Bâtiment EARHAR
ZAC Grenoble Air Parc
38590 St Etienne de St Geoirs
–France

www.pragmatec.net

PICos18 v 2.01

Pragmatec

Products and services dedicated to real time embedded systems

PICos18

Операционная Система Реального Времени для PIC18.

Руководство разработчика.

Bâtiment EARHAR
ZAC Grenoble Air Parc
38590 St Etienne de St Geoirs
–France

www.pragmatec.net

PICos18 v 2.01

PICos18 является продуктом компании PRAGMATEC S.A.R.L., свободно распространяемый по лицензии GPL. Эта лицензия гарантирует PICos18 всегда оставаться свободной для доступа.

PICos18 это ядро операционной системы реального времени (ОСРВ), основанный на стандарте OSEK/VDX™ для автомобильной промышленности, для семейства PIC18 производства Microchip Technology Inc.

Этот документ разработан PRAGMATEC S.A.R.L. и содержит обучение и руководство пользователя в помощь пользователям операционной системы (ОС) PICos18. Реализация ОС технически и практически обоснована для устройств с PIC18F452, MPLAB и компилятора C18.

Содержание

1. Введение

Многозадачная ОС

В мире встроенных систем мы слышим, как много людей говорят об операционных системах реального времени без точного знания, что это такое. В целях более легкого понимания, что такое многозадачные системы реального времени, мы считаем необходимым рассказать о 3 разных аспектах:

Операционная система…

Операционная система это набор функций, большинство из которых сгруппированы термином «**сервисы**». Например, для Linux в состав ОС входят менеджер задач, менеджер доступа к устройствам, файловый менеджер… Оболочка это программа, и она не является частью ОС. Функция malloc, которая занимается динамическим выделением памяти, вызывает сервис ОС, потому что ОС отвечает за управление ресурсами, а память – это ресурс.

Одна из самых известных функций ОС (которая не является сервисом) – это **планировщик**, отвечающий за параллельное выполнение процессов.

…многозадачная…

Итак, ОС контролирует ресурсы и приложениям лучше пользоваться набором **сервисов** для безопасной и эффективной работы. ОС принимает решение выполнять готовое к выполнению приложение из многих независимых, гарантируя стабильность всей системы. Эта **многопрограммная** способность позволяет многим разработчикам открывать свои собственные программы без мысли о том, что разработано в другой задаче системы. Каждый разработчик может чувствовать себя единственным разработчиком.

Если ОС позволяет, то можно сделать разные программы, выполняющиеся параллельно, как будто каждая задача выполняется как единственная (но более медленно, чем если бы задача выполнялась одна на самом деле).

Если же операционная система – это совершенный менеджер задач (планирование задач это работа ОС) мы говорим, что ОС является **многозадачной вытесняющей**. Иначе каждая задача должна заботиться о процессе планирования сама, обращаясь к планировщику ОС время от времени. Такую ОС мы называем **многозадачной кооперативной**.

PICos18 является многозадачной вытесняющей ОС.

…реального времени

Многозадачная ОС может точно управлять параллелизмом задач путем разделения системного времени на небольшие отрезки, в течение которого задача представлена в памяти. Проблемой является то, что задачи не нуждаются в одинаковом доступе к ЦПУ, а некоторые задачи, выполняемые очень редко, требуют все возможности ЦПУ.

Задачам в таком случае присваивают разные **приоритеты** и могут стартовать немедленно при необходимости. ОС гарантирует постоянное **время отклика**, которое называется **детерменизмом**. Лучшее время отклика – это ноль, что невозможно.

Время отклика PICos18 составляет 50 мкс.

PICos18: операционная система реального времени для PIC18

Серия PIC16 не дает возможности создать такую ОС. В самом деле, основное свойство многозадачной ОС – выполнять различные задачи вместе, а это значит, что нам необходимо управлять стеком вызываемых функций (смотри описание на PIC18F452 DS39564A, стр. 37, раздел 4.2 «Стек адресов возврата»). Представьте, что будет, если все имеющиеся в системе задачи используют один и тот же стек: ОС останавливает текущую задачу для активации другой, а следующая инструкция return извлекает из стека адрес предыдущей задачи, где она была остановлена!

PIC18 может управлять аппаратным стеком адресов возврата вызываемых функций (благодаря инструкциям PUSH и POP и свободному перемещению указателя стека), дает возможность привести стек в правильное состояние перед переходом к следующей задаче.

Было обоснованной необходимостью тогда определить список сервисов ОС и как она управляет задачами и ресурсами. Лучше тогда разработать ОС на основе простых решений, и компания Pragmatec решила взять за основу стандарт OSEK/VDX.

Стандарт OSEK/VDX

PICos18 основан на стандарте OSEK/VDX (www.osek-wdx.org). Это большой проект в автоиндустрии, поддерживаемый большинством французских и немецких автопроизводителей. Целью данного проекта является определение стандарта управления и работы встроенных архитектур на современных автомобилях. Современные автомобили содержат до 30 блоков (управление двигателем, бортовой компьютер, контроллер дверей, ABS, ESP…), которые предают данные через общие сети (CAN, VAN, LIN, MOST шины).

Для совместной работы этих вычислительных блоков есть определенные способы:

- Определение одинаковых платформ разработки, одинаковых процессов разработки и одинаковых обоснованных процессов;
- Точно определить общий язык для производителей, подрядчиков и третьих сторон;
- Использовать общую обоснованную архитектуру разработки в проектах.

Аббревиатура OSEK означает «Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug» (Открытая система и интерфейсы передачи данных для встроенной автомобильной электроники). Буквы VDX означают «Vehicle Distributed eXecutive».

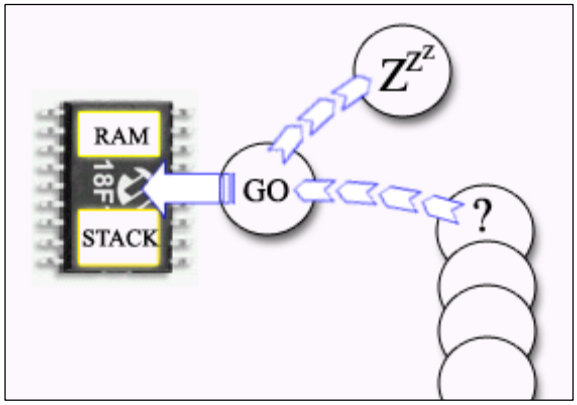
Операционная система стандарта OSEK включена теперь в VDX.

На сегодня этот стандарт в авто- и роботостроении и определяет особенности ОС вокруг трех осей: Операционная система (Operating system, OS), Коммуникации (Communication, COM) и управление сетями (Network Management, NM). В данном случае в PICos18 применена только спецификация OS.

Стандарт OSEK/VDX хорошо подходит к ядру PIC18. Приложение PICos18 составлено из множества задач, обозначенных кружочками в представленных далее рисунках.

Главной особенностью является то, что только одна задача может иметь доступ к PIC18 для выполнения (более точно, к процессору, памяти и стеку).

В ходе решения, какой задаче позволить выполняться в некоторый момент времени, ОС PICos18 инспектирует все задачи приложения и выбирает задачу в состоянии готовности с наибольшим приоритетом.



После активации задача может ожидать определенное событие и заснуть на некоторое время, чтобы позволить выполниться другой задаче с меньшим приоритетом. Когда ожидаемое событие наступает, ядро пробуждает задачу, ранее переведенную в состояние ожидания.

PICos18 использует следующие состояния задач: **READY**, **SUSPENDED**, **WAITING**, **RUNNING**.

Особенности PICos18

Операционная система PICos18 состоит из нескольких слоев:

- ✓ **Ядро ОС, kernel core,** (Init + Scheduler + Task Manager) отвечает за управление задачами приложения, а также принимает решение активировать следующую задачу согласно состояниям и приоритетам задач.
- ✓ **Системный таймер** и **таймауты** (Alarm Manager). Они связаны с ядром ОС и используют прерывания от TIMER0 для периодического обновления системного таймера и счетчиков таймаутов, используемых приложением.
- ✓ **Пользовательские функции** (Hook routines) частично включены в ядро ОС и позволяют разработчику перейти к собственному коду и что-то добавить или изменить. Действуя таким образом, возможно получить контроль за процессами ядра в течение короткого времени для отладки приложения в случае необходимости.
- ✓ **Диспетчер задач** (Task or Process Manager) это набор сервисов ОС, которые предлагают необходимые функции по управлению состоянием задач (изменение состояния задачи, сцепить две задачи, активировать задачу,…).
- ✓ **Диспетчер событий** (Event Manager) это набор сервисов ОС, которые предлагают необходимые функции по управлению событий ожидаемых, либо посланных от/к задаче (ожидать событие, послать событие, сбросить событие, прочитать набор принятых событий…).
- ✓ **Диспетчер прерываний** (INT manager) это набор сервисов ОС для разрешения или запрещения прерываний (высокого и низкого приоритетов).

PICos18 является операционной системой модульного типа, а это значит, что доступ к периферии PIC18 (драйверы, файловая система и т.д.) описывается как задача, независимая от ядра.

Этот подход дает возможность вам привыкнуть использовать в PICos18-приложениях различные программные слои (различные задачи и библиотеки, как это предлагает компания Pragmatec) для достижения каких-либо необходимых вам свойств, разрабатывая быстрее и легче.

Лицензия GPL

Операционная система PICos18 распространяется как «open-source» по лицензии GPL (General Public License). Это значит, что все исходные тексты ОС, написанные на языке C и ассемблере, всегда доступны без всяких ограничений. Это также значит, что все полностью свободно, и вы не платите ничего авторам.

В начале каждого файла PICos18 с исходным текстом вы найдете текст с GPL баннером. Если вам понадобится изменить или использовать один из файлов PICos18, вы должны сохранить этот баннер или в основном соответствовать ему:

```
/*********************************************************************/
/*                                                                   */
/* File name: filename of the source file                            */
/*                                                                   */
/* Since:     date of creation                                       */
/*                                                                   */
/* Version:   3.xx (current release of PICos18)                      */
/*                                                                   */
/* Author:    Designed by Pragmatec S.A.R.L.        www.pragmatec.net */
/*            MONTAGNE Xavier [XM]      xavier.montagne@pragmatec.net */
/*            LASTNAME Firstname [xx]                                */
/*                                                                   */
/* Purpose:   file content explanation                               */
/*                                                                   */
/*  The GPL licence from Boston (USA)(see « Free Software Foundation) */
/* Distribution: This file is part of PICos18.                       */
/*            PICos18 is free software; you can redistribute it      */
/*            and/or modify it under the terms of the GNU General    */
/*            Public License as published by the Free Software       */
/*            Foundation; either version 2, or (at your option)      */
/*            any later version.                                     */
/*                                                                   */
/*            PICos18 is distributed in the hope that it will be     */
/*            useful, but WITHOUT ANY WARRANTY; without even the     */
/*            implied warranty of MERCHANTABILITY or FITNESS FOR A   */
/*            PARTICULAR PURPOSE. See the GNU General Public         */
/*            License for more details.                              */
/*                                                                   */
/*            You should have received a copy of the GNU General     */
/*            Public License along with gpsim; see the file          */
/*            COPYING.txt. If not, write to the Free Software        */
/*            Foundation, 59 Temple Place - Suite 330,               */
/*            Boston, MA 02111-1307, USA.                            */
/*                                                                   */
/*          > A special exception to the GPL can be applied should   */
/*            you wish to distribute a combined work that includes   */
/*            PICos18, without being obliged to provide the source   */
/*            code for any proprietary components.                   */
/*                                                                   */
/* History:                                                          */
/* 2004/09/20 [XM]  Create this file.                                */
/*                                                                   */
/*********************************************************************/
```

Лицензия GPL гарантирует, что исходные тексты PICos18 могут быть получены свободно без всяких ограничений или патентов каких-либо компаний, организаций или лиц. Более того, компания Pragmatec, разработавшая PICos18, допускает поддерживать ОС

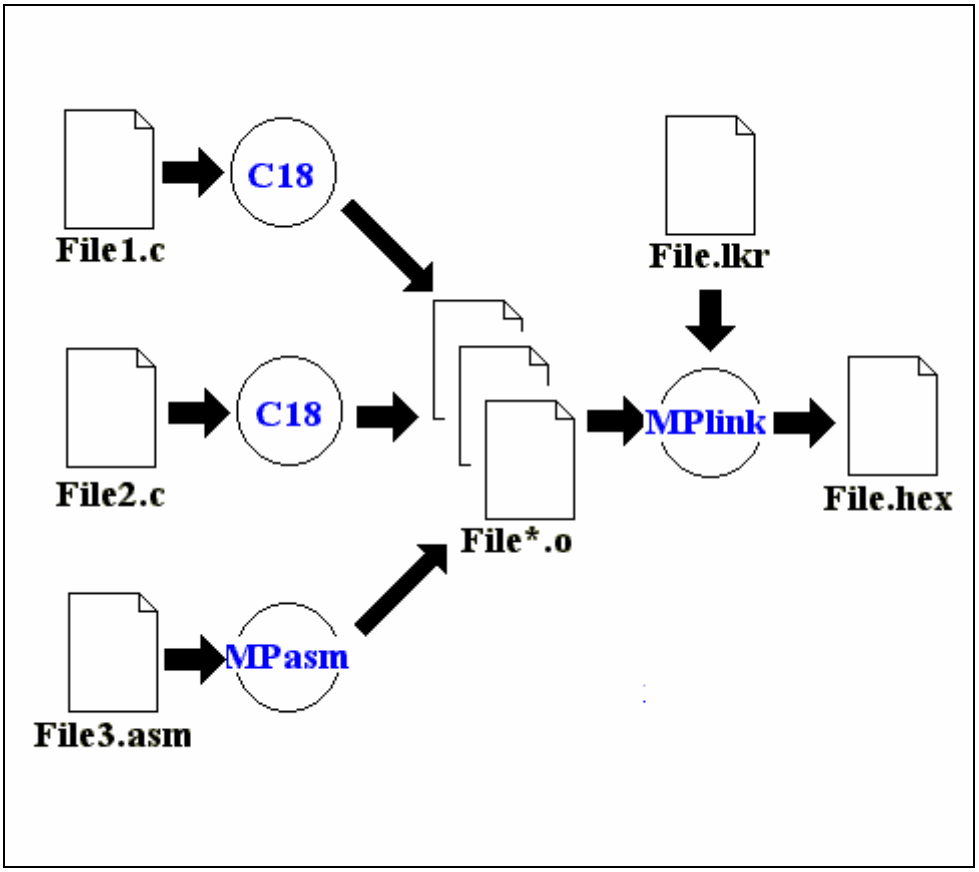
и обновлять так часто, как это возможно и всегда в соответствии с GPL лицензией. Конечно, каждый может изменять исходные тексты PICos18 для собственных нужд или может принять участие в совершенствовании ядра.

Специальный параграф был добавлен в этот баннер GPL лицензии. Этот параграф написан по-английски специально, так как позволяет добавлять какие-либо файлы в PICos18 без усилий по применению GPL лицензии к этим новым файлам, что означает, что вам нет необходимости хранить собственные исходные тексты. Но запомните, что вы можете хранить исходные тексты ядра у себя, поскольку это находится под защитой GPL лицензии.

Компилятор Microchip

Вы можете начать программировать свою первую программу на C с PICos18. Ядро само состоит из файлов на C и файлов на ассемблере (kernel.asm).

Эти разные фалы будут компилироваться и компоноваться все вместе с целью получить один файл: HEX-файл, который будет загружен в ваш PIC18.



Средства компиляции Microchip состоят из трех элементов:

- MPASM ассемблер для преобразования файлов .asm в .o файлы.
- MCC18 компилятор для преобразования .c файлов в .o файлы.
- MPLINK компоновщик для компоновки всех .o файлов в один .hex файл.

Файл, называемый скриптом компоновки (linker script или File.lkr) очень важен для создания .hex файла. В самом деле, .o файлы не могут разместить код и переменные в

памяти. Например, функция main() была оттранслирована в ассемблерные инструкции, которые PIC18 может выполнить, но не может расположить в определенном месте.

Назначение скрипта компоновки – определить положение переменных и кода в памяти (RAM и ROM). PICos18 содержит некоторые скрипты для наиболее употребительных и известных микроконтроллеров семейства PIC18. Вы можете легко адаптировать свое приложение или добавить еще один файл для управления новым PIC18.

В ходе компиляции и компоновки проекта другие различные файлы (*.map, *.lst, *.cod) могут быть сгенерированы дополнительно. За разъяснениями обращайтесь к документации Microchip.

Обучение было написано в помощь пользователям, чтобы понять и правильно использовать ОСРВ PICos18. По техническим причинам это было сделано с применением MPLAB и скомпилировано компилятором C18 студенческой версии для микроконтроллера PIC18F452.

Демопрограмма была протестирована под Microsoft Windows XP, в среде разработки MPLAB v8.00 и компилятором C18 v3.14 student edition от Microchip.

