می شوند بنابراین AVR ها می توانند بسیار سریعتر عمل کنند و همچنین کدهای با حجم بالایی را اجرا کنند.به عنوان مثال کارایی یک AVR که با سرعت4 MHzکار می کند با کارایی میکروکنترلر PIC با سرعت 16MHz و همچنین میکروکنترلر 8051 با سرعت 48 MHzبرابر است.

3-1 ثباتهایAVR

میکروکنترلرهای هشت بیتی AVR دارای32 ثبات هشت بیتی همه منظوره به نامهای r0 تا . r31 می‌باشند.

سه ثبات آدرس شانزده بیتی با نام مستعار X و Y و Z که هر کدام از این سه ثبات دو ثبات از همان 32 ثبات 8 بیتی هستند یعنی . ((X(r27:r26), Y(r29:r28), Z(r31:r30

یک ثبات 16 بیتی به منظور اشاره‌گر پشته که در آدرسهای ورودی/خروجی (0x3e(SPH:و (0x3d(SPL قرار گرفته‌اند.

یک ثبات 8بیتی به منظور سنجش وضعیت یا همان ثبات پرچم با نام. SREG که بیتهای آن به صورت ITHSVNZC نامگذاری شده‌اند. شرح این بیت ها به صورت زیر می‌باشد:

I : فعال ساز و غیرفعال ساز عمومی وقفه SREG7 یا Global Interrupt Enable/Disable Flag

T : بیت انتقالی مورد استفاده دستورالعملهای BLD و BSTبا نام SREG6

H: Half Carry Flag, SRE5 G

S : بیت علامت یا Signed tests Instruction Set, SREG4

V : سرریزنما برای مکمل دو یا Two's Complement Overflow Indicator, SREG3

N : بیت منفی یا Negative Flag, SREG2

Z : بیت صفر یا Zero Flag, SREG1

C : Carry Flag, SREG0

در شکل 1، نقشهٔ ثباتهای AVR نمایش داده شده است.

	7 … 0	Addr.	
General Purpose Working Registers	R0	$00	
	R1	$01	
	R2	$02	
	...		
	R13	$0D	
	R14	$0E	
	R15	$0F	
	R16	$10	
	R17	$11	
	...		
	R26	$1A	X-register Low Byte
	R27	$1B	X-register High Byte
	R28	$1C	Y-register Low Byte
	R29	$1D	Y-register High Byte
	R30	$1E	Z-register Low Byte
	R31	$1F	Z-register High Byte

شکل 1: **نقشهٔ ثباتهای AVR**

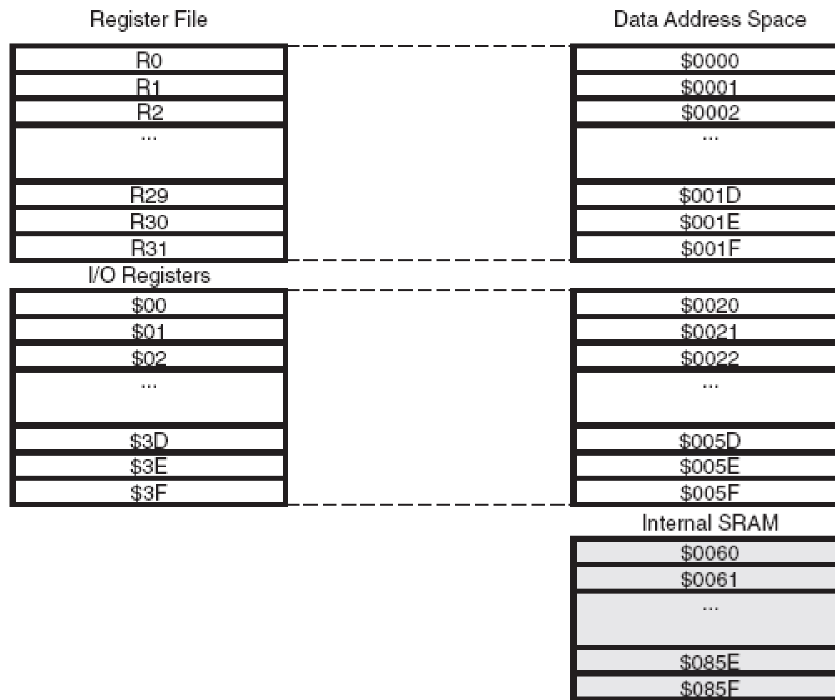
2-3 **حافظه داده و ثباتهای**AVR

32آدرس اول حافظه یعنی (x0000 تا x001f0) متعلق به ثباتهای r0 تا r31 هستند. البته در برخی MCU (Master Controller Unit)ها برای ثباتها از فضای حافظی داده استفاده می‌شود .

آدرسهای (x0020 تا x005f) از حافظه‌ی داده در دسترس آدرسهای ورودی/خروجی است

از آدرس x0060 حافظه‌ی داده به بعد فقط شامل حافظه استاتیک است.

در شکل 2 نقشهٔ حافظه میکروکنترلر AVR ارائه شده است.



شکل 2: **نقشهٔ حافظهٔ میکروکنترهای AVR**

3-3 خانواده های محصولاتAVR

1-3-3 Tiny AVR

این مدل به منظور انجام یک عملیات ساده بهینه سازی شده و در ساخت وسایلی که به میکروهای کوچک احتیاج است کاربرد فراوان دارند. کارایی عظیم آنها برای ارزش و بهای وسایل موثر است.

2-3-3 Mega AVR

این مدل از میکروکنترلر ها دارای امکانات زیر می‌باشند:

فرکانس کاری 16مگاهرتز

حافظهٔ RAM داخلی

حافظهٔ EEPROM داخلی

حافظه فلش

مبدل های آنالوگ به دیجیتال

USART و SPI و TWI بر طبق واسطه های سریاسل

واسطه ی JTAG بر طبق IEEE 1149.1

TWI: Two Wire Interface is a byte oriented interface
USART: Universal Serial Asynchronous Receiver/Transmitter
SPI: Serial Peripheral Interface

```
JTAG available only on devices with 16KB Flash and up
```

واسط JTAG فقط در میکروهای با بیش از 16 کیلوبایت حافظه فلش موجود است.

نامگذاری این مدل به صورت ATMEGAxx می‌باشد که xx مقدار حافظهٔ فلش (حافظهٔ برنامه) را برحسب مگابایت مشخص می‌کند. هر کدام از این تراشه‌ها دارای امکانات فوق می‌باشند و تفاوت آنها در حجم و تعداد آنها می‌باشد. در پیوست 1، خصوصیات تراشهٔ ATMEGA32 شرح داده شده است.

3-3-3 **LCD AVR**

این مدل با بالاترین یکپارچگی و انعطاف پذیری ممکن طراحی شده است و با داشتن درایور LCD و کنترلر اتوماتیک وضوح تصویر ،بهترین واسطه را با انسان دارند و دارای توان مصرفی پایین و کارایی بالایی هستند.اولین عضو این خانواده 100 سگمنت داشت و دارای یک UART و SPI به منظور ارتباط به صورت سریال بود.

4 **نرم‌افزار**

میکروکنترلرهای AVR به سه زبان زیر برنامه نویسی می‌شوند که هر کدام نرم افزارهای خاص خود را برای امر برنامه نویسی و کامپایل کردن نیاز دارند:

1- زبان برنامه نویسی اسمبلی

2- زبان برنامه نویسی c

3- زبان برنامه نویسی بیسیک

زبان برنامه نویسی اسمبلی زبان اصلی بوده و برای یادگیری آن نیاز به خواندن کتاب یا جزوه خاصی نیست.تمامی دستورالعملهای برنامه‌نویسی و ساختار سخت‌افزاری خانواده میکروکنترلرهای AVR در DATA SHEETهای آماده شده توسط شرکت ATMEL آورده شده است.همچنین با مراجعه به سایتهای ذیل میتوان با مثالهای این زبان برنامه نویسی آشنا شد:

www.avr-asm-tutorial.net/avr_en
www.avrfreaks.net
www.cygan.com/avr

نرم افزار معروف شبیه سازی و رفع اشکال برنامه نوشته شده به زبان اسمبلی AVR STUDIO است .

زبان برنامه نویسی C ، قابلیتهای زیادی از جمله ارتباط بهتر با سخت‌افزار رادارد و دست برنامه‌نویسان را برای کدنویسی کاملا باز گذاشته است. برنامه نویسان میتوانند برای اجزاء سخت افزاری مختلف قابل اتصال به میکرو کدنویسی کنند. نرم افزار محبوب مورد استفاده برای این برنامه نویسی CodevisionAVR است. برای نوشتن سیستم عامل بلادرنگ مورد بحث نیز از زبان C و برای کمپایل آن از نرم‌افزار فوق استفاده شده است

زبان برنامه نویسی بیسیک که ساده ترین راه برای یادگیری برنامه نویسی AVR است اما قابلیتهای زبانهای برنامه نویسی C و اسمبلی را ندارد. نرم‌افزار محبوب مورد استفاده برای این نوع برنامه‌نویسی BASCOM است.

5 طرح پیشنهادی

اکثر سیستم‌عامل‌های بلادرنگی که تا کنون پیاده سازی شده‌اند به این صورت می‌باشند که پس از اتصال تغذیه به سیستم، سیستم عامل اجرا شده و با توجه به فرامین کاربر Taskها به صورت دینامیک بوجود می‌آیند و اجرا می‌گردند. چنین سیستم‌عاملی اولا حجم زیادی از حافظه را اشغال می‌کند و پیچیدگی زیادی دارد. با توجه به اینکه در میکروکنترلرهای AVR یا هر نوع میکرو کنترلر دیگر محدودیت حافظه موجود است و معمولا Taskها از ابتدا معلوم می‌باشند و نیازی به فراهم آوردن محیطی برای ایجاد Task به صورت دینامیک وجود ندارد طرحی بهینه به صورت زیر ارائه می‌گردد.

هر طرحی از دو قسمت سیستم عامل و برنامه‌های مورد نیاز برای اجرا تشکیل می‌شود و معمولا در این طرح ها سیستم‌عامل و برنامه‌ها از یکدیگر جدا هستند و جداگانه کامپایل می‌شوند. در طرح پیشنهادی هدف این است که taskها و سیستم عامل بلادرنگ تحت عنوان یک برنامه کامپایل شوند. در این روش با توجه به سیستمی که قرار است به صورت بلادرنگ کار کند بررسی می‌گردد و تعداد Taskهای مورد نیاز و اولویت آنها و زمانهای مورد نیاز برای آنها استخراج می‌شود و برنامه نویسی می‌گردند و سپس این Taskها در مکانهای خاصی از جهارچوب ارائه شده برای سیستم عامل قرار داده می‌شوند و مجموعاً تحت عنوان یک کد کامپایل شده و تراشه توسط آن پراگرام می‌گردد. در واقع سیستم عامل بلادرنگ نوشته شده محیطی را فراهم می‌آورد که هر Task مستقل از بقیهٔ Task ها نوشته می‌شود و هر Task تصور می‌کند که تمام سخت‌افزار را در اختیار دارد، نیازی به هماهنگی در نحوهٔ استفاده از رجیسترها نیست و هر Task تمام 32 رجیستر را در اختیار دارد. همچنین Taskها به صورت همزمان اجرا می‌گردند بدون اینکه کاربر نیازی به نوشتن کدی داشته باشد. با توجه به نوع سیستم کاربر می‌تواند الگوریتم Scheduling را نیز تغییر دهد.

