Dreamcatcher Kernel

Manual

Stephen Niedzielski

EEL 270 – Spring, 2007
Dr. Rania Hussein / Christopher Theriault

[Last Modified: April 19, 2007]

Table of Contents

Overview

This document describes the installation and use of the Dreamcatcher Kernel.

The Dreamcatcher Kernel is an open source minimal operating system designed for the MCF52233DEMO and is written in C for compilation using the CodeWarrior Development Studio 6.3. The kernel provides a basic framework for allowing multiple threads of execution utilizing a proportional sharing system.

All values within this manual should be assumed decimal unless otherwise specified. However, a prefix of "0x" indicates a hexadecimal base.
get the mucos book and use it as a reference

This kernel does not fully support standard IO features, such as printf.
OS_Yield
Monitor
static functions
DK_InitalizeTask should be able to return error.
preemptive kernel
PAGE 89 OF PIC.PDF SHOWS ACTUAL INTERRUPT TIMING
Consider saving the math data registers.
DMA control of MCF52233
user enable-able and definable hooks
plenty of exmples
keep system dependent code in one file.
keep application dependent code in one file.
math and rounding errors on different systems with different type sizes
priority
priority inheritance
other scheduling algorithms
monitor
IO management, other OS services?
handling interrupts
get time a needed function?
memory management
operatings systems features overview at beginning
SIE
classic C
traditional, not extended instruction set.
More MuCOS and kernel goal and thought
verify timer accuracy and durations and formulas and such
dk_ to DK_
read more documentation, look at more examples, do more tests
need os_yield
need variable task resources
need monitor
need to enable priorities
need better context saving
need to tidy files
union stack pointer, byte, int, function pointer, etc

Examine FreeRTOS
file io queue
USB hardware, HRRN

ccrt
These formulas are probably wrong, the duration code is proper.

$$\text{Interrupt Frequency} = \frac{\frac{\text{Clock Ticks}}{\text{Second}}}{[2^{Prescale} \cdot 4 \cdot \text{Modulo}]} = \frac{\text{Clock Frequency}}{[2^{Prescale} \cdot 4 \cdot \text{Modulo}]}$$

Prescale must be between 0 and 7

$$\text{Interrupt Frequency} = \frac{\text{Clock Frequency}}{[2^{Prescaler+2} \cdot 4 \cdot (65\,535 - \text{TMR0} + 2 \cdot 4)]}$$

$$\text{Duration} = \frac{1}{\text{Interrupt Frequency}} = \frac{[2^{Prescaler+2} \cdot 4 \cdot (65\,535 - \text{TMR0} + 2 \cdot 4)]}{\text{Clock Frequency}}$$

$$\frac{\text{Duration} \cdot \text{Clock Frequency}}{2^{Prescaler+2} \cdot 4} = (65\,535 - \text{TMR0} + 2 \cdot 4)$$

$$\frac{-\text{Duration} \cdot \text{Clock Frequency}}{2^{Prescaler+2} \cdot 4} + 65\,535 + 2 \cdot 4 = \text{TMR0}$$

TMR0 is a two byte register, TMRH TMRL. Musn't be assigned a value greater than 65 535.

19.0714 Hz expected.

19.0686 Hz with 10 saves
19.0715 Hz with no additional saves

180 book for ideas

pcritical section
- mucos management
  - what should I be managing?
  - don't share pins, why would you?
  - memory, files, i/o, semaphores

OS_Yield()
monitor
- lock block, pointer, bytes to lock

PTP
custom slicing
kill_task, modify_task, ...
only things I will use?

dspic

correct procedure for removing a task
went in order, A, B, C, D

07.01.11	Create a bare bones template project.
07.01.15	Create a USB 2.0 slave application recognized as a HID for PC.
07.01.15	Integrate a NES controller without damaging or modifying it.
07.01.16	Make operating system cross-platform with PIC.

Settings / C / Codewarrior / windows / more examples

state diagram
idle – ready – unning
Test application
get rid of floats
needs fpu for floating point support
fP_coldfire
c_4i_cf_runtime

must be explicit and convert to double
10.0

.0001 milliseconds to 10 seconds.
and no less than (Clock Frequency / 4)^-1

Tasks with the prefix "initialize" are meant to be called once.
Configures may be called any amount, but still normally before starting whatever .

stack size / memory management / no need a6 / floating point support, lose all stack data in main

LOOK AT EXAMPLE CODE FOR RTOS / MCF AND PIC, INCLUDING MTRX SLIDES

must not ever succeed max tasks size, including dynamicly.

Examine other RTOS'.

This is not a traditional quantum based system. Time quantums are exactly what you want.

Circular diagram

Figure out why assertion is broken.

Idle task should not prop share.

