LEANSLICE

Lean Multitasking
using Code Slicing

Version 1.1

User's Manual

B Knudsen Data
Trondheim - Norway

This manual covers LEANSLICE version 1.1 and related topics. New versions may contain changes without prior notice.

BUG REPORTS:

The software has been carefully tested and debugged. It is, however, not possible to guarantee a 100 % error free product.

Please report cases of bad generated code and other serious program errors.

1) Investigate and describe the problem. If possible, please provide a complete C example program that demonstrates the problem. A fragment from the generated assembly file is sometimes enough.
2) This service is intended for difficult compiler problems (not application problems).
3) Language: English
4) State the compiler version.
5) Send your report to support@bknd.com, or by fax to (+47) 73 96 51 84.

Document version: **B**

CONTENTS

1 INTRODUCTION

1.1 Why Multitasking

A TASK is typically a sequence of operations that are strongly related to each other and thus needs to be executed or handled in a certain sequence. If there are no interrupts, then a simple program will usually satisfy this condition, using loops, conditions and subroutine calls to operate.

However, it is often possible to identify tasks or subtasks that operate independent of each other. Independent means that it is not possible to determine the state of the other tasks when looking at the state of one of them. The word state is in this context not necessarily related to state machines.

An example of a task that operates independent of other tasks is reading the input data stream from an UART. Such a data stream contains start and stop bits, and data bits that are defined by their position in the stream. Thus, if the operations are reading single bits, then the interpretation of a bit depends on where in the sequence it was detected. If there are several UART channels, then each channel typically operates independent of the others. Clearly, reading UART channels is most easily solved if it possible to deal with ONE channel at a time. This is where multitasking comes in, because it is possible to deal with one task at a time and ignore what is happening elsewhere.

Implementing a sequential program that handle parallel and independent tasks is a challenge, and usually requires dividing each task into smaller pieces that are executed in one chunk, and using state variables to determine the current state of each task. The program structure often becomes difficult to read and maintain. And the parallelism achieved is often limited.

A multitasking system should:
1) Allow code for each task to be written independently of other tasks when this is most natural
2) Provide a task switching mechanism and multitasking library functions

1.2 The LEANSLICE Concept

The LEANSLICE multitasking concept is slightly different from the traditional approaches. The goal have been to meet several challenging demands:

1) Enable multitasking on small 8 bit CPU's such as PICmicro and similar architectures that have a fixed hardware stack that can not be accessed by other means than CALL and RETURN instructions.
2) Use state machines at the low level to get very fast response times.
3) Allow each task to look like a procedure with proper multitasking abstractions.

These design goals have been satisfied, and the resulting LEANSLICE concept is both powerful and simple to use. In addition the code is compact.

1.3 Code Slicing versus Time Slicing

To describe the LEANSLICE multitasking mechanism, the words **code slicing** are appropriate. The fundamental idea is to divide the task code in advance into smaller distinct parts that is executed without interruption. This is basically the way **state machines** operates. However, the code for state machines usually looks quite different because focus is on states, using signals to trigger state transitions. LEANSLICE allows the application source code to look very similar to ordinary C functions, with "state" information hidden. State machines and LEANSLICE shares the ability to protect critical program regions automatically.

In general, state machines have many good properties. They can be found in many places, from small functional blocks in hardware circuits, to being the modelling concept in very large real-time multitasking applications. UML (Universal Modelling Language) allows large systems to be designed using graphical notation and is often supported by automatic code generation tools.

The term **cooperative** multitasking is also used for multitasking systems that only switch to another task at predefined positions in the application code. The LEANSLICE concept takes this a step further, by using many task switching positions for each task, and allows thus each code slice to be very small. Other systems may switch task at a single position, or when a task goes into an idle state.

Another popular multitasking approach is **time slicing**, where a multitasking kernel interrupts each task (process) after some milliseconds to give control to another task. Thus, each task is given CPU processing time at regular intervals. The fundamental issue is that a task can be interrupted anytime. Preemtive scheduling is frequently used to improve response time. Time slicing is a general mechanism often used in large systems, but requires normally a powerful CPU (that can switch stack, save registers and local context). Performing a context switch is unfortunately time consuming. The great advantage is that each task looks like an ordinary sequential program that executes “concurrently” with other tasks, and allows task switching in deep call structures. It is possible to use **reentrant** functions that can be called from several tasks without restrictions.

Reentrant Code

Reentrant code means that several tasks can execute the same code without disturbing each other. This means that each task should not use global or static variables, but instead have its own local context. This is typically a stack that allows parameters and local variables to be stored dynamically. Reentrant functions and libraries are important in time slicing systems. Note that only CPU’s with built-in stack relative addressing can support reentrant code efficiently.

LEANSLICE currently does not support or make use of reentrant code. However, it is possible to create several tasks of the same type **statically**, using **separate** code for each instance to make the context private. This improves performance at the cost of more code.

Preemptive Scheduling

Time slicing systems often use preemptive scheduling to improve response time. The idea is to give control to the right task immediately in response to IO signals, timeouts and events.

The LEANSLICE way of improving response time approach is to use fast context switch and custom scheduling of tasks. It is possible to calculate the response timing delay in advance quite precise.

Lean Multitasking

LEANSLICE allows the application to be divided into independent tasks just like other multitasking concepts. An important difference is that each task must be divided into small code slices. Currently, the slice size is fully decided by the application source code at design time. All code slices are enumerated and the next slice to be executed is stored in a state variable. The enumeration is either done automatically, or by the application designer.

LEANSLICE is a “lean” multitasking concept because of a highly efficient low level representation, combined with static allocation. There are thus **no dynamic runtime lists** and **no runtime kernel**. The task size grows slowly when using multitasking library calls which frequently only need few instructions.

1.4 Performance and Timing

On RISC CPU architectures, most instructions need a fixed number of clock cycles to execute, and this time is termed one **instruction cycle**. This interpretation means that some instructions need more than one instruction cycle to execute. Measuring response times in instruction cycles makes sense because this is independent of the actually CPU oscillator used.

How fast an external event needs to be serviced tells how timing critical the event is. The following three timing classes have been defined as examples:

a) 30 - 300 Instruction Cycles : Very timing critical
b) 300 - 3000 Instruction Cycles : Timing critical
c) > 3000 Instruction Cycles : Moderate timing critical

Clearly, to provide response times down to 30 instruction cycles, clever coding is required. Even interrupt may have problems with such demands. The good news is that LEANSLICE is suited for very timing critical application needs. It will be required to use few tasks, custom scheduling and instruction cycle counting to ensure that the timing requirements are satisfied. But it is possible.