6 کدهای سیستم‌عامل بلادرنگ

در این قسمت به شرح کدهای سیستم‌عامل مزبور می‌پردازیم

1-6 متغیرهای عمومی

os_central_table 1-1-6

os_central_table استراکچری است که داده‌های اصلی مورد نیاز سیستم عامل را نگهداری می‌کند، فیلدهای این استراکچر به صورت زیر است:

```
typedef struct os_central_table {
// Timer ticks since last boot os_systime;       ULONG
os_dispcntr;//dispatch counter       ULONG
os_idlecntr;// Counter for processor idle time       ULONG
// OS version os_version;       UBYTE
// OS running flag os_running;       UBYTE
os_schedrun;//scheduler flag       UBYTE
// Pointer to used TCBs  os_tcb *tcb_used_list;
// Pointer to TCB freelist  os_tcb *tcb_free_list;
// Pointer to current running task TCB    os_tcb *tcb_current;
os_tcb *tcb_runq;//Pointer to highest priority runnable task TCB
*scb_used_list; // Pointer to used semaphores      os_scb
*scb_free_list; // Pointer to semaphores freelist      os_scb
*mcb_used_list; // Pointer to used message mailboxes      os_mcb
*mcb_free_list; // Pointer to mailbox freelist      os_mcb
} os_cent_tbl;
```

شرح فیلدهای این استراکچر در جدول 2 نمایش داده شده است

جدول 2: فیلدهای استراکچر os_central_table

نام فیلد	نوع	توضیحات
os_systime	32-bit unsigned integer	تعداد تیکهای ساعت را شمارش می کند.
os_dispcntr	32-bit unsigned integer	تعداد دفعاتی که تعویض متن انجام شده را نشان می‌دهد
aos_idlecntr	32-bit unsigned integer	تعداد دفعاتی که NULL-task احضار شده است
aos_version	8-bit unsigned integer	نشان دهندهٔ نسخهٔ سیستم عامل است
os_running	8-bit unsigned integer	فلگی است که نشان می‌دهد آیا سیستم‌عامل در حال اجرا می‌باشد یا خیر
os_schedrun	8-bit unsigned integer	فلگی است که نشان می‌دهد scheduler فعال است یا خیر
tcb_used_list	TCB pointer	اشاره گر به اولین Task استفاده شده
tcb_free_list	TCB pointer	اشاره گر به اولین Task آزاد

tcb_current	TCB pointer	اشاره گر به TCB مربوط به Task در حال اجرا
tcb_runq	TCB pointer	اشاره گر به TCB مربوط به Taskی که بالاترین اولویت را دارد و قابل اجرا می‌باشد
scb_used_list	SCB pointer	اشاره‌گر به اولین سمافور استفاده شده
scb_free_list	SCB pointer	اشاره گر به اولین سمافور آزاد.
mcb_used_list	MCB pointer	اشاره‌گر به اولین MailBox استفاده شده
mcb_free_list	MCB pointer	اشاره‌گر به اولین MailBox آزاد

2-1-6 **ثابتها**

سیستم عامل دارای تعدادی constant می‌باشد که قبل از کامپایل کردن آن باید با مقدار صحیح مقدار دهی شوند. در واقع این constant برای اختصاص فضای حافظه مورد نیاز به متغیر ها استفاده می‌شود. برای اینکه سیستم‌عامل زیاد پیچیده نباشد، برای این سیستم عامل واحد مدیریت حافظه در نظر گرفته نشده است و احتصاص فضای حافظه به صورت استاتیک می‌باشد. تمام constantهایی که توسط کاربر باید مقدار دهی شود در فایلrtos.h قرار دارند. این constantها عبارتند از:

OS_TASK_MAX, Taskماکزیمم تعداد ها

OS_SEM_MAX,ماکزیمم سمافورهای مورد نیاز

OS_MBOX_MAX,مورد نیاز هایMail Boxماکزیمم

OS_TASK_HSTK_SIZE,اندازهٔ استک سخت‌افزاری

OS_TASK_DSTK_SIZE,اندازهٔ استک نرم‌افزاری

مقدار ثابت OS_TICK_TIME نیز توسط کاربر قابل مقدار دهی می‌باشد و کاربر می‌تواند با مقدار دهی این ثابت قبل از کامپایل مدت هر Slice time را برای Taskها انتخاب کند. سیستم‌عامل از این ثابت برای مقدار دهی تایمر استفاده می‌کند. مقدار این ثابت زمان را بر حسب میلی ثانیه مشخص می‌کند و می‌تواند بین 1 تا 16 میلی‌ثانیه باشد، در صورتی که کاربر زمانهای کوتاهتر یا طولانی تری نیاز داشته باشد، باید مود تایمر در قسمت Main عوض شود. در پیاده سازی فعلی از مود CTC تایمر صفر استفاده شده است و کلاک تایمر 15,624کیلو هرتز انتخاب شده است.(کلاک اصلی تقسیم بر 1024).

در مود CTC تایمر از صفر شروع به شمارش می‌کند تا به مقدار تعیین شده در رجیستر هشت بیتی OCR0 برسد، در این حالت تایمر وقفه می‌دهد و شمارش را از صفر شروع می‌کند. فاصلهٔ زمانی بین دو وقفه به مقدار

OCR0 و کلاک تایمر بستگی دارد.کلاک تایمر از تقسیم کلاک اصلی میکرو(16Mhz) بر مقدار تعیین شده در رجیستر TCCR0 تعیین می‌شود. در پیاده سازی کنونی مقدار رجیستر TCCR0 ثابت در نظر گرفته شده است و مقدار دهی رجیستر OCR0 با توجه به ثابت OS_TICK_TIME انجام می‌پذیرد.

در صورتی که به رزولوشن بیشتری نیاز داشته باشید باید کلاک تایمر صفر را با مقدار دهی رجیستر TCCR0 در حالتی قرار دهید که کلاک تایمر سریعتر باشد.

3-1-6 **نوع داده‌های تعریف شده**

نوع های زیر را در فایل rtos.h تعریف شده است.

```
typedef unsigned char UBYTE;
typedef unsigned int UWORD;
typedef unsigned long int UDWORD;
typedef unsigned long int ULONG;
```

2-6 **Tasks‌ها**

1-2-6 **مقدمه**

Task به یک زیر برنامه یا تابع ساده گفته می‌شودکه محیط کاری مربوط به خودش را دارد و دارای یک Context می‌باشد. سیستم‌عامل همیشه هنگام وقوع وقفهٔ تایمر، task آماده به اجرایی را برای اجرا انتخاب می‌کند که بالاترین اولویت را داشته باشد.

سیستم‌عامل پیاده سازی شده عمل محافظتی از حافظه انجام نمی‌دهد و همهٔ Taskها از یک فضای آدرس مشترک استفاده می‌کنند همچنین تمام متغیرهای عمومی از جمله متغیرهای مربوط به سیستم عامل توسط Taskها قابل دسترسی می‌باشد. بنابراین کاربران باید هنگام نوشتن برنامه(Taskها)به این نکته دقت کنند.

2-2-6 **بلوک کنترل Task (TCB)**

هر Task در سیستم‌عامل دارای یک TCB می‌باشد. در TCB تمام داده‌های مهم مربوط به کنترل Task توسط سیستم‌عامل نگهداری می‌شود. تمام TCBهای مربوط به Taskها توسط سیستم عامل به هم پیوند می‌خورند و یک لیست پیوندی یک طرفه ایجاد می‌کنند

فیلدهای TCBبه صورت زیر تعریف شده است:

```
typedef struct os_task_control_block {
// Pointer to task's hardware stack           *hwstk;  void
// Pointer to task's data stack          *datastk;  void
// Status of task             status; UBYTE
// Priority of task               prio; UBYTE
      // Nbr of ticks to keep task sleeping      delay; UWORD
//for keep track task time         ULONG   ttime;
//for keep track stack use       *hstktop ;   void
```

```
//for keep track stack use        *dstktop ;  void
struct os_task_control_block *next_tcb;//Pointer to next task's TCB
// Next task on semaphore waiting list        void *tcb_sem_q;
} os_tcb;
```

در جدول 3 توضیحات فیلدهای این استراکچر شرح داده شده است:

جدول3: فیلدهای استراکچر os_task_control_block

نام فیلد	نوع	توضیحات
hwstk	Pointer	اشاره گر به استک سخت افزاری
datastk	Pointer	اشاره گر به استک نرم‌افزاری
status	8-bit unsigned integer	نشان دهندهٔ وضعیتTask که می‌تواند یکی از حالات RUNNABLE, RUNNING DELAYD WAITING باشد.
prio	8-bit unsigned integer	الویت Task را مشخص می‌کند. عددی بین صفر تا 255. صفر بالاترین اولویت و 255 پایین ترین اولویت می‌باشد
delay	16-bit unsigned integer	مدت زمانی است که TASK نباید اجرا شود. توسط کاربر تاخیر خورده است. این زمان برحسب تیک ساعت می‌باشد که در فایل rtos.h مشخص شده است.
ttime		مدت زمانی که پردازنده به این Task اختصاص داده است.
hstktop		برای دنبال کردن استک استفاده دارد.
dstktop		برای دنبال کردن استک استفاده دارد.

next_tcb	TCB pointer	اشاره‌گر به TCB بعدی در لیست
tcb_sem_q	TCB pointer	اشاره‌گر به TCB بعدی که منتظر سمافور است

3-2-6 ایجاد task

تمامی taskها توسط تابع ()os_task_create که جزئی از هستهٔ سیستم‌عامل پیاده سازی شده می‌باشد، ایجاد می‌گردند. این تابع به صورت داخلی توسط سیستم‌عامل برای ایجاد NULL-task فراخوانی می‌شود تا در صورتی که کاربر Taskی تعریف نکند، برای سیستم‌عامل مشکلی رخ ندهد. سیستم عامل اولویت این Task را کمترین مقدار ممکن قرار می‌دهد تا در صورت وجودTask کاربر، این task هیچگاه اجرا نگردد.
تابع ()os_task_create دارای چهار پارامتر می‌باشد که عبارتند از : اشاره گر به کد این Task، اشاره گر به فضای اختصاص داده شده برای استک سخت افزاری، اشاره گر به فضای اختصاص داده شده برای استک نرم افزاری و نهایتا اولویت Task.
در این تابع ابتدا استک سخت‌افزاری تولید می‌شود. این استک باید طبق فرمت خاصی باشد و با ساختاری که نرم‌افزار CodevisionAVR انتظار دارد مطابقت داشته باشد به همین دلیل آدرس برگشت به Task در آن قرار داده می‌شود سپس تمام رجیسترهای عمومی بجز R28 و R29 روی آن قرار می‌گیرددر این استک رجیستر SREG که بیت مربوط به اینتراپت در آن فعال شده است روی استک قرار داده می‌شود.

4-2-6 استک

برنامهٔ CodevisionAVR برای تبدیل برنامه‌های نوشته شده از زبان C به زبان اسمبلی از دو استک استفاده می‌کند که اصطلاحاً آنها را استک سخت‌افزاری و نرم‌افزاری نامیده‌اند. استک سخت‌افزاری توسط مکانیزم اینتراپت و صدا زدن زیر برنامه‌ها برای نگهداری آدرس برگشت استفاده می‌شود. هنگامی که از دستورات Push یا Pop استفاده می‌شود در واقع از این استک استفاده می‌شود. از استک نرم‌افزاری برای ارسال پارامترها و دریافت مقدار برگشتی از زیر برنامه استفاده می‌شود. از استک نرم‌افزاری برای ذخیره کردن متغیرهای محلی که در زیر برنامه‌ها استفاده شده‌اند نیز استفاده می‌شود. هر Task باید دارای این دو استک مخصوص به خودش باشد. این استکها دارای اندازهٔ ثابت می‌باشند که در فایل rtos.h تعیین شده است.

5-2-6 Null task

Null-task نیز مشابه Taskهای تعریف شده توسط کاربر می‌باشد ولی دارای چند خصوصیت متفاوت می‌باشد. اولا این task دارای کمترین الویت می‌باشد و وضعیت آن همیشه Runnable می‌باشد.این دو خصوصیت باعث می‌شود که اگر هیچ کدام از Taskهای کاربر در وضعیت اجرا نباشند، این task برای اجرا انتخاب شود. این task در واقع زمانی اجرا می‌شود که سیستم در حالت idle قرار داشته باشد. از این task می‌توان برای روشن کردن یک LED استفاده کرد که روشنایی آن باعث می‌شود میزان بودن سیستم را نشان دهد.

6-2-6 یک نمونه برنامه

در این قسمت به عنوان مثال یک نمونه سیستم که با استفاده از این سیستم‌عامل پیاده سازی شد، معرفی می‌کنیم. در این مثال چهار task به نامهای task1, task2, task3 and task4 در نظر گرفته شد و برنامهٔ مربوط به این taskها در فایل user_tasks.c نوشته شد. Task1 توسط برنامهٔ main که در فایل rtos_core.c قرار دارد ایجاد می‌شود و بقیهٔ taskها توسط این task تولید شده، ایجاد می‌شوند.

Taskهای 2و3 و4، شامل یک حلقهٔ بینهایت هستند که باعث روشن و خاموش شدن Led متناظر خود روی برد می‌شوند. این Taskها برای ایجاد تاخیر بین روشن و خاموش شدن Ledها از تابع os_sleep() استفاده می‌کنند. این تابع task را برای مدت معینی که در پارامتر آن ذکر می‌شود، غیر فعال می‌کند.

Task1 پورت سریال میکرو را مقدار دهی می‌کند و منتظر دریافت کاراکتری از طرف کامپیوتر PC می‌شود. این task نیز با دریافت هر کاراکتر Led متناظر خودش را Toggle می‌کند. با توجه به کاراکتر ارسال شده

توسط PC فرمان مربوطه را که فعال یا غیر فعال کردن یک از Taskها می‌باشد انجام می‌دهد. همچنین این برنامه می‌تواند اطلاعات زمانی در مورد taskها را برای PC ارسال کند تا کاربر از وضعیت آنها با خبر شود.

