1. Введение.

1.1 Почему многозадачность?

Задача – это обычно последовательность операций, которые жестко связаны между собой и поэтому должны выполняться в строгой последовательности. Если задачи выполняются последовательно друг за другом, без прерываний, тогда простая программа удовлетворяет этим условиям, используя циклы, условные переходы и вызовы подпрограмм.

Как бы то ни было, часто возможно выделить задачи и подзадачи способные функционировать независимо друг от друга. Независимо – это означает невозможность определить состояние какой-либо задачи исходя из состояния других задач приложения. Значение слово состояние в данном контексте необязательно относится к состоянию автоматов.

Примером задачи, которая функционирует независимо от других, может служить чтение входящего потока данных UART. Если передача ведется по нескольким каналам UART, то обычно каждый канал функционирует независимо от других. Наиболее просто эта задача может быть решена, если в каждый момент времени будет вести обработка данных только одного канала. В подобном приложении наиболее просто организовать многозадачность, т.к. возможно работать только с одной задачей в каждый момент времени, не обращая внимания на состояния других.

Реализация последовательной программы, поддерживающей параллельные и независимые задачи довольно трудна, и обычно требует разбиения каждой задачи на маленькие куски, которые выполняются в период одного обращения к задаче, и использования переменных состояний для определения состояний каждой задачи в момент обращения к ней. Подобные приложения часто становятся трудночитаемыми и запутанными. Кроме того, возможно достичь только ограниченного уровня параллельности.

Таким образом, система многозадачной компиляции должна:

1) Позволять код для каждой задачи писать независимо от остального приложения.
2) Обеспечить механизм переключений между задачами.
3) Обеспечить необходимые библиотеки для упрощенного написания многозадачных приложений.

1.2 Концепция LEANSLICE.

Концепция LEANSLICE немного отличается от традиционных подходов. Задачей при ее создании было удовлетворить нескольким важным требованиям:

1) Позволить реализацию многозадачных приложений на небольших 8-ми разрядных микроконтроллерах, таких как серия 1886 ЗАО “ПКК Миландр” и подобных, имеющих фиксированный аппаратный стек, доступ к которому не может быть получен никакими другими средствами, кроме команд CALL и RETURN.

2) Использование автоматов на низких уровнях для обеспечения маленьких времен отклика.

3) Позволить каждой задачи выглядеть как процедуре с независимым от остального приложения кодом.

Вышеперечисленные требования были полностью удовлетворены, и в итоге получилась концепция одновременно мощная и простая в использовании. Кроме того, программный код получается достаточно компактным.

1.3 Разделение по коду или разделение по времени.

Механизм LEANSLICE работает по принципу разделения по коду. Основная идея состоит в разбиении кода задачи на некоторые небольшие части, которые исполняются непрерывно. Т.е. каждая задача системы функционирует по сути как конечный автомат. При описании конечных автоматов обычными средствами, часто получается трудночитаемый, довольно нетрадиционный код, так как программисту приходится включать в код информацию о состояниях и переключениях. Компилятор LEANSLICE позволяет скрывать информацию о состояниях и описывать код программы автомата почти так же, как обычный код на языке С. Общим для автоматов и LEANCLICE является возможность автоматической защиты критических частей программы.

Автоматы имеют множество замечательных особенностей. Их можно найти в самых разнообразных приложениях, от небольших функциональных блоков в схемах до огромных многозадачных приложений высокого уровня. UML (Universal Modeling Language- Универсальный Язык Моделирования) позволяет разрабатывать большие системы используя их графическое представление.

Термин объединенная многозадачность (cooperative multitasking) также используется для многозадачных систем, которые переключают задачи только на предопределенные позицию в коде приложения. Компилятор LEANSLICE продвинулся на шаг вперед, используя несколько позиций в пределах одной задачи для переключения, что позволяет разбивать код на довольно маленькие части. Аналогичные системы могут переключать задачи только на одну и ту же позицию, либо только в момент достижения ими состояния “IDLE”.

Другой популярный подход к проектированию многозадачных приложений – это разделение по времени. Ядро приложения прерывает каждый процесс через некоторое время и отдает управление другому процессу. Таким образом, каждая задача получает доступ к вычислительным ресурсам на определенные интервалы времени. В результате, каждый процесс может быть остановлен в любой момент времени. Для уменьшения времени отклика часто используется приоритетное планирование. Механизм разделение по времени наиболее часто используется в мощных системах, способных за короткое время переключится на другую задачу и при этом сохранить контекст текущей. Основным преимуществом такого метода является то, что каждая задача выглядит как обычная последовательная программа, которая выполняется параллельно с другими, и позволяет переключение процессов в любом месте (in deep call structure). Таким образом, возможно применять повторно используемые функции (reentrant functions), которые могут быть вызваны из нескольких процессов без каких либо ограничений.

Повторно используемый код

Повторно используемый код – это код, который могут использовать несколько задач, не мешая друг другу. Это означает, что каждый процесс должен использовать свой собственный контекст и не должен использовать глобальные или статические переменные. Обычно это реализуется как стек, который позволяет динамически сохранять параметры и локальные переменные. Повторно используемые функции – это важная особенность многозадачных систем с разделением по времени. Эффективно поддерживать функцию повторно используемого кода могут только процессоры со встроенным стеком относительной адресации.