LEANSLICE enables precise timing because of its static nature. It is possible to do timing analysis of the generated assembly code by instruction counting.

A time slicing system will have to use preemptive scheduling do deal with response times in the region around 300 instruction cycles. The reason is that a context switch is costly because of the register saving. Also, if more than one task have high priority, then response time may exceed 300 instruction cycles in the worst case. This depends on the CPU architecture and the kernel performance.

When using LEANSLICE, a task will normally execute in 15-30 instruction cycles. This means that round robin scheduling of 10 tasks easily satisfy response times of 300 instruction cycles for **all** tasks.

Microcontroller Performance

LEANSLICE depends on computed goto to operate. Most microcontrollers offer this feature. A single call level is used to schedule a task and return the next state to be executed.

LEANSLICE is intended and suited for use on Microchip PIC12/14/16/17/18, Ubicom SX and compatible RISC microcontrollers. Note that it is difficult to achieve tight timing on advanced microcontrollers like PIC17 and PIC18. The reason is that each CALL have to finish to do a task switch. But, the ability to do custom scheduling allows groups of tasks to scheduled in certain application modules, or at certain processing steps. The interrupt mechanism and hardware IO can then be utilised to handle the urgent timing needs of the application. This means that the LEANSLICE mechanism is useful on the larger microcontrollers also, as an alternative to state machines.

Device families such as PIC18, dsPIC and IP2000 offer access to the call stack and are thus suitable for time slicing multitasking approaches also. However, doing a context switch on the PIC18 requires saving registers like W, STATUS, BSR, FSR0L, FSR0H, FSR1L, FSR1H, FSR2L, FSR2H, PRODH, PRODL, TBLPTRL, TBLPTRH, TBLPTRH, TABLAT, PCLATH and PCLATU. That is more than 70 instruction cycles just to handle the vital registers. In addition comes the operations on the dynamic lists in the kernel. This limits the response time of time slicing systems.

1.5 LEANSLICE Features

The LEANSLICE concept currently have the following features:

- Multitasking support is integrated into the compiler for an **efficient** implementation
- The implementation is fully **static**, with no dynamic lists to maintain
- **No** multitasking kernel that consumes code
- Interrupt is **not** used, and can thus be dedicated for application usage
- **Small** multitasking code overhead for each task
- Very **fast** task switch: 10-15 instruction cycles
- **Round robin** scheduling is default
- Full **custom** task scheduling is possible to enhance performance, define priorities, or design a hierarchy of tasks that are scheduled on demand
- A **state** can be defined at each task switching point, allowing state information to be read outside the task for debugging or error recovery

- Task switch is performed only at well defined program code points which makes it easy to **protect** critical program regions
- User definable **delay** function for precise timing using multiple software timers
- Tasks can be stopped and started individually, restarted, suspended and resumed
- **Libraries** for timers, events, semaphores, binary semaphores and mailboxes

Fast Context Switch

Context switch means to switch execution from one task to another task. In the LEANSLICE concept, this mean to save the next state to be executed. At the syntax level, a task switch requires execution of the function waitState(). This is not a real function, but an abstraction that the compiler translates into proper instructions. The code between each waitState() is executed without interruption. Each waitState() adds **0-2 instructions** to the application. A context switch is normally done in **10-15 instruction cycles**.

Current Limitations

LEANSLICE enables fast and compact multitasking programs. However, there are some limitations to be aware of:

1) LEANSLICE is best suited when the tasks are small.
2) It is not possible to switch task in a function that is called from a task.
3) Tasks are not reentrant, which means that parallel code have to be generated when there are more than one task of the same type.
4) Local variables have to be defined static if they should survive a task switch.
5) Task priority is not available, but can be controlled by scheduling each task separately.

1.6 Installation and System Requirements

LEANSLICE version 1.1 is supported by:
1) CC5X EXTENDED edition, version 3.1G or later
2) CC8E EXTENDED edition, version 1.0F or later
3) CC7A EXTENDED edition, version 0.5B or later
4) CC1B EXTENDED edition, version 0.5E or later

The LEANSLICE library/header files needs to be copied to a suitable directory.

Summary of Delivered Files

```
MVENTIL.C     : multitasking example: ventilation control

DELAY.H       : timing / delay library
EVENT.H       : event library
BINSEM.H      : binary semaphore library
SEMAPHOR.H    : semaphore library
MSG.H         : mailbox library
```

1.7 Example with 3 Tasks: Ventilation Control

The following example shows how simple it is to create 3 tasks that operates independently or cooperates. The application needs only 151 code words on a PIC16C84, and 15 bytes of RAM. That is very little for a fully functional multitasking program. The application uses 3 software timers of 16 bit each that allows delays from 10 milliseconds up to 10.9 minutes. Using other software timers (8 or 24 bit) is simply controlled by a definition in application.

The first task ‘generatePulses’ generates 100 pulses of 0.5 second each with a 0.5 second off period. The task goes into a stopped condition after finishing, but can be restarted again. The purpose of this task is to show how a task can be started from another task (‘ventilationControl’), and then run independently.

Task ‘ventilationControl’ controls the ventilation on/off switch. Ventilation is switched on immediately after the temperature goes above a limit defined by the IO-pin ‘highTemperature’. After temperature goes below the limit again, ventilation is switched off after a 100 seconds period have expired. This ensures a hysteresis that prevents frequent on-off switching.