3-6 زمانبندی taskها

زمانبندی و سوئیچ بین برنامه‌ها توسط Scheduler و Dispatcher انجام می‌شود.

Scheduler 1-3-6

وظیفهٔ scheduler ، انتخاب و تصمیم گیری در مورد taskای می‌باشد که در تیک زمانی بعدی باید اجرا گردد. تصمیم گیری به وضعیت و اولویت ، Taskها بستگی دارد. این تابع ()os_schedule_ نام دارد و در فایل rtos_core.c قرار دارد. علامت _ که در ابتدای نام این تابع استفاده شده است برای مشخص کردن این است که این تابع توسط Task های کاربر صدا زده نمی‌شود و مخصوص خود سیستم عامل است.

الگوریتم زمانبندی سیستم‌عامل پیاده سازی شده از نوع Pre-emptive می‌باشد. این زمانبندی این امکان را می‌دهد که taskها و وقفه می‌توانند بلافاصله باعث عوض شدن task قابل اجرا شوند، بدون اینکه برنامهٔ جاری به اتمام برسد. در صورتی که task بعدی همان task فعلی نباشد آنگاه باید عمل task Switch توسط تابع ()os_dispatch_ انجام شود که شامل ذخیره کردن Context برنامهٔ جاری و بار کردن Context برنامهٔ جدید می‌باشد.

Dispatcher 2-3-6

وظیفهٔ dispatcher، سوئیچ واقعی بین taskها می‌باشد که اصطلاحا تعویض متن نامیده می‌شود. تعویض متن شامل سه مرحلهٔ زیر می‌باشد:

1-ذخیره کردن رجیسترهای task قبلی در استک سخت‌افزاری و ذخیره کردن اشاره‌گر استک در TCB

2- بارکردن مقدار اشاره گر استک از TCB مربوط به task جدید و بارکردن مقدار رجیسترها از استک سخت‌افزاری task جدید

3-رفتن به task جدید

این تابع ()os_dispatch_ نام دارد و در فایل rtos_core.c قرار دارد. علامت _ که در ابتدای نام این تابع استفاده شده است برای مشخص کردن این است که این تابع توسط Task های کاربر صدا زده نمی‌شود و مخصوص خود سیستم عامل است.

4-6 ارتباط بین taskها

1-4-6 سمافورها

مقدمه

سمافورها برای ایجاد هماهنگی بین taskها استفاده می‌شود. و برای پیاده‌سازی ناحیهٔ بحرانی نیز استفاده می‌گردد. در حالت کلی می‌توان گفت سمافور یک متغیر عمومی است که مقدار اولیه آن می‌تواند 0 تا 65535 باشد.

بلوک کنترلی سمافور

به هر سمافور یک بلوک کنترلی به نام SCB تخصیص داده می‌شود. این بلوک کنترلی، حاوی تمام داده‌های مورد نیاز سمافور مربوط به خودش می‌باشد. این بلوکهای کنترلی به هم دیگر پیوند خورده‌اند و یک لیست پیوندی را تشکیل داده اند. فیلدهای بلوک کنترل سمافور به صورت زیر تعریف می‌شود.

```
typedef struct os_sema {
// Semaphore value            count; UWORD
// Semaphore queue *scb_sem_q;      os_tcb
struct os_sema *next_scb;// Pointer next semaphore's SCB
} os_scb;
```

جدول4: فیلدهای استراکچر os_sema

فیلد	نوع	توضیحات
count	16-bit unsigned integer	مقدار سمافور
scb_sem_q	TCB pointer	اشاره‌گر به اولین TCB که منتظر این سمافور می‌باشد.
next_scb	SCB pointer	اشاره گر به SCB استفاده شدهٔ بعدی

ایجاد سمافور

سمافور با استفاده از تابع سیستمی os_sem_create() ایجاد می‌گردد. این تابع اولین SCB،آزاد را از لیست انتخاب می‌کند و فیلدهای آن را مقدار دهی می‌کند و اشاره‌گر به آن SCB را برای Task فراخوانی کنندهٔ این تابع برمی‌گرداند. از این اشاره‌گر به عنوان Handle به ان سمافور استفاده می‌شود. در صورتی که SCB آزاد موجود نباشد، این تابع مقدار تهی را برمی‌گرداند.

نحوهٔ عملکرد سمافور

سمافور توسط دو تابع os_wait() و os_signal() مورد استفاده قرار می‌گیرد. تابع os_wait() مقدار سمافور را چک می‌کند، اگر مقدار آن مثبت باشد آنگاه یک واحد از آن کم می‌کند و سپس اجرای Taskی که این

تابع را صدا زده است ادامه می‌یابد. اگر مقدار سمافور صفر باشد، آنگاه task فراخوانی کننده معلق می‌شود و در صف انتظار سمافور قرار داده می‌شود تا اینکه تابع os_signal() توسط task دیگری فراخوانی شود.

تابع os_signal()ابتدا صف انتظار سمافور را چک می‌کند، اگر taskای در صف انتظار وجود داشته باشد اولین task را از صف خارج می‌کند و زمانبند صدا زده می‌شود. اگر صف انتظار خالی باشد آنگاه مقدار سمافور یک واحد اضافه می‌شود و task فراخوانی کننده ادامه می‌یابد.

چون توابع سمافور باید بدون انقطاع اجرا شوند بنابراین باید در ناحیهٔ بحرانی انجام شوند. با غیر فعال کردن وقفه‌ها این کار به اسانی قابل انجام است.

6-4-2 **Mailboxes**

مقدمه

Mailbox مکانیزمی برای ارسال پیام بین taskها می‌باشد. Mailbox مکانی از حافظه است که حاوی اشاره‌گری به آن پیام می‌باشد

بلوک کنترل Mailbox

هر Mailbox دارای یک بلوک کنترلی به نام MCB می‌باشد که برای نگهداری داده‌های مورد نیاز Mailbox می‌باشد. همهٔ Mailbox ها به هم پیوند خورده‌اند و یک لیست پیوندی ایجاد کرده‌اند. فیلدهای بلوک کنترل Mailbox به صورت زیر است:

```
typedef struct os_mail_box {
// Semaphore for sending to mailbox           *msend;      os_scb
// Semaphore for receiving from            *mrecv;      os_scb
mailbox
// Pointer to message              *mmsg;  void
// Pointer to next mailbox      struct  os_mail_box *next_mcb;
} os_mcb;
```

جدول5: فیلدهای استراکچر os_mail_box

فیلد	نوع	توضیحات
msend	SCB pointer	اشاره‌گر به سمافوری که برای ارسال استفاده می‌شود
mrecv	SCB pointer	اشاره‌گر به سمافوری که برای دریافت استفاده می‌شود
mmsg	Pointer	اشاره‌گر به پیام
next_mcb	MCB pointer	اشاره‌گر به بلوک کنترل mailbox بعدی

ایجاد mailbox

mailbox با استفاده از تابع سیستمی ()os_ mbox _create ایجاد می‌گردد. این تابع اولین MCB ،آزاد را از لیست انتخاب می‌کند و فیلدهای آن را مقدار دهی می‌کند و اشاره‌گر به آن SCB را برای Task فراخوانی کنندهٔ این تابع برمی‌گرداند. از این اشاره‌گر به عنوان Handle به ان mailbox استفاده می‌شود. در صورتی که MCBآزاد موجود نباشد، این تابع مقدار تهی را برمی‌گرداند. Taskی که با mailbox کار می‌کند باید خودش مواظب اندازهٔ پیام‌ها باشد.

نحوهٔ عملکرد mailbox

Mailbox توسط دو تابع ()os_mbox_send و () os_mbox_ recv قابل دسترسی می‌باشند. تابع ()os_mbox_send ، اگر پیامی درmailbox نباشد آن را به mailbox ارسال می‌کند. اگر mailbox در حال استفاده باشد، task فراخوان کنندهٔ این تابع به حالت تعلیق می‌رود تا اینکه mailboxخالی شود. تابع () os_mbox_ recv پیام را از mailbox دریافت می‌کند و به task فراخوانی کننده برمی‌گرداند. اگر mailbox خالی باشد آنگاه Task فراخوانی کننده به حالت تعلیق برده می‌شود تا اینکه پیامی دریافت شود.

5-6 تایمر صفر

تایمر صفر در سیستم عامل پیدا سازی شده نقش مهمی ایفا می‌کند. این تایمر با توجه به مقداری که در ثابت OS_TICK_TIME در فایل rtos.h تعیین شده است وقفه می‌دهد. سرویس روتین این وقفه taskهایی را که با استفاده تابع ()os_sleep غیر فعال شده‌اند در موقع مناسب فعال از حالت delayed خارج و آنها را فعال می‌کند. سرویس روتین از طریق لیست پیوندی TCB فرایندها را چک می‌کند و اگر فیلد delay بزرگتر از صفر بود از آن یکی کم می‌کند و اگر به صفر رسید آنگاه فرایند را به حالت runnable تغییر وضعیت می‌دهد و scheduler را فراخوانی می‌کند. سرویس روتین همچنین مقدار uptime را افزایش می‌دهد.

6-6 سورس برنامه‌ها

rtos.h 1-6-6

```
//define macro and data structure and constants
#define OS_ENTER_CRITICAL  {    \
#asm("cli")     \
os_crit_count++; \
}


#define OS_EXIT_CRITICAL   {    \
--os_crit_count;  \
\     if(!os_crit_count){
#asm("sei")    \
\  }
}
int os_crit_count;


0b00000000     #define OS_TASK_RUNNABLE
0b00000001       #define OS_TASK_RUNNING
0b00000010       #define OS_TASK_WAITING
0b00000100        #define OS_TASK_DELAYD
0b00001000     #define OS_TASK_SUSPENDED
#define OS_TASK_SUSPENDED_WAIT  0b00010000
0b00100000      #define OS_TASK_SUSPENDED_DELA


// start of User configurable Constants:
#define OS_TICK_TIME            1  //mSec (maximum value is 16ms)
5            #define OS_SEM_MAX
5           #define OS_TASK_MAX
2           #define OS_MBOX_MAX
100     #define OS_TASK_HSTK_SIZE
200     #define OS_TASK_DSTK_SIZE
//Scheduler task time acculumulator
//When OS_CHCK==1 the scheduler uses timer1!
1    //sched time enable               #define OS_CHCK
// end  of User configurable Constants:


#pragma regalloc-   #define OS_GLOBAL_VAR
#pragma regalloc+   #define OS_GLOBAL_END


typedef unsigned char UBYTE;
typedef unsigned int UWORD;
typedef unsigned long int UDWORD;
typedef unsigned long int ULONG;
```

```
typedef struct os_task_control_block {
// Pointer to task's hardware stack           *hwstk;  void
// Pointer to task's data stack   *datastk;  void
// Status of task           status; UBYTE
// Priority of task       prio; UBYTE
        // Nbr of ticks to keep task sleeping      delay; UWORD
//for keep track task time         ULONG   ttime;
//for keep track stack use *hstktop ;   void
//for keep track stack use  *dstktop ;  void
// Pointer to next task's    struct os_task_control_block *next_tcb;
TCB
                                                  void *tcb_sem_q;
// Next task on semaphore waiting list
} os_tcb;


typedef struct os_sema {
// Semaphore value            count; UWORD
// Semaphore queue *scb_sem_q;      os_tcb
struct os_sema *next_scb;// Pointer next semaphore's SCB
} os_scb;


typedef struct os_mail_box {
// Semaphore for sending to mailbox           *msend;      os_scb
// Semaphore for receiving from            *mrecv;      os_scb
mailbox
// Pointer to message      *mmsg;        void
// Pointer to next mailbox      struct  os_mail_box *next_mcb;
} os_mcb;


typedef struct os_central_table {
// Timer ticks since last boot os_systime;       ULONG
//dispatch counter      os_dispcntr;       ULONG
// Counter for processor idle time      os_idlecntr;       ULONG
// OS version os_version;       UBYTE
// OS running flag os_running;       UBYTE
os_schedrun;    //scheduler flag       UBYTE
// Pointer to used TCBs        os_tcb *tcb_used_list;
// Pointer to TCB freelist        os_tcb *tcb_free_list;
// Pointer to current running                os_tcb *tcb_current;
task TCB
// Pointer to highest priority              os_tcb *tcb_runq;
runnable task TCB
// Pointer to used semaphores  *scb_used_list;       os_scb
// Pointer to semaphores freelist  *scb_free_list;       os_scb
// Pointer to used message          *mcb_used_list;      os_mcb
mailboxes
// Pointer to mailbox freelist         *mcb_free_list;      os_mcb
} os_cent_tbl;


// Kernel services and data-types that are visible to user tasks
typedef os_scb os_semaphore;
```

```
typedef os_mcb os_mailbox;
typedef os_tcb os_task;
extern os_tcb *os_task_create( void (*task)(void), UBYTE *hstack, UBYTE
*dstack, UBYTE p );
extern void os_sleep( UWORD ticks );
extern ULONG os_uptime( void );
extern os_scb *os_sem_create( UWORD ival );
extern void os_wait( os_scb *semaphore );
extern void os_signal( os_scb *semaphore );
extern os_mcb *os_mbox_create( void );
extern void os_mbox_send( os_mcb *mbox, void *msg );
extern void *os_mbox_recv( os_mcb *mbox );
```