LEANSLICE не поддерживает повторно используемый код, но в то же время позволяет создавать несколько задач одного и того же типа, использующих раздельный код для отделения переменных друг о друга. Это увеличивает быстродействие и в то же время сильно увеличивает объем кода.

Приоритетное планирование.

Системы с разделение по времени часто используют приоритетное планирование для улучшения времени отклика. Идея приоритетного планирования состоит в передачи управления нужному процессу сразу при получении входных сигналов, сигналов будильников и других событий.

Компилятор LEANSLICE позволяет уменьшить время отклика системы, используя быстрое переключение контекстов и специальное планирование процессов. При создании приложений возможно довольно точно просчитать время отклика каждой задачи.

Многозадачность.

Компилятор LEANSLICE позволяет разделять приложения на независимые части (задачи), так же, как и другие многозадачные системы. Важное отличие состоит в том, что каждая такая часть может быть разделена на еще меньшие куски. В итоге, размер каждой исполняемой части задачи определяется на этапе создания приложения. Каждая часть пронумерована, и номер требуемой к исполнению части хранится в переменной состояния. Нумерация может производиться как автоматически, так и разработчиком приложения.

1.4 Исполнение и согласование по времени.

Архитектура RISC подразумевает, что большинство команд требуют определенное количество тактов для выполнения инструкции. Это время называется машинным циклом. Этот термин подразумевает, что некоторые инструкции требуют более одного цикла для выполнения. Использование такой единицы измерения, как машинный цикл имеет важное преимущество: измерение времени выполнения в процессорных циклах независимо от используемого генератора.

То, как быстро должно быть обработано событие, говорит о том, насколько критичным по времени оно является. Например, возможно такое разделение:

a) 30-300 процессорных циклов: Очень критичное.

b) 300-3000 процессорных циклов: Критичное

c) >3000 процессорных циклов: Некритичное.

Для уменьшения времени отклика до 30 процессорных циклов необходимо очень аккуратное программирование. Алгоритм LEANSLICE позволяет создавать приложения с подобными задержками. Для этого потребуется несколько процессов, специальное планирование и расчет времени выполнения каждой критичной части кода.

Алгоритм LEANSLICE позволяет предопределять время выполнения. Для выполнения анализа по времени сгенерированного ассемблерного кода требуется подсчет инструкций.

Системе с разделение по времени потребуется приоритетное планирование для того, чтобы обеспечить времена отклика системы в пределах 300 процессорных циклов. Причиной такой медлительности является необходимость переключения контекста и сохранения регистров. Так же, если более одного процесса будут иметь высокий приоритет, то время отклика может превысить необходимую границу.

Алгоритм LEANSLICE подразумевает выполнение одной задачи на протяжении 15-30 циклов. Это означает, что в случае использования кругового планирования (ROUND-

ROBIN) для 10 процессов, время отклика каждого из них легко уложится в 300 машинных циклов.

Исполнение программ. Алгоритму LEANSLICE требуется инструкция GOTO для функционирования. В большинстве микроконтроллеров подобная команда реализована. Для планирования процессов и исполнения следующего состояния используется уровень простого перехода.

LEANSLICE может работать с микроконтроллерами Ubicom и микроконтроллерами серии 1886ВЕ и совместимыми ними (PIC 12/14/16/17/18).

1.5 Особенности LEANSLICE.

Концепция LEANSLICE на данный момент имеет следующие особенности:

- Поддержка многозадачности интегрирована в компилятор для эффективной реализации приложения.
- Реализация полностью статическая.
- Не используется ядро многозадачности.
- Не используются прерывания, и таким образом свободны для использования из в приложении.
- Небольшой добавочный код для переключения процессов.
- Быстрое переключение процессов:10-15 инструкций.
- Использование циклического планирования по умолчанию.
- Возможность использования специального планирования. С помощью него можно увеличить быстродействие или определить приоритеты процессов.
- В каждой задаче могут быть определены состояния, как точки переключения, при этом состояние процесса можно определить извне для отладки или устранения ошибок.
- Переключение процессов происходит только в определенном месте кода, что позволяет защитить критические части кода приложения.
- Функция пользовательских задержек использует множество программных таймеров для определения задержек.
- Процессы могут стартовать, останавливаться, подвешиваться, завершаться и перезапускаться независимо друг от друга.
- Включены библиотеки для таймеров, событий, семафоров, двоичных семафоров и почтовых ящиков.

Быстрое переключение контекста. Переключение контекста означает переключение выполнения с одно процесса на другой. В концепции LEANSLICE это означает сохранение состояния, которое будет выполнятся следующим при возврате к

процессу. Переключение процесса запускается функцией waitState(). Это не настоящая функция, а некоторое подобие директивы для компилятора, которую он переводит в ряд необходимых инструкций. Код между каждым waitState() выполняется без прерываний. Каждый waitState() добавляет 0-2 инструкции к ассемблерному коду приложения. Переключение контекста обычно происходит за 10-15 инструкций.