Средства разработки Microchip можно скачать с веб-сайта компании Microchip: www.microchip.com.

Компания Pragmatec

Представленное обучение является собственностью PRAGMATEC S.A.R.L.

PICos18 является продуктом компании Pragmatec и распространяется свободно под лицензией GPL.

Компания Pragmatec разрабатывает и предлагает расширения PICos18, позволяющие разработчику создавать свои приложения, основанные на свободных программных слоях (RS232 драйвер, драйвер CAN шины для передачи данных через периферию CAN шины PIC18F458, USB, I2C, и т.д.).

Компания Pragmatec также имеет набор дополнительного ПО и аппаратных средств для PICos18 и PIC18, позволяющие разработчику отлаживать и контролировать приложение из Windows-приложения, программировать PIC18, конвертировать данные CAN шины…

2. PICos18 : обучение

Целью данного обучения является показ того, как просто программировать с PICos18. Стандарт OSEK определяет основные особенности собственно операционной системы и определяет интерфейс между ОС и приложением.

Следуя данному примеру, вы научитесь как правильно настроить MPLAB, использовать PICos18 и программировать многозадачные приложения для PIC18.

Средства разработки

В первом разделе вы откроете полноценный проект, основанный на PICos18. Для этого вы будете использовать среду разработки MPLAB и компилятор C18 от Microchip.

Первая задача

Теперь вы можете успешно скомпилировать проект PICos18 под MPLAB, и теперь настало время открыть и посмотреть выполнение в симуляторе вашей первой задачи. Вы научитесь писать такие задачи и узнаете, что такое системный тик в 1 мс.

Вытеснение

Вы великолепно разрабатываете приложение только с одной задачей, но это приложение невозможно использовать в реальности! Добавив новую задачу, вы исследуете все механизмы вытеснения одной задачи другой.

Многозадачное приложение

Этот раздел посвящен механизмам синхронизации между задачами в PICos18. Третья задача позволяет завершить приложение и отправить сообщение другой задаче с целью ее активизации. Мы научим использовать сообщения, посланные от задачи или разделяемых ресурсов.

Прерывания

Пользуясь микроконтроллерами PIC18, вы, вероятно, надеялись работать с периферией. Делая это, вы могли рассматривать прерывания. Вы научитесь работать с прерываниями в PICos18 и писать свои обработчики прерываний.

Использование драйверов

Вы желаете связываться через последовательный порт или CAN интерфейс? Теперь вы знаете, как обрабатывать прерывания и писать задачи и вы знаете, как управлять таймаутами и сообщениями, и это очень облегчает использование и написание настоящих драйверов для PICos18.

Пример приложения

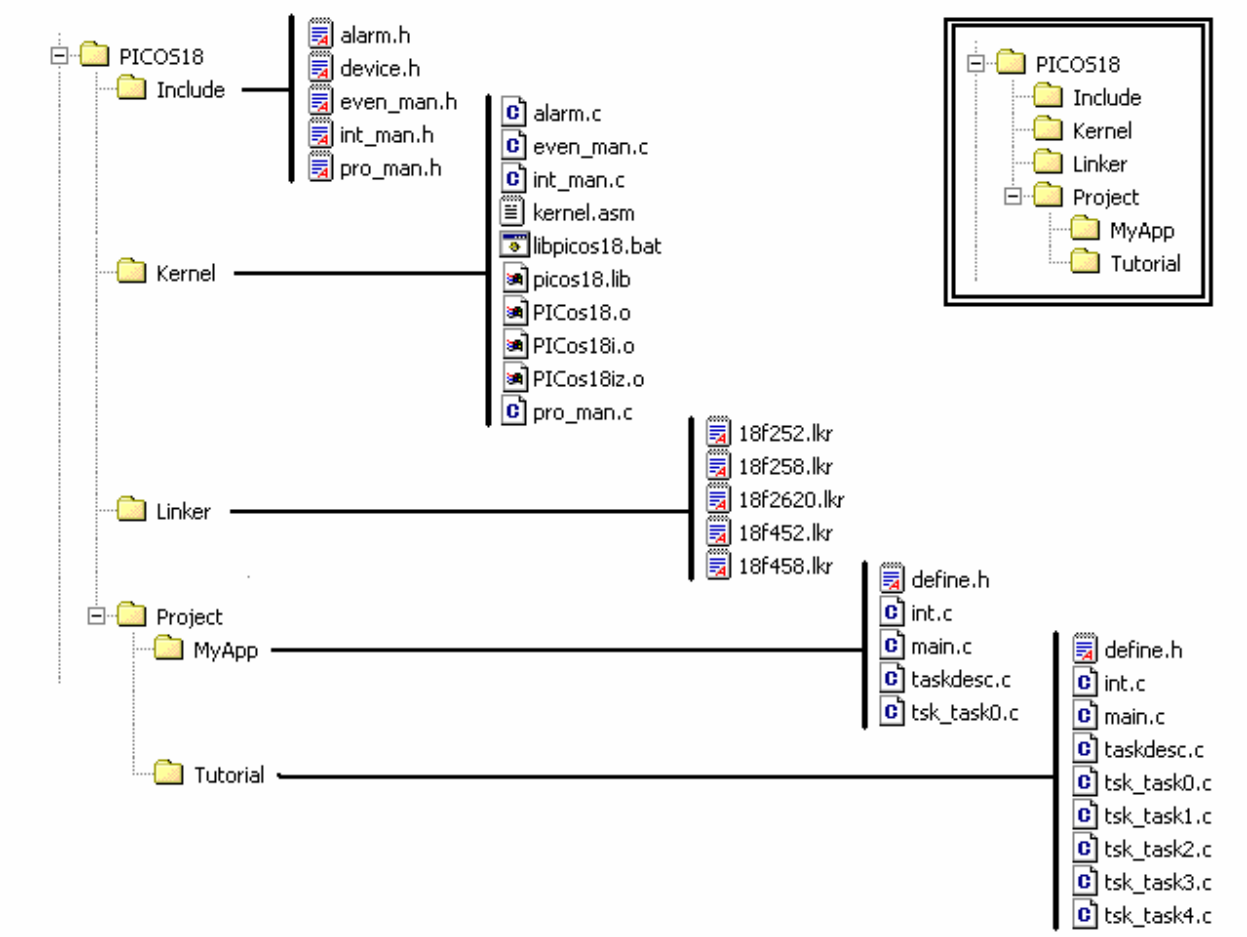
Обучение заканчивается на этом последнем разделе по разработке PICos18. Исходные тексты этого типового приложения находятся в директории Project/Tutorial дистрибутивов PICos18. Этот пример представлен, чтобы показать вам, как легко разрабатывать приложения в комплексе с PICos18.

3. Средства разработки

В первом разделе вы откроете полноценный проект, основанный на PICos18. Для этого вы будете использовать среду разработки MPLAB и компилятор C18 от Microchip.

> Исходные файлы проекта

Перед запуском MPLAB скачайте последнюю версию PICos18, включая исходные тексты ядра и приложение для обучения. Это следующие файлы:



Include	Эта папка содержит все заголовочные файлы ОС PICos18. Заголовочные файлы приложений и драйверов расположены в своих папках (в Project). Заголовочные файлы существуют для каждой подсистемы ОС: таймауты (alarms), события (events), прерывания (interruption) и процессы. Файл device.h содержит определения для ОС, например, возвращаемые значения функций сервисов PICos18 (API).
Kernel	Здесь вы найдете все исходные тексты ОС. Файлы *.c сгруппированы по сервисам ОС (API). Файл kernel.asm, только он написан на ассемблере, содержит все основные свойства ядра ОС: алгоритм перепланировки задач, их порядок согласно приоритетам и обработка ошибок. Исходные тексты содержат для вас информацию, которую вы при желании и необходимости можете изменить. Для облегчения применения PICos18 мы добавили библиотеку PICos18: picos18.lib. Код запуска программы от Microchip не очень хорошо подходит для PICos18, поэтому мы изменили его. Новый адаптированный код находится в файлах типа PICos18*.o (PICos18iz.o, например). Вам не надо волноваться по поводу библиотеки PICos18.lib и кода запуска, потому что они автоматически включаются в ваш PICos18 проект!
Linker	Когда вы создаете свое приложение с PICos18, вы пишете программу на языке С. Затем вы компилируете программу с помощью С18 (получаете фалы *.o соответственно каждому *.c файлу) и затем компонуете с помощью MPLINK. Вы можете скомпоновать *.o файлы только с использованием скрипта компоновки, *.lkr файлы включены в эту директорию. В этой директории для наиболее употребительных микроконтроллеров семейства PIC18 представлен свой LKR-скрипт с целью облегчить вам управление этой сложной частью процесса компиляции. В будущем в эту директорию будут добавляться скрипты для каждого нового PIC18, работающего под PICos18.
Project/ MyApp	Эта директория содержит все файлы, содержащие основу проекта на PICos18. Как вы можете заметить, это всего лишь несколько файлов: файл main.c, который является первым выполняющимя файлом приложения PICos18; файл int.c, который содержит обработчики прерываний обоих приоритетов; файл taskdesc.c с точным описанием всех настроек вашего приложения (таймауты, счетчики, настройки задач, …), и файлы на С для каждой задачи приложения.
Project/ Tutorial	Демонстрационная программа, основанная на типовом приложении PICos18. Проходя разделы, вы научитесь программировать с PICos18 и создавать свои собственные приложения. Представленные в этой директории файлы станут законченной программой к моменту окончания вашего обучения. Это подвигнет кого-нибудь больше позаниматься с этой начальной сложности программой на уровне исходного кода.

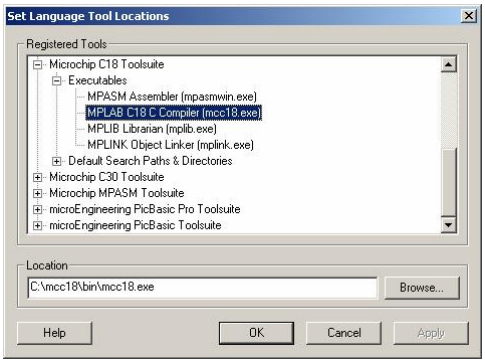
Распакуйте архив PICos18_v3_xx.zip в корневой каталог C:\. Все разделы обучения расположены в директории под названием PICos18 в корневом каталоге C:\. Нет необходимости здесь что-либо изменять.

На настоящий момент демопрограмма проверена в среде разработки MPLAB v8.00 совместно с компилятором C18 v3.14 SE от Microchip. Вы можете скачать бесплатные версии этих программ с вебсайта Microchip. Бесплатная версия (SE, Student Edition, учебная версия) такая же, как и полная, но без оптимизации. Фактически, вы отключите такие опции для PICos18, которые не подходят для многозадачных приложений.

Учебная версия C18 хорошо подходит для PICos18 и PIC18, сгенерированный ими ассемблерный код хорошо совпадает с кодом платной версии.

Целевым контроллером выбран PIC18F452 (не забудьте выбрать его в настройках MPLAB).

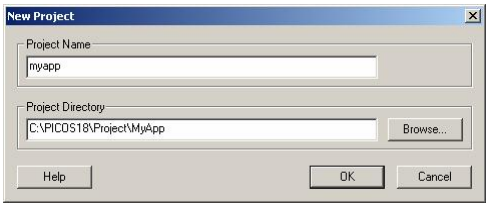
> Компилятор C18



Начните с установки MPLAB и компилятора C18 для семейства PIC18 Microchip. По умолчанию мы приняли путь “C:\mcc18\”.

В MPLAB кликните на “Project / Set Language Tool Location…” и задайте расположение MPASM, MPLAB C18 и MPLINK.

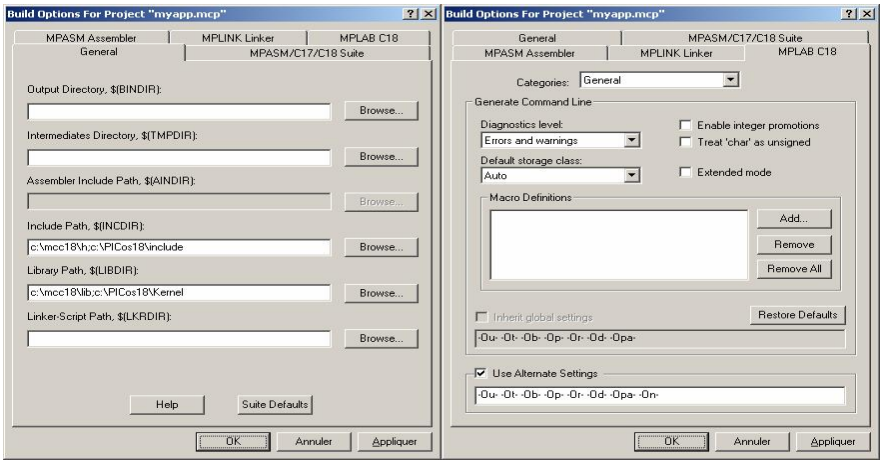
> Открытие проекта



Откройте новый проект в MPLAB. Кликните на “Project / New…” и задайте имя проекта длиной не более 8 символов. Я выбрал “picos.pjt”. Будьте внимательны: путь не должен содержать пробелы!

Затем кликните на “Project / Select Language Toolsuite…” и выберите “Microchip C18 Toolsuite”.

> Настройки проекта

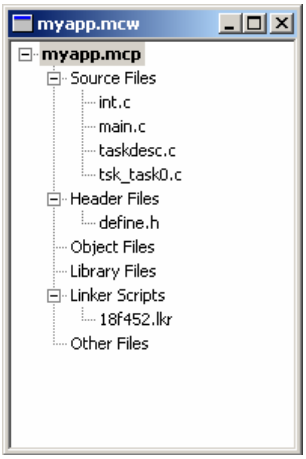


Кликните на “Project / Build Options…” и задайте папки Include и Library для C18 и компилятора. Будьте внимательны: путь к папке не должен содержать пробелы!

Кликните на вкладку “MPLAB C18” и установите галочку на “Use alternate settings” и добавьте “-On” в командную строку.

Это заставит компилятор отключить межбанковую оптимизацию памяти, не эффективную в PICos18.

> Добавление файлов в проект

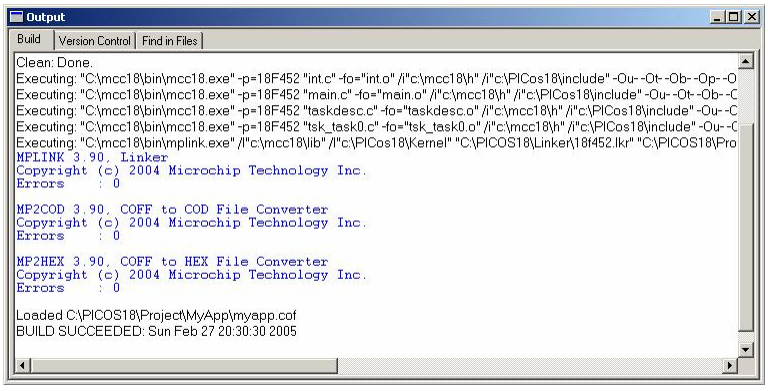


Затем кликните правой кнопкой мыши на окне проекта и добавьте следующие файлы:

- int.c, main.c, taskdesc.c и tsk_task0.c, находящиеся в папке MyApp;
- define.h, находящийся там же;
- 18f452.lkr, находящийся в папке Linker

Файлы picos18.lib и PICos18iz.o будут автоматически включены в проект компилятором.

> Компиляция проекта

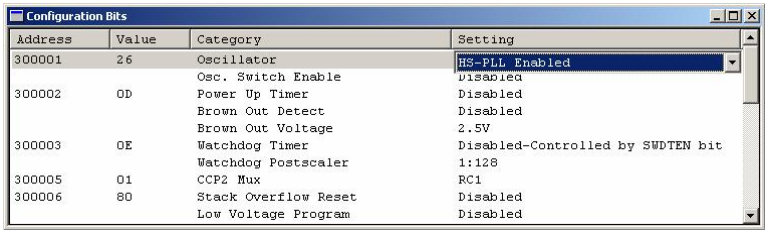


Наконец, скомпилируйте проект, нажав кнопку **F10**.

Компилятор скомпилирует все файлы один за другим. Не требуется компилировать какие-нибудь файлы ядра, поскольку библиотека PICos18 уже добавлена в проект.

Затем компоновщик соберет вместе все .o файлы, чтобы создать фалы, необходимые для работы симулятора и программирования PIC18.