Change to use int instead of float

verify int truncation is O.K. in division operations

dk_idle must be a user definable length since it allows a chance to be interrupted

need to get rom linker file

revise naming convention

Resources consumed by kernel
uses pit0
take the system out of supervisor mode.
uses interrupt controller 0
have hussein actually implement from documentation
compiler settings
functions available in mucos?
- Process management.
- Interprocess communication.
- Address space management.
- Hardware abstractions.
- Prioritization – you select management method.

Traps and user / supervisor
RTOS tests and features
statistics for debugging and performance reasons
IO blocking and blocking
Need to adjust idle task to occupy only a small stack size.
Need semaphores? What can I even do with this thing?
No ROM.

Co-requisite Documentation

The following is a listing essential documentation for MCF52233DEMO development. Download the most recent documentation from http://www.freescale.com/ or http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MCF52233.

MCF52235RM.pdf — MCF52235 Integrated Microcontroller Reference Manual
The most important document to have, the chip specific reference manual.

CFPRM.pdf — ColdFire Family Programmer's Reference Manual.
Describes the ColdFire family at a very low level, including a listing of the instruction set.

CCOMPILERRM.pdf — C Compilers Reference
Describes the CodeWarrior C compiler.

CF_Assem_Ref.pdf — CodeWarrior Development Studio Assembler Reference
Describes the assembler and assembly syntax.

ColdFire_Build_Tools_Reference.pdf — CodeWarrior Development Studio Build Tools Reference
Describes the linker and compiler.

M52233DEMO_SCH_E1_0.pdf — MCF52233DEMO Schematic

M52233DEMO_UG.pdf — MCF52233DEMO Overview

MCF52235RMAD.pdf — MFC52235 Reference Manual Errata.

V4ECFUM.pdf — ColdFire CF4e Core User's Manual
Describes an overview of the ColdFire processor.

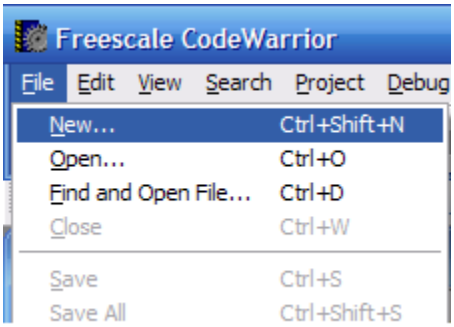
MSL_C_Reference.pdf — MSL C Reference

Targeting_ColdFire.pdg — CodeWarrior Development Studio Targeting Manual
Describes application development in CodeWarrior.

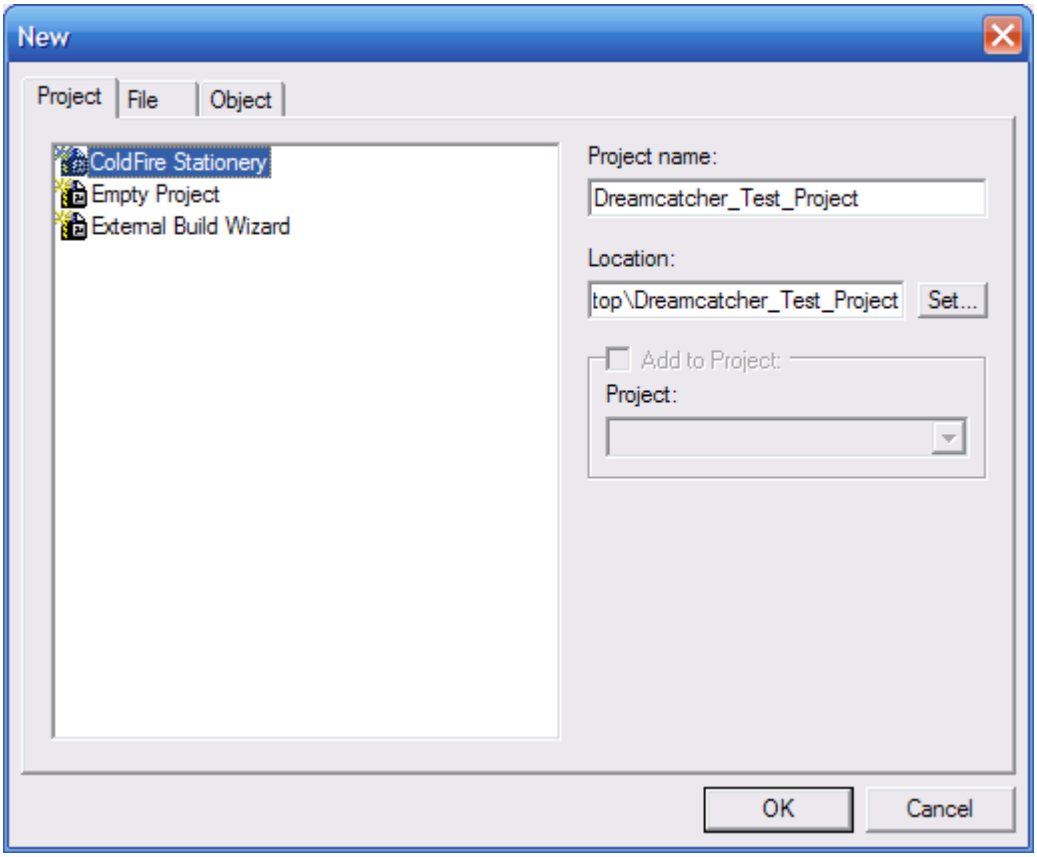
As some of the aforementioned documents are largely incomplete, you may find additional support and troubleshooting information at the Freescale forums, located at http://forums.freescale.com.