Существующие ограничения. LEANSLICE позволяет реализовать быстрое и компактное многозадачное приложение. В то же время существуют некоторые ограничения для его использования:

1) LEANSLICE наилучшим образом подходит если задачи небольшие.
2) Невозможно переключать процессы из функций, порожденных процессами.
3) Невозможно использовать один и тот же код для нескольких процессов.
4) Локальные переменные должны определится как статические, если необходимо их сохранить при переключении процессов.
5) Установка приоритетов процессов невозможна, но может контролироваться специальным планированием выполнения задач (вместо кругового, используемого по умолчанию).

1.6 Установка и системные требования.

LEANSLICE версии 1.1 поддерживается компиляторами:

1) CC5X EXTENDED EDITION, version 3/1G или позднее.
2) CC5X EXTENDED EDITION, version 3/1G или позднее.
3) CC5X EXTENDED EDITION, version 3/1G или позднее.
4) CC5X EXTENDED EDITION, version 3/1G или позднее.

LEANSLICE библиотечные/заголовочные файлы требуется скопировать в соответствующую директорию.

Список поставляемых файлов.

MVENTIL.C *: Пример использования многозадачности. Контроль вентиляции.*

DELAY.H **:** *Библиотека задержек и таймеров.*

EVENT.H *: Библиотека событий.*

BINSEM.H *: Библиотека двоичных семафоров.*

SEMAPHOR.H *: Библиотека семафоров.*

MSG.H *: Библиотека почтовых ящиков.*

1.7 Пример реализации проекта с тремя процессами: Контроль вентиляции.

Приведенный ниже пример показывает как просто создать многозадачное приложение, содержащее три процесса, которые могут работать совершенно независимо друг от друга или взаимодействовать. Приложение требует только 151 кодового слова при реализации на PIC16C84, и 15 байт ОЗУ. Это довольно мало для полностью многозадачной программы. Приложение использует три программных счетчика по 16 бит каждый, на которых строятся задержки от 10 миллисекунд до 10,9 минут.

Первая задача <генерация импульсов> формирует 10 сигналов с частотой 1 Гц и скважностью ½. Процесс останавливается после завершения, но может быть перезапущен снова. Назначение этой задачи продемонстрировать как одна задача может быть вызвана другой и далее функционировать независимо.

Процесс <контроль вентиляции> включает и выключает вентиляцию. Вентиляция включается сразу же, как только температура поднимается выше предела определенного выводом <высокая температура>. После того как температура опускается ниже предела, вентиляция выключается после 100 гистерезиса.

Третий процесс <Индикация>, функционирует полностью независимо от остальных.

Он генерирует сигналы частотой 1Гц, со скважностью зависимой от состояния процесса <генерация импульсов>.

```
// VENTILATION CONTROL
// ventilationControl: control ventilation on/off, delayed off
// generatePCM: send PCM pulses depending on ventilation on/off state
// generatePulses: send 100 pulses when ventilation goes on or off
#pragma chip PIC16C84
#pragma taskOptions 2
enum { TimerVC, TimerPulse, TimerPCM }; // Timer identifiers
#define FreqOsc 4000000 // Oscillator frequenzy, 4 MHz
#define FreqTimer 100 // Timer frequency, 100 Hz
#define TimerType uns16 // uns8, uns16 or uns24
#define NoOfTimers 3 // 1 .. 8
#define DELAY_TYPE 1
#include "delay.h"
enum {Off, On};
bit highTemperature @ PORTA.0;
bit ventilation @ PORTA.1;
bit outPCM @ PORTA.2;
```

```
bit outPulse @ PORTA.3;
#define INIT_PORTA 0b00000000
#define INIT_TRISA 0b00000001

Task generatePulses( void)
{
        static char pulses;
        pulses = 100;
        do {
                outPulse = 1;
                delay( TimerPulse, 50); // 0.5 seconds
                outPCM = 0;
                delay( TimerPulse, 50); // 0.5 seconds
            } while (--pulses > 0);
}

Task ventilationControl( void)
{
        ventilation = Off;
        while (!highTemperature)  waitState();
        ventilation = On;
        startTask( generatePulses);
        VENTILATION_ON:
        while (highTemperature) waitState();
        startTimer(TimerVC, 10000); // 100 seconds
        while (!timeout(TimerVC))
        {
                if (highTemperature)
                goto VENTILATION_ON;
                waitState();
        }
        startTask( generatePulses);
        restartTask(); // restart (and then switch ventilation off)
}
```

```
Task generatePCM( void)
{
        /* NOTE: pulse width may not be correct when
        ventilation change state from 0 to 1 or from 1 to
        0. Ensuring that PCM width is always correct will
        require more code.
        */
        while (1) {
                outPCM = ventilation;
                delay( TimerPCM, 10); // 0.1 seconds
                outPCM = !ventilation;
                delay( TimerPCM, 90); // 0.9 seconds
            }
}

void main( void)
{
        PORTA = INIT_PORTA;
        TRISA = INIT_TRISA;
        startTask( ventilationControl);
        startTask( generatePCM);
        clearTask( generatePulses);
        initTimers();
        while (1) {
                timerTick();
                taskSlicer();
            }
}
```

1.8 Что делать дальше.