rtos_core.c 2-6-6

```
//Core of operating system(  _os_schedule   timer interrupt
_init_free_list os_start
//os_sleep os_uptime   NULLTASK main)  are in this file
//Main function is in this file
// Central table                os_cent_tbl os_ctbl;
// Pointers to Task Control Blocks  os_tcb all_tcb[OS_TASK_MAX];
// Pointers to Semaphore Control     all_scb[OS_SEM_MAX];      os_scb
Blocks
// Pointers to Message Queue Control    all_mcb[OS_MBOX_MAX];      os_mcb
Blocks
// Straight pointer to null-                        os_tcb *NULL_TASK;
process' PCB
// Stacks for NULL task
UBYTE DNULL[OS_TASK_DSTK_SIZE], HWNULL[OS_TASK_HSTK_SIZE];


// Low level function which makes actual context switch
void _os_dispatch( void )
{
UBYTE *dstk_highrdy, *hstk_highrdy;
os_tcb *curr_tcb;
OS_ENTER_CRITICAL;


os_ctbl.os_dispcntr++;   //dispatch counter
  // Get address to            curr_tcb = os_ctbl.tcb_current;
current TCB
dstk_highrdy = os_ctbl.tcb_runq->datastk; // Get highest priority
ready tasks data stack
  // Get highest priority    hstk_highrdy = os_ctbl.tcb_runq->hwstk;
ready task's hw stack
  // Make first in run-   os_ctbl.tcb_current = os_ctbl.tcb_runq;
queue current
os_ctbl.tcb_current->status = OS_TASK_RUNNING;


/*
Context switch.
```

```
*dstk_highrdy -> Y+4           ;
*hstk_highrdy -> Y+2           ;
*curr_tcb -> Y+0           ;


1. Save HW SP and data SP to current task's PCB
2. Load SP from highest priority task's TCB
3. Restore context and return to new task


      */
#asm
; Save all registers except SW stack pointer
R0  PUSH
R1  PUSH
R2  PUSH
R3  PUSH
R4  PUSH
R5  PUSH
R6  PUSH
R7  PUSH
R8  PUSH
R9  PUSH
R10  PUSH
R11  PUSH
R12  PUSH
R13  PUSH
R14  PUSH
R15  PUSH
R16  PUSH
R17  PUSH
R18  PUSH
R19  PUSH
R20  PUSH
R21  PUSH
R22  PUSH
R23  PUSH
R24  PUSH
R25  PUSH
R26  PUSH
R27  PUSH
R30  PUSH
R31  PUSH
R0,SREG          IN
R0  PUSH


; Save current HW stack pointer, low first
; hstk_current -> X           R26,Y+0         LDD
R27,Y+1         LDD


  ; *hstk_current = SP;     R30,SPL          IN
X+,R30          ST
R30,SPH          IN
```

```
X+,R30          ST


; Save current SW stack pointer, low first
R30,R28   ; LOW(Y) -> R30         MOV
; Restore Y value (3 pointers * 2             R30,6  ADIW
bytes each)
; *dstk_current = Y;            X+,R30          ST
X,R29          ST


; Load new HW stack pointer, low first
R30,Y+2   LDD
SPL,R30   OUT
R30,Y+3   LDD
SPH,R30   OUT


; Load new SW stack pointer, low first
R30,Y+4         LDD
R31,Y+5         LDD
R28,R30         MOV
R29,R31         MOV


; Pop all registers except SW stack pointer
R0         POP
SREG,R0         OUT
R31         POP
R30         POP
R27         POP
R26         POP
R25         POP
R24         POP
R23         POP
R22         POP
R21         POP
R20         POP
R19         POP
R18         POP
R17         POP
R16         POP
R15         POP
R14         POP
R13         POP
R12         POP
R11         POP
R10         POP
R9         POP
R8         POP
R7         POP
R6         POP
R5         POP
R4         POP
R3         POP
R2         POP
R1         POP
```

```
R0         POP


RETI
#endasm
}


void _os_schedule( void )
{
os_tcb *tptr, *highest_tcb;
UBYTE highest;


//don't do anything if scheduler is off
if (!(os_ctbl.os_schedrun)) return;


OS_ENTER_CRITICAL;
if (OS_CHCK) //Uses TIMER1!!!!
{
TCCR1B=0;
os_ctbl.tcb_current->ttime++;//=(unsigned long)TCNT1 ;
//Increment current task time
    }
                                                    highest = 255;
// NULL-task's priority
// NULL-task                 NULL_TASK->status = OS_TASK_RUNNABLE;
is always runnable


if( os_ctbl.tcb_current->status & OS_TASK_RUNNING ) // If current
task is running here
os_ctbl.tcb_current->status = OS_TASK_RUNNABLE;   // it is
still runnable


// Go through all          tptr = (os_tcb *)os_ctbl.tcb_used_list;
runnable tasks
                // and get the one having                    do {
highest priority
// Is task           if( OS_TASK_RUNNABLE==tptr->status ) {
runnable? runnable=0
// Yes, is it's        if( tptr->prio <= highest ) {
priority higher than highest allready?
// Yes, save it         highest = tptr->prio;
highest_tcb = (os_tcb *)tptr;
}
}
tptr = (os_tcb *)tptr->next_tcb;
}
                                                    while( tptr );
// Was it last one?
// Put              os_ctbl.tcb_runq = (os_tcb *)highest_tcb;
highest priority task in run-queue
```

```
// If next in          if( os_ctbl.tcb_current != os_ctbl.tcb_runq )
run-queue is different task
                                 _os_dispatch();
// we have to dispatch
    if (OS_CHCK)  //USES timer 1!!!
{
TCNT1=0;
TCCR1B=5;
}
OS_EXIT_CRITICAL;


}


//#pragma savereg-


// Timer 0 output compare interrupt service routine
// Timer tick ISR
interrupt [TIM0_COMP] void timer0_comp_isr(void)
{
volatile os_tcb *tptr;
volatile UBYTE sched_n_disp;


OS_ENTER_CRITICAL;
// Is multitasking on?                if( os_ctbl.os_running ) {
                                           sched_n_disp = 0;
// Flag of need for scheduling and dispatching
// Point to first    tptr = (os_tcb *)os_ctbl.tcb_used_list;
TCB
                                            do {
// Go through all PCBs
if(tptr->status==OS_TASK_DELAYD){  //Is the task
sleeping?
if(tptr->delay>0)            //Still some time
left?
tptr->delay--;          // Yes, decrement
   //no, make                              else {
it runnable
tptr->status = OS_TASK_RUNNABLE;   //make
the task runnable again
sched_n_disp = 1;                  //re-
schedule
}
}
tptr = (os_tcb *)tptr->next_tcb;// Get next TCB
}
                                              while( tptr );
  // Last in PCB chain?


// Increment system                os_ctbl.os_systime++;
uptime
```

```
// If some task became                 if( sched_n_disp ) {
runnable
_os_schedule();                  // we have to re-
schedule
}
}
OS_EXIT_CRITICAL;
}


//#pragma savereg+


void _init_free_list( void )
{
UBYTE i;
os_tcb *tcb_this, *tcb_next;
os_scb *scb_this, *scb_next;
os_mcb *mcb_this, *mcb_next;


OS_ENTER_CRITICAL;
// Process PCBs
// Link all                         for(i=0;i<OS_TASK_MAX-1;i++) {
PCBs in free list
tcb_this = (os_tcb *)&all_tcb[i];
tcb_next = (os_tcb *)&all_tcb[i+1];
tcb_this->next_tcb = (os_tcb *)tcb_next;
}
// Last PCB           all_tcb[OS_TASK_MAX].next_tcb = (os_tcb *)0;
in chain


// Process SCBs
// Link all SCBs in                    for(i=0;i<OS_SEM_MAX-1;i++) {
free list
scb_this = (os_scb *)&all_scb[i];
scb_next = (os_scb *)&all_scb[i+1];
scb_this->next_scb = (os_scb *)scb_next;
}
// Last SCB            all_scb[OS_SEM_MAX].next_scb = (os_scb *)0;
in chain


// Process MCBs
// Link all                         for(i=0;i<OS_MBOX_MAX-1;i++) {
MCBs in free list
mcb_this = (os_mcb *)&all_mcb[i];
mcb_next = (os_mcb *)&all_mcb[i+1];
mcb_this->next_mcb = (os_mcb *)mcb_next;
}
// Last SCB           all_mcb[OS_MBOX_MAX].next_mcb = (os_mcb *)0;
in chain


OS_EXIT_CRITICAL;
}
```

```
void os_start( void )
{
// Is there tasks created? if( os_ctbl.tcb_used_list ) {
// Start multi-tasking os_ctbl.os_running = 1;
}
}


void os_sleep( UWORD ticks )
{
OS_ENTER_CRITICAL;
     // Put task to sleep for  os_ctbl.tcb_current->delay = ticks;
'ticks' time.
os_ctbl.tcb_current->status = OS_TASK_DELAYD;
_os_schedule();
OS_EXIT_CRITICAL;
}


ULONG os_uptime( void )
{
return( os_ctbl.os_systime );
}


void NULLTASK( void )
{
while( 1 );


}




void main(void)
{


// NULL-task's stacks           UBYTE *nt_dstk, *nt_hstk;


///***********************************
//Application dependent setup
//most of this should be moved to task definitions
//(EXECPT for timer 0 setup


// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: 16 MHz
// Mode: Output Compare
//OC0 output: Disconnected
// Timer/Counter 0 is cleared on compare match
```

```
//TCCR0=0b00001101; devide source clk by 1024 and timr to CTC mode
//ASSR=0x00;
TCNT0=0x00;
//The following sets the task-switch rate!!!
//internal clock (16Mhz) divided by 1024 so each tick is 64 microseconds
//64 usec x 15.6 = 1mSec   OCR0=15.6 * OS_TICK_TIME;
//OCR0=78 ; //5 ms
//OCR0=39 ; //2.5 ms
//OCR0=20 ; //1.25 ms
//OCR0=0 ; //960 usec
//BUT don't start clock until OS is started
// Timer(s)/Counter(s) Interrupt(s) initialization
TIMSK=0x02;  //timer 0 compare match


// Initialize PCB & SCB freelist
_init_free_list();


// Set up OS central table
// Not running                    os_ctbl.os_running = 0;
//enable scheduler                   os_ctbl.os_schedrun = 1;
// No created processes           os_ctbl.tcb_used_list = (os_tcb *)0;
yet
os_ctbl.tcb_free_list = (os_tcb *)&all_tcb[0];  // First TCB in free list
// No created semaphores           os_ctbl.scb_used_list = (os_scb *)0;
yet
os_ctbl.scb_free_list = (os_scb *)&all_scb[0];  // First SCB in free list
// No created mailboxes           os_ctbl.mcb_used_list = (os_mcb *)0;
yet
os_ctbl.mcb_free_list = (os_mcb *)&all_mcb[0];  // First MCB in free list


// Allways create NULL task which is allways runnable
// and which has the lowest priority
NULL_TASK = os_task_create( NULLTASK, HWNULL, DNULL, 255 );


// Use NULL-task's stacks prior to first context switch
nt_dstk = (UBYTE *)NULL_TASK->datastk;
nt_hstk = (UBYTE *)NULL_TASK->hwstk;
#asm
; Load new HW stack pointer, low first
R30,Y+0   LDD
SPL,R30   OUT
R30,Y+1   LDD
SPH,R30   OUT


; Load new data stack pointer, low first?
R30,Y+2         LDD
R31,Y+3         LDD
R28,R30         MOV
R29,R31         MOV


#endasm
```

```
os_ctbl.tcb_current = (os_tcb *)NULL_TASK;

///**************************************************
// Create at least one user task
//Modify for task and stack names if nescessary
//os_task_create( task1, hw_stack1, data_stack1, 50 );
os_task_create(OS_FIRST_TASK_NAME, OS_FIRST_TASK_HSTACK_NAME
,OS_FIRST_TASK_DSTACK_NAME, OS_FIRST_TASK_PRIO);
///**************************************************

// Start multitasking
os_start();
TCNT0=0;
TCCR0=0b00001101; //devide source clk by 1024 and timr to CTC mode
_os_schedule();

// This causes program to jump actual NULL process
#asm
R0          POP
SREG,R0          OUT
R31          POP
R30          POP
R27          POP
R26          POP
R25          POP
R24          POP
R23          POP
R22          POP
R21          POP
R20          POP
R19          POP
R18          POP
R17          POP
R16          POP
R15          POP
R14          POP
R13          POP
R12          POP
R11          POP
R10          POP
R9          POP
R8          POP
R7          POP
R6          POP
R5          POP
R4          POP
R3          POP
R2          POP
R1          POP
R0          POP
```

```
RETI


#endasm
while (1)
{
// We won't never, ever reach this point
#asm("nop");
};
}
```