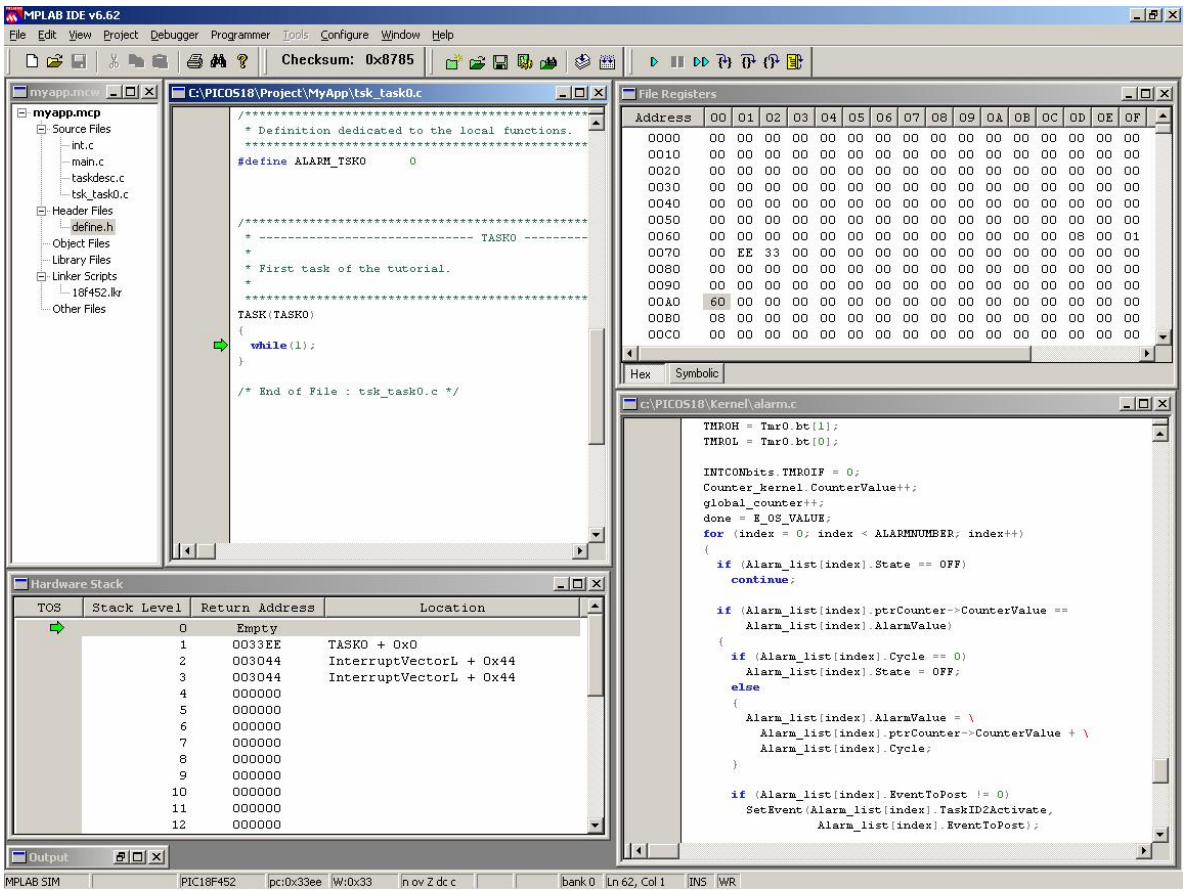
> Отключение сторожевого таймера



Симулятор MPLAB использует настройки битов конфигурации PIC18.

Настройте симулятор так, как показано на скриншоте. *Сторожевой таймер должен быть отключен!*

> Выполнение проекта в симуляторе



Начните трассировку в симуляторе, выбрав “Debugger / Select Tool / MPLAB SIM”, затем F9 (Run). Подождите несколько секунд и остановите с помощью кнопки **F5** (Stop).

Трассировка остановится на команде “while(1)” задачи 1. Но эта команда не заблокирует PIC18 в бесконечном цикле, потому что ОС может быть вызвана при необходимости: потратьте немного времени и остановитесь на одном из мест ОС.

Исходный код ОС, на котором вы остановились, будет отображен на дисплее, но вы никогда не добавляли исходный код ОС в проект!

Библиотека PICos18 была создана с тем же путем к папке, в которую вы распаковали библиотеку и исходные файлы PICos18 из архива.

4. Первая задача

Итак, вы можете успешно скомпилировать проект PICos18 в MPLAB, и теперь настало время открыть и посмотреть выполнение в симуляторе вашей первой задачи. Вы научитесь писать такие задачи и узнаете, что такое системный тик в 1 мс.

> Требования

Самое первое, что необходимо сделать, это открыть новый проект в MPLAB из папки PICos18.

Также вам необходимо установить и настроить компилятор Microchip C18, как об этом сказано в предыдущем разделе.

> Состояние одной задачи

В нашем приложении есть только одна задача, определенная в файле “tsk_led.c”:

```
TASK(TASK0)
{
  while(1);
}
```

Нам будет легче понять суть состояний задачи через бесконечный цикл.

В MPLAB поставьте точку останова на строке с содержанием “ `while(1);`”, а затем стартуйте приложение, нажав **F9**.

Вы увидите, что симулятор остановил приложение на точке останова. Но как ОС знает о представленной здесь задаче, выполняет ее код в бесконечном цикле,… как будто бы и нет этой полностью блокирующей команды?

На самом деле задача описывается для ОС в файле taskdesc.c.

```
/*****************************************************************
* ---------------------------- task 0 ----------------------------
*****************************************************************/
rom_desc_tsk rom_desc_task0 = {
   TASK0_PRIO,                    /* prioinit from 0 to 15          */
   stack0,                        /* stack address (16 bits)        */
   TASK0,                         /* start address (16 bits)        */
   READY,                         /* state at init phase            */
   TASK0_ID,                      /* id_tsk from 1 to 15            */
   sizeof(stack0)                 /* stack size    (16 bits)        */
};
```

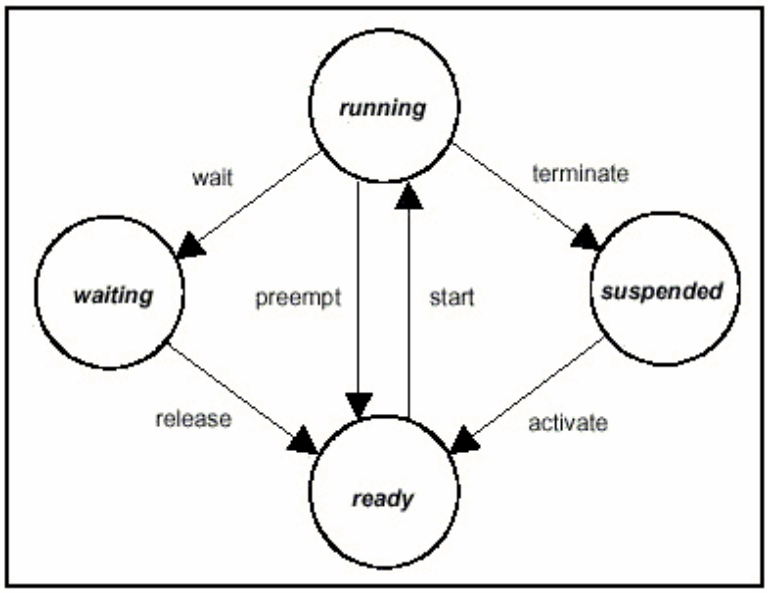
На выделенной жирным шрифтом строке вы можете указать состояние задачи в момент ее старта. Фактически, когда задача существует в проекте (она скомпилирована компилятором), нет острой необходимости ОС активировать ее.

В PICos18 согласно стандарту OSEK задача может иметь 4 возможных состояния:

SUSPENDED: задача представлена в приложении, но не принимается в расчет ядром ОС.

READY: задача доступна к выполнению ОС, когда планировщик посчитает это необходимым.

WAITING: задача в спящем состоянии, т.е. она как бы и приостановлена (SUSPENDED) и будет готова (READY) как только произойдет событие.



RUNNING: только одна задача может выполняться в каждый определенный момент времени: Это задача в состоянии READY с наивысшим приоритетом.

Говоря о нашем приложении, задача находится в состоянии READY. Пока она в этом состоянии, ОС будет выполнять ее…

Замените **READY** на **SUSPENDED** в файле taskdesc.c, затем перекомпилируйте проект и запустите в симуляторе: вы можете заметить, что программа не останавливается на точке останова.

> Описание задачи

Наряду с описанием состояния задачи есть еще ряд особенностей, описанных в файле taskdesc.c. В самом деле, весь рассказ о вашем приложении находится в этом файле, как например: ID задачи, ее приоритет, программный стек, таймауты, используемые в проекте…

Мы находим первый из таймаутов приложения:

```
AlarmObject Alarm_list[] =
{
/*****************************************************************
 * ------------------------- First task --------------------------
 *****************************************************************/
  {
    OFF,                                  /* State                */
    0,                                    /* AlarmValue           */
    0,                                    /* Cycle                */
    &Counter_kernel,                      /* ptrCounter           */
    TASK0_ID,                             /* TaskID2Activate      */
    ALARM_EVENT,                          /* EventToPost          */
    0                                     /* CallBack             */
  },
};
```

Эта таблица – список таймаутов, каждый из которых представляет собой структуру. Таймаут является частью объекта TIMER0, управляемый программно из ядра ОС. Фактически у PICos18 есть таймер с периодом в 1 мс, построенный из аппаратного и программного драйверов. Поэтому есть способ создать программный таймер, основанный на тике в 1 мс: таймаут (alarm).

Таймаут подключается к счетчику (это Counter_kenel с периодом в 1 мс) и к задаче (с идентификатором TASK0_ID). Когда таймаут достигает определенной величины, он посылает сообщение этой задаче (ALARM_EVENT в нашем примере) или вызывает С функцию (CallBack механизм).

Мы далее находим список ресурсов приложения:

```
/*********************************************************************
 * --------------------- COUNTER & ALARM DEFINITION -------------------
 *********************************************************************/
Resource Resource_list[] =
{
  {
    10,                                 /* priority                 */
    0,                                  /* Task prio                */
    0,                                  /* lock                     */
  }
};
```

Ресурс это другой объект, управляемый ОС. В основном ресурсы предназначены для разделения периферии между разными задачами, например, 8-битный порт PIC18. Он имеет собственный приоритет (поле priority) и может быть занят (поле lock) задачей (поле Task prio). Далее, когда задача захочет получить доступ к 8-битному порту, она может иметь безопасный доступ к ресурсу и не дать другим задачам доступ в течение короткого времени. Управление ресурсами будет описано в разделе о многозадачности.

Теперь мы находим список программных стеков:

Существует 2 класса переменных: статические и автоматические,… но чтобы полнее понять о чем идет речь, то это глобальные и локальные переменные, даже если это не вполне корректно. Глобальные переменные (переменные, объявленные вне С-функции) располагаются в области RAM по абсолютным адресам, которые никогда не изменяются.

Локальные переменные (объявленные внутри С-функции) компилируются С18 так, что они располагаются в специфическом буфере памяти, называемым программным стеком. Программный стек это динамическая область памяти, что-то содержащая по текущему указателю. Когда вы входите в С-функцию, PIC18 начинает заталкивать в стек локальные переменные, а когда покидаете ее, то выталкивает (удаляет) зарезервированную область памяти для локальных переменных, тем самым высвобождает память. Этот механизм позволяет вам экономить много памяти, когда вам нужно много С-функций, а дополнительный код больше не нужен.

Управление программным стеком у PICos18 и С18 совпадают. Каждая задача имеет свой собственный стек, так что каждая задача приложения может работать как единственная в собственном пространстве, не беспокоясь о других процессах. Более того, PICos18 автоматически определяет переполнение стека, что случается, когда программный стек становится больше, чем ожидалось (kernel_panic флаг).

```
/*********************************************************************
 * ----------------------- TASK & STACK DEFINITION --------------------
 *********************************************************************/
#define DEFAULT_STACK_SIZE      128
DeclareTask(TASK0);

volatile unsigned char stack0[DEFAULT_STACK_SIZE];
```

Каждый раз, когда вы добавляете задачу в свое приложение, не забывайте добавить программный стек простым копированием строки сверху и изменения имени стека (stack1, например).

- Минимальный размер стека составляет 80 байтов.
- Типичный размер стека обычно колеблется в пределах 96 – 128 байтов.
- Максимальный размер стека ограничивается размером доступной памяти и вашей разумностью.

Любые другие значения также допустимы.

Далее мы находим список задач приложения:

```
/*********************************************************************
 * ---------------------------- task 0 -------------------------------
 *********************************************************************/
rom_desc_tsk rom_desc_task0 = {
   TASK0_PRIO,                      /* prioinit from 0 to 15          */
   stack0,                          /* stack address (16 bits)        */
   TASK0,                           /* start address (16 bits)        */
   READY,                           /* state at init phase            */
   TASK0_ID,                        /* id_tsk from 1 to 15            */
   sizeof(stack0)                   /* stack size (16 bits)           */
};
```

Вы уже видели это описание задачи, но обратим ваше внимание на некоторые дополнительные детали:

- **prioinit**: это приоритет задачи в пределах от 1 до 15 включительно. 1 это низший приоритет, 15 – высший. Нулевой приоритет закреплен за ядром и не должен использоваться в вашем приложении.
- **stack address**: адрес начала программного стека задачи. Каждая задача имеет свой стек, где хранятся локальные переменные, когда задача выполняется, а также контекст задачи (регистры, аппаратный стек PIC18, дополнительная область данных), когда задача не выполняется.
- **start address**: это имя задачи, означающую имя функции точки входа задачи. Компилятор транслирует ее в физический адрес, используемый ядром для первого запуска задачи.
- **state init**: это стартовое состояние задачи. В нашем приложении это READY, что означает немедленный запуск задачи как только ядро закончит инициализацию.
- **stack size**: это размер стека в байтах. Ядро использует это значение для контроля целостности стека, например, при переполнении.

> Прерывания с периодом в 1 мс

Даже если ваше приложение не использует прерывания, вам, вероятно, понадобится внутренний программный таймер. Сделать так позволяют некоторые определения в файле int.c и одна из разновидностей программной таблицы векторов прерываний.

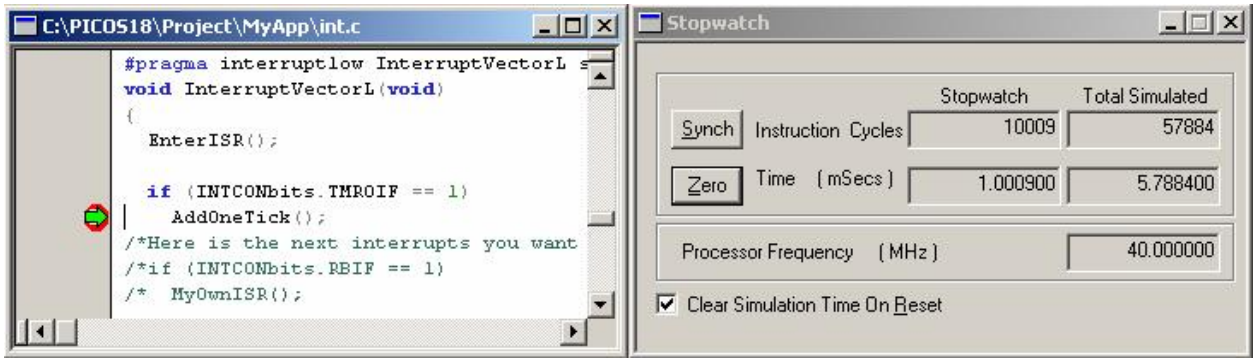
В самом деле, функция очень облегченная и позволяет вам обрабатывать прерывания без первоначального чтения тяжелого даташита на PIC18:

```
#pragma code _INTERRUPT_VECTORL = 0x003000
void InterruptVectorL(void)
{
  SAVE_TASK_CTX(stack_low, stack_high);
  EnterISR();

  if (INTCONbits.TMR0IF == 1)
    AddOneTick();
/*Here is the next interrupts you want to manage */
/*if (INTCONbits.RBIF == 1)                       */
/*  MyOwnISR();                                   */

  LeaveISR();
  RESTORE_TASK_CTX;
}
#pragma code
```

Как вы можете себе представить, когда происходит переполнение TIMER0 (TMR0IF флаг), вызывается функция AddOneTick для добавления 1 мс каждому таймауту приложения. При достижении заданного значения таймаут с помощью ОС посылает сообщение связанной с ним задаче, по существу выводит задачу в состояние готовности быть активированной.

Вы можете заметить, что ОС всегда в готовности, и что никакая задача не может остановить ее, даже при наличии бесконечного цикла ‘while(1)”.

Для проверки этого факта установите в симуляторе частоту генератора 40 МГц (Debugger / Settings…), поставьте точку останова на вызываемую функцию AddOneTick() и откройте экран MPLAB Stopwatch (Debugger / Stopwatch):



> Запуск приложения

Наше приложение содержит файл main.c, необходимый компилятору.

В соответствии со стандартом OSEK этот main файл не является частью операционной системы, но является частью приложения, хотя вся его работа заключается в запуске ОС (потому что возможно запустить и остановить работу ОС из этого файла).