The third task ‘generatePCM’, operates totally independent of the other tasks. The job is to generate a 1 Hz PCM signal where the pulse width depends on the state of the ventilation switch.

```
// 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 What to do next

It is recommended to do a initial study of the application to decide if it is suited for a multitasking solution. There are no simple answers to this. It is always possible to write a traditional program using a loop in main that decide which operation to do next. Interrupts allows urgent IO needs to be serviced immediate. Interrupts often solve most needs for fast responses on parallel activities.

Issues that counts for using multitasking:

- activities are very much INDEPENDENT of each other
- there are many operations that can not be ordered in a certain sequence
- tasks needs to be serviced fast; there are timing constraints
- many tasks have similar operation, but need to be executed independent of each other

Issues that counts against using multitasking:
- there are strong dependencies between the activities
- most operations can be performed in a predefined sequence

Question is often: *How to solve the application needs easy and make a quality implementation that can be easily extended and maintained later.*

Summary of LEANSLICE advantages

- save code writing and state book-keeping
- get more readable code
- works for **all** PICmicro devices
- write tasks that operate independently or cooperate
- take advantage of the compact implementation
- use compact and efficient libraries for message passing, events, semaphores, binary semaphores and delays
- get the best of both approaches: the efficiency of state machines combined with easy task code writing

It is recommended to read the whole manual to learn all about the LEANSLICE concept.

2 LEANSLICE EXAMPLE

The purpose of following example is to show how LEANSLICE works. This is done by discussing the implementation and explaining the assembly code generated.

2.1 The Application : Reading Serial Data

The task used in this section is reading the input serial data stream. This is a trivial task that often is solved by on-chip hardware on many microcontrollers. The main reason to use this example is that it is reasonably well known. The aim is to show how multitasking can implement software UART's, handling several independent input channels simultaneously.

A serial data stream contains start and stop bits, and data bits that are defined by their position in the stream.

2.2 Task Switching

Task switching is done by calling the built-in function waitState(). This tells the compiler to insert a task switch. This most naturally done in wait loops, but also to break up large code sequence to ensure that timing requirements of other tasks are satisfied.

A simple example of how to use this function is when waiting for the start bit of an input serial IO line. When the input is 1, there is nothing else to do for this task than wait. And the waiting time should be used for processing other tasks. Note that for each waitState(), there will be an entry in the computed goto array in the beginning of the task. The implementation is very efficient.

```
while (input_serial == 1)
    waitState();

; generated assembly code
BTFSC 0x05,input_serial
RETLW .1
```

Solving timing requirements using multitasking requires a hardware timer. This timer should be free running to enable several tasks to use it. Testing on fixed timer values is not recommended when the timer increments reasonably fast (1 timer increment for each 8 instruction cycle is very fast). The generated code for waiting for the next data bit in the input stream follows:

```
                ;wakeupSerial += SERIALBIT_INTERVAL;
m003 MOVLW .13
     ADDWF wakeupSerial,1
                ;while ( !((wakeupSerial - TMR0) & 0x80))
m004 MOVF  TMR0,W
     SUBWF wakeupSerial,W
     ANDLW .128
     BTFSC 0x03,Zero_
                ;    waitState();  // wait for next bit
     RETLW .3
```

2.3 Scheduling

The default scheduling is round robin. This simply means that each task is scheduled in sequence, and then there should be a loop to do it all over again. The implementation of receiving serial data may thus look like (51 instructions on PIC16C54):

```
// Assuming 4 MHz and 9600 baud, prescaler divide TMR0 by 8
```

```
#pragma taskOptions 1

#define SERIALBIT_INTERVAL  13

bit input_serial @ PORTA.0;
bit output_serial @ PORTA.1;

char wakeupSerial;
char bitCounter;
char serialData;

Task receiveSerial( void)
{
    while (input_serial == 1)
        waitState();

    wakeupSerial = TMR0 + SERIALBIT_INTERVAL / 2;
    while ( !((wakeupSerial - TMR0) & 0x80))
        waitState();

    if (input_serial == 1)  // check start bit
        restartTask();

    bitCounter = 8;
    do  {
        wakeupSerial += SERIALBIT_INTERVAL;
        while ( !((wakeupSerial - TMR0) & 0x80))
            waitState();  // wait for next bit
        Carry = input_serial;
        serialData = rr( serialData);
    } while ( --bitCounter > 0);

    // check the stop bit
    wakeupSerial += SERIALBIT_INTERVAL;
    while ( !((wakeupSerial - TMR0) & 0x80))
        waitState();  // wait for next bit

    if ( input_serial == 0)
        restartTask();  // wrong stop bit

    // store/handle received byte

    restartTask();
}

void main( void)
{
    // initialize
    startTask( receiveSerial);
    OPTION = 2;  // TMR0 period is 8 microseconds at 4 MHz
    while (1)
        taskSlicer();
}
```

Custom Scheduling

When using the default round robin scheduling, the receiving task uses active waiting until the stop bit is detected. When using task type 1, the receiving task needs around 11 instruction cycles during the idle time. This should be acceptable in most cases, but custom scheduling may improve the performance. How to do custom scheduling efficiently depends on the application. For the bit receiving task, it is possible to move the timer checking outside the task, and only schedule the task when the next bit is ready. The renewed code follows (46 instructions on PIC16C54).

```
Task receiveSerial( void)
{
    if ( input_serial == 1)  {  // first check start bit
        receivingFlag = 0;  // error, too short start bit
        restartTask();
    }
    bitCounter = 8;
    do  {
        waitState();  // wait for next bit
        Carry = input_serial;
        serialData = rr( serialData);
    } while ( --bitCounter > 0);

    waitState();  // wait for stop bit
    receivingFlag = 0;
    if (input_serial == 0)
        restartTask();  // wrong stop bit

    // store/handle received byte

    restartTask();  // wait for next start bit
}

void main( void)
{
    // initialize
    startTask( receiveSerial);
    OPTION = 2;  // TMR0 period is 8 microseconds at 4 MHz
    while (1)  {

        if (receivingFlag)  {
            if ((wakeupSerial - TMR0) & 0x80)  {
                taskSlicer( receiveSerial);
                wakeupSerial += SERIALBIT_INTERVAL;
            }
        }
        else if (input_serial == 0)  {
            wakeupSerial = TMR0 + SERIALBIT_INTERVAL / 2;
            receivingFlag = 1;
        }
    }
}
```

3 LEANSLICE DETAILS

3.1 Available Task Types

It is required to chose a task type when using LEANSLICE. Three types are available:

1) Type 1 tasks are always active. The main benefit is slightly more compact code.

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

2) Type 2 tasks may stop execution (state 0xFF). Note that stopped tasks requires some fixed execution overhead (10-15 instruction cycles). The execution overhead for active tasks is the same as type 1.

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

3) Type 3 tasks allows execution to be suspended in all states: suspendTask(), resumeTask() and waitUntilResume(). The main advantage is that the execution overhead of suspended tasks is low (3 instruction cycles). The disadvantage is slightly slower execution of active tasks (2 instruction cycles extra for each scheduling).

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

Which task type to chose depends on the application. If all tasks are always active, then type 1 is preferred. If some tasks are allowed to stop execution, then type 2 or 3 must be chosen. If most tasks are active, then type 2 is best. However, type 3 have some more functionality because tasks can be stopped and resumed in any state. Note that type 1 tasks should be considered when using custom scheduling because task execution then is fully controlled by the application.

Task Options Summary

```
#pragma taskOptions 1  // tasks never stops executing
#pragma taskOptions 2  // task may stop (state 0xFF)
#pragma taskOptions 3  // suspend task in any state (bit 7)
```

NOTE: One of the options 1, 2 or 3 are required to enable multitasking.