user_task.c 3-6-6

```
os_tcb *os_task_create( void (*task)(void), UBYTE *hstack, UBYTE *dstack,
UBYTE p )
{
UBYTE *stk, *dstk, *hstk;
UBYTE i;
os_tcb *tcb_new;


OS_ENTER_CRITICAL;
tcb_new = (os_tcb *)os_ctbl.tcb_free_list;// Get next free TCB
// TCBs left?                        if( tcb_new ) {
// Set up the hardware stack
hstk = (UBYTE *)hstack + (OS_TASK_HSTK_SIZE - 1);// Stack
begins in the end of array
stk = hstk;
  // Low part of task's address       *stk-- = (UBYTE)(task);
  // High part of task's   *stk-- = (UBYTE)(task >> 8);
address
for(i=0;i<30;i++)                    // Clear all other
registers
*stk-- = 0;
// SREG with interrupts enabled          *stk-- = 0x80;


// Set up data stack
dstk = (UBYTE *)dstack + (OS_TASK_DSTK_SIZE - 1);// Stack
begins in the end of array


os_ctbl.tcb_free_list = (os_tcb *)tcb_new->next_tcb;//
Release TCB from freelist
tcb_new->next_tcb = (os_tcb *)os_ctbl.tcb_used_list;// Link
new TCB into TCB used list
os_ctbl.tcb_used_list = (os_tcb *)tcb_new;
// Load data stack    tcb_new->datastk = (UBYTE *)dstk;
pointer to TCB
// Load hw stack             tcb_new->hwstk = (UBYTE *)stk;
pointer to TCB
//for keep track stk  tcb_new->hstktop = (UBYTE *)hstk;
//for keep track stk   tcb_new->dstktop = (UBYTE *)dstk;
// Make task runnable tcb_new->status = OS_TASK_RUNNABLE;
// Set task's priority           tcb_new->prio = (UBYTE)p;
```

```
// No initial delay          tcb_new->delay = (UWORD)0;
// Initialize semaphore    tcb_new->tcb_sem_q = (os_tcb *)0;
queue
}


OS_EXIT_CRITICAL;
return( tcb_new );
}


///****************************************************
//utility which allows one task to sleep another
//no call to the scheduler is needed here since the
//task is not the current task.
// USE os_sleep if you are sleeping the current task!!


void os_sleep_task(os_tcb *task, UWORD ticks)
{
OS_ENTER_CRITICAL;
task->delay = ticks;
task->status = OS_TASK_DELAYD;
OS_EXIT_CRITICAL;
}


///****************************************************
//utility which allows one task to UN-sleep another
//if the task is dalayed. Task will restart on next tick.


void os_unsleep_task(os_tcb *task)
{
OS_ENTER_CRITICAL;
if (task->status == OS_TASK_DELAYD)
task->delay = 0;
OS_EXIT_CRITICAL;
}


///****************************************************
//utility to allow tasks to suspend each other
//need to schedule if the current task suspends itself
void os_suspend_task(os_tcb *task)
{
OS_ENTER_CRITICAL;
if (task->status<2)  //running or runnable
task->status = OS_TASK_SUSPENDED;
else if (task->status==OS_TASK_WAITING)
task->status = OS_TASK_SUSPENDED_WAIT;
    else if (task->status==OS_TASK_DELAYD)
task->status = OS_TASK_SUSPENDED_DELA;
(os_ctbl.tcb_current==task)   if
_os_schedule();
OS_EXIT_CRITICAL;
}
```

```
///*****************************************************
//utility to allow tasks to resume another task
//need to reschedule IF the task has priority
void os_resume_task(os_tcb *task)
{
char change ;
OS_ENTER_CRITICAL;
change=0;
if (task->status == OS_TASK_SUSPENDED){
change=1;
task->status = OS_TASK_RUNNABLE;
}
else if (task->status == OS_TASK_SUSPENDED_WAIT){
change=1;
task->status = OS_TASK_WAITING;
}
else if (task->status == OS_TASK_SUSPENDED_DELA){
change=1;
task->status = OS_TASK_DELAYD;
}
//reschedule if reactivated task has priority
if (change==1 && task->prio < os_ctbl.tcb_current->prio)
_os_schedule();
OS_EXIT_CRITICAL;
}
///*****************************************************
//Find the high point of a task
//hardware stack use
int os_task_hstk_chck(os_tcb *task)
{
char *hbottom  ;
int free ;
OS_ENTER_CRITICAL;
hbottom = (char *)task->hstktop - OS_TASK_HSTK_SIZE + 1 ;
while (*hbottom++ == 0) free++;
return free;
OS_EXIT_CRITICAL;


}
///*****************************************************
//Find the high point of a task
//hardware stack use
int os_task_dstk_chck(os_tcb *task)
{
char *dbottom  ;
int free ;
OS_ENTER_CRITICAL;
dbottom = (char *)task->dstktop - OS_TASK_DSTK_SIZE + 1 ;
while (*dbottom++ == 0) free++;
return free;
OS_EXIT_CRITICAL;


}
```

```
///****************************************************
//get the task time
long os_task_gettime(os_tcb *task)
{
OS_ENTER_CRITICAL;
task->ttime ;     return
    OS_EXIT_CRITICAL;
}


///****************************************************
//set the task time
void os_task_settime(os_tcb *task, ULONG time)
{
OS_ENTER_CRITICAL;
task->ttime = time ;
OS_EXIT_CRITICAL;
}


///****************************************************
//get the dispatch counter
long os_getdispatch()
{
OS_ENTER_CRITICAL;
os_ctbl.os_dispcntr ;     return
    OS_EXIT_CRITICAL;
}


///****************************************************
//set the dispatch counter
void os_setdispatch(ULONG count)
{
OS_ENTER_CRITICAL;
os_ctbl.os_dispcntr = count ;
OS_EXIT_CRITICAL;
}


///****************************************************
//get a task's status
UBYTE os_task_getstatus(os_tcb *task)
{
    OS_ENTER_CRITICAL;
task->status ;     return
    OS_EXIT_CRITICAL;
}


///****************************************************
//lock scheduler
void os_sched_lock(){
    OS_ENTER_CRITICAL;
    os_ctbl.os_schedrun = 0 ;
    OS_EXIT_CRITICAL;
}
```

```
///*****************************************************
//UNlock scheduler
void os_sched_unlock(){
    OS_ENTER_CRITICAL;
    os_ctbl.os_schedrun = 1 ;
    OS_EXIT_CRITICAL;}
```

Rtos_semaphore.c 4-6-6

```
os_scb *os_sem_create( UWORD init_val )
{
os_scb *scb_new;


OS_ENTER_CRITICAL;
 // Get next             scb_new = (os_scb *)os_ctbl.scb_free_list;
free SCB
 // SCBs left?                                   if( scb_new ) {
os_ctbl.scb_free_list = (os_scb *)scb_new->next_scb;//
Release SCB from freelist
scb_new->next_scb = (os_scb *)os_ctbl.scb_used_list;// Link
new SCB into SCB used list
os_ctbl.scb_used_list = (os_scb *)scb_new;
// Set semaphore's                 scb_new->count = init_val;
initial value
// No task pending          scb_new->scb_sem_q = (os_tcb *)0;
on semaphore
}
OS_EXIT_CRITICAL;
return( scb_new );
}


/*
About wait(), signal() and HW locking:


Since we are in uniprocessor environment, indivisibility is
achieved by simply disabling interrupts. After that there is
no way for task to loose the processor. Otherwise somekind of
hardware locking would be needed. In multi-processor
environments
possibilities are 'test-and-set'-instruction or instruction
to
interchange two memory locations.
*/


//=================================================================
```

```
void os_wait( os_scb *sem )
{
os_tcb *ptcb;


OS_ENTER_CRITICAL;
// Semaphore value >0 ?     {  if( sem->count )
// Yes, decrement the value      sem->count--;
}
else {                          // No, we have to wait
ptcb = (os_tcb *)sem->scb_sem_q;
// Any tasks in queue?      if( ptcb ) {
while( ptcb->tcb_sem_q ) {// Yes, find queue's end
because
ptcb = (os_tcb *)ptcb->tcb_sem_q;// this is a
FIFO, not a stack.
}
ptcb->tcb_sem_q = (os_tcb *)os_ctbl.tcb_current;// Link
current process to wait queue
}
else {
// No,  sem->scb_sem_q = (os_tcb *)os_ctbl.tcb_current;
this is the first one
}
os_ctbl.tcb_current->tcb_sem_q = (os_tcb *)0; // Terminate
queue properly
os_ctbl.tcb_current->status = OS_TASK_WAITING;// Now we're
waiting
_os_schedule();                              // Re-schedule
run-queue
}
OS_EXIT_CRITICAL;
}


//==============================================================
void os_signal( os_scb *sem )
{
os_tcb *ptcb;


OS_ENTER_CRITICAL;
ptcb = (os_tcb *)sem->scb_sem_q;
// Is there pending                              if( ptcb ) {
wait()s?
sem->scb_sem_q = (os_tcb *)ptcb->tcb_sem_q;  // Yes, dequeue
the first one
     // Make task     ptcb->status = OS_TASK_RUNNABLE;
runnable
     // Re-schedule run-                       _os_schedule();
queue
}
// No, just increment the                           else {
semaphore
sem->count++;
}
OS_EXIT_CRITICAL;
```

```
}

//==================================================================
//semaphore accept
UWORD os_sem_accept( os_scb *sem )
{
    UWORD temp;
    OS_ENTER_CRITICAL;
    temp = sem->count;
    if (temp>0) sem->count--;
    return temp;
    OS_EXIT_CRITICAL;
}
```

rtos_mbox.c 5-6-6

```
os_mcb *os_mbox_create( void )
{
*mcb_new;      os_mcb


mcb_new = 0;
OS_ENTER_CRITICAL;
mcb_new = (os_mcb *)os_ctbl.mcb_free_list;// Save pointer to next
free mcb
// mcbs left?                      if( mcb_new ) {
//initilize mailbox          mcb_new->mmsg = NULL;
// Yes,   os_ctbl.mcb_free_list = (os_mcb *)mcb_new->next_mcb;
release mcb from freelist
// Link   mcb_new->next_mcb = (os_mcb *)os_ctbl.mcb_used_list;
new mcb into mcb used list
os_ctbl.mcb_used_list = (os_mcb *)mcb_new;
//      = (os_scb *)os_sem_create( 0 );         mcb_new->mrecv
Semaphore for receiving messages
//      = (os_scb *)os_sem_create( 1 );         mcb_new->msend
Semaphore for sending messages
// Clear       = (void *)0;                 mcb_new->mmsg
message pointer
}
OS_EXIT_CRITICAL;
return( mcb_new );
}


//======================================================================
// Send a message to mailbox
void os_mbox_send( os_mcb *mcb, void *msg )
{
// Is there room in the mailbox    os_wait( (os_scb *)mcb->msend );
// Put message in the mailbox               mcb->mmsg = msg;
// Signal the queue's receive   os_signal( (os_scb *)mcb->mrecv );
semaphore
}
```

```
//===================================================================
// Receive a message from mailbox
void *os_mbox_recv( os_mcb *mcb )
{
void *msg;

os_wait( (os_scb *)mcb->mrecv );// Wait for a message
msg = mcb->mmsg;                // Get the message from mailbox
//and wipe the old message       mcb->mmsg = NULL;
// Signal the queue's sending   os_signal( (os_scb *)mcb->msend );
semaphore

return( msg );
}

//===================================================================
//receive a message without waiting
//You MUST check the retrun value to see if it is NULL
void *os_mbox_accept( os_mcb *mcb )
{
    void *msg;
    OS_ENTER_CRITICAL;
// Get the message from mailbox     msg = mcb->mmsg;
//if there is a message                          if (msg!=NULL) {
 mcb->mmsg = NULL ;         //clear the mailbox
// Signal the sending semaphore  os_signal( (os_scb *)mcb->msend );
    }
    return msg;
    OS_EXIT_CRITICAL;
}
```