Kernel Setup

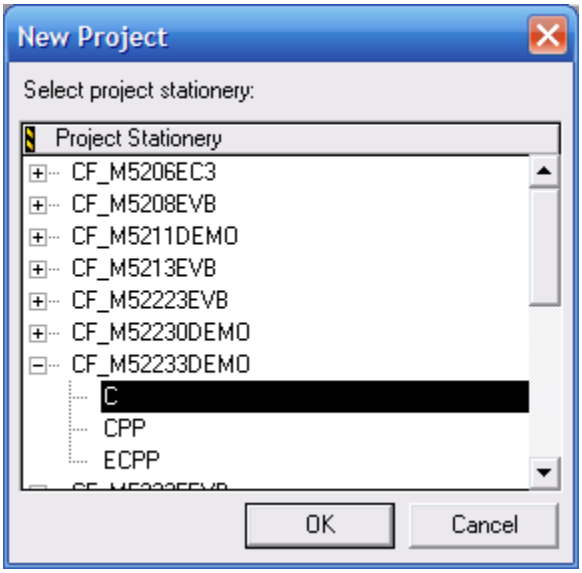
Launch CodeWarrior and select File > New...



Enter a project name and location, then select ColdFire stationary from the right and click O.K.



Click CF_M52233DEMO, and select C. Click O.K.

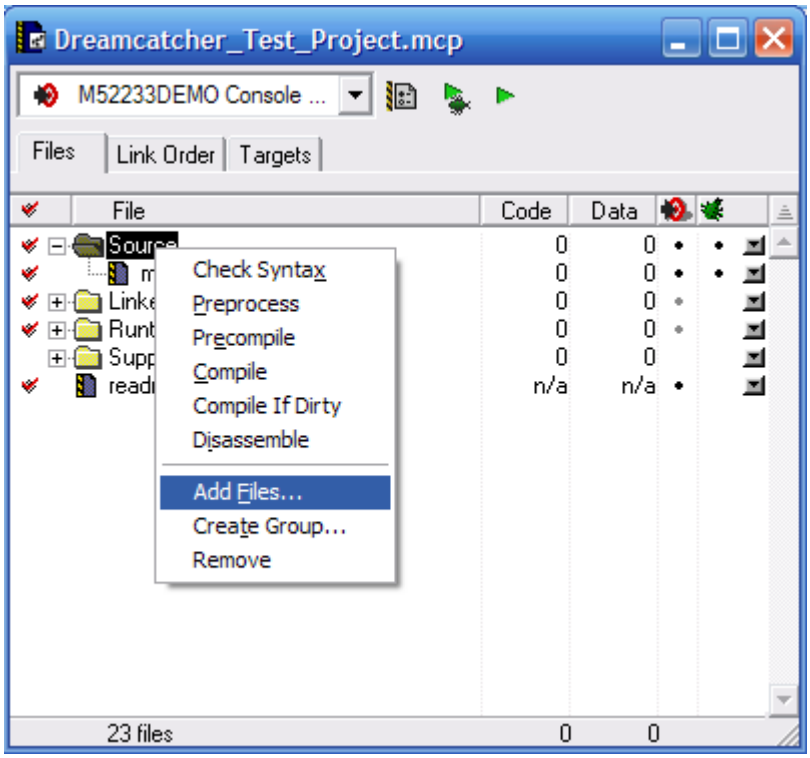


You should now have successfully created a project. Open up the project folder from Windows Explorer and copy the following files into your source folder, ignoring any overwrite warnings:

dreamcatcher_kernel.c
dreamcatcher_kernel.s
dreamcatcher_kernel.h
mcf52233_vectors.s