Рекомендуется выполнить начальную стадию приложения, для того чтобы определить, годится оно для многозадачной реализации или нет. На этот вопрос не так просто ответить. В любом случае, возможно написать обычную программу используя петлю в основной функции, которая и будет определять, что делать дальше. Прерывания при этом, позволят немедленно обрабатывать возникающие события.

Приложения, использование LEANSLICE для которых будет наиболее эффективно, должны обладать следующими признаками.

- Работа приложения обладает выраженными независимыми направлениями.
- В приложении присутствует много задач, которые не могут быть расположены последовательно.
- Задачи имеют ограничения по времени.
- Задачи похожи друг на друга, но их требуется выполнять независимо.

При этом существуют и противопоказания:

- Сильная зависимость между процессами в приложении
- Большинство операций могут быть выполнены в определенной последовательности.

Преимущества LEANSLICE.

- Код легче читается.
- Пригоден для всех микроконтроллеров серии 1886 а также всех PIC.
- Задачи пишутся независимо друг от друга.
- Код приложения занимает гораздо меньше места.
- Включает библиотеки передачи сообщений, событий, семафоров, двоичных семафоров и задержек.
- Сочетает эффективность конечных автоматов с легким написанием кода.

2. Принцип реализации многозадачности с использованием LEANSLICE.

Приложение с реализацией многозадачности в режиме LEANSLICE представляет собой набор задач, описанных независимо друг от друга и основную функцию, описывающую порядок переключения процессов и некоторые общие для всего приложения действия. Каждая задача представляет собой конечный автомат, после каждого выполнения кода текущего состояния которого, приложение переключается на следующую задачу. При этом, информация о состоянии задач и необходимых переключениях вставляется в код программы автоматически при компиляции.

Рассмотрим для примера диаграмму, приведенную на рисунке 1. Приложение содержит два параллельных процесса. Кружками на рисунке обозначены состояния процессов (некоторые законченные этапы работы) а стрелочками показаны переключения между ними. Допустим, выполнение приложения начинается с выполнения этапа A первого процесса. После выполнения этой части кода, управления будет передано этапу B второго процесса. После однократного выполнения кода любого этапа какого либо

процесса, программа обязательно возвращается в основную функцию, и после выполнения очередных этапов всех остальных процессов, либо повторно выполняет код текущего этапа, либо переходит к следующему этапу данного процесса. В примере, иллюстрированном диаграммой на рисунке 1, после завершения этапа В осуществляется переход к этапу С процесса 1 (переход 2). Это означает, что либо условие выхода из этапа А было выполнено, либо был осуществлен безусловный переход на следующий этап. После выполнения кода этапа С, управление передается этапу D процесса 2 (переход 3), и после его выполнения снова возвращается к этапу С (переход 4). Т.е. условие выхода из этапа С выполнено не было. Далее, после каждой передачи управления первому процессу, будет проверятся условия выхода из состояния С, и в случае его удовлетворения будет выполнятся код этапа А, а в противном случае повторно код этапа С. После однократного выполнения кода любого этапа процесса управление передается основной функции, и далее всем другим активным процессам системы (в нашем примере это только процесс 2).

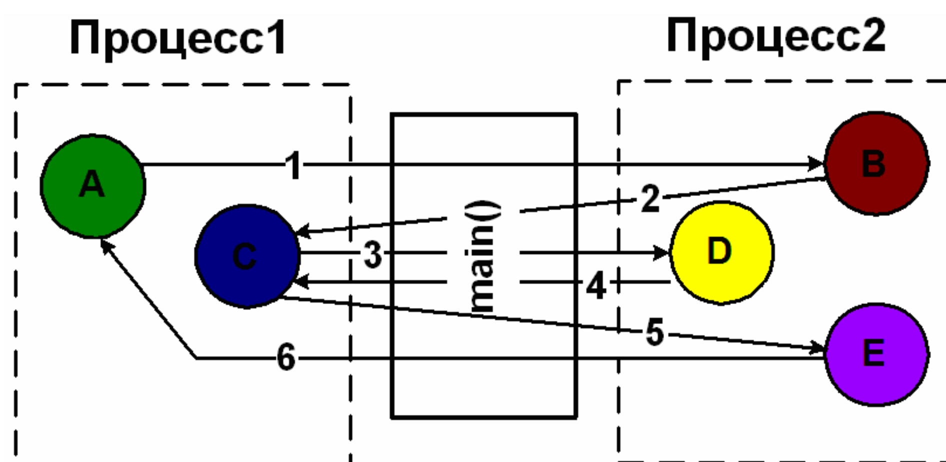

Рисунок 1. Диаграмма переключения процессов.

Разработчику для описания состояний конечного автомата процесса требуется лишь вставить специальные функции waitState() для завершения каждого этапа в коде задачи, а для переключения процессов одну или несколько функций taskSlicer() в основной функции. Заметим, что ни waitState() ни taskSlicer() ни другие подобные функции не являются функциями в обычном понимании, это своеобразные директивы компилятора, заменяемые при компиляции требуемым кодом. Здесь и далее для удобства будем называть их функциями.