```
#pragma taskOptions 7  // fast test of invalid states
#pragma taskOptions 8  // fast test of invalid states
#pragma taskOptions 9  // test for invalid states
#pragma taskOptions 10 // test for invalid states
#pragma taskOptions 11 // NOT computed goto for state selection

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

3.2 Defining Code Slices

It is currently required to split each task into a sufficient number of fragments (code slices) by using waitState() or waitUntilResume(). These are not real functions, but abstractions that tells that control (CPU execution) is passed to other tasks. How large each fragment should be depends on the application and the timing requirements.

It is especially important to pass control to other tasks in loops waiting on an external IO condition. Inspection of the generated assembly code is always recommended.

Task States

The compiler will assign a state to each task code slice. The reason is simply that the state identify which code slice to execute. The states are enumerated from 0 to enable computed goto into a table containing goto's to the start of the code slices.

Note that waitState() and waitUntilResume() can ONLY be performed in functions of type Task, not in any other function.

```
Task tx( void)
{
    //..
    waitState(1);
    //..
    waitState();  // hidden state, set to 2 by the compiler
    //..
    waitState(4);  // state out of sequence is OK
    //..
    waitState(3);
    //..
}
```

Fixed Task States

Interpretation of states:

- State 0 : startup
- State 1 .. N : active states
- State 0xFF: Task type 2 stopped or killed
- State 0x80: Task type 3 stopped or killed
- State Bit 7 = 1: Task type 3 suspended

Automatic or Custom Defined Task States

The function waitState() allow the compiler to select a state when the state parameter is missing. The actual state is then found counting the number of previous waitState(). It is therefore possible to use tasks with mixed waitState() and waitState(n).

If state parameters are used, then it is recommended to enumerate the states used by waitState(n) and waitUntilResume(n) to get a meaningful symbolic representation. Note that is NOT required to use the states in sequence. Thus, waitState(5) can proceed waitState(3). However, there can not be any "missing" states. This causes an error message.

```
enum {StartupT1, State1, WaitingForXXX, .. };
```

NOTE that state 0 (StartupT1) can not be used inside a task. It is for startup use only. Starting enumeration from state 1 is therefore an alternative:

```
enum {State1 = 1, WaitingForXXX, .. };
```

Reading the Task State

It is possible to read the address of the task state variables. This is mainly useful for type 3 tasks, and enables general synchronization mechanisms to be constructed. The state address of an suspended process can be stored in a variable. Later, when the right condition is satisfied, this task can be activated by clearing bit 7 through an indirect variable access.

```
pt = getStateAddress();
waitUntilResume();
```

```
..
//Code executed in another task or function:
*pt &= ~0x80;  // resume execution of idle task

resumeTask( task);  // when the waiting task name is known
```

Invalid Task States

Testing for invalid states is optional, and the default setting is to **not** test for invalid states. Invalid states may occur on missing task initialisation or when faulty application code overwrites RAM. (There may also be a very small probability that environmental noise can corrupt RAM contents, but this will probably cause the whole application to fail anyway). The available test modes are:

1) Stop task (task type 2,3) in case of illegal state. Tasks of type 1 are restarted because they are always running. The test requires 3-4 instructions for each task and slows down execution.

```
#pragma taskOptions 10  // test for invalid states
```

2) Same as the first test type, except that type 3 tasks are restarted on invalid states.

```
#pragma taskOptions 9   // test for invalid states
```

3) Fast (1 instruction cycle) test of invalid states by using ANDLW and extra RETLW's. Invalid states are thus mapped to a valid state. This ensures that the computed goto never jumps to a random destination. The fast test is only performed if the task have less than 31 states (30 states for type 2 tasks). Otherwise an error message is generated. Note that the insertion of extra RETLW's may increase the code size if the task have 8-10 or 16-26 states compared to a type 1 test.

```
#pragma taskOptions 8    // fast test of invalid states
```

4) Same as test type 3, except that the compiler will use a type 2 test if the task have 16-26 (or more than 32) states to save code.

```
#pragma taskOptions 7    // fast test of invalid states
```

3.3 Task Scheduling

The TaskSlicer

Function taskSlicer() implements round robin scheduling of tasks. This simply means that all defined tasks are scheduled in sequence, one by one. It is recommended to use taskSlicer() in main().

Custom Scheduling

The alternative is to schedule each task separately by taskSlicer(taskN). This allows priority levels to be constructed according to the application needs. It is also possible to only schedule tasks at certain conditions. Flags can be used to enable/disable a group of tasks. Example of custom defined scheduling:

```
while (1)  {
    //.. perform IO
    taskSlicer( task1);
    //.. perform IO
    taskSlicer( task2);
    taskSlicer( task3);
    //.. perform IO
    taskSlicer( task1);   // high priority
    //.. perform IO
```

```
    if (enableTaskSetX)  {
        taskSlicer( taskX1);
        taskSlicer( taskX2);
        //.. perform IO
        taskSlicer( task1);   // high priority
        taskSlicer( taskX3);
    }
}
```

Note that custom scheduling frequently will increase code size and RAM usage. And performance may be slower for certain operations. So it is required to do instruction cycle counting and overall timing analysis to verify that total performance is improved when doing custom scheduling.

Creating and Scheduling a Hierarchy of Tasks

When timing is less critical, it is possible to schedule groups of tasks in called functions. Thus, only tasks relevant for the current module need to be processed. It is possible to initialise the tasks for each call to the module, or alternatively use the state from previous call. This open for using LEANSLICE tasks instead of state machines in conventional designed programs (i.e. programs that run in loops and call IO service routines on demand). Very few multitasking systems have this flexibility because they take control of the whole system. LEANSLICE tasks can be used on demand when needed, and limited to certain parts of the application.