user_task.c 6-6-6

```
//definition of task are in this file
#include <mega32.h>
#include <stdio.h>

#include "rtos.h"
#include "user_tasks.h"
#include "rtos_core.c"
#include "rtos_task.c"
//for impelement semaphores    #include "rtos_semaphore.c"
//for impelement mailboxes        #include "rtos_mbox.c"
//for impelement UART        #include "rtos_uart.c"

//global variables for all tasks

OS_GLOBAL_VAR //force the data to be in RAM using pragma
```

```
os_mailbox *task23_pipe ;     //string from task2 to task3
os_mailbox *task11_pipe ;     //string from task1 to itself
//shared memory task 1, 3, 4
UWORD sleep_time=10  ;
//memory lock semaphore
os_semaphore *sem_memory;
//task control block pointer
os_tcb *t1, *t2, *t3, *t4    ;
//allow register vars again
OS_GLOBAL_END
bit flag=1;
bit key_flag=1;
//*********************************************************
void task1( void )
{
char t1str[32];
char t1astr[20];
char*msg ;           //
     char msg[5]="Ali" ;
//get the current task control block
t1 = os_ctbl.tcb_current;


//start 3 tasks and retrun control block pointers
t2=os_task_create( task2, hw_stack2, data_stack2, 100 );
t3=os_task_create( task3, hw_stack3, data_stack3, 150 );
t4=os_task_create( task4, hw_stack4, data_stack4, 200 );


//allocate UART structures, turn on UART
//parameter is baud rate given as an integer
//This routine uses the clock value set in the
//Project...Config dialog!!
os_init_uart(9600);
//init portA to output
DDRA = 0xff ;
PORTA = 0xff;
//init portD to input
DDRD.6 =0;
PORTD.6 =1;


while(1)
{
PORTA.0 ^= 1;
os_sleep(1);


if (PIND.6==0 && key_flag==1){
sprintf(t1astr,"Uptime: %d\r\n",os_uptime()) ;
     os_puts(t1astr);
                     flag=!flag;
                     key_flag=0;
}


if (PIND.6==1 && key_flag==0)
```

```
key_flag=1;
        msg[0]=getchar();


   if(msg[0]=='2')
{
PORTA.2=0;
os_suspend_task(t2);
}
if(msg[0]=='3')
{
PORTA.4=0;
os_suspend_task(t3);
}
if(msg[0]=='4')
{
PORTA.6=0;
os_suspend_task(t4);
}






   if(msg[0]=='6')
os_resume_task(t2);
if(msg[0]=='7')
os_resume_task(t3);
if(msg[0]=='8')
os_resume_task(t4);
if(msg[0]=='9')
                       {

sprintf(t1astr,"\r\nUptime:%d\r\n",os_uptime()%65000) ;
        os_puts(t1astr);


sprintf(t1astr,"Task1
time:%d\r\n",os_task_gettime(t1)%65000) ;
        os_puts(t1astr);


sprintf(t1astr,"Task2
time:%d\r\n",os_task_gettime(t2)%65000) ;
        os_puts(t1astr);


sprintf(t1astr,"Task3
time:%d\r\n",os_task_gettime(t3)%65000) ;
        os_puts(t1astr);


sprintf(t1astr,"Task4
time:%d\r\n",os_task_gettime(t4)%65000) ;
        os_puts(t1astr);
```

```
}
               msg[0]='p';


}
}


//*********************************************************
void task2( void )
{
while(1)
{
//toggle the led
PORTA.2 ^= 1;
os_sleep( sleep_time );
}
}


//*********************************************************
void task3( void )
{
while(1)
{
//toggle the led
PORTA.4 ^= 1;
os_sleep( 2*sleep_time );
}
}


//*********************************************************
void task4( void )
{
while(1)
{


//toggle the led
PORTA.6 ^= 1;
os_sleep( 3*sleep_time );
}
}


//*********************************************************
```

user_task.h 7-6-6

```
//Portotype of user task are in this file
//Code of user tasks are in user_task.c file
//Need this to identify the first task to os_core.c
#define OS_FIRST_TASK_NAME task1
#define OS_FIRST_TASK_DSTACK_NAME data_stack1
#define OS_FIRST_TASK_HSTACK_NAME hw_stack1
#define OS_FIRST_TASK_PRIO     240 //priority


//user defined names for tasks
//The task names will be used during calls
//to os_task_create
//NOTE that the creation of task1 is done in os_core.c
void task1(void);
void task2(void);
void task3(void);
void task4(void);


//stack definitions for eaxh task
//The stack names will be used during calls
//to os_task_create
UBYTE data_stack1[OS_TASK_DSTK_SIZE], hw_stack1[OS_TASK_HSTK_SIZE];//
Stacks for task1
UBYTE data_stack2[OS_TASK_DSTK_SIZE], hw_stack2[OS_TASK_HSTK_SIZE];//
Stacks for task2
UBYTE data_stack3[OS_TASK_DSTK_SIZE], hw_stack3[OS_TASK_HSTK_SIZE];//
Stacks for task3
UBYTE data_stack4[OS_TASK_DSTK_SIZE], hw_stack4[OS_TASK_HSTK_SIZE];//
Stacks for task4
```

rtos_uart.c 8-6-6

```
// UART communication code


#pragma regalloc-      //need this to ensure no reg variables
char uart_recv_index ;
char uart_send_index ;
char* uart_send_ptr ;
char* uart_recv_ptr ;
// Semaphore for writing UART  os_semaphore *sem_uart_xmit;
// Semaphore for reading UART  os_semaphore *sem_uart_recv;
#pragma regalloc+


//*********************************************************
//macros to package up semaphores and hide interrupts
#define os_puts(str) \\
do{               \\
os_wait(sem_uart_xmit); \\
uart_send_index = 0; \\
uart_send_ptr = &str[0]; \\
```

```
if (str[0]>0)    \\
{       \\
putchar(str[0]); \\
UCSRB.5=1;  \\
os_wait(sem_uart_xmit); \\
}  ;     \\
os_signal(sem_uart_xmit); \\
}while(0)


 #define os_gets(str)  \\
do{               \\
os_wait(sem_uart_recv);\\
uart_recv_index=0; UCSRB.7=1;\\
uart_recv_ptr = str;  \\
os_wait(sem_uart_recv);\\
os_signal(sem_uart_recv);\\
}while(0)


//*********************************************************
//define the structures and turn on the UART
void os_init_uart(long baud)
{
sem_uart_xmit = (os_semaphore *)os_sem_create( 1 );
sem_uart_recv = (os_semaphore *)os_sem_create( 1 );
UCSRB=0x18; // activate UART
//uses the clock freq set in the config dialog box
UBRRL=0x19;// _MCU_CLOCK_FREQUENCY_ /(baud*16L) - 1L;
}


//receive ISR
///*********************************************************
//UART character-ready ISR
interrupt [USART_RXC] void uart_rec(void)
{
char r_char;
r_char=UDR;   //get a char
//build the input string
if ((r_char!='\r') && (r_char!='\n'))
uart_recv_ptr[uart_recv_index++] = r_char;
else
{
putchar('\n');  //use putchar to aoid overwrite    //
uart_recv_ptr[uart_recv_index] = 0x00 ; //zero terminate
UCSRB.7=0; //stop ISR after <CR>
os_signal(sem_uart_recv);
}
}


//*********************************************************
//UART xmit-empty ISR
interrupt [USART_DRE] void uart_send(void)
{
char t_char;
```

```
t_char = uart_send_ptr[++uart_send_index];
if (t_char == 0)
{
UCSRB.5=0; //kill isr
os_signal(sem_uart_xmit);
}
else
{
putchar(t_char);     //send the char
}
}
//********************************************************
```

7 روش استفاده از سیستم‌عامل

برای استفاده از سیستم‌عامل بلادرنگ ارائه شده کافی است مطابق روش ذکر شده در ضمیمهٔ 2، یک پروژه در نرم‌افزار CodevisionAVR ایجاد گردد و تنظیمات ذکر شده انجام پذیرد. اگر کاربر بخواهد سیستم‌عامل فوق را برای MEGA32 استفاده کند کافی است، دو فایل user_tasks.h و user_tasks.c را مطابق منظور خودش بازنویسی کند. در فایل user_tasks.h تعریف کلی task ها انجام می‌شود و در فایل user_tasks.c کد مربوط به آن taskها نوشته می‌شود. همچنین اندازهٔ استک مورد نیاز برای هر task در فایل user_tasks.h باید مشخص گردد.

اگر بخواهید سیستم‌عامل را برای تراشهٔ دیگری از خانوادهٔ AVR استفاده کنید فقط وقفهٔ مربوط به تایمر و تنظیمات آن باید مطابق با تراشهٔ انتخاب شده پیکربندی شود. در صورتی که تایمر صفر آن تراشه را بتوان در حالت compre mach IRQ و CTC برنامه‌ریزی کرد، نیازی به تغییر وقفه نیز نمی‌باشد و فقط در فایلها به جای include کردن فایل mega32.h باید فایل megaxx.h را include کرد.

اگر بخواهید سیستم‌عامل را برای میکرو کنترلرهای دیگری بجز AVR استفاده کنید، قسمتهایی از فایل rtos_core.c را که به زبان اسمبلی نوشته شده است باید عوض شوند و همچنین برنامه‌ریزی وقفهٔ تایمر نیز مطابق با آن میکروکنترلر انجام شود. البته سرویس روتین وقفه تغییری نمی‌کند.

8 ضمیمهٔ 1 (مشخصات Atmega32)

امکانات کلی این تراشه به صورت زیر است:

32 مگابایت حافظهٔ فلش برای ذخیره سازی برنامه

یک کیلو بایت حافظهٔ EEPROM

دو کیلو بایت حافظهٔ SRAM

امکانات امنیتی برای حفظ برنامه‌ها

پشتیبانی از استاندارد Jtag

دو تایمر هشت بیتی

یک تایمر شانزده بیتی

چهار کانال PWM

هشت مبدل آنالوگ به دیجیتال 10 بیتی

امکانات سریال TWI، USART، SPI،

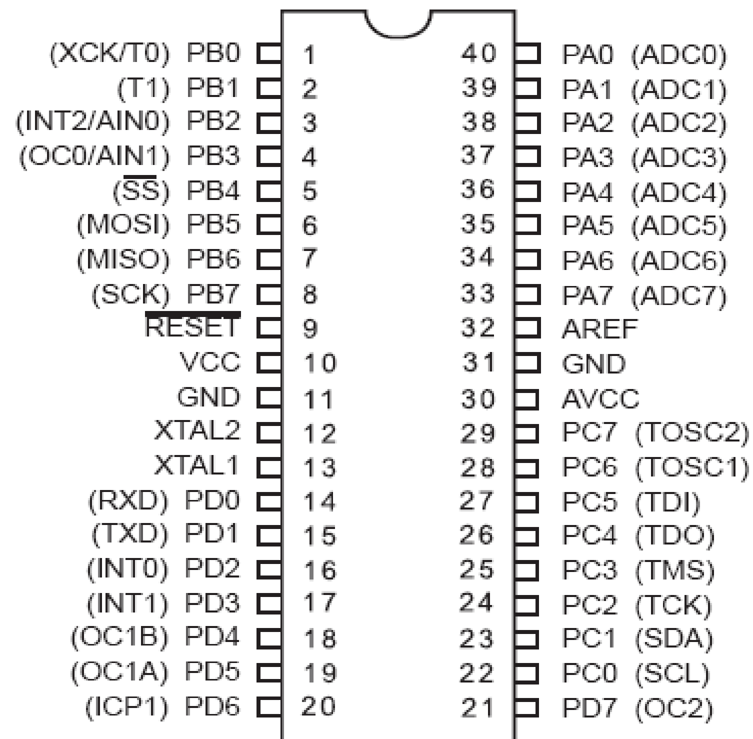
شمارندهٔ Watch dog

مقایسه کنندهٔ آنالوگ

امکان Power on reset

کریستال داخلی

در شکل 3 پایه‌های این تراشه نمایش داده شده است. برای راه اندازی این تراشه نیازی به هیچ قطعهٔ جانبی نیست و اتصال پایه‌های تغذیه کفایت می‌کند.