Функция taskSlicer() может явно указывать на следующий процесс, в этом случае необходимо передать в качестве параметра имя следующей задачи. Код основной функции может выглядеть, например, следующим образом:

void main();

startTask(<Процесс 1>);

```
 startTask(<Процесс 2>);
<инициализация переменных>
taskSlicer(<Процесс 1>);
While(1)
{
if (<условие>)
taskSlicer(< Процесс 2>);
else  taskSlicer(<Процесс 1>);
}
```

Если функции taskSlicer не указывать явно номер процесса, то компилятор автоматически построит код таким образом, что процессы будут переключаться по кругу. В этом случае основная функция должна иметь вид:

```
void main();
 startTask(<Процесс 1>);
 startTask(<Процесс 2>);
<инициализация переменных>
While(1)
{
taskSlicer();
}
```

В обоих примерах, программа, выполнив инициализацию всех задач и переменных, входит в основного этапа первой задачи, осуществляется возврат в основную программу и далее выполнение следующий задачи в соответствии с кодом основной функции.

Код текущего состояния задачи считается законченным при достижении функций waitState() или changeState(). Разница между ними состоит в том, что waitState() обычно включается в тело цикла условия перехода к следующему состоянию, а changeState() жестко переключает состояние автомата на следующее (указанное явно или неявно). Например, в приведенном ниже примере, операторы состояния 1 будут выполняться до достижения <условия 1>, при этом после каждого цикла выполнения, приложение будет вываливаться из задачи в основную функцию. После удовлетворения <условия 1>, выполнив <последовательность действий 2>, задача переключит состояния на указанное в <номер состояния перехода>, но перед этим осуществится выход в основную функцию и выполнение остальных задач приложения.

```
task <имя задачи>(void)
{while(<условие 1>)
```

```
{ <последовательность действий состояния1>;
waitState();
}
<последовательность действий состояния2>;
changeState(<номер состояния перехода>)}
……
}
```

3. Подробное описание LEANSLICE.

3.1 Выбор типа задач.

Перед началом написания программы необходимо выбрать один из трех доступных типов задач:

1). Задачи первого типа всегда активны. Основное их преимущество это наиболее компактный код.

```
#pragma taskOptions 1          // enable type 1 tasks
```

2).Задачи второго типа могут быть остановлены (состояние 0xFF). Остановка задачи требует выполнения еще 10-15 инструкций. Активные задачи работают так же как задачи первого типа.

```
#pragma taskOptions 2          // enable type 2 tasks
```

3).Задачи третьего типа могут быть подвешены в любом состоянии. Основное преимущество таких задач состоит в том, что для обработки задачи в подвешенном состоянии требуется всего около 3 машинных циклов. Недостаток – это более медленное выполнение активных задач(на 2 инструкции больше для каждой задачи).

```
#pragma taskOptions 3          // enable type 3 tasks
```

Какой тип задач выбрать зависит от приложения. Если все задачи постоянно активны, тогда идеально подойдут задачи первого типа. Если для некоторых задач допустима остановка, то более подходящими окажутся задачи второго или третьего типа. При этом, если большинство задач активно, то более подходит второй тип. Третий тип более функционален, т.к. процессы могут быть подвешены в любом состоянии и далее в нужный момент времени продолжить работу. Заметим, что первый тип задач больше всего подходит в случае использования специального планирования, т.к. выполнение задач полностью контролируется приложением.

Все задачи приложения должны обладать одним и тем же типом. Выбор нескольких типов задач недопустим.

3.2 Другие возможные опции LEANSLICE.

В LEANSLICE существует возможность автоматической реализации проверки процессов на ошибочные состояния. Существует несколько возможных режимов тестирования:

1. #pragma taskOptions 10 // test for invalid states

В случае ошибочного состояния задачи останавливаются (типы 2 и 3), а задачи первого типа перезапускаются. Требует выполнения 3-4 дополнительных инструкции для каждой задачи.

2. #pragma taskOptions 9 // test for invalid states

Аналогичен предыдущему, за исключением того, что задачи третьего типа не останавливаются, а перезапускаются.

3. #pragma taskOptions 8 // fast test of invalid states

Быстрое тестирование(1 машинный цикл). Некорректные состояния преобразуются в корректные. Таким образом исключена возможность перехода на случайный адрес. Данный режим подходит только для процессов с количеством состояний меньше 31 (для задач типа 2 - меньше 30).

4. #pragma taskOptions 7 // fast test of invalid states

Аналогичен предыдущему, за исключением того, что задачи второго типа могут иметь от 16 до 26 или больше 32 состояний.

Также возможными опциями для LEANSLICE являются:

#pragma taskOptions 11 // NOT computed goto for state selection

#pragma taskOptions 3, 10, 11 // multiple options

3.3 Разбивка кода задач на этапы.

Для разбиения кода задачи на фрагменты(состояния), т.е. для того чтобы указать место окончания фрагмента кода текущей задачи и перехода к другим необходимо вставить в С-код программы одну из следующих функций:

waitState(); - Обычно используется в условных циклах. После исполнения этой функции и возврата в задачу выполнения продолжается со следующей за waitState() инструкции. Данной функции можно указать номер состояния либо явно (в качестве параметра), либо состояние будет присвоено автоматически.

changeState(); - Аналогичен GOTO с той лишь разницей что перед переходом происходит выход из задачи и обработка всех других активных процессов приложения. После возврата в задачу выполнения осуществляется с состояния указанного в качестве параметра функции (changeState(5);). Если в качестве параметра ничего не указано то со следующего состояния.

waitUntilResume(); - Функция подвешивает процесс. Выполнения процесса можно возобновить если в нужном месте активной части приложения разместить функцию resumeTask(<имя процесса>); Данная функция пригодна только для процессов третьего типа.