For example, if the instruction cycle is 1 microsecond and the application needs timer resolution of 0.1 seconds, then tasks that depend on timeout do not need to be scheduled frequently.

3.4 Built-in Multitasking Functions

Restarting a task

The syntax for restarting a task is calling a built-in function. This simply translates to a return.

```
restartTask();  // the task restarts itself

; generated assembly
RETLW .0
```

It is also possible to start and restart any task. Note that this is a brute force start/restart that simply set the state variable of the task to 0 (startup state). This function should be performed at program startup for each task. Any other use requires care. Examples of such usage is to resolve error or exception conditions of the application.

```
startTask( task1);   // start/restart task

; generated assembly
CLRF _TaskS1
```

Performing a Task Switch

The built-in function waitState() allow a task switch to be performed. Supplying state information is optional. If not supplied, the compiler will select a free state number for this action.

```
// set state and switch to another task (remain active)
waitState();         // RETLW .1
waitState(3);        // RETLW .3
```

Changing to an existing state is done by changeState(state). This is near equivalent to using “goto Label”, except that a task switch is performed first.

```
changeState( 3);  // resume execution from state 3
```

Note that changeState(state) does not introduce any new state, but requires that the state is defined somewhere else in the task. Execution is resumed in the specified state.

Suspending and Resuming a Task

When using tasks of type 3 (#pragma taskOptions 3) it is possible to suspend and resume tasks. Bit 7 of the task state variable is used to define the suspended state. A suspended task depends on active parts of the application to release it from the suspended state. A task can suspend itself, or it can be suspended from another task or any other function in the application.

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

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

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

Killing a Task

It is possible to clear the current execution of a task and set it to an idle state. This function is not available for type 1 tasks. The killed task will not restart execution until a startTask() or resumeTask() is performed. Execution will then start from the beginning of the task.

```
clearTask(task1); // kill task
```

Scheduling a Task

Scheduling a task means to execute task code (a code slice) from the position identified by the current task state and continue execution until the task pass control by returning the next state of itself (i.e. the state to be executed at next schedule).

```
taskSlicer();  // schedule all tasks
taskSlicer( task1);  // schedule task1 only
```

Reading the State of a Task

It is possible to read the state of a task:

```
state = getState( task1);  // read task state
ps = getStateAddress( task1); // state var address
ps = getStateAddress(); // state address of current task
```

3.5 Creating Multiple Tasks of the Same Type

Reentrant code is currently not supported. This would frequently not be sufficient, because even if most of the code of the task instances are identical, each instance often needs to operate on different IO pins. The answer is to duplicate code by defining the task as a macro and use macro expansion to define several task instances statically. The advantage of static definitions is that there never will be too little RAM at runtime. Using macro arguments allow each instance to be customised. Newer versions of compilers from B Knudsen Data have improved error/warning printing in macros (recursive printing), and better readability of generated assembly because macro source lines are printed in the assembly file.

3.6 Task Details

Local Variables in Tasks

No parameters or local variables can be defined when waitState() or waitUntilResume() is performed. An error is printed if the compiler finds active local variables. Solutions: Add modifier 'static' to the local variable or use a narrow scope that is terminated before waitState() is executed:

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

NOTE that a static variable consumes RAM locations. Only local variable space can be reused. Extra blocks { .. } ensures that local variables are undefined when a task switch is performed (waitState()). Therefore, static should be used only if the variable contents have to survive a task switch!

Task Startup

ALL tasks needs to be initialized before taskslicer() is called. Otherwise the task state may be undefined.

```
 startTask( task1);   // start task (at program startup)
 clearTask( task1);   // idle task at startup
```

There is currently no check to verify that the tasks are initiated correctly. Initialisation should be done by startTask() or clearTask() before the first taskSlicer() is performed.

Note that clearRAM() will set all tasks to running because 0 is the startup state. Therefore, after using clearRAM() it is recommended to use startTask() or clearTask() to define the initial state of the tasks.

Terminating a Task

Tasks are allowed to terminate. Type 2 and 3 tasks are stopped on return or function termination. Type 1 tasks are restarted.

Task Prototypes

The task have to be known before it is possible to do operations like startTask() etc. Task prototypes are sometimes required. Note that the state variable is allocated when the prototype is defined. Also, unlike other prototypes, an error will occur if the task is not defined somewhere in the application.

```
Task taskX( void);  // task prototype
```

Adjusting Task Start Address

It is sometimes required to adjust the hex code address of the tasks. The 12 bit core requires that computed goto are limited to reside within the first 256 address words on each code page. The 14 bit core requires that no 256 word boundary is crossed during the state selection. The compiler generates an error message when a 256 word boundary is crossed. The cure in case of such errors is to:

a) The compiler can insert a series of instructions (12/14 bit cores: XORLW+BTFSC+GOTO) instead of computed goto. This requires much more instructions if the task have more than 2-3 states.

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

b) RECOMMENDED. Insert a dummy function above the task which caused the error. Add dummy code which push the beginning of the computed goto below the 256 word page boundary. The address of the instructions can be checked in the generated list file. The dummy function can be removed later when the application matures, for example by using conditional compilation (#if .. #endif).

c) Use #pragma origin to skip some locations

d) Move functions (change the definition sequence) until the error message disappear.

Using RAM Banks

The following code describes the easiest way to locate the task state variables in a RAM bank DIFFERENT from the bank used for local variables and parameters. The reason for this can be space limitations in mapped/common RAM.

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

3.7 Interrupts

The interrupt mechanism is not required, and can operate fully independent of LEANSLICE tasks. There are thus no restrictions or guidelines for interrupts. However, note that interrupts will delay the task scheduling and overall timing.

4 Multitasking Libraries

The LEANSLICE multitasking libraries implements common multitasking features. The libraries complies to the lean approach (with no dynamic data structures). This means that there are no queues of waiting tasks. Tasks that have to wait will use active waiting or the suspend mechanism. The multitasking libraries use most frequently active waiting.

4.1 Delays and Timing

The timer module (delay.h) implements 8, 16 or 24 bit software timers that can be used in the tasks to get precise delays. The timer increments have to be much slower than the instruction cycles. This means that the software timer operates around 100 or 1000 Hz when the microcontroller run at 4 - 20 MHz. Around 1000 instruction cycles for each software timer decrement is normally sufficient. Note that math operations are time consuming. It is possible to modify library delay.h if the application timing requirements is outside the implemented options.