Также в main файле содержится функция Init(), позволяющая сделать первоначальные настройки приложения по частоте без необходимости открытия даташита на PIC18:

```
void Init(void)
{
  FSR0H = 0;
  FSR0L = 0;

  /* User setting : actual PIC frequency */
  Tmr0.lt = TMR0_PRESET;

  /* Timer OFF - Enabled by Kernel */
  T0CON = 0x08;
  TMR0H = Tmr0.bt[1];
  TMR0L = Tmr0.bt[0];
}
```

Выделенная линия необходима для настройки ВНУТРЕННЕЙ частоты кристалла PIC18. На самом деле PIC18 имеет PLL модуль – умножитель частоты на 4, который может быть включен или нет. Обычно устройство на PIC18 содержит кварцевый резонатор с двумя конденсаторами по 15 пФ и совместно с PLL может работать на частоте до 40 МГц.

Командный цикл PIC18 выполняется за 4 такта, и производительность процессора соответственно составляет 10 MIPS (миллионов инструкций в секунду) согласно даташиту. PIC18 такой маленький 8-разрядный микроконтроллер, но очень мощный с развитой периферией для ваших приложений.

Если у вас есть желание использовать другую частоту, просмотрите файл define.c на предмет определения других частот. Вы можете использовать одну из готовых констант, а можете воспользоваться магической формулой.

5. Вытеснение

Вы великолепно разрабатываете приложение только с одной задачей, но это приложение невозможно использовать в реальности! Добавив новую задачу, вы исследуете все механизмы вытеснения одной задачи другой.

> Периодическое просыпание задачи

Вам еще до сих пор трудно понять, что интересного в такой операционной системе, когда у вас только одна задача… да и то с бесконечным циклом!

Итак, мы подошли к тому, чтобы добавить маленький бит интереса в нашу задачу, изменив код, который будет мигать светодиодом с определенной частотой.

Измените код задачи следующим образом:

```
TASK(TASK0)
{
  TRISBbits.TRISB4 = 0;
  LATBbits.LATB4   = 0;

  SetRelAlarm(ALARM_TSK0, 1000, 200);

  while(1)
  {
    WaitEvent(ALARM_EVENT);
    ClearEvent(ALARM_EVENT);

    LATBbits.LATB4 = ~LATBbits.LATB4;
  }
}
```

Наша задача содержит две фазы: первая находится перед “while(1)”, которая производит инициализацию, а вторая составляет тело бесконечного цикла.

Две первые инструкции управляют портом ввода-вывода PIC18. Ключевое слово “TRISx” используется для настройки порта на вход или выход, а ключевое слово “TRISxbits” – структура, позволяющая изменять состояние бита порта.

Мы определили порт RB4 как выход (“0”- выход, “1”- вход) и значение выхода, равным нулю (ключевое слово LATxbits).

Затем функция SetRelAlarm используется для программирования таймаута задачи. В предыдущем разделе мы рассмотрели, что такое таймаут: объект таймера с периодом в 1 мс. В файле taskdesc.c ID таймаута равен 0 (ALARM_TSK0) и связан с задачей TASK0:

```
AlarmObject Alarm_list[] =
  {
   /*****************************************************************
    * ------------------------- First task --------------------------
    *****************************************************************/
   {
     OFF,                                 /* State                */
     0,                                   /* AlarmValue           */
     0,                                   /* Cycle                */
     &Counter_kernel,                     /* ptrCounter           */
     TASK0_ID,                            /* TaskID2Activate      */
     ALARM_EVENT,                         /* EventToPost          */
```

```
    0                       /* CallBack                */
  },
};
```

Функция SetRelAlarm имеет три важных поля:

- **Alarm ID**: индекс таймаута в таблице Alarm_list
- **TIMER 1**: число миллисекунд перед первой посылкой сообщения
- **TIMER 2**: число миллисекунд между двумя сообщениями

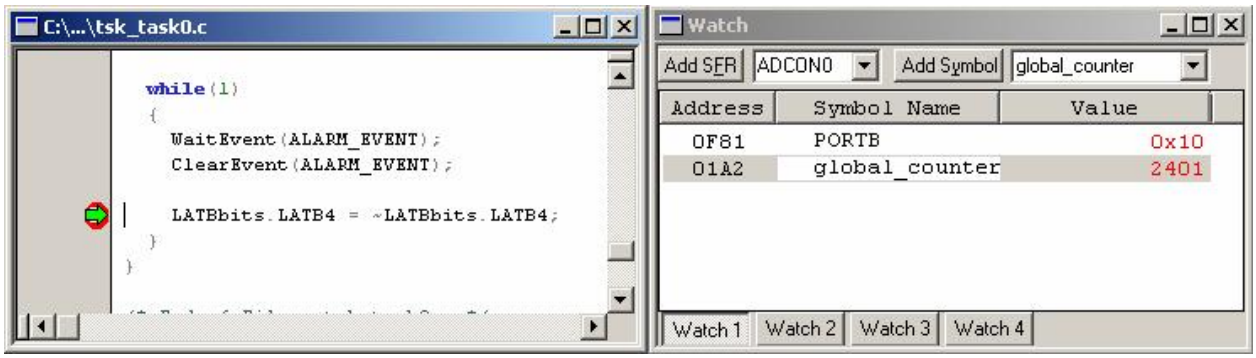
В нашем случае это значит, что таймаут 0 ждет 1000 мс до отправки сообщения ALARM_EVENT, а затем посылает сообщение каждые 200 мс.

Если таймаут запрограммирован, задача может работать. В цикле while(1) первая функция (WaitEvent) разрешает задаче заснуть, пока не придет сообщение **ALARM_EVENT**.

Вы можете заставить заснуть задачу для любого сообщения, но должны помнить, что сообщения представляют собой числа степени 2, определяемые в файле define.c. Это следующие значения: 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02 и 0x01.

> Задача 0. Трассировка в симуляторе

Для проверки, как задача мигает светодиодом с периодом в 200 мс, вы можете в симуляторе, поставив точку останова на строке, которая переключает бит RB4:



Более того, вы можете проследить за периферией PIC18 или глобальными переменными PICos18, особенно переменной “global_counter”, являющуюся десятичным счетчиком прошедших миллисекунд. Получить десятичное значение можно, кликнув в окошке Watch правой кнопкой мыши на значении переменной и выбрав “Dec”.

Запустите выполняться программу в симуляторе несколько раз (F9), чтобы посмотреть **переключение PORTB между 0x00 и 0x10**, а также значение глобальной переменной, увеличенной на 200. Начальное значение переменной было 1001 (ядру необходимо 1 мс для загрузки, поэтому вы получите значение 1001, вместо ожидаемой 1000).

Это означает, что порт RB4 включится после 1 секунды, а затем будет переключаться с периодом в 200мс.

А теперь измените последних два параметра функции SetRelAlarm и посмотрите, что произойдет. К примеру, измените последний параметр на 0, и вы увидите, что RB4 останется в 1, и вы никогда не вернетесь в задачу.

Посмотрите руководство по PICos18 API, чтобы узнать об этих функциях больше.

> Открытие второй задачи

Теперь вы можете разработать одну периодическую задачу и пора перейти к открытию второй задачи и синхронизировать их.

1) В MPLAB сохраните файл “tsk_task0.c” как “tsk_task1.c”

2) Измените все относящееся к новой задаче в файле taskdesc.c:

```
#define DEFAULT_STACK_SIZE 128
DeclareTask(TASK0);
DeclareTask(TASK1);

volatile unsigned char stack0[DEFAULT_STACK_SIZE];
volatile unsigned char stack1[DEFAULT_STACK_SIZE];

/*************************************************************************
 * ---------------------- TASK DESCRIPTOR SECTION ----------------------
 *************************************************************************/
#pragma              romdata          DESC_ROM
const rom unsigned int descromarea;
/*************************************************************************
 * ----------------------------- task 0 -----------------------------
 *************************************************************************/
rom_desc_tsk rom_desc_task0 = {
  TASK0_PRIO,                       /* prioinit from 0 to 15          */
  stack0,                           /* stack address (16 bits)        */
  TASK0,                            /* start address (16 bits)        */
  READY,                            /* state at init phase            */
  TASK0_ID,                         /* id_tsk from 1 to 15            */
  sizeof(stack0)                    /* stack size (16 bits)           */
};
/*************************************************************************
 * ----------------------------- task 1 -----------------------------
 *************************************************************************/
rom_desc_tsk rom_desc_task1 = {
  TASK1_PRIO,                       /* prioinit from 0 to 15          */
  stack1,                           /* stack address (16 bits)        */
  TASK1,                            /* start address (16 bits)        */
  READY,                            /* state at init phase            */
  TASK1_ID,                         /* id_tsk from 1 to 15            */
  sizeof(stack1)                    /* stack size (16 bits)           */
};
```

3) Затем измените тело функции task 1:

```
unsigned char hour, min, sec;

/*************************************************************************
 * ----------------------------- TASK1 -----------------------------
 *
 * Second task of the tutorial.
 *
 *************************************************************************/
TASK(TASK1)
{
  hour = min = sec = 0;

  while(1)
  {
    WaitEvent(TASK1_EVENT);
    ClearEvent(TASK1_EVENT);

    sec++;
    if (sec == 60)
```

```
    {
      sec = 0;
      min++;
      if (min == 60)
      {
        min = 0;
        hour++;
      }
    }
  }
}
```

Вы, возможно, догадались, что эта задача управляет своего рода программными часами с часы/минуты/секунды: каждый раз задача просыпается, переменная секунд увеличивается на 1, а переменные минут и часов обновляются при необходимости.

4) Итак, задача должна вызываться каждые 1000 мс. Измените задачу 0 так, чтобы она посылала сообщение нашей новой задаче каждую секунду:

```
SetRelAlarm(ALARM_TSK0, 1000, 1000);
while(1)
{
  WaitEvent(ALARM_EVENT);
  ClearEvent(ALARM_EVENT);

  LATBbits.LATB4 = ~LATBbits.LATB4;
  SetEvent(TASK1_ID, TASK1_EVENT);
}
```

5) Теперь вам необходимо обновить определения в файле “define.c”:

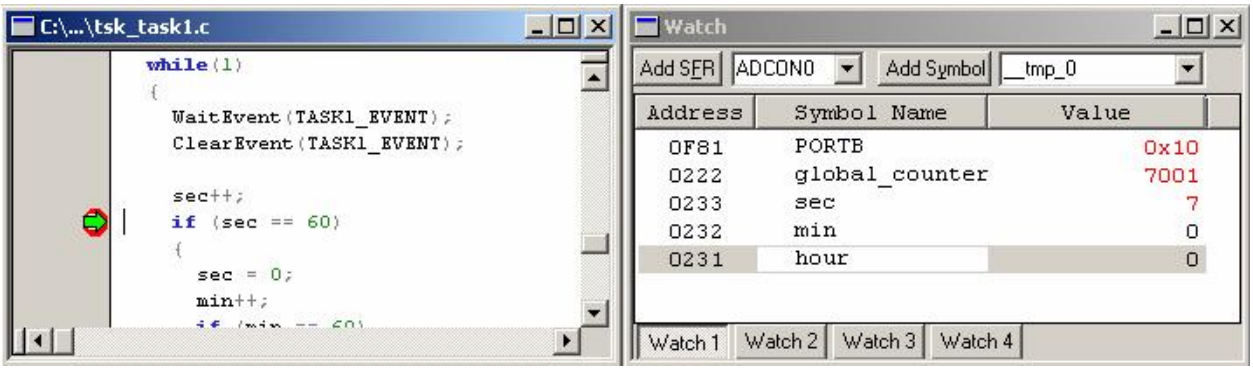
```
/*****************************************************************
 * ---------------------------- Events -----------------------------
 *****************************************************************/
#define ALARM_EVENT 0x80
#define TASK1_EVENT 0x10

/*****************************************************************
 * ---------------------------- Task ID ----------------------------
 *****************************************************************/
#define TASK0_ID 1
#define TASK1_ID 2

#define TASK0_PRIO 7
#define TASK1_PRIO 6
```

6) Наконец, вы можете добавить новый файл в проект (“Project / Add files to Project…”).

Перекомпилируйте проект (**F10**).

> Задача 1. Трассировка в симуляторе

В порядке проверки, успешно ли обновляются переменные часов, минут и секунд каждую секунду, запустите программу в симуляторе с точкой останова:

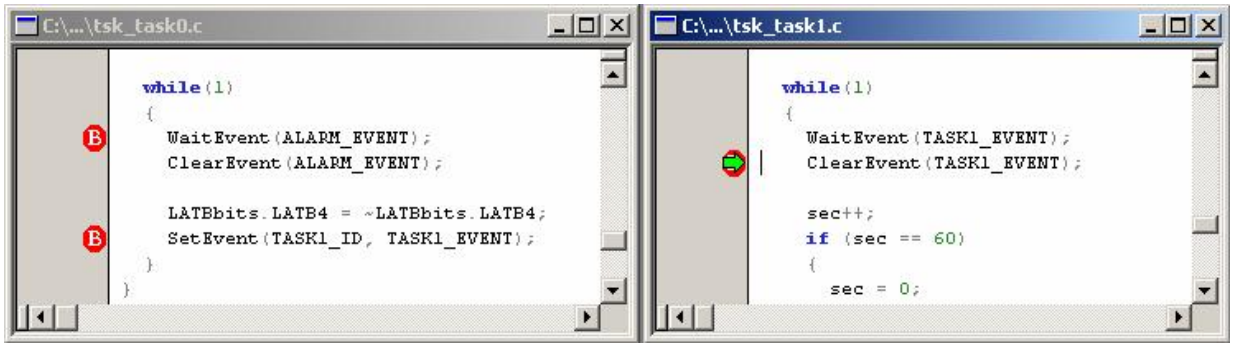


Симуляция 1 секунды требует некоторого времени для MPLAB, поэтому, если вы желаете протестировать переменные минут и часов до 60, то вам лучше изменить таймаут ALARM_TASK0 для более быстрого выполнения применительно к симулятору, конечно.

> Вытеснение

Сейчас ваше приложение состоит из 2 задач, и теперь это многозадачное приложение. Но это не значит, что обе задачи выполняются в одно и тоже время, поскольку это невозможно для процессора PIC18, который может выполнить только одну команду за раз. Максимум, что мы можем сказать, что задачи выполняются параллельно в смысле одна функция после другой по набору правил, описанных в ядре ОС,… к примеру, правила приоритетов.

Для лучшего понимания определений “вытеснение”, “многозадачность” и “реальное время” добавьте несколько точек останова и запустите программу в симуляторе:



Указатель будет останавливаться в точках останова в следующем порядке:

- **WaitEvent в задаче 0**
- **SetEvent в задаче 0**
- **WaitEvent в задаче 0**
- **ClearEvent в задаче 1**

В самом деле, задача 0 имеет приоритет выше, чем задача 1, и когда она посылает сообщение задаче 1, то она продолжает работу, пока не заснет, выполнив функцию WaitEvent. В этот момент задача 1 освобождается для выполнения, и указатель останавливается на функции ClearEvent.

А сейчас измените приоритет задачи 1 в файле “define.h”, дав ей более высокий приоритет:

```
/*********************************************************************
 * ---------------------------- Task ID ------------------------------
 *********************************************************************/
#define TASK0_ID        1
#define TASK1_ID        2

#define TASK0_PRIO      7
#define TASK1_PRIO      10
```

Перекомпилируйте программу и запустите ее в симуляторе: теперь новый порядок точек останова следующий:

- **WaitEvent в задаче 0**
- **SetEvent в задаче 0**
- **ClearEvent в задаче 1**
- **WaitEvent в задаче 0**

Поскольку задача 1 теперь является задачей с более высоким приоритетом, когда происходит событие, то текущая задача сразу же останавливается, чтобы позволить задаче с более высоким приоритетом выполняться: здесь можно сказать, что это и есть **вытеснение** (preemptive означает подвесить).

Вытеснение это одно из важнейших свойств многозадачных операционных систем реального времени. Действительно, в ОС без вытеснения задача 0 будет продолжать работать, пока сама не впадет в спячку, чтобы позволить задаче 1 выполняться. Мы называем такие ОС кооперативными, так как текущая задача несет ответственность за решение, когда отдать управление обратно приложению. ОС **SALVO**, которая также управляет PIC18, является хорошим примером кооперативной ОС.

Но вытеснение не является достаточным основанием сказать, что ОС является ОСРВ или нет. Чтобы это сделать, мы поговорим о **детерменизме**, или о способности ОС гарантировать постоянное время переключения одной задачи на другую. Для PICos18 это время составляет 50 мкс, т.е. каждый раз, когда задача 0 получает сообщение от задачи 1, то время переключения с задачи 1 на задачу 0 составляет 50 мкс.

PICos18 является вытесняющей, многозадачной операционной системой реального времени, что позволяет при необходимости переключиться на задачу с повышенным приоритетом немедленно, если произошло событие, и необходимо только 50 мкс сделать это, каким бы сложным не было приложение. Теперь вы можете понять, как такой бесконечный цикл “while(1)” в задаче с высшим приоритетом будет блокировать остальное приложение. Попробуйте сделать это в симуляторе!