Возможные состояния процесса:

№ состояния	Назначение
0	**Начальное состояние**
1..N	**Рабочие состояния**
0xFF	**Задача типа 2 завершена.**
0x80	**Задача типа 3 завершена.**
0x81..0xFF	**Задача типа 3 подвешена.**

3.4 Управление процессами

3.4.1 Инициализация процессов. В начале основной функции каждой программы необходимо инициализировать все процессы. По сути, инициализация – это перевод задачи в начальное состояние. Синтаксис этой функции имеет вид startTask(<имя процесса>);

Эта операция может быть выполнена и во время выполнения программы. Для этого используется функция restartTask(<имя процесса>);

3.4.2 Планирование процессов. Планирование процессов возможно при помощи функции taskSlicer(); в основной функции приложения. При этом имеется возможность явно указывать номер следующего выполняющегося процесса(taskSlicer(<номер процесса>);).

В противном случае (taskSlicer();) все процессы приложения будут выполнятся один за другим по кругу.

3.4.3 Остановка, подвешивание и запуск процессов. Процессы третьего типа возможно подвешивать и возобновлять их выполнение.

Процесс может быть подвешен из остальной части приложения:

```
                                    // suspend a task from outside
suspendTask( task1);                // BSF _TaskS1,7
```

Либо подвешивать сам себя:

```
                                    // a task can suspend itself
waitUntilResume();                  // RETLW 0x82
waitUntilResume(5);                 // RETLW 0x85
```

Возобновление функционирования процесса возможно только извне:

```
                                      // resume a suspended task
resumeTask( task1);                   // BCF _TaskS1,7
```

3.4.4 Завершение процессов. В LEANSLICE существует возможность завершать процессы для всех типов, кроме первого.

clearTask(task1); // kill task

В этом случае выполнение процесса заканчивается. Возобновить выполнения можно из основной части программы при помощи resumeTask(); или startTask(); Процесс будет выполняться с нулевого состояния.

3.4.5 Получение состояния процесса.

В LEANSLICE реализована возможность получения состояния процесса из остальной части программы.

State = getState(<имя процесса>); получение значения состояния процесса.

StA = getStateAddress(<имя процесса>); получение значения адреса состояния процесса.

StA = getStateAddress(); получение значения адреса состояния текущего процесса из самого процесса.

3.5 Работа с переменными. Никаких параметров локальных переменных не должно быть определено, когда выполняются функции waitState() или waitUntilResume(). Т.е. когда осуществляется выход из процедуры. Если компилятор обнаружит активные локальные переменные возникнет ошибка. Возможным решением является использовать определение static для всех локальных переменных или использовать скобки, которые закрываются до выполнения waitState(). При этом, если использовать скобки, значение переменных не будет доступно остальной части системы.

```
static char s;                        // static local variables are allowed
{ /* narrow local variable scope */
char i = 0;
...
} /* OK: variable 'i' is undefined at next waitState() */
waitState();
```

3.6 Работа с банками памяти. Приведенный код иллюстрирует самый простой способ для расположения переменной состояния процесса в банке, отличном от банка использованного для локальных переменных и параметров.

```
#pragma rambank 1                    // locate state variables in RAM bank 1
Task task1( void);
Task task2( void);
Task task3( void);
#pragma rambank –
```

4. Описание встроенных библиотек.

4.1 Библиотека таймеров.

Библиотека таймеров (delay.h) содержит 8,16 и 24 разрядные программные таймеры, которые могут быть использованы в задачах для получения необходимых задержек. Инкремент таймеров должен производится гораздо реже чем частота работы контроллера. Это означает, что программные счетчики работают на частотах от 100 до 1000 Гц, в то время как частота работы микроконтроллера составляет 4МГц и выше. Около 1000 инструкций на один инкремент программных таймеров нормальное явление. Библиотеку delay.h можно модифицировать, если требуются недоступные в исходном ее варианте возможности. В библиотеке реализована возможность назначить задержку каждой задаче. Далее приведен пример кода с использованием библиотечных таймеров.

Определение таймеров:

Инициализация и инкремент:

Использование таймеров в качестве задержек:

Заметим, что delay(1) не может быть точной, поскольку таймер опирается на простой аппаратный таймер, и невозможно предсказать, когда произойдет следующее его переключение. Длительность delay(1) будет в интервале от 0.0 до 1.0 периода. Например, если пройдет половина периода до следующего переключения аппаратного таймера, тогда период будет составлять 0.5. Это означает, что несколько задержек будут более предсказуемы, чем одна.

Быстродействие.