The module delay.h offers both simple delays or active waiting. It is possible to define a timer for each task. The definitions in the application program should look like:

```
enum {TimerA, TimerB};     // Timer identifiers
#define FreqOsc      4000000 // Oscillator frequenzy
#define FreqTimer    1000    // Timer frequency, 1000 or 100 Hz
#define TimerType    uns16   // uns8, uns16 or uns24
#define NoOfTimers   2       // 1 .. 8
#define DELAY_TYPE   1
#include "delay.h"
```

Initialization and increments:

```
initTimers();  // initialization on application startup

// The timers need to be incremented by making frequent calls
// to timerTick()in main. The calls must not be done more infrequent
// than the timer decrement period
timerTick();
```

Using the timers by active waiting or simple delays:

```
// active waiting means starting the timer and checking for timeout:
startTimer(TimerA, 100);
if (timeout(TimerA)) ..

// Simple delays have built-in task switching:
delay(TimerA, 1000); // delay up to 1000 timer ticks
```

Note that delay(1) will not be precise because the timer depends on a single hardware timer, and it is not possible to tell when the next hardware increment occurs. The duration of delay(1) will be in the interval 0.0 to 1.0 period. For example, if half of the period to the next hardware timer increment have expired, then the actual period will be 0.5. This means that a series of delays will be more precise than a single delay. The reason for this is that several task may need service at a timeout, and some tasks may have to wait a fraction of the next timer increment before the task is scheduled. On the other hand, if the timer is started because of an external IO event, then the duration of delay(100) will be in the interval 99.0 - 100.0.

Performance

DELAY_TYPE 1: The software timers are implemented using the 8 bit hardware timer TMR0 and the prescaler. Up to 8 timers is allowed.
initTimers() : 6 instruction words. Initialize prescaler. Set timeout to TRUE on all software timers.
timerTick() : 23+ instruction words. The size increases depending on the timer size (8, 16 or 24 bit) and the number of timers. Three 16 bit timers requires 46 instruction words. The function requires minimum 12 instructions cycles when the next hardware timeout period have not expired. This increases when software timers needs to be decremented (around 49 instruction cycles when decrementing three active 16 bit timers).
delay(T,C) : 6-7 instructions when using 16 bit timers
startTimer(T,C) : 4-5 instructions when using 16 bit timers
if (timeout(T)) : 1-2 instructions

4.2 Events

Events can be used to broadcast information between tasks. The implemented event type is simple and each event maps to a single bit. Tasks waiting for an event will use active waiting. When using round robin scheduling, it is enough to set the event to TRUE and then do an task switch. This ensures that the tasks waiting for the event will detect it. The event can be CLEARED at the next scheduling of the task that set it to true. It is also possible to use the event mechanism in other ways.

Required definitions in the application program:

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

Initialization:

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

Using events:

```
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
```

Performance

EVENT_TYPE 1: Single bit event representation, up to 32 events possible.
initEvents() : 1 instruction (1-8 events), .., 4 instructions (25-32 events)
setEvent(E) : 1 instruction
clearEvent(E) : 1 instruction
waitUntilEvent(E) : 2-3 instructions

4.3 Semaphores

Semaphores are frequently used for protecting critical common data, regions or application resources. The value of the implemented semaphores can range from 0 to 255, and is also called a **counting** semaphore. A granted semaphore normally represent a permission to use a shared resource. The actual interpretation depends on the application. Note that binary semaphores normally satisfy simple needs to protect data. However, if many tasks are allowed to read or write shared data simultaneously, then a counting semaphore is worth considering.

Required definitions in the application program:

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

Initialization:

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

Using semaphores:

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

Semaphore types:

- Type 1: **blocking** implementation: the highest semaphore request blocks other requests. Example: a semaphore is given the value 5. Up to 5 tasks hunts for one semaphore unit each to enter a critical region. A single task needs to update the data in the region and asks for all 5 semaphore units. The reading tasks are then blocked until the writing task is granted all 5 semaphore units and have returned them. The blocking approach simulates a queue with 2 levels.

- Type 2: **non-blocking** implementation: free semaphores are always granted. Tasks requesting for many semaphore units never blocks tasks requesting for an available number of semaphore units. Tasks requesting many semaphore units may thus have to wait long.

- Type 3: **non-blocking** implementation, same as type 2. Slower execution, but possibly less code when semaphore identifiers are allocated dynamically in the application (i.e. the semaphore identifier is stored in a variable instead of being represented by a fixed constant identifier).

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 Binary Semaphores

Binary semaphores are used for protecting critical common data or regions. This ensures that only one task is granted access at the same time.

Required definitions in the application program:

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

Initialization:

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

Using binary semaphores:

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

The available single binary semaphore type allows up to 8, 16, 24 or 32 binary semaphores.

Performance

BINSEM_TYPE 1: Single bit semaphore representation, up to 32 semaphores possible.
initBinSemaphores() : 1 instruction (1-8 semaphores), .., 4 instructions (25-32 semaphores)
getBinSemaphore(B) : 3 instructions
freeBinSemaphore(B) : 1 instruction

4.5 Mailboxes

Mailboxes are intended for exchanging byte messages between tasks. The available mailbox interface allows various degrees of synchronization between the tasks. Note that a mailbox can only contain a **single** message. There is no queue for storing multiple messages waiting to be processed. However, the sender can check if the receiver have removed the message from the mailbox, and receive an acknowledge message.