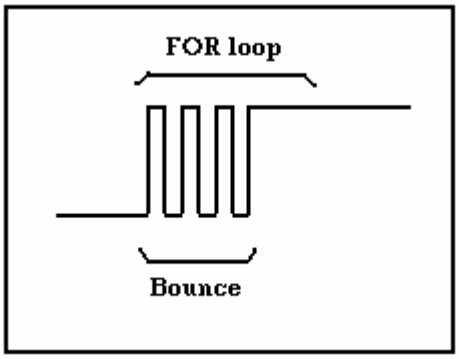
6. Многозадачное приложение

Этот раздел посвящен механизмам синхронизации между задачами в PICos18. Третья задача позволяет завершить приложение и отправить сообщение другой задаче с целью ее активизации. Мы научим пользоваться сообщениями, посланными от задачи или разделяемых ресурсов.

> Создание фоновой задачи

В предыдущих разделах мы создали приложение, состоящее из 2 задач, которые управляют классом программных часов с 3 переменными: hour, min и sec.

Многозадачность PICos18 дает не только возможность вам разделить проект на отдельные задачи для сокращения сложности приложения, но еще и создавать разные независимые подсистемы. Для примера мы откроем 2 дополнительные задачи для постоянной проверки портов RB0 и RB1 как 2 нажатые кнопки.



Нажатие кнопки не является обычным сигналом, который переходит из состояния 0 в состояние 1 и остается в положении 1 до следующего изменения.

Когда вы нажимаете на кнопку (чаще механическую), то обычно происходит дребезг в течение нескольких миллисекунд до получения стабильного сигнала 1. Мы тогда пишем код, который дожидается этого стабильного состояния перед чтением состояния нажатой кнопки.

Прежде всего, мы начнем с открытия первой задачи, которая проверяет порт RB0, к которому подключена кнопка BTN0.

1) В MPLAB сохраните файл “tsk_task0.c” как “**tsk_task2.c**”
2) Измените связанные с новой задачей определения в файле taskdesc.c:

```
#define DEFAULT_STACK_SIZE 128
DeclareTask(TASK0);
DeclareTask(TASK1);
DeclareTask(TASK2);

volatile unsigned char stack0[DEFAULT_STACK_SIZE];
volatile unsigned char stack1[DEFAULT_STACK_SIZE];
volatile unsigned char stack2[DEFAULT_STACK_SIZE];

/*************************************************************************
 * ---------------------- TASK DESCRIPTOR SECTION ---------------------
 *************************************************************************/
#pragma              romdata          DESC_ROM
...
/*************************************************************************
 * ----------------------------- task 1 -------------------------------
 *************************************************************************/
rom_desc_tsk rom_desc_task2 = {
  TASK2_PRIO,                       /* prioinit from 0 to 15          */
  stack2,                           /* stack address (16 bits)        */
```

```
  TASK2,                    /* start address (16 bits)      */
  READY,                    /* state at init phase          */
  TASK2_ID,                 /* id_tsk from 1 to 15          */
  sizeof(stack2)            /* stack size (16 bits)         */
};
```

3) Затем измените тело задачи 2 как показано ниже:

```
extern unsigned char hour;

/*************************************************************************
 * ---------------------------- TASK2 -----------------------------------
 *
 * Third task of the tutorial.
 *
 *************************************************************************/
TASK(TASK2)
{
  unsigned int i;

  while(1)
  {
    while(PORTBbits.RB0 == 0);
    for (i = 0; i < 10000; i++)
      Nop();
    if (PORTBbits.RB0 == 1)
      hour++;
  }
}
```

Задача имеет бесконечный цикл без ожидания события в функции WaitEvent. Поэтому задача может блокировать остальное приложение, более точно, задачу с более низким приоритетом... Тогда ее приоритет должен быть одним из самых низких в приложении: мы говорим, что такая задача является фоновой задачей.

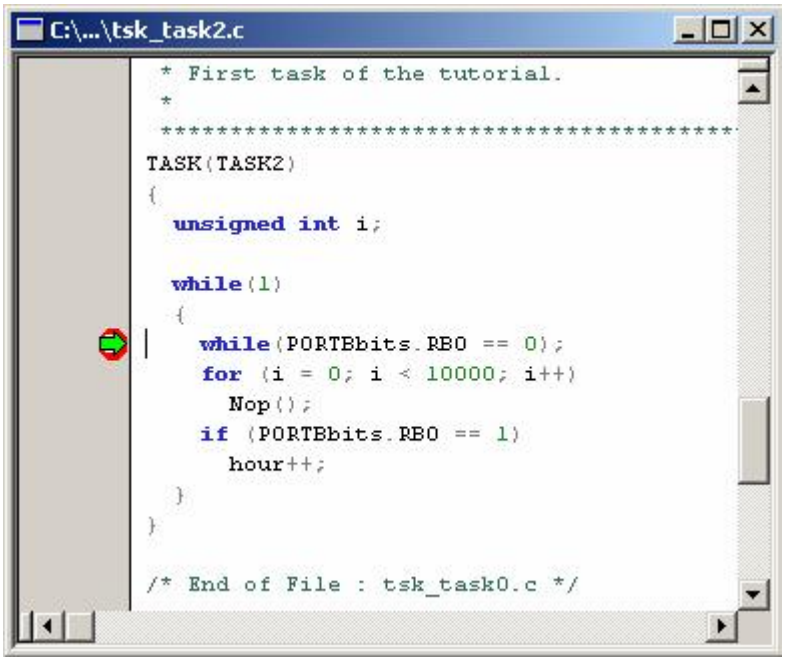
Она начинает отслеживать состояние порта RB0 и остается в бесконечном цикле так долго, пока кнопка не нажата. Как только кнопка нажимается, задача дожидается конца дребезга благодаря циклу FOR. Затем, если RB0 в состоянии ON, переменная “hour” увеличивается. Это позволяет обновить программные часы, например.

4) Теперь вам необходимо обновить связанные с новой задачей определения в файле “define.h”:

```
/*************************************************************************
 * ---------------------------- Task ID ---------------------------------
 *************************************************************************/
#define TASK0_ID 1
#define TASK1_ID 2
#define TASK2_ID 3

#define TASK0_PRIO 7
#define TASK1_PRIO 10
#define TASK2_PRIO 1
```

5) Наконец, вы можете добавить новую задачу в проект (“Project / Add files to Project…”).

Вы можете перекомпилировать и запустить программу (**F10**).



> Карусель

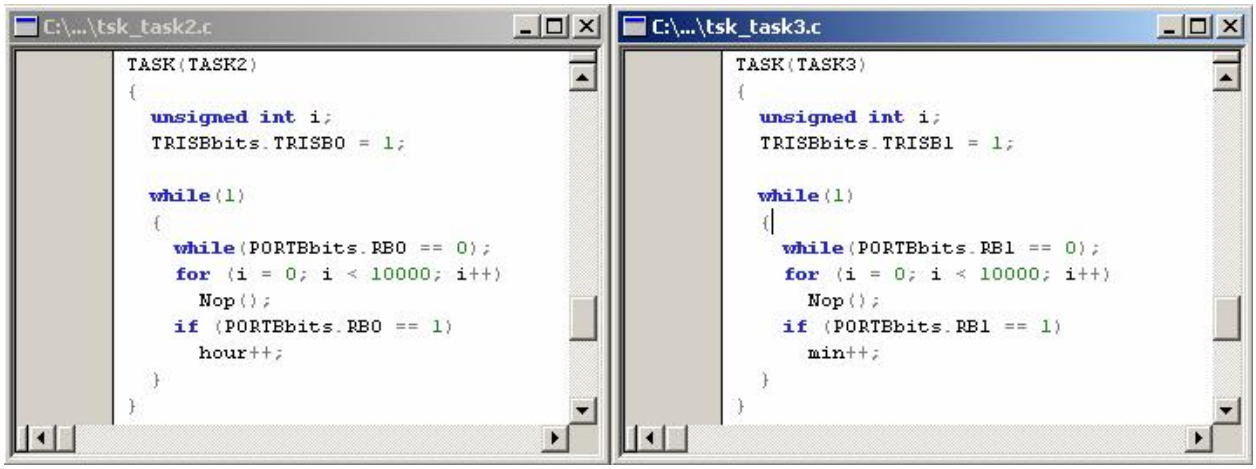
Теперь мы можем открыть вторую фоновую задачу. На самом деле PICos18 позволяет вам сделать много разных задач с одним приоритетом. В нашем приложении мы удивляемся, как это все может работать... В действительности планировщик PICos18 использует карусельный (round-robin) алгоритм.

В соответствии с алгоритмом фоновая задача может **удерживать процессор PIC18 в течение отрезка времени в 1 мс**. Поскольку задача имеет очень низкий приоритет, она может работать только тогда, когда ни одной из задач не нужно выполняться, и только в течение 1 мс максимум. По истечении этого времени другая задача с таким же приоритетом будет выполняться в течение такого же отрезка времени.

Создайте задачу 3 с таким же кодом (за исключением того, что проверяется порт RB1) и с таким же приоритетом:

```
/*********************************************************************
 * ---------------------------- Task ID ------------------------------
 *********************************************************************/
#define TASK0_ID 1
#define TASK1_ID 2
#define TASK2_ID 3
#define TASK3_ID 4

#define TASK0_PRIO 7
#define TASK1_PRIO 10
#define TASK2_PRIO 1
#define TASK3_PRIO 1
```

Не забудьте добавить новый файл в проект и перекомпилировать его (**F10**).

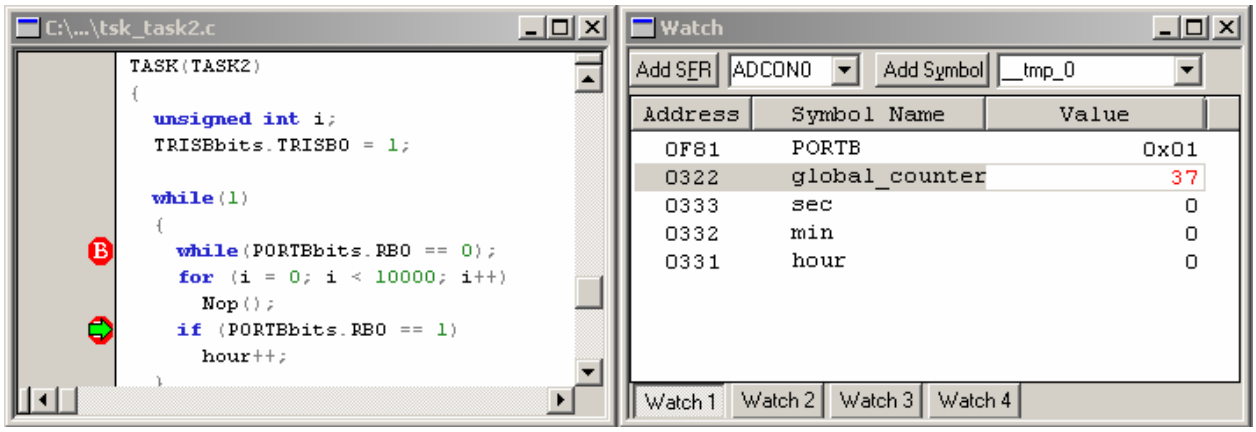


Поставьте несколько точек останова перед командой “**while(PORTBbits.RBx == 0);**” и запустите приложение в симуляторе. Вы увидите, что указатель остановился в задаче 2. Отключите точку останова, где находится указатель, и запустите приложение: указатель остановился теперь в задаче 3 спустя 1 мс…

Если теперь вы поставите точки останова в задаче 0 и 1, вы увидите указатель, остановившийся спустя 1 секунду в одной из 2 задач: это работает вытеснение, потому что более высокий приоритет у обеих задач.

> Трассировка в симуляторе

Для имитации нажатия кнопки в положение ON вы можете использовать окно просмотра Watch:



Запустите приложение и подождите, пока указатель не остановится на первой точке останова задачи 2. Кликните на окошке Watch для задания значения 0x01 порта PORTB. Затем запустите приложение и проверьте точку останова в задаче 3, если она до сих пор разрешена.

Вы увидите указатель, остановившийся в задаче 3 перед достижением второй точки останова в задаче 2. Фактически через 1 мс планировщик решил запустить другую задачу…

7. Прерывания

Пользуясь микроконтроллерами PIC18, вы, вероятно, планировали работать с периферией. Делая это, вы могли рассматривать прерывания.

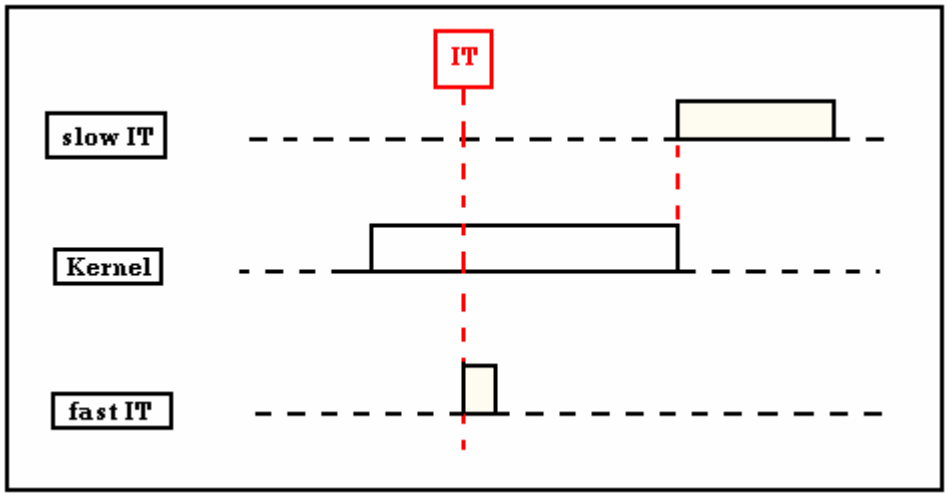
Вы научитесь работать с прерываниями в PICos18 и писать свои обработчики прерываний.

> Особенности файла int.c

Сейчас мы подключим обе кнопки к выводу PIC18 с внешними прерываниями, что позволит нам обрабатывать дребезг кнопки только тогда, когда она нажата.

Прежде всего, мы посмотрим, что представляет собой файл int.c в плане управления прерываниями. PIC18 может управлять двумя группами прерываний: с низким приоритетом (IT_vector_low) и с высоким приоритетом (IT_vector_high). Ну и что…?!

На самом деле вы должны знать, что ОС PICos18 производит ряд операций с задачей и что нельзя прерывать течение этого процесса (к примеру, в течение вытеснения текущей задачи новой). Тогда для упрощения, насколько это возможно, ядро ОС сделано непрерываемым. Но мы также видим детерминизм PICos18 в 50 мкс (также называемым временем задержки, или отклика), как своего рода черная дыра, когда прерывания не могут быть распознаны.