DELAY_TYPE 1: Программные счетчики реализуются используя восьмибитный аппаратный счетчик TMR0 и предделители. Возможно использование до 8 таймеров.

initTimers() : 6 машинных циклов. Инициализирует предделитель. Устанавливает тайм-аут как TRUE для всех программных счетчиков.

timerTick() : Более 23 машинных циклов. Размер увеличивается в зависимости от размера счетчика (8, 16 or 24 bit) и количества таймеров. 3 шестнадцатибитных счетчика требуют 46 инструкций. Минимальное требование функции – это 12 инструкций осуществляется когда заканчивается следующий аппаратный период. Увеличивается,

если программные счетчики необходимо декрементировать (около 49 машинных циклов когда присутствуют 3 активных 16 разрядных таймера).

delay(T,C) : 6-7 инструкций когда используются 16 разрядные счетчики.

startTimer(T,C) : 4-5 инструкций когда используются 16 разрядные счетчики.

if (timeout(T)) : 1-2 инструкции.

4.2 События.

События могут использоваться для распространения информации между задачами. Реализация событий проста и каждое событие занимает один бит. Задача ждущая событие, будет использовать активное событие. Если используется круговое планирование достаточно установить значение события как TRUE и переключить задачу. Это означает, что задачи ждущие событие получат его. Событие может быть стерто в следующий раз во время исполнения этой задачи. Так же возможны другие варианты использования механизма событий.

Определение событий:

```
enum {EventA, EventB};          // Event identifiers
#define NoOfEvents 2
#define EVENT_TYPE 1
#include "event.h"
```

Инициализация:

```
initEvents();                   // initialization in main()
```

Использование событий:

```
setEvent(EventA);               // set event to TRUE
waitState();                    // enable waiting tasks to detect the event
                                // the event can be set to false immediate or delayed
clearEvent(EventA);             // set event to FALSE
waitUntilEvent(EventA);         // task waiting for event, active waiting
```

Быстродействие.

EVENT_TYPE 1: Представление событий одиночными битами. Возможно использовать до 32 событий.

initEvents() : 1 инструкция (1-8 событий), .., 4 инструкции (25-32 событий)

setEvent(E) : 1 инструкция

clearEvent(E) : 1 инструкция

waitUntilEvent(E) : 2-3 инструкции

4.3 Семафоры.

Семафоры часто используются для защиты общих данных, областей памяти или ресурсов приложений. Значение семафоров могут быть от 0 до 255 (счетные семафоры). Значение семафора обычно показывает возможность использования общих ресурсов. Конкретная интерпретация значения зависит от приложения. Заметим, что двоичные семафоры обычно удовлетворяют простые потребности для защиты данных. Как бы то ни было, если существует много процессов, которым позволено считывать и записывать общие данные одновременно, то счетные семафоры могут также необходимо рассматривать в качестве оптимального решения.

Определение семафоров в приложении:

```
enum {Semaphore1, SM2};              // Semaphore identifiers
#define NoOfSemaphores 2
#define SEM_TYPE 1
#include "semaphore.h"
```

Инициализация:

```
initSemaphore(SM2,3);                // initialize semaphore (in main())
```

Использование семафоров:

```
getSemaphore(SM2,2);                 // (wait) and get semaphore
                                     //.. process protected region/data/resource
freeSemaphore(SM2,2);                // free semaphore (on exit)
```

Типы семафоров:

- Тип1: Блокирующие. Высший запрос блокирует другие запросы. Пример: значение семафора 5. Вплоть до пяти процессов пытаются обратиться к критической области памяти. Одному из них требуется обновить общие данные и он запрашивает все 5 единиц семафора. Остальные процессы вынуждены ожидать пока записывающий не освободит все 5 единиц семафора. Блокирующий подход представляет собой двух уровневую очередь.
- Тип2:Неблокирующие. Свободные семафоры всегда доступны. Процессы, запрашивающие несколько единиц никогда не блокируют процессы запрашивающие остальные доступные единицы. Таким образом, процессы запрашивающие несколько единиц могут довольно долго ожидать.
- Тип3:Неблокирующие, такой же как тип 1. Медленнее функционируют, но так же имеют меньший код, когда идентификаторы семафоров динамически

располагаются в приложении (т.е. идентификаторы семафоров запоминаются в переменной, вместо того чтобы быть определенными в фиксированной константе.)

Быстродействие:

Performance

SEM_TYPE 1: Blocking

initSemaphore(S,V) : 3 instructions for each semaphore

getSemaphore(S,V) : 3 instructions for each semaphore

freeSemaphore(S,V) : 2 instructions for each semaphore

Fixed instructions : 13 (1 semaphore), 20 (more than one semaphore)

SEM_TYPE 2: Non-blocking

initSemaphore(S,V) : 2 instructions for each semaphore

getSemaphore(S,V) : 6 instructions for each semaphore

freeSemaphore(S,V) : 2 instructions for each semaphore

Fixed instructions : 0

SEM_TYPE 3: Non-blocking (same as type 2 when 1 semaphore)

initSemaphore(S,V) : 2 instructions for each semaphore

getSemaphore(S,V) : 6 instructions for each semaphore

freeSemaphore(S,V) : 2 instructions for each semaphore

Fixed instructions : 10 (when more than one semaphore)

4.4 Двоичные семафоры. Двоичные семафоры служат для защиты критичных общих данных или областей памяти. Они гарантируют, что только один процесс имеет доступ в каждый момент времени.