شکل 3: **پایه‌های تراشهٔ Atmega32**

9 ضمیمهٔ 2 (نحوهٔ کامپایل و اجرا)

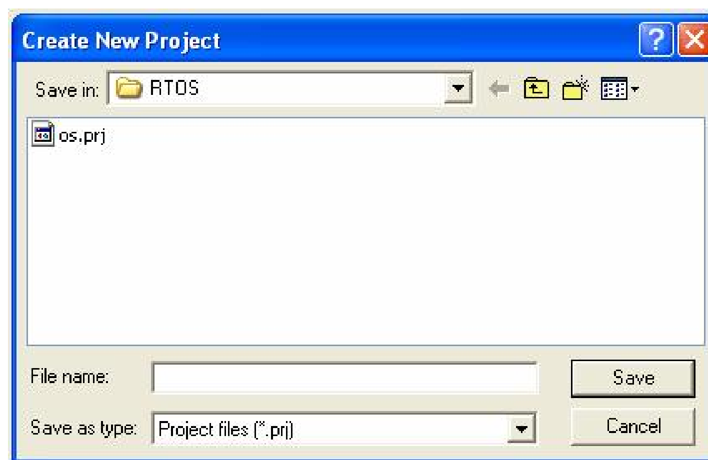
برای کامپایل کردن سیستم عامل باید ابتدا توسط نرم افزار CodeVisionAVR یک پروژه ایجاد کرد. برای اینکار باید از منوی فایل گزینهٔ New را انتخاب کرد سپس در پنجرهٔ ظاهر شده گزینهٔ Project را انتخاب کنید(شکل 4).

شکل 4: نقشهٔ حافظهٔ میکروکنترهای AVR

بعد از انتخاب این گزینهٔ پنجرهٔ زیر ظاهر می‌شود که باید گزینهٔ No انتخاب شود(شکل 5).

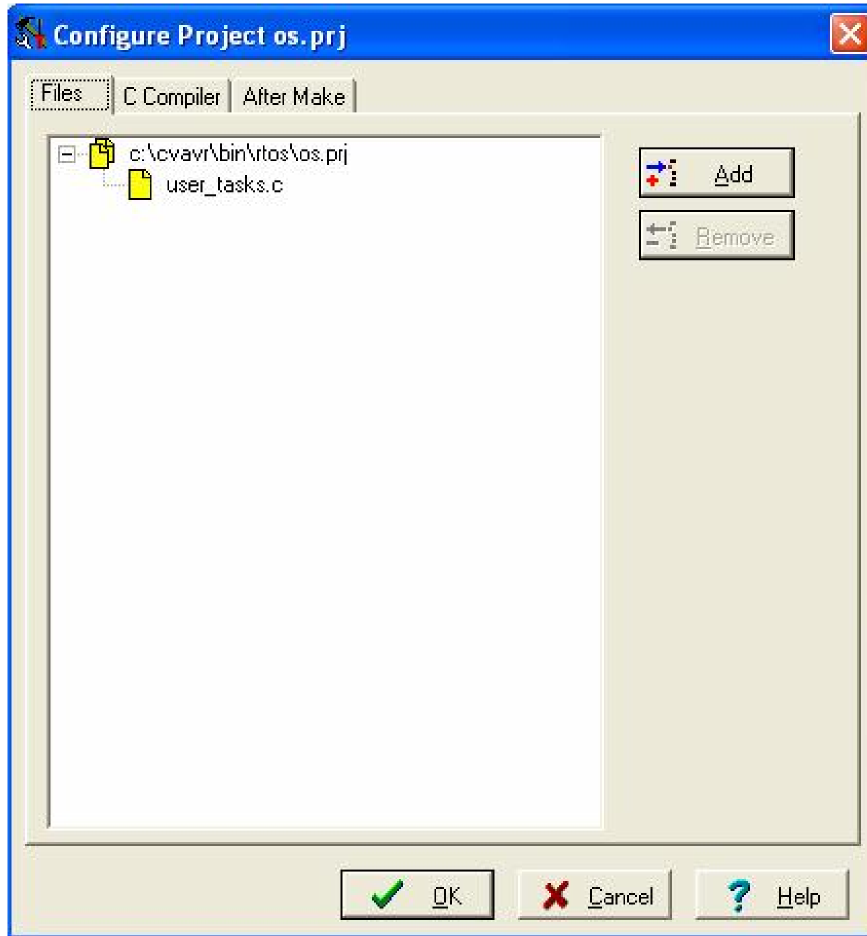
شکل 5: نقشهٔ حافظهٔ میکروکنترهای AVR

در مرحلهٔ بعدی نام پروژه را تایپ کنید، دقت شود که پروژه در فولدری ذخیره شود که سورس فایلهای سیستم عامل قرار دارند(شکل 6)



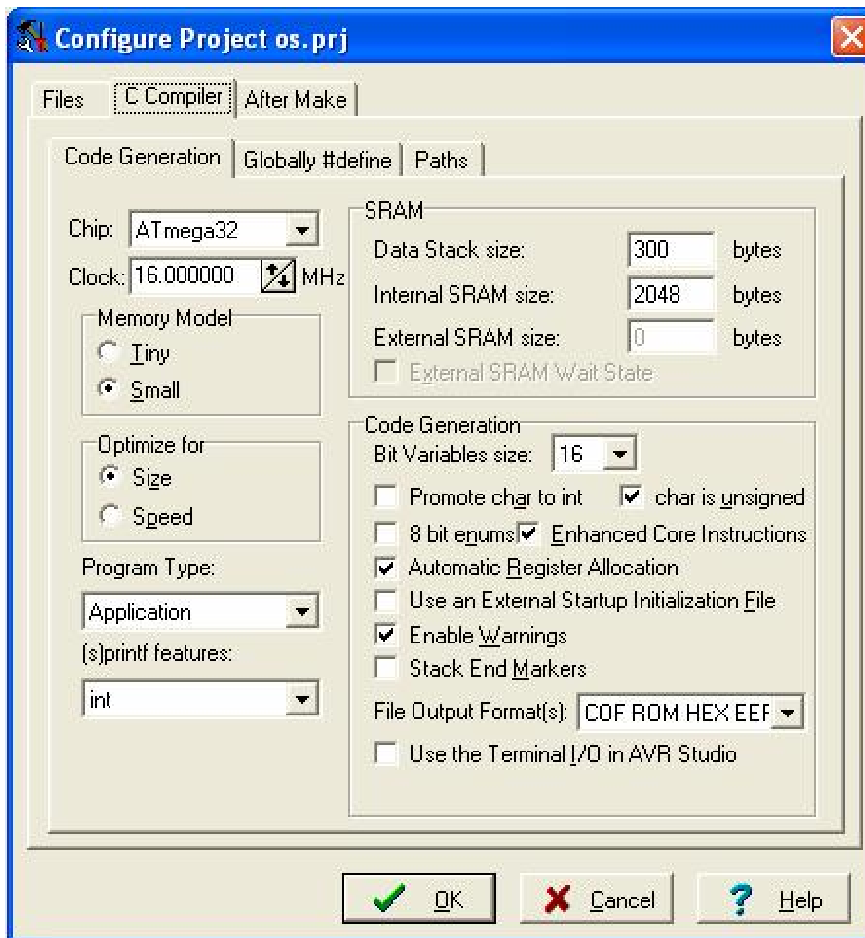
شکل 6: نقشهٔ حافظهٔ میکروکنترهای AVR

بعد از ذخیره شدن پروژه پنجرهٔ زیر ظاهر می‌شود. در قسمت Files با زدن دکمهٔ Add ، فایل user_tasks.c را انتخاب کنید. دقت شود که بقیهٔ فایلها نباید اضافه شوند(شکل 7).



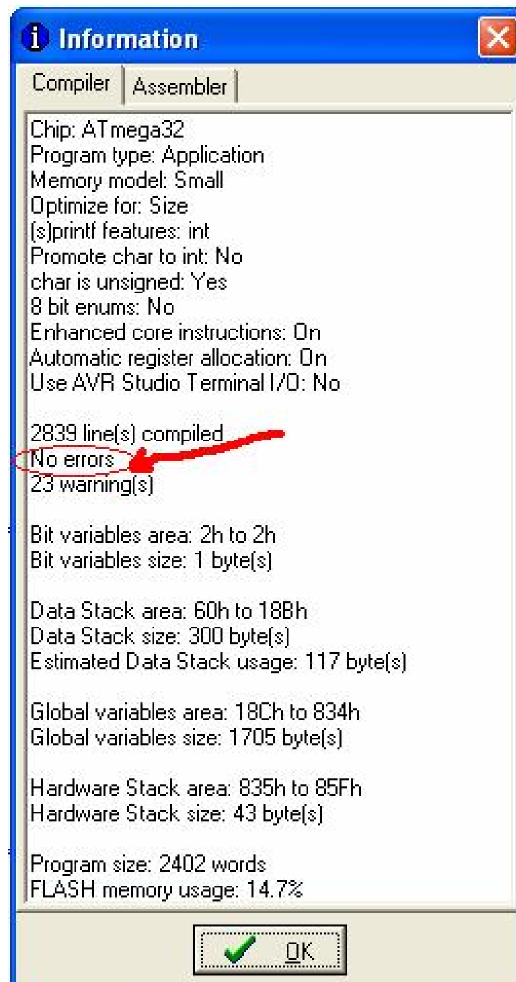
شکل 7: نقشهٔ حافظهٔ میکروکنترلرهای AVR

در پنجرهٔ قبلی قسمت C Compiler را انتخاب کنید و سپس تنظیمات شکل زیر را وارد کنید. گزینه‌هایی که باید تغییر کنند عبارتند از: نام چیپ، ATmega32 انتخاب شود. Clock، 16 انتخاب شود و در قسمت Data Stack size عدد 300 وارد کنید(شکل 8).



شکل 8: نقشهٔ حافظهٔ میکروکنترلرهای AVR

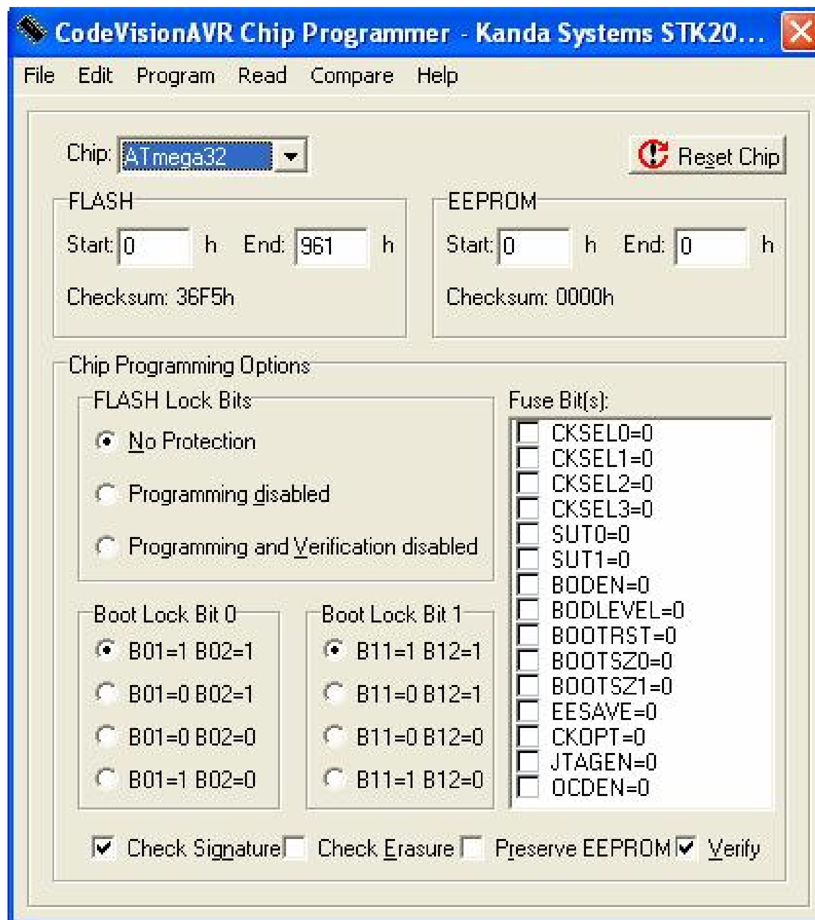
بعد از انجام تنظیمات فوق باید برنامه را کامپایل کرد، برای این منظور باید از منوی Project گزینهٔ make را انتخاب کنید. در صورتی که اشکالی در برنامه نباشد پنجره‌ای به صورت زیر ظاهر می‌شود در این پنجره باید No Error نوشته شده باشد(شکل 9).



شکل 9: **نقشهٔ حافظهٔ میکروکنترهای AVR**

برای پراگرام کردن باید ابتدا در منوی setting گزینهٔ Programmer را انتخاب کنید و سپس در پنجرهٔ ظاهر شده، نوع پراگرامر را انتخاب کنید.

سپس از منوی tools گزینهٔ Chip programmer را انتخاب کنید تا پنجرهٔ زیر ظاهر شود(شکل 10).