Required definitions in the application program:

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

Initialization:

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

Interface:

```
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)
```

```
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
```

Using mailboxes for exchanging byte messages:

```
// 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
```

Message interpretation:

Type 1: Return messages and empty mailbox share representation (most compact code)

```
0-0x7F : messages
0x80-0xFF: empty mailbox or acknowledge messages
```

Type 2: No return messages

```
0 : empty mailbox (mail removed)
1-0xFF: messages
```

Type 3: Includes return messages and specific empty mailbox representation

```
0 : empty mailbox (mail removed)
1-0x7F: messages
0x80-0xFF: return/acknowledge messages
```

Type 4: Can only be used together with type 3 tasks. Receiver is suspended until message is ready. The advantage is faster execution because receiver is suspended. The disadvantage is need for more code and more RAM. Also note that task state addresses and mailbox addresses have to reside in RAM below address 0x100 in this implementation.

```
0-0x7F : messages
0x80-0xFF: empty mailbox or acknowledge messages
```

Performance

MSG_TYPE 1,2,3,4:
initMbox(B) : **1,2,3,4**: 1 instruction
putMsg(B) : **1,2,3**: 2 instructions, **4**: 9 instructions
waitUntilMsgRead(B) : **1,4**: 3 instructions, **2**: 4 instructions, **3**: 6 instructions
deliverMsg(B, M) : **1,2**: 6 instructions, **3**: 8 instructions, **4**: 12 instructions
waitUntilMsgAck(B) : **1,3,4**: 3 instructions
getMsgCopy(B) : **1,2,3,4**: 2 instructions
hasMsg(B) : **1,2,4**: 1-2 instructions, **3**: 5 instructions
noMsg(B) : **1,2,4**: 1-2 instructions, **3**: 4 instructions
clearMsg(B) : **1,2,3,4**: 1 instruction
ackOnMsg(B) : **1,3,4**: 1-2 instructions
setAckOnMsg(B) : **1,3,4**: 1 instruction
getMsg(B, V) : **1,2,3,4**: 3 instructions
getNextMsg(B, V) : **1**: 6 instructions, **2**: 7 instructions, **3,4**: 9 instructions
waitForMsg(B) : **1**: 3 instructions, **2**: 4 instructions, **3,4**: 6 instructions

5 DEBUGGING

Debugging a LEANSLICE multitasking application is similar to debugging other applications. However, there are some extra challenges. The main challenge is to ensure that the timing and response time constraints of the application are satisfied. The scheduling mechanism introduce more statistical timing variations, which may require analysis to deal with tight timing requirements. Fortunately it is possible to count instruction cycles by reading the generated assembly code to find worst case scenarios.

Another problem with multitasking is deadlocks. A deadlock may occur if two tasks are hunting for the same resources, but can not complete the operation because they are waiting for each other to release an allocated resource. Deadlocks should not be a problem when using LEANSLICE, because there are few dynamic allocated entities. However, it is strongly recommended to allocate the resources in a certain sequence, or only allocate groups of resources. Common resources are often protected by semaphores, and semaphore allocation is a place to look for deadlocks.

A third challenge with multitasking is rare events. The statistical operation of multitasking may mean that faulty combinations of events occur very rare. This is in general one of the deep challenges with multitasking, but it is also apparent in large programs with lots of dynamic structures. The more dynamism introduced, the more likely is it that special fault combinations that occurs rare may exist. The answer to rare fault events is careful code reading and code simulation. The advantage of simulation is that the time scale can be compressed to allow years of operation to be simulated in a few hours. It is also possible to limit simulation to certain modules and put these under heavy stress to provoke faulty situation to happen much more frequently than it will under normal use. Such simulation is recommended for all program types, both multitasking and not multitasking.

Rare events are the kind of problems that you will hate to deal with, because they can frequently not be reproduced. Ability to reproduce faults is a very strong arguments for using simulation because the simulator can be restarted, even if it takes hours to get to the fault again. Reproducing errors in the real world may fail if the events and external conditions leading to this error is complex.

However, multitasking applications built by LEANSLICE are small, have mainly a static structure and operates predictive. The debugging challenges should be moderate, and comparable to conventional single task programs.

5.1 Timing

There are two ways of ensuring that response times are within constraints. The first is instruction cycle counting, and the second is to perform timing measurements (real-time or simulated). The important factors are average, minimum and maximum.

5.2 Inspecting State Variables

Ability to inspect the state variables in a great advantage during debugging, but also during production testing. A collection of state variables reveal much of the inner operation of the application. The ability of giving each state a symbolic name is also useful. Inspecting the states from the outside is easiest when the state number is translated to a name string.

An debugging setup that proved to be very useful for a specific application was to send all state variables (and other vital RAM variables) as serial output data at each iteration in the main loop. This data stream was read by an external Personal Computer that translated the information to a screen picture. This made debugging and later production testing of produced copies very easy. This application used delays in the range of several seconds.

6 FUTURE ENHANCEMENTS

The LEANSLICE concept can be improved in several ways. A brief list of planned new features follows.

6.1 New Task Type

In some applications it is a disadvantage that idle tasks consumes CPU cycles. When most tasks are idle it will be most efficient to totally avoid executing code to check the state of these. The disadvantage is that this task type will require a dynamic list of active tasks. And it will not be possible to save instruction cycles when using custom scheduling.

6.2 Task Switch in Called Functions

Ability to perform a task switch in called function will be a great advantage. This will remove one of the most important disadvantages compared to time slicing systems.

Automatic Insertion of Task Switch Code

Automatic insertion of task switch code will provide better abstraction and enable usage of the standard math libraries. Functions that are shared between tasks should be automatically protected. This will be a major step in moving towards source code that looks like multithreaded applications.

6.3 Timing Analysis

A tool that can perform timing analysis of an application solution will be very useful. Such timing analysis is expected to require simulation of application code. A tool providing such ability should also offer:

1) Simulation of stress situations
2) Simulation of heavy load situations
3) Offer fast simulation to provoke faulty event combinations and situations that are rare.

7 MICROCONTROLLER NOTES

In general, LEANSLICE multitasking is suited for both small and large microcontrollers. Making timing critical multitasking solutions will be easiest on small microcontrollers (typically up to 2k words of code). The reason is that larger microcontrollers often contains advanced hardware on-chip IO that are suited for service by interrupt processing. Using LEANSLICE to support timing-critical solutions is difficult in large programs because it is normal to use deep call structures, and task switching is not possible in called functions. Instead it will be the state machine properties of the tasks that are important. Building a hierarchy of tasks that are executed on-demand is attractive.

7.1 Microchip 12 bit core

The 12 bit core (PIC16C5X etc.) have some limitations. First, each page contains 512 instructions. However, it is only possible to use CALL and computed goto to the first 256 addresses of each page. The CC5X compiler checks that these limitations are not violated. The restriction on computed goto means that it may be required to move tasks to another position in the source code if the goto table is not located in the first page half.

A task switch (type 1 and 2) requires 10 instruction cycles, which is very fast:

```
CALL  receiveSerial ;2: call next task
MOVF  _TaskS1,W     ;1: read task current state
ADDWF PCL,1         ;2: perform computed goto
GOTO  m001          ;2: jump to task slice of current state
..                  ;   perform task code
RETLW .3            ;2: return next state of current task
MOVWF _TaskS1       ;1: store state after return
```

A task of type 3 needs 2 more instruction cycles (total 12) to do a task switch. Also note that updating the page bits requires extra instructions. Because of the small bank size, it will be required to update the bank bits more frequently.