Определение семафоров в приложении:

```
enum {BinSemaphore1, BS2};
#define NoOfBinSemaphores 2
#define BINSEM_TYPE 1
#include "binsem.h"
```

Инициализация:

```
initBinSemaphores();            // initialize in main()
```

Использование двоичных семафоров:

```
getBinSemaphore(BS2);     // get semaphore (wait until free)
                          //.. process protected region
freeBinSemaphore(BS2);    // release semaphore
```

Доступный тип битовых двоичных семафоров позволяет использовать 8, 16,24 или 32 двоичных семафора.

Быстродействие.

BINSEM_TYPE 1: Двоичное битовое представление семафоров, возмодно до 32 семафоров.

initBinSemaphores() : 1 инструкция (1-8 семафоров), .., 4 инструкции (25-32 семафоров).

getBinSemaphore(B) : 3 инструкции.

freeBinSemaphore(B) : 1 инструкция.

4.5 Почтовые ящики. Почтовые ящики служат для обмена байтовыми сообщениями между процессами. Доступный интерфейс почтовых ящиков позволяет различные степени синхронизации между процессами. В любом случае, отправитель может проверить, удалил ли приемник сообщение из ящика, и получить сообщение подтверждения.

Определение в приложении:

```
enum {MboxTask1, MB2};        // Mailbox identifiers
#define NoOfMailBoxes 2
#define MSG_TYPE 1
#include "msg.h"
```

Инициализация:

```
initMbox(MboxTask1);          // initialize in main()
```

Интерфейс:

```
putMsg(MboxNo, Msg)           // write message, overwrite previous
waitUntilMsgRead(MboxNo)      // sender waits until message is removed
deliverMsg(MboxNo, Msg)       // write message, then wait until removed
waitUntilMsgAck(MboxNo)       // wait for acknowledge (type 1,3,4)
getMsgCopy(MboxNo)            // read message without removing it
```

```
hasMsg(MboxNo)                  // test if mailbox contain a message
noMsg(MboxNo)                   // test if mailbox is empty
clearMsg(MboxNo)                // delete message
ackOnMsg(MboxNo)                // test if acknowledge on message (type 1,3,4)
LEANSLICE B Knudsen Data 26
setAckOnMsg(MboxNo)             // set acknowledge on message (type 1,3,4)
getMsg(MboxNo, mVar)            // read message and remove it (acknowledge)
getNextMsg(MboxNo, mVar)        // wait for message and read (remove) it
waitForMsg(MboxNo)              // wait until message arrives
```

Использование почтовых ящиков для передачи байтовых сообщений:

```
                                // sending task
putMsg(MB2, 9);                 // write message 9 to mailbox MB2
waitUntilMsgRead(MB2);          // wait for receiver to remove the message
                                // receiving task
char mm;
waitForMsg(MB2);                // wait until message arrives
getMsg(MB2, mm);                // store message in mm and clear mailbox
```

Интерпретация сообщений:

- Тип1: Поддерживаются ответные сообщение и коды пустого ящика.

  0-0x7F : messages

  0x80-0xFF: empty mailbox or acknowledge messages

- Тип2: Ответные сообщения не поддерживаются.

  0 : empty mailbox (mail removed)

  1-0xFF: messages

- Тип3: Включают ответные сообщения и специальное представление почтовых ящиков.

  0 : empty mailbox (mail removed)

  1-0x7F: messages

  0x80-0xFF: return/acknowledge messages

- Тип4: Может использоваться только с процессами 3-го типа. Приемник подвешен, до момента прихода сообщения. Преимущества это более быстрое функционирование. Недостаток это больший код. Так же заметим, что при использовании типа 4, адреса задач и почтовых ящиков должны находиться до адреса 0x100 в ОЗУ.

0-0x7F : messages

0x80-0xFF: empty mailbox or acknowledge messages

Быстродействие:

MSG_TYPE 1,2,3,4:

initMbox(B) : **1,2,3,4**: 1 инструкция

putMsg(B) : **1,2,3**: 2 инструкции, **4**: 9 инструкций

waitUntilMsgRead(B) : **1,4**: 3 инструкции, **2**: 4 инструкции, **3**: 6 инструкций

deliverMsg(B, M) : **1,2**: 6 инструкций, **3**: 8 инструкций, **4**: 12 инструкций

waitUntilMsgAck(B) : **1,3,4**: 3 инструкции

getMsgCopy(B) : **1,2,3,4**: 2 инструкции

hasMsg(B) : **1,2,4**: 1-2 инструкции, **3**: 5 инструкций

noMsg(B) : **1,2,4**: 1-2 инструкции, **3**: 4 инструкции

clearMsg(B) : **1,2,3,4**: 1 инструкция

ackOnMsg(B) : **1,3,4**: 1-2 инструкции

setAckOnMsg(B) : **1,3,4**: 1 инструкция

getMsg(B, V) : **1,2,3,4**: 3 инструкции

getNextMsg(B, V) : **1**: 6 инструкций, **2**: 7 инструкции, **3,4**: 9 инструкции.

waitForMsg(B) : **1**: 3 инструкции, **2**: 4 инструкции, **3,4**: 6 инструкции.