Чтобы обойти эту проблему, мы используем быстрое прерывание PIC18, использующее аппаратный механизм по сохранению простых регистров (как WREG, STATUS, …). К несчастью, при использовании быстрого прерывания PIC18 вы должны писать очень простой код на С…

Если же вам необходимо писать более сложный код, как вызов С-функции, например, вы должны пользоваться медленными прерываниями… с этой черной дырой в 50 мкс!

```
/*********************************************************************
 * Function you want to call when an IT occurs.
 ********************************************************************/
  extern void AddOneTick(void);
/*extern void MyOwnISR(void); */
```

```
  void InterruptVectorL(void);
  void InterruptVectorH(void);
/*****************************************************************
 * General interrupt vector. Do not modify.
 *****************************************************************/
#pragma code IT_vector_low=0x18
void Interrupt_low_vec(void)
{
  _asm goto InterruptVectorL _endasm
}
#pragma code

#pragma code IT_vector_high=0x08
void Interrupt_high_vec(void)
{
  _asm goto InterruptVectorH _endasm
}
#pragma code

/*****************************************************************
 * General ISR router. Complete the function core with the if or switch
 * case you need to jump to the function dedicated to the occuring IT.
 *****************************************************************/
#pragma code _INTERRUPT_VECTORL = 0x003000
void InterruptVectorL(void)
{
  SAVE_TASK_CTX(stack_low, stack_high);
  EnterISR();

  if (INTCONbits.TMR0IF == 1)
    AddOneTick();
/*Here is the next interrupts you want to manage */
/*if (INTCONbits.RBIF == 1) */
/* MyOwnISR(); */

  LeaveISR();
  RESTORE_TASK_CTX;
}
#pragma code

/* BE CARREFULL : ONLY BSR, WREG AND STATUS REGISTERS ARE SAVED */
/* DO NOT CALL ANY FUNCTION AND USE PLEASE VERY SIMPLE CODE LIKE */
/* VARIABLE OR FLAG SETTINGS. CHECK THE ASM CODE PRODUCED BY C18 */
/* IN THE LST FILE. */
#pragma code _INTERRUPT_VECTORH = 0x003300
#pragma interrupt InterruptVectorH
void InterruptVectorH(void)
{
  if (INTCONbits.INT0IF == 1)
    INTCONbits.INT0IF = 0;
}
#pragma code
```

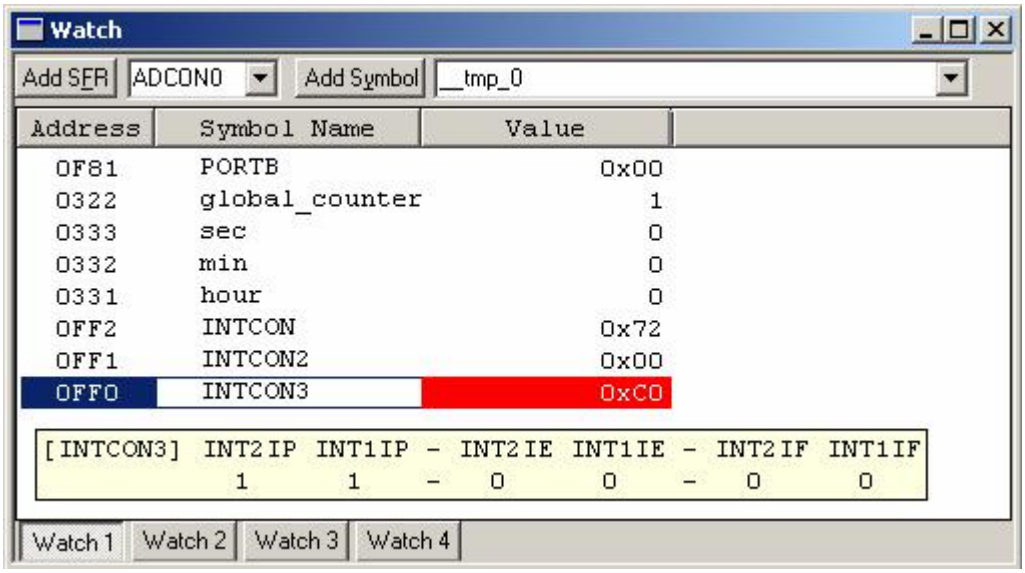
На первый взгляд файл int.c кажется очень трудным для прочтения, но на самом деле он очень прост для понимания. Достаточно будет изменять только 2 области: это функции InterruptVectorL() и InterruptVectorH(). Почти в 99% случаев медленные прерывания будут использоваться вашим приложением, поэтому реально вам необходимо будет изменять только функцию InterrupVectorL().

> Режим «быстрых прерываний»

Ниже представлена функция обработки “быстрого прерывания”:

```
/* BE CARREFULL : ONLY BSR, WREG AND STATUS REGISTERS ARE SAVED */
/* DO NOT CALL ANY FUNCTION AND USE PLEASE VERY SIMPLE CODE LIKE */
/* VARIABLE OR FLAG SETTINGS. CHECK THE ASM CODE PRODUCED BY C18 */
/* IN THE LST FILE. */
#pragma code _INTERRUPT_VECTORH = 0x003300
#pragma interrupt InterruptVectorH
void InterruptVectorH(void)
{
  if (INTCONbits.INT0IF == 1)
  INTCONbits.INT0IF = 0;
}
#pragma code
```

Внешние прерывания **INT0 (RB0 вывод), INT1 (RB1 вывод) и INT2 (RB2 вывод)** по умолчанию находятся в режиме быстрого прерывания, поэтому в момент прерывания выполнение кода программы останавливается и выполняется функция InterruptVectorH().



Прерывание INT0 не настраиваемое, что означает вызов только функции InterruptVectorH(). В противоположность этому INT1 и INT2 могут быть настроены на вызов функции InterruptVectorL(). Чтобы это сделать, измените в регистре **INTCON3** биты INT1IP и INT2IP.

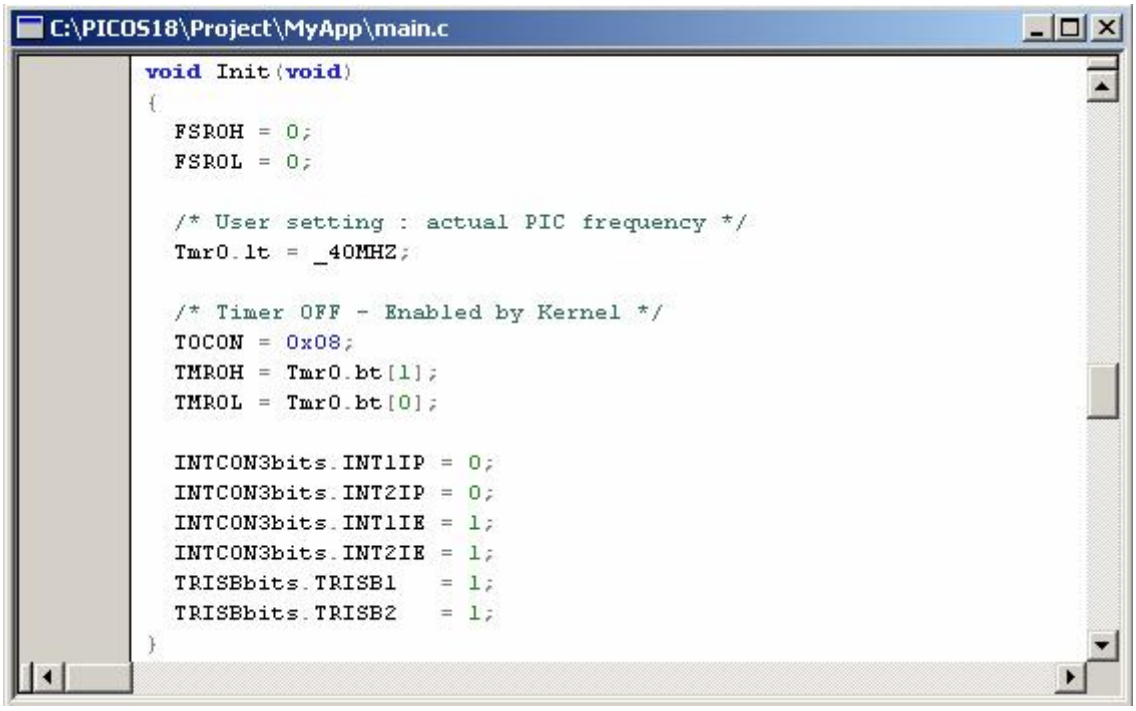
Если вы желаете подключить две кнопки на выводы INT1 и INT2, вам необходимо написать код для подавления дребезга в функции InterruptVectorH(). Но вы видите, что код обработки быстрого прерывания должен быть очень простым и не содержать вызовов функций… Только одну вещь мы можем делать – это увеличивать или уменьшать глобальную переменную!...

Обоснование очень простое: вы не можете использовать регистры PIC18, кроме WREG, STATUS и BSR. Если вы колеблетесь в выборе между быстрым и медленным прерываниями, просмотрите листинг ассемблерного кода, который генерирует компилятор C18, чтобы узнать, можно ли использовать другой регистр. Когда у нас нет

необходимости в точности управления менее 50 мкс теми же кнопками, мы выбираем режим “**медленных прерываний**”.

> Режим «медленных прерываний»

В большинстве случаев возможно применить режим “медленных прерываний”. По умолчанию только 3 прерывания находятся в режиме “быстрых прерываний”. Это INT0, INT1 и INT2. Поскольку режим INT0 невозможно изменить, мы будем использовать INT1 и INT2 в режиме “медленных прерываний”:



Для использования INT1 и INT2 в режиме “медленных прерываний”, измените регистр INTCON3, как это показано ранее на стадии инициализации в файле main.c.

Затем измените файл int.c для захвата прерываний от INT1 и INT2:

```
/*************************************************************************
 * Function you want to call when an IT occurs.
 ************************************************************************/
  extern void AddOneTick(void);
  extern void MyOwnISR(void);
  void InterruptVectorL(void);
  void InterruptVectorH(void);

...

/*************************************************************************
 * General ISR router. Complete the function core with the if or switch
 * case you need to jump to the function dedicated to the occuring IT.
 ************************************************************************/
#pragma code _INTERRUPT_VECTORL = 0x003000
void InterruptVectorL(void)
{
  SAVE_TASK_CTX(stack_low, stack_high);
```

```
  EnterISR();

  if (INTCONbits.TMR0IF == 1)
    AddOneTick();
/*Here is the next interrupts you want to manage */
  if (INTCON3bits.INT1IF || INTCON3bits.INT2IF)
    MyOwnISR();

  LeaveISR();
  RESTORE_TASK_CTX;
}
#pragma code
```

Нам необходимо теперь добавить функцию “MyOwnISR” в какой-нибудь C-файл с одной из двух фоновых задач, обрабатывающих кнопку. Для примера мы добавляем эту функцию в файл “tsk_task2.c”:

```
void MyOwnISR(void)
{
  if (INTCON3bits.INT1IF)
  {
    INTCON3bits.INT1IF = 0;
    /* ... */
  }
  if (INTCON3bits.INT2IF)
  {
    INTCON3bits.INT2IF = 0;
    /* ... */
  }
}
```

Не забудьте объявить функцию в начале файла:

```
#include "define.h"

void MyOwnISR(void);

/*************************************************************************
 * Definition dedicated to the local functions.
 *************************************************************************/
#define ALARM_TSK0      0
```

Теперь можно перекомпилировать проект (**F10**).

> Управление фоновыми задачами

На данный момент функция “MyOwnISR” не посылает никакой информации о кнопке, когда та нажата. Более того, фоновая задача продолжает работать в бесконечном цикле. Тогда нам надо погрузить в сон фоновые задачи и пробуждать их только при нажатии кнопки. Чтобы это сделать, изменим обе задачи быть основной задачей:

```
TASK(TASK2)
{
  unsigned int i;
  for (i = 0; i < 10000; i++)
    Nop();
  if (PORTBbits.RB1 == 1)
```

```
    hour++;
  TerminateTask();
}
```

Как вы можете заметить, задача номер 2 больше не имеет никакого бесконечного цикла “while(1)” и заканчивает работу вызовом функции “TerminateTask()”.

Вот особенности основной (BASIC) задачи:

- отсутствует вызов функции “WaitEvent()”,
- отсутствует бесконечный цикл “while(1)”,
- в конце задачи вызывается функция “TerminateTask()”.

Целью является сначала перевести задачу в состояние **SUSPENDED**, а затем активизировать ее, вызвав функцию “ActivateTask()”.

Измените файл “taskdesc.c”, задав по умолчанию состояние задач 2 и 3 SUSPENDED:

```
/*********************************************************************
 * ---------------------------- task 2 -----------------------------
 *********************************************************************/
rom_desc_tsk rom_desc_task2 = {
  TASK2_PRIO,                       /* prioinit from 0 to 15          */
  stack2,                           /* stack address (16 bits)        */
  TASK2,                            /* start address (16 bits)        */
  SUSPENDED,                        /* state at init phase            */
  TASK2_ID,                         /* id_tsk from 1 to 15            */
  sizeof(stack2)                    /* stack size (16 bits)           */
};
/*********************************************************************
 * ---------------------------- task 3 -----------------------------
 *********************************************************************/
rom_desc_tsk rom_desc_task3 = {
  TASK3_PRIO,                       /* prioinit from 0 to 15          */
  stack3,                           /* stack address (16 bits)        */
  TASK3,                            /* start address (16 bits)        */
  SUSPENDED,                        /* state at init phase            */
  TASK3_ID,                         /* id_tsk from 1 to 15            */
  sizeof(stack2)                    /* stack size (16 bits)           */
};
```

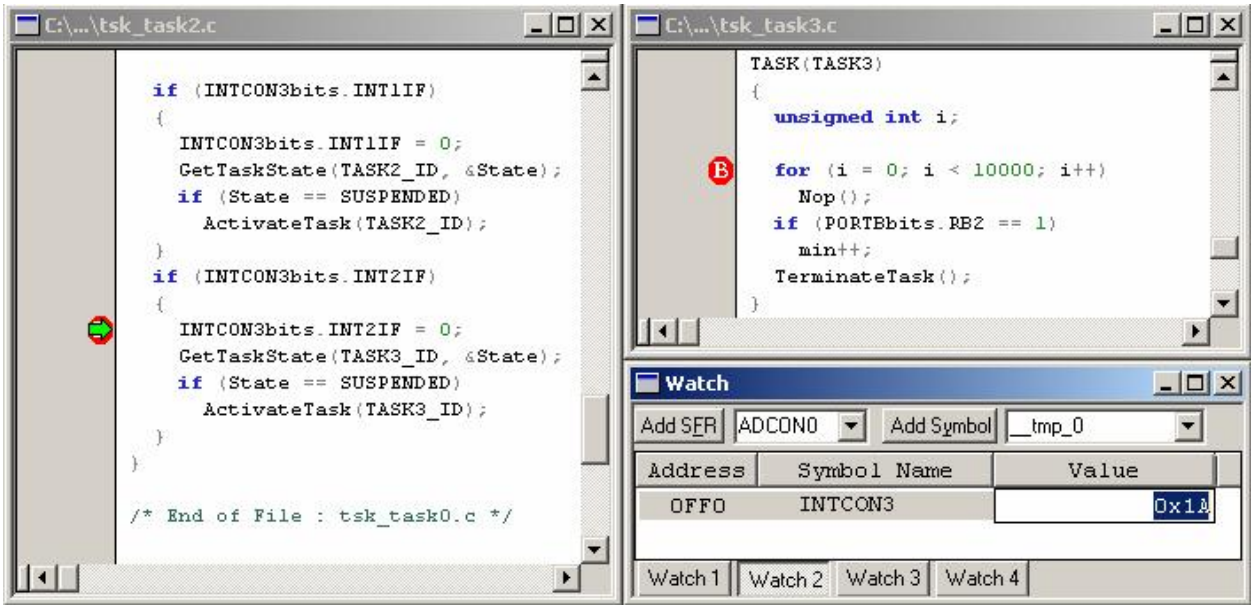
Измените задачу 3, чтобы она стала BASIC задачей:

```
TASK(TASK3)
{
  unsigned int i;

  for (i = 0; i < 10000; i++)
    Nop();
  if (PORTBbits.RB2 == 1)
    min++;

  TerminateTask();
}
```

Измените функцию “MyOwnISR”, чтобы активировать задачу, предназначенную для обработки кнопки:

```
void MyOwnISR(void)
{
  TaskStateType State;

  if (INTCON3bits.INT1IF)
  {
    INTCON3bits.INT1IF = 0;
    GetTaskState(TASK2_ID, &State);
    if (State == SUSPENDED)
      ActivateTask(TASK2_ID);
  }
  if (INTCON3bits.INT2IF)
  {
    INTCON3bits.INT2IF = 0;
    GetTaskState(TASK3_ID, &State);
    if (State == SUSPENDED)
      ActivateTask(TASK3_ID);
  }
}
```

Перекомпилируйте проект (**F10**) и запустите программу в симуляторе (**F9**).



Поставьте одну точку останова в задаче 3, а другую точку в функции “MyOwnISR”. Запустите программу и остановите спустя какое-то время (**F5**). Измените INTCON3, чтобы вызвать прерывание искусственно (задайте, к примеру, значение 0x1A для прерывания от INT2): вы увидите, что указатель остановится сначала на “MyOwnISR”, а затем на задаче 3.

Если вы продолжите дальше выполнение программы, вы увидите, что указатель не будет входить в задачу 3 никогда, до следующего прерывания: **задача находится в состоянии остановки**.