7.1 Microchip 14 bit core and PIC17

The 14 bit core (PIC16F87X etc.) and the 16 bit PIC17 core are similar. A task switch (type 1 and 2) requires 12 instruction cycles, which is fast. This is reduced to 11 instruction cycles when PCLATH is 0.

```
CALL  receiveSerial ;2: call next task
MOVLW .3            ;1 load constant to W
MOVWF PCLATH        ;1 update PCLATH
MOVF  _TaskS1,W     ;1: read task current state
ADDWF PCL,1         ;2: perform computed goto
GOTO  m001          ;2: jump to task slice of current state
..                  ;   perform task code
RETLW .3            ;2: return next state of current task
MOVWF _TaskS1       ;1: store state after return
```

A task of type 3 needs 2 more instruction cycles (total 14 (13)) to do a task switch. Also note that updating the page bits requires extra instructions.

7.3 Microchip PIC18

The PIC18 unfortunately does not have computed branch, which would have been very useful. However, if the computed goto does not cross a 256 byte address boundary, then it is enough to add an offset to PCL. Otherwise both PCLATH and PCLATU may need to be updated.

A task type 1 switch requires 11 instruction cycles:

```
CALL/RCALL rcvSerial ;2: call next task
MOVF  PCL,W,0        ;1: update PCLATH and PCLATU
RLNCF  _TaskS1,W     ;1: read task current state
ADDWF PCL,1          ;2: perform computed goto
BRA   m001           ;2: jump to task slice of current state
..                   ;   perform task code
RETLW .3             ;2: return next state of current task
MOVWF _TaskS1        ;1: store state after return
```

A task of type 2 needs 1 more cycle and task type 3 needs 2 more instruction cycles (total 13) to do a task switch.

Note that 3 extra instruction cycles are needed when a 256 byte boundary is crossed. These extra instructions are automatically inserted unless the -GS command line option is used. When generating relocatable assembly (option -r), the long skip format is always used. If the table contains more than 124 single word elements, even more instructions are needed. On devices with more than 65536 bytes of code, it will be required to add instructions that generates carry from PCLATH to PCLATU. All this means slower execution of the computed goto.

7.2 Ubicom SX : 12 bit core

The timing for the Ubicom SX devices are identical to the 12 bit Microchip core.

Task switch:
1. Type 1 tasks: 10 instruction cycles
2. Type 2 tasks: 10 instruction cycles
3. Type 3 tasks: 12 instruction cycles

Updating the page bits requires extra instructions. Because of the small bank size (16 bytes), it is often required to update the bank bits frequently.

8 APPLICATION NOTES

8.1 Comparing LEANSLICE and State Machines

To show the difference between state machines and the equivalent LEANSLICE code, a simple example is considered. The task ventilationControl() controls the ventilation on/off switch depending on a temperature limit. The behaviour is:

1. Switch ventilation off (initial state).
2. Wait until high temperature.
3. Switch on ventilation.
4. Wait until the temperature goes below the limit again.
5. Wait 100 seconds before stopping ventilation to prevent frequent on-off switching. Go to step 2 if the temperature goes high again while waiting.
6. Turn ventilation off and go to step 1.

An implementation of ventilationControl() as a state machines may look like:

```
char stateVC;
void ventilationControl(void)
{
    switch (stateVC)  {

      case 0:
        ventilation = Off;
        stateVC = 1;

      case 1:
        if (!highTemperature)
            break;

        stateVC = 2;
        ventilation = On;

      case 2:
        if (highTemperature)
            break;

        stateVC = 3;
        startTimer1(100);

      case 3:
        if (!timeout1)  {
            if (highTemperature)
                stateVC = 2:
            break;
        }
        stateVC = 0;
        break;
    }
}
```

Note that there are no wait loops in a state machine. At each iteration, a short check is performed to determine if it is time to change state. The iterations are performed by making calls to this function regularly.

The LEANSLICE concept allows the task to be written like a procedure or process. Note that the implementation is fully equivalent to the previous state machine definition.

```
Task ventilationControl(void)
{
    ventilation = Off;

    while (!highTemperature)
        waitState();

    ventilation = On;

   VENTILATION_ON:
    while (highTemperature)
         waitState();

    startTimer1(100);

    while (!timeout1)  {
        if (highTemperature)
            goto VENTILATION_ON;
        waitState();
    }
    restartTask(); // restart this task
    //turning ventilation off is performed at startup
}
```

The main difference from the state machine definition is the hiding of state information and insertion of waitState(). This enables the compiler to select the state during compilation. The size of the above task is only 22 instructions, plus code in main for starting and calling the task (3 instructions).
The ventilationControl() implementation uses active waiting while the timer is decremented. This allows the delay to be truncated if the temperature goes high again. Unconditional waiting can be implemented using the delay() function which hides the task switching.

The main program is simple:

```
void main( void)
{
    // initialize IO

    startTask( ventilationControl);
    // start other tasks

    while (1)  {
        // handle global IO: sampling AD channels, etc.
        // decrement software timers

        taskSlicer();  // run the tasks, round robin
    }
}
```

APPENDIX

A1 List of Built-in Multitasking Functions

```
void waitState( char);    // switch to next task
      // missing parameter means "hidden" state

void changeState( char);    // switch task
      // change to existing state

void startTask( Task (*task)()); // start task (or restart)
void restartTask( void);  // the task restarts itself

void clearTask( Task (*task)()); // make task idle (or kill task)
      // not available for type 1 tasks

void taskSlicer( Task (*task)());  // task scheduling
      // missing parameter means ALL tasks

char getState( Task (*task)());  // read task state
char getStateAddress( Task (*task)()); // read state var address
      // missing parameter means current task

/* Type 3 tasks only: */
void suspendTask( Task (*task)()); // suspend task
void resumeTask( Task (*task)()); // resume execution

void waitUntilResume( char);  // set state and suspend
      // missing parameter means "hidden" state
```

A2 News

LeanSlice 1.1 offer the following improvements compared to version 1.0:

1. PIC17 support
2. New built-in function changeState(existingState) which simplifies task design.