شکل 10: نقشهٔ حافظهٔ میکروکنترهای AVR

در پنجرهٔ ظاهر شده از منوی Program گزینهٔ Erase چیپ را انتخاب کنید تا حافظهٔ میکرو را پاک کند، بعد از اینکار از همان منو گزینهٔ Flash را انتخاب کنید تا میکرو پراگرام شود.

میکروکنترلر با استفاده از لینک سریال به کامپیوتر متصل است. در طرف کامپیوتر یک برنامه به زبان C نوشته شده است سورس این فایل در ادامه آورده شده است

9-1 سورس فایل rtos.c

این فایل روی کامپیوتر اجرا می شود. این برنامه قادر است 3 کار متفاوت انجام دهد که عبارتند از معلق کردن یا را اندازی مجدد taskها.همچنین این برنامه قادر است اطلاعات آماری مربوط به taskها را از میکرو دریافت کند و نمایش دهد. در صورتی که task ی معلق شود، LED مربوط به آن روی بورد خاموش می‌شود.

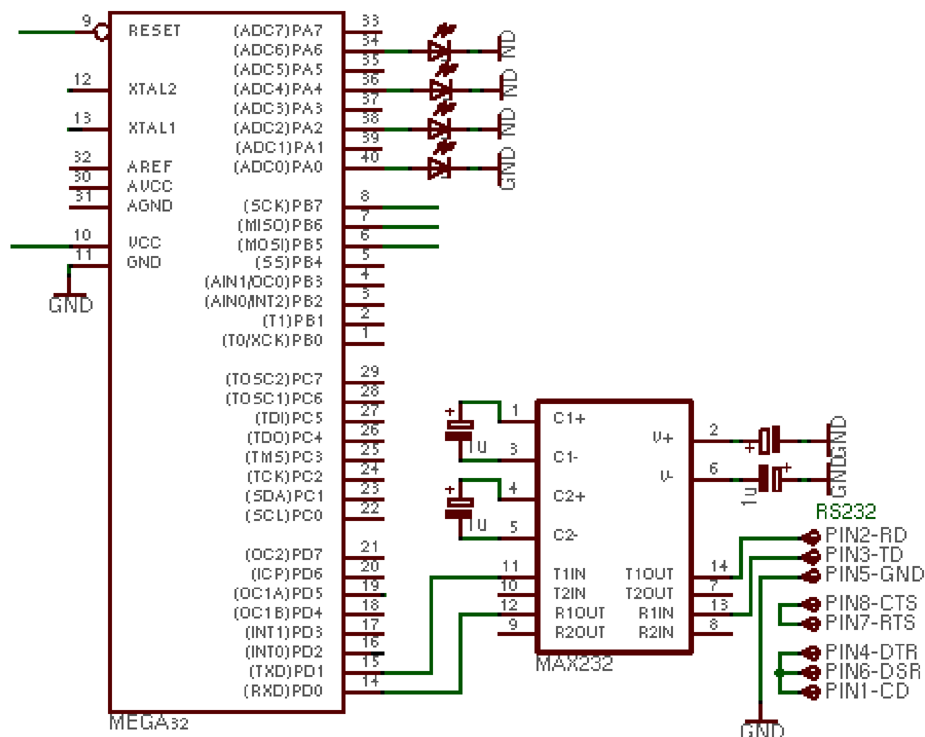
```
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define PORT1 0x3F8  /* Port Address Goes Here */
#define INTVECT 0x0C /* Com Port's IRQ here (Must also change PIC
setting) */
  /* Defines Serial Ports Base Address */
  /* COM1 0x3F8                        */
       */                /* COM2 0x2F8
       */                /* COM3 0x3E8
       */                /* COM4 0x2E8
int bufferin = 0;
int bufferout = 0;
char ch;
char buffer[1025];
void InitPort(void);
void ClosePort(void);
void ClosePort(void);
void GetData(void);
void interrupt PORT1INT(void);
void interrupt (*oldport1isr)();


void main(void)
{
 int c;
 int i;
 int ch;
// printf("\n Press ESC to quit \n");
InitPort();
 c=1;
 do {
 GetData();
     if(c!=0)
     {
     clrscr();
     window(1,10,80,25);
     cprintf("1.Suspend process\r\n");
     cprintf("2.Resume  process\r\n");
     cprintf("3.Get process Information \r\n");
     cprintf("Enter your choice=>");
     }
     c=0;
     if (kbhit())
 c = getch();
     switch(c-'0'){
     case 1 :
      cprintf("\r\n\r\nEnter Process Number(2 or 3 or 4)=>");
      ch=getch();
      ch=((ch-1)%4)+1;
      outportb(PORT1,'0'+ch);
      break;
```

```
      case 2 :
       cprintf("\r\n\r\nEnter Process Number(2 or 3 or 4)=>");
       ch=getch();
       ch=((ch-1)%4)+5;
       outportb(PORT1,'0'+ch);
       break;
      case 3 :
       clrscr();    //
       while(!kbhit())    //
       {    //
       outportb(PORT1,'9');
       for(i=0;i<=10000;++i) GetData();    //
       delay(700);    //
       getch();    //
       }    //
       break;
      }
  outportb(PORT1, c);}    //


//     if (kbhit()){c = getch();
  outportb(PORT1, c);}          //


    } while (c !=27);


ClosePort();


}


void GetData(void)
{
 //    cprintf("A ");
     if (bufferin != bufferout){ch = buffer[bufferout];
bufferout++;
if (bufferout == 1024) {bufferout = 0;}
cprintf("%c",ch);
/*
if(ch=='\n')
{
buffer[bufferin+2] = 0;
bufferin = 0;
cprintf("%s",buffer);
bufferout = 0;
ch=0;


}
*/
     }
}
```

```
void ClosePort()
{
 outportb(PORT1 + 1 , 0);       /* Turn off interrupts - Port1 */
 outportb(0x21,(inportb(0x21) | 0x10));  /* MASK IRQ using PIC */
 /* COM1 (IRQ4) - 0x10  */
 /* COM2 (IRQ3) - 0x08  */
 /* COM3 (IRQ4) - 0x10  */
 /* COM4 (IRQ3) - 0x08  */
 setvect(INTVECT, oldport1isr); /* Restore old interrupt vector */
}
void InitPort()
{
 outportb(PORT1 + 1 , 0);        /* Turn off interrupts - Port1 */
 oldport1isr = getvect(INTVECT); /* Save old Interrupt Vector of later
    recovery */
 setvect(INTVECT, PORT1INT);     /* Set Interrupt Vector Entry */
 /* COM1 - 0x0C */
 /* COM2 - 0x0B */
 /* COM3 - 0x0C */
 /* COM4 - 0x0B */


 /*         PORT 1 - Communication Settings         */
 outportb(PORT1 + 3 , 0x80);  /* SET DLAB ON */
 outportb(PORT1 + 0 , 0x30);  /* Set Baud rate - Divisor Latch Low Byte
*/
      /* Default 0x03 =  38,400 BPS */
      /*         0x01 = 115,200 BPS */
      /*         0x02 =  57,600 BPS */
      /*         0x06 =  19,200 BPS */
      /*         0x0C =   9,600 BPS */
      /*         0x18 =   4,800 BPS */
      /*         0x30 =   2,400 BPS */
 outportb(PORT1 + 1 , 0x00);  /* Set Baud rate - Divisor Latch High Byte
*/
 outportb(PORT1 + 3 , 0x03);  /* 8 Bits, No Parity, 1 Stop Bit */
 outportb(PORT1 + 2 , 0xC7);  /* FIFO Control Register */
);  /* Turn on DTR, RTS, and OUT2 */     outportb(PORT1 + 4 , 0x0B
 outportb(0x21,(inportb(0x21) & 0xEF));  /* Set Programmable Interrupt
Controller */
 /* COM1 (IRQ4) - 0xEF  */
 /* COM2 (IRQ3) - 0xF7  */
 /* COM3 (IRQ4) - 0xEF  */
 /* COM4 (IRQ3) - 0xF7  */
 outportb(PORT1 + 1 , 0x01);  /* Interrupt when data received */
}
void interrupt PORT1INT()  /* Interrupt Service Routine (ISR) for PORT1
*/
{
 int c;
 do { c = inportb(PORT1 + 5);
      if (c & 1) {buffer[bufferin] = inportb(PORT1);
  bufferin++;
  if (bufferin == 1024) {bufferin = 0;}}
    }while (c & 1);
 outportb(0x20,0x20);
}
```

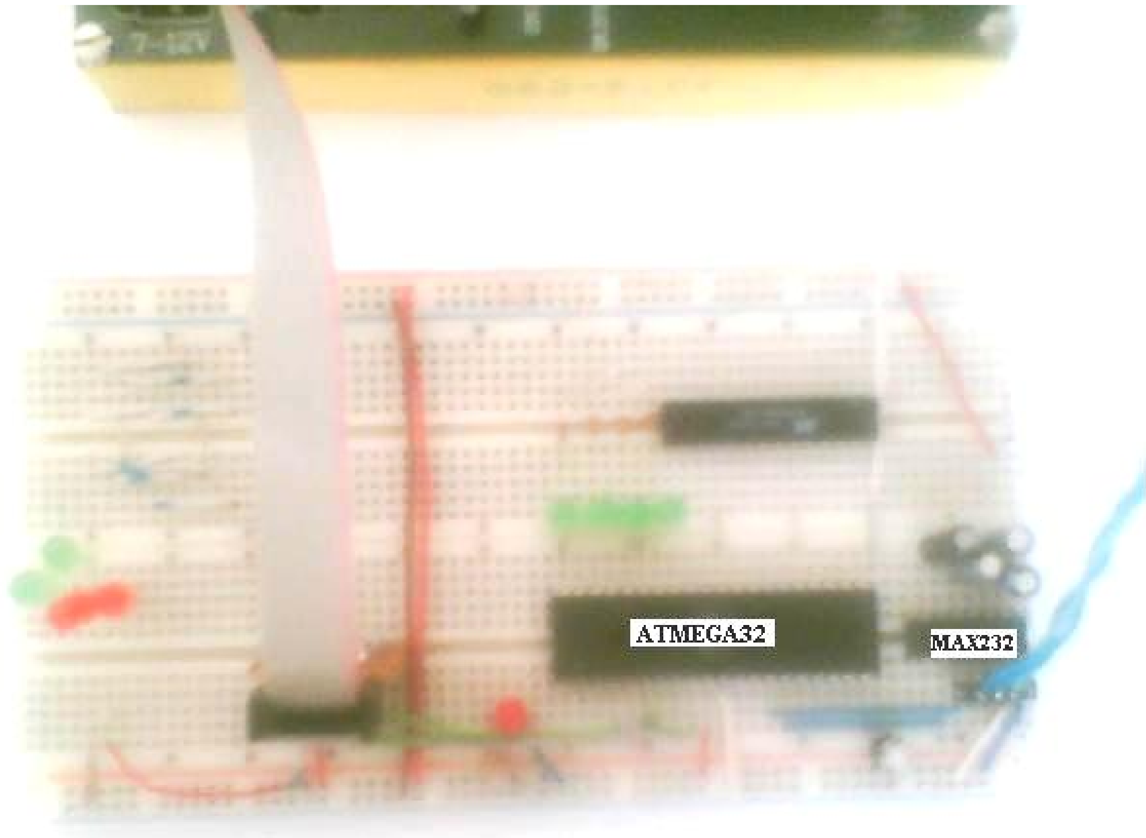
2-9 پیاده‌سازی سیستم‌عامل روی سخت‌افزار

برای پیاده سازی سیستم‌عامل از یک تراشهٔ ATMEGA32 استفاده شد، این تراشه با استفاده از کابل ISP پراگرام می‌شود. در taskهایی که برای این نمونه در نظر گرفته شد، چهار task می‌باشند که چهار LED برای هریک از این taskها در نظر گرفته شده است که نحوهٔ روشن و خاموش شدن آنها در قسمت قبل آورده شده است. برای اینکه یک محیط برای تست سیستم‌عامل فراهم شود،تراشهٔ ATMEGA32 با استفاده از یک تراشهٔ MAX232 به پورت سریال کامپیوتر متصل شده است تا بتواند با کامپیوتر داده تبادل نماید. در شکل 11، شماتیک این مدار نمایش داده شده است.



شکل 11: شماتیک سخت افزار مورد نیاز برای تست سیستم عامل

در شکل 12، تصویری از مدار شکل 11 که روی برد بورد بسته شده است و برای آزمایش استفاده گردید نمایش داده شده است.



شکل 12: تصویری از پیاده‌سازی سیستم‌عامل روی تراشهٔ ATmega32

10 مراجع

[1]Labrosse, Jean J., (1992), uC/OS The Real-Time Kernel
[2]Tanenbaum, Andrew S.,Operating Systems: Design and Implementation