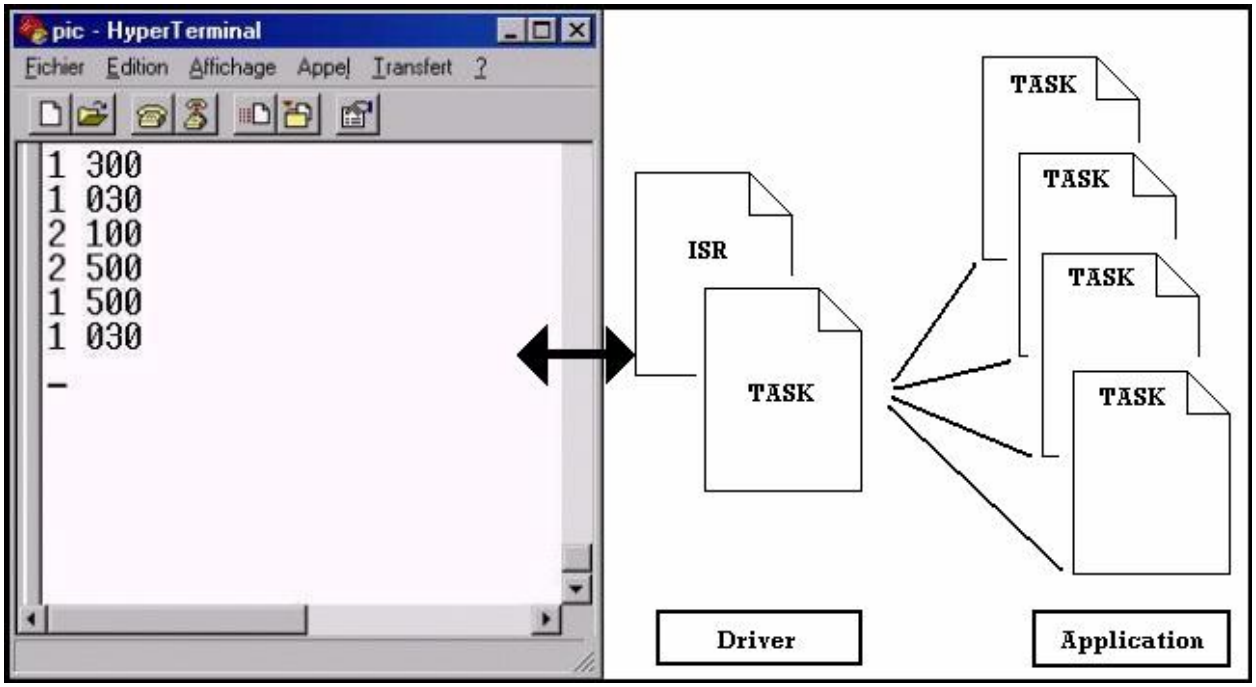
8. Использование драйверов

Вы желаете связываться через последовательный порт или CAN интерфейс? Теперь вы знаете, как обрабатывать прерывания и писать задачи и вы знаете, как управлять таймаутами и сообщениями, и это очень облегчает использование и написание настоящих драйверов для PICos18.

> Особенности драйверов

Драйвер часто сравнивают с библиотекой, но на самом деле имеются некоторые различия. Библиотека является набором функций, написанные на языке программирования (на C, например) и используется для упрощения процесса разработки. В случае передачи данных через порт RS232 библиотека предлагает, к примеру, функцию по открытию или закрытию канала, или по приему и передаче данных…

Драйвер это слой, написанный согласно специфике управления периферией. Задача драйвера не предлагать набор функций по использованию периферии, а это и есть работа драйвера быть ответственным за использование периферии.



Драйвер обычно состоит из ISR (Interrupt Service Routine, Обработчик Прерывания) и задачи. Мы уже рассматривали прерывание в предыдущем разделе в функциях InterruptVectorL() и InterruptVectorH().

ISR настроен на захват прерывания (IT) и для последующего информирования задачи драйвера о произошедшем прерывании. Поэтому миссией задачи является сделать что-то необходимое по управлению этим прерыванием.

> Использование программного слоя

PICos18 распространяется как OCPB вместе с обучающей программой. Но на странице **DOWNLOAD** вебсайта мы можем найти также свободно доступные драйверы и библиотеки. Это позволяет вам разрабатывать приложения как мозаику, с множеством **свободных слоев**, которые вы можете добавлять в свой проект, не тратя времени зря на изучение, как это работает.

Давайте изучим, как использовать программный слой. Точнее, мы будем использовать драйвер RS232 для PICos18 для отображения наших “программных часов” с глобальными переменными “hour”, “min” и “sec”.

Скачайте драйвер RS232 со странички DOWNLOAD PICos18:

Пакет содержит текстовый файл (с расширением TXT), который показывает, как включить и использовать драйвер в вашем приложении (как задать скорость или как зарегистрировать задачу RS драйвера).

```
_____________________________________________________________________
>                                                                    <
>                          RS232 driver v1.06                        <
>                                for                                 <
>                          PICos18 release 2.00                      <
>                                                                    <
>              PICos18 - Real-time kernel for PIC18 family           <
>                                                                    <
>                                                                    <
> www.picos18.com                                  www.pragmatec.net <
>____________________________________________________________________<

...
```

Теперь вы можете вставить задачу в свой проект, написать ISR, настроить ваше приложение благодаря файлу “taskdesc.c”, затем, следуя рекомендациям текстового файла, шаг за шагом включить драйвер в ваше приложение.

Вам не надо добавлять ссылки задачи “Example”, такие как таймаут в структуре Alarm_list (раздел IV в текстовом файле) или определение EXAMPLE_ID в файле define.h. Не берите ничего из раздела VII, поскольку это будет использоваться в следующем параграфе.

> Использование драйвера

Для использования драйвера создайте новую задачу под именем “TASK4” взамен имени задачи “Example” в текстовом файле. Не забудьте добавить описание задачи и настройки таймаута ALARM_TASK4 в файле “taskdesc.c”.

Вот как должна выглядеть задача TASK4:

```
#include "define.h"
#include "drv_rs.h"
#include <stdio.h>
#include <string.h>

#define ALARM_TSK4 1

int Printf (const rom char *fmt, ...);
extern unsigned char hour;
extern unsigned char min;
extern unsigned char sec;

/*************************************************************************
 * Definition dedicated to the local functions.
 *************************************************************************/
RS_message_t RS_msg;
unsigned char buffer[80];

/*************************************************************************
 * ------------------------------ TASK4 -------------------------------
 *
 * Fifth task of the tutorial.
 *
 *************************************************************************/
TASK(TASK4)
{
  SetRelAlarm(ALARM_TSK4, 1000, 1000);
  Printf(" _____________________________________________________________________ \r\n");
  Printf(">                                                                     <\r\n");
  Printf(">                        PICos18 Tutorial                             <\r\n");
  Printf(">                                                                     <\r\n");
  Printf(">                                                                     <\r\n");
  Printf(">          PICos18 - Real-time kernel for PIC18 family                <\r\n");
  Printf(">                                                                     <\r\n");
  Printf(">                                                                     <\r\n");
  Printf("> www.picos18.com                                 www.pragmatec.net <\r\n");
  Printf(">_____________________________________________________________________<\r\n");
  Printf(" \r\n");
  Printf(" \r\n");

  while(1)
  {
    WaitEvent(ALARM_EVENT);
    ClearEvent(ALARM_EVENT);
    Printf("%02d : %02d : %02d\r", (int)hour, (int)min, (int)sec);
  }
}
```

```
/*********************************************************************
 * Function in charge of structure registration and buffer transmission.
 *
 * @param string IN const string send to the USART port
 * @return void
*********************************************************************/
int Printf (const rom char *fmt, ...)
{
  va_list ap;
  int n;
  RS_enqMsg(&RS_msg, buffer, 50);
  va_start (ap, fmt);
  n = vfprintf (_H_USER, fmt, ap);
  va_end (ap);
  SetEvent(RS_DRV_ID, RS_NEW_MSG);
  WaitEvent(RS_QUEUE_EMPTY);ClearEvent(RS_QUEUE_EMPTY);
  return n;
}
```

Первая часть кода используется для подключения головных файлов драйвера RS232 и функции “printf”. Затем вам необходимо объявить идентификатор ALARM_TASK4 и еще объявить как “extern” внешние переменные “hour”, “min” и “sec”, чтобы указать компилятору C18, что их определения отсутствуют в этом C файле.

Мы добавили в память 2 вида переменных:

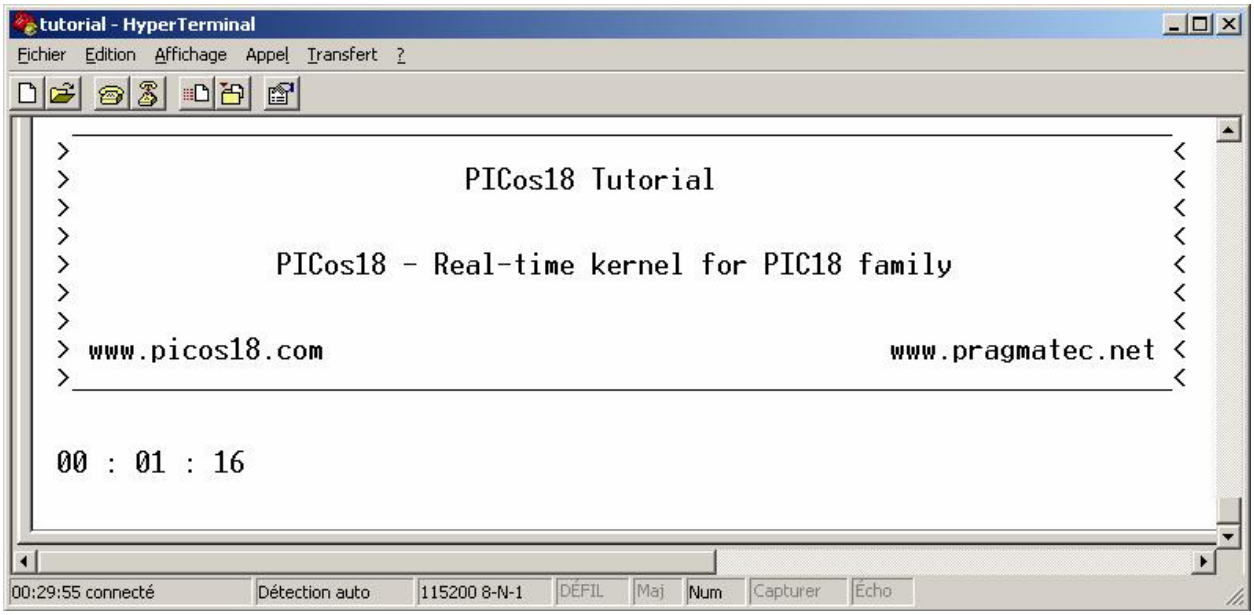
- **RS_message_t RS_msg**: Этой структуре предписано отправлять что-нибудь через RS-драйвер
- **unsigned char buffer[80]**: это буфер для отправки сообщений

Далее исследуйте тело задачи TASK4 с таймаутом в 1000 мс, т.е. 1с. Функция “Printf” вызывается для отправки постоянных строк, конвертирования данных из одного типа в другой,… Все форматы поддерживаются C18, за исключением переменных FLOAT…!

Функции “Printf” необходима подстрока “\r\n” в конце строки, если вам хочется перейти на новую строку: “\r” используется для возврата каретки к началу строки, а “\n” – для перехода на новую строку.

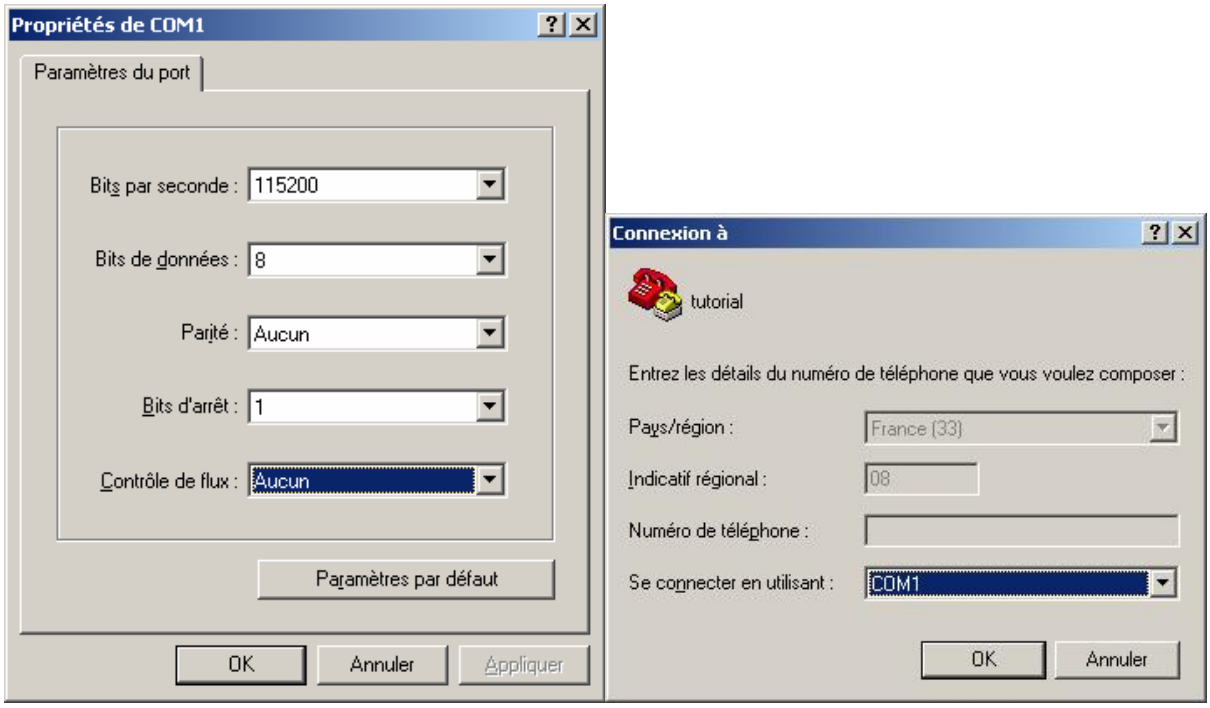
Для отображения баннера нам нужно сделать “\r\n” и отобразить время в одном и том же месте, используя “\r”.

В итоге мы получим следующее сообщение в HyperTerminal:



> Настройки HyperTerminal

Будьте внимательны с настройками COM-порта: установите скорость 115200 бит в секунду и без аппаратного бита четности.



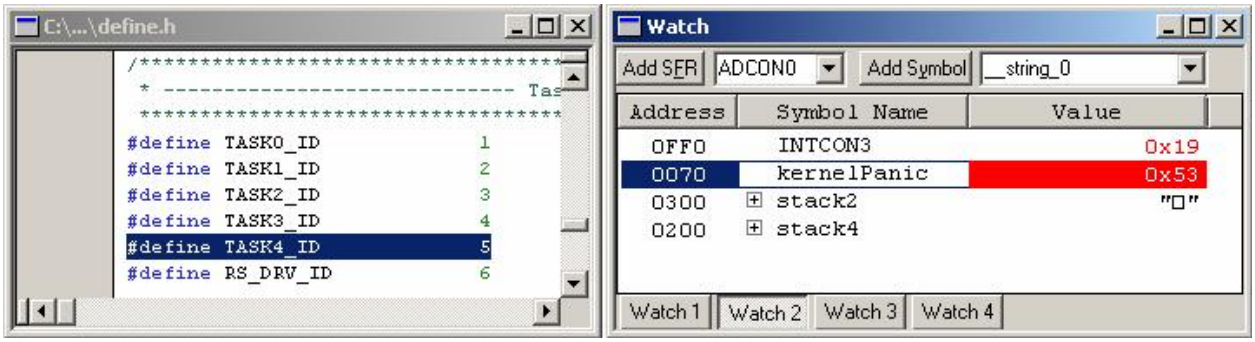
9. Пример приложения

Обучение заканчивается на этом последнем разделе по разработке PICos18. Исходные тексты этого типового приложения находятся в директории дистрибутивов Project/Tutorial PICos18. Этот пример представлен, чтобы показать вам, как легко разрабатывать приложения в комплексе с PICos18.

> Множественные прерывания

Если вы запустите приложение прямо сейчас и нажмете одну из двух кнопок, то вы увидите достаточно быстро такую проблему: PIC18 как будто бы «замерзает» после нажатия, и это случается, если вы нажимаете кнопку слишком быстро!

Остановите приложение, если вы пользуетесь Microchip ICD2 в режиме отладчика, или остановите симулятор, если вы в симуляторе, и посмотрите на следующие регистры:



Вы должны заметить переменную “kernel_panic”, которая имеет значение, отличное от нуля. В нашем случае это 0x53.

В настоящий момент эта переменная показывает, что произошло переполнение стека. Когда стек пересекает свою границу, то ядро автоматически обнаруживает это и останавливает задачу как источник проблемы. Это позволяет сохранить ваше приложение от дальнейшего разрушения, а также отладить PICos18 после обнаружения, что же случилось.

Переменная kernel_panic имеет значение 0x53.

Правая тетрада, или нибл, (“3”) означает, что задача с ID 3 имеет переполнение стека.

Левая тетрада, или нибл, (“5”) означает, что задача с ID 5 имеет стек, испорченный в результате переполнения.

Ядро затем принимает решение остановить обе задачи, поскольку поведение обеих задач небезопасно. Этот механизм безопасности позволяет PIC18 работать, даже если стек переполнен, и позволяет вам исправить ошибку.

Задача 2, предназначенная для обработки кнопки, работает ненормально, и ее стек переполнился, который, в свою очередь, разрушил смежный стек задачи 4.

Но как произошло переполнение?

Здесь возможны 2 случая:

- код задачи слишком велик, и вы недооценили необходимый вам размер стека. Просто увеличьте размер стека как источник проблем.
- Бесконечные прерывания заставляют задачу сохранять контекст (регистры PIC18 и аппаратный стек) в цикле, что подразумевает переполнение стека.

Мы жмем на кнопки, и мы знаем из предыдущего раздела, что эти кнопки имеют некоторый дребезг, когда их нажимают. Возникающие в результате прерывания и вызывают перегрузку PIC18. От множества прерываний текущая задача вынуждена сохранять свой контекст, но прерывание возникает опять, и мы, в конце концов, оказываемся в бесконечном цикле обработчика прерывания InterruptVectorL() вплоть до наступления переполнения стека…

Такое поведение случается довольно часто, даже если вы не используете OCPB. С одним только отличием, что PICos18 следит за приложением в “реальном времени” и, если случается переполнение стека, то останавливает задачу при необходимости. Без ОС PIC18 рухнул бы после того как… На самом деле это определяется не архитектурой, а трудностью кода, который мы исправим в следующем разделе.

> Процесс отладки

Проблема возникает из-за того, что мы всегда разрешаем прерывания от портов RB1 и RB2, когда мы даже не начали обработку первого прерывания. Обрабатывать каждое прерывание от кнопки нет необходимости: нет нужды знать, что кнопка до стабилизации сигнала имела дребезг 50 раз. Нам только надо знать, что она была нажата.

Измените задачи TAS2 и TASK3:

```
TASK(TASK2)
{
  unsigned int i;

  hour++;
  if (hour == 24)
    hour = 0;
  INTCON3bits.INT1IF = 0;
  INTCON3bits.INT1IE = 1;
  TerminateTask();
}

...

TASK(TASK3)
{
  unsigned int i;

  min++;
  if (min == 60)
    min = 0;
  INTCON3bits.INT2IF = 0;
  INTCON3bits.INT2IE = 1;
  TerminateTask();
}
```

И измените ваш обработчик прерываний:

```
void MyOwnISR(void)
{
  TaskStateType State;

  if (INTCON3bits.INT1IF)
  {
    INTCON3bits.INT1IE = 0;
    GetTaskState(TASK2_ID, &State);
    if (State == SUSPENDED)
      ActivateTask(TASK2_ID);
  }
  if (INTCON3bits.INT2IF)
  {
    INTCON3bits.INT2IE = 0;
    GetTaskState(TASK3_ID, &State);
    if (State == SUSPENDED)
      ActivateTask(TASK3_ID);
  }
}

...

void InterruptVectorL(void)
{
  SAVE_TASK_CTX(stack_low, stack_high);
  EnterISR();

  if (INTCONbits.TMR0IF == 1)
    AddOneTick();
/*Here is the next interrupts you want to manage */
  if ((INTCON3bits.INT1IF & INTCON3bits.INT1IE)||
      (INTCON3bits.INT2IF & INTCON3bits.INT2IE))
    MyOwnISR();
  if ((PIR1bits.RCIF)&(PIE1bits.RCIE))
    RS_RX_INT();
  if ((PIR1bits.TXIF)&(PIE1bits.TXIE))
    RS_TX_INT();

  LeaveISR();
  RESTORE_TASK_CTX;
}
```

Вы можете увидеть, что MyOwnISR() запрещает прерывания (INTxIE = 0 означает Interrupt Enable запрещены), и задача может обработать флаг разрешения прерывания один раз и закончить процесс обработки прерывания.

Теперь вы можете запрограммировать ваш PIC18 для тестирования приложения: вы сможете изменить показания ваших программных часов, нажимая обе кнопки. Но есть еще проблема: если вы быстро нажимаете кнопку много раз, то новое значение не появится сразу, а только каждую секунду, когда дисплей обновится. Для исправления нам нужны способности реального времени PICos18!

> Обновления в режиме реального времени

Для обновления дисплея каждый раз при нажатии на кнопку, мы должны информировать дисплей каждый раз, когда появляется новое значение… но не беспокоя дисплей, который работает раз в секунду, что значит, не нарушая поведение наших программных часов.

Сделаем так. Нам необходимо послать сообщение задаче TASK4:

```
TASK(TASK2)
{
  unsigned int i;

  hour++;
  if (hour == 24)
  hour = 0;
  SetEvent(TASK4_ID, UPDATE_EVENT);
  INTCON3bits.INT1IF = 0;
  INTCON3bits.INT1IE = 1;
  TerminateTask();
}

...

TASK(TASK3)
{
  unsigned int i;

  min++;
  if (min == 60)
    min = 0;
  SetEvent(TASK4_ID, UPDATE_EVENT);
  INTCON3bits.INT2IF = 0;
  INTCON3bits.INT2IE = 1;
  TerminateTask();
}
```

Затем задача TASK4 должна дождаться этого нового сообщения:

```
...

  while(1)
  {
    WaitEvent(ALARM_EVENT | UPDATE_EVENT);
    GetEvent(id_tsk_run, &Time_event);

    if (Time_event & ALARM_EVENT)
      ClearEvent(ALARM_EVENT);
    if (Time_event & UPDATE_EVENT)
      ClearEvent(UPDATE_EVENT);

    Printf("%02d : %02d : %02d\r", (int)hour, (int)min, (int)sec);
  }

...
```

Не забудьте добавить переменную “EventMaskType Time_event” как глобальную или локальную переменную.

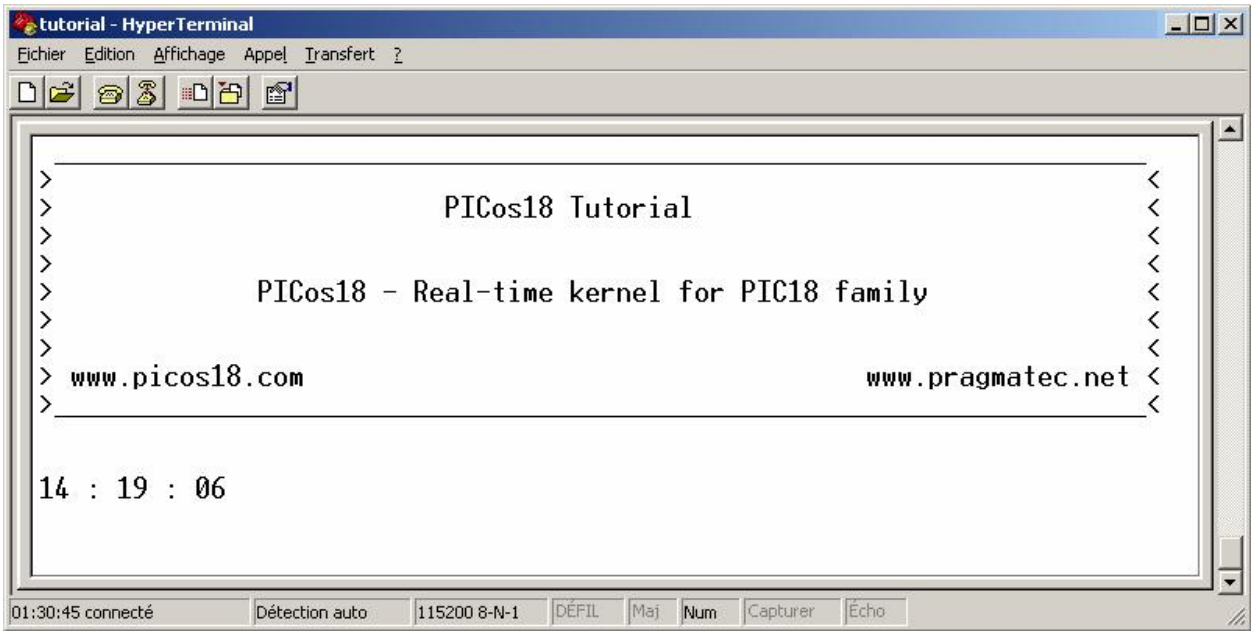
Определение UPDATE_EVENT неизвестно нашему приложению, и тогда добавьте его в файл define.h:

```
/*****************************************************************
 * ---------------------------- Events -----------------------------
 *****************************************************************/
#define ALARM_EVENT 0x80
#define TASK1_EVENT 0x10
#define UPDATE_EVENT 0x02
```

```
#define RS_NEW_MSG 0x10
#define RS_RCV_MSG 0x20
#define RS_QUEUE_EMPTY 0x10
#define RS_QUEUE_FULL 0x20
...
```

> Законченное приложение

Теперь вы можете перекомпилировать приложение, загрузить в PIC18F452 и протестировать его. Вы увидите программные часы, отображаемые в HyperTerminal, и вы сможете настраивать их в реальном времени, нажимая кнопки.



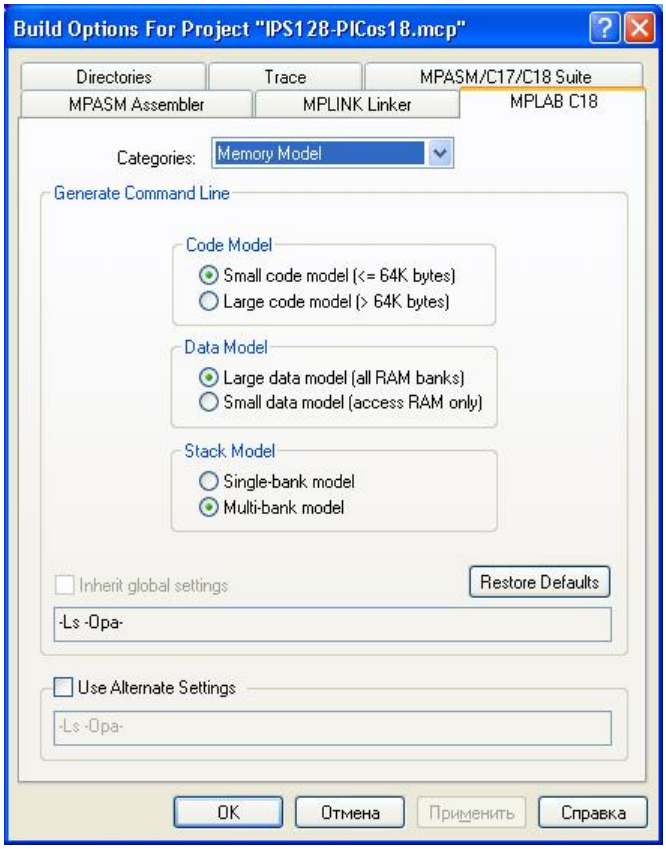
Конечно, это простое приложение. Это не значит, что все сделано, просто обучение упрощено так, насколько это возможно. Вы можете добавить собственный код и воспользоваться драйверами PICos18: например, вы можете добавить RTC (Real Time Clock, часы реального времени) с использованием шины I2C (в таком случае вам понадобится драйвер I2C master) и настраивать их из HyperTerminal…

> Послесловие и дополнение к русскому переводу

Жизнь не стоит на месте, и нет предела совершенству. В результате использования операционной системы PICos18 v2.13 были выявлены существенные ошибки как самой ОС, так и компилятора C18. Анализ и дальнейшее совершенствование ОС привели к появлению обновленной версии PICos18 v3.00. Помимо приведения алгоритма работы ядра ОС в полное соответствие руководству, был добавлен своего рода индикатор использования стека задачи и сняты существовавшие ограничения на размер и размещение стеков.

PICos18 v 2.01

Вы уже, очевидно, скачали последнюю версию PICos18. Теперь настройте компилятор так, как показано на скриншоте.

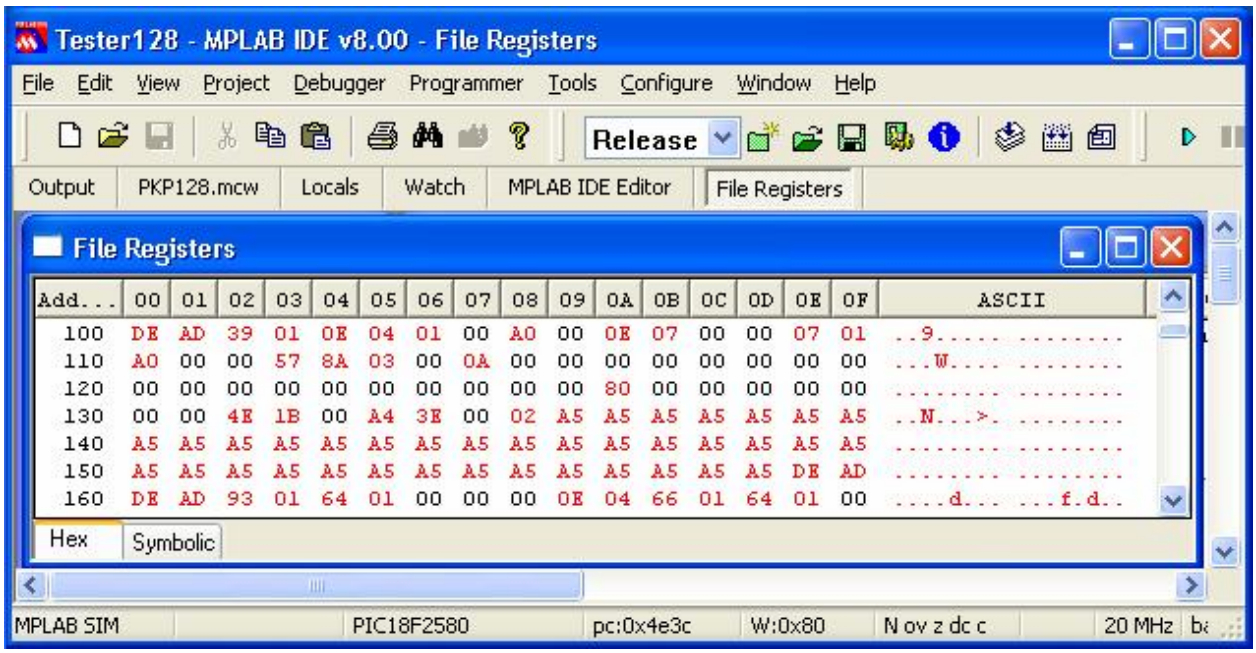


Как вы можете видеть, выбрана Large Data Model. Это дает некоторое увеличение скомпилированного кода, зато позволяет избежать ограничений по размеру больших объектов, например, массивов. Кстати, стеки представляют собой массивы байтов, которые могут пересекать границы банков памяти. Благодаря этому программисту уже не надо отвлекать свое внимание на отслеживание подобных моментов.

Multi-bank Stack Model реализует полноценное управление стеком задач, особенно при пересечении границы банков памяти. Кроме того, теперь сняты ограничения на размер стека и его расположение, которые существовали до версии v3.00.

Вы можете выбрать удобную для себя оптимизацию. Но это уже другая вкладка.

Теперь о возможности отследить использование стека задачи с помощью примитивного индикатора. Стек задачи ограничивается «часовыми» (sentinel) 0xDEAD снизу и сверху. Именно их проверяет ядро ОС при переключении задач на целостность стека. Если Ядро не находит «часовых» на своих постах-адресах, то впору поднимать тревогу – не разрешать запускать задачу.



Суть примененного способа проста: при запуске ОС, когда происходит инициализация стеков задач, в отвечающей непосредственно за стек область записывается код 0xA5. По мере использования стека задача заменяет это предопределенное значение другим. Совпадения достаточно редки, и вы сразу увидите в окне File Registers этот «хвост» из знаков 0xA5.

На скриншоте представлен стек одной из задач, выполнявшейся некоторое время в симуляторе. Стек занимает адреса с 0x100 по 0x15F. «Часовые» занимают адреса 0x100, 0x101 и 0x15E, 0x15F. Адреса 0x102, 0x103 содержат адрес старта задачи, а все, что выше – стек задачи. Здесь содержатся локальные переменные задачи и вызываемых ею функций. Здесь же хранится контекст задачи, когда она не выполняется. Немного терпения, и вы можете выделить основные области стека. Но не будем отвлекаться, и определим, сколько свободного места имеется в стеке на момент остановки приложения.

Эта задача на самом деле очень простая. Находим «часового» 0xDEAD в старших адресах стека (0x15E, 0x15F) и отсчитываем от него в сторону уменьшения адресов количество знаков 0xA5. Таковых здесь насчитывается 37. Таким образом, стек имеет запас в 37 байтов.

Будьте внимательны, чтобы не ошибиться в «часовых» - они у разных задач такие одинаковые!

Теперь, когда у вас имеется такое простое, но эффективное средство, вы можете легко определить причину странного поведения вашего микроконтроллера. Посмотрите на стеки задач и определите, не произошло ли переполнение. Если же вам не хватает памяти данных, то этот индикатор позволит вам определить, какой задаче вы дали «лишний кусок».

Ну, а теперь за дело!

10. Ссылки в интернете

Узнать больше о стандарте OSEK и использовании PIC18 вместе с PICos18 можно из некоторых ссылок в Интернете.

www.osek-vdx.org : стандарты OSEK/VDX™ (PDF файлы).

www.picos18.com : последняя версия и новости о PICos18.

www.pragmatec.net : если вам интересна продукция с использованием PICos18.

www.microchip.com : официальный вебсайт MICROCHIP позволит вам получить:

- последние описания PIC18 (даташиты)
- последнюю версию MPLAB®
- последнюю версию C18

www.fsf.org : лицензия GPL.

Перевод с английского С.Н.Кушнир www.picos18@tut.by