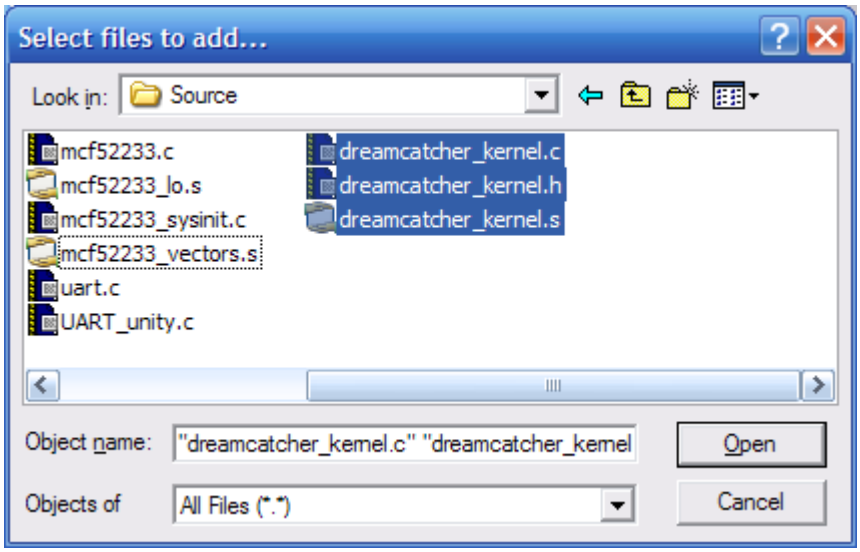
In addition, you will need to copy and overwrite the following file in your cfg folder:
M52233DEMO_SRAM.lcf

Open CodeWarrior again and right click on the source folder in project browser.

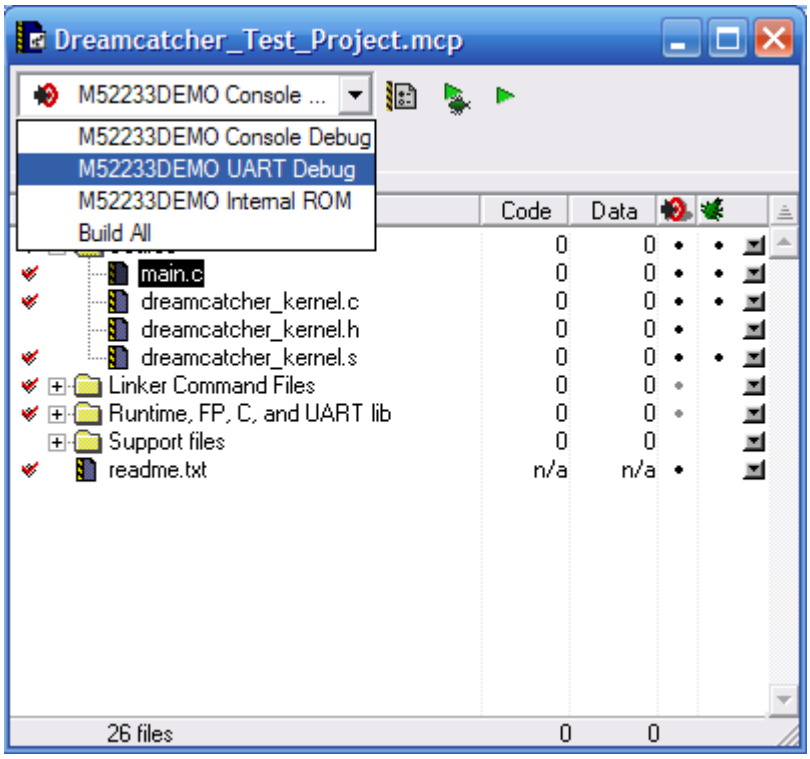


Click add files and select the following files:
dreamcatcher_kernel.c
dreamcatcher_kernel.s
dreamcatcher_kernel.h

Click open.



Again, in your project browser window, select the drop down menu and click M52233DEMO UART Debug.



Open main.c by double clicking it in the project browser. Edit main.c to appear as such:

```
/*
 * File:          main.c
```

```
 * Purpose:       sample program
 *
 */

#include "dreamcatcher_kernel.h"
#include "common.h"


/* A simple task. */
void TaskA()
{
   while(1)
   {
      /* Toggle an LED. */
      MCF_GPIO_PORTTC ^= 1;
   }
}

/* A simple task. */
void TaskB()
{
   while(1)
   {
      /* Toggle an LED. */
      MCF_GPIO_PORTTC ^= 2;
   }
}

/* A simple task. */
void TaskC()
{
   while(1)
   {
      /* Toggle an LED. */
      MCF_GPIO_PORTTC ^= 4;
   }
}

/* A simple task. */
void TaskD()
{
   while(1)
   {
      /* Toggle an LED. */
      MCF_GPIO_PORTTC ^= 8;
   }
}

/* Initializes LED's for use. */
unsigned InitializeLEDs()
{
    /* Configure TIN0-3 for IO operation. */
    MCF_GPIO_PTCPAR &= ~0xF;

    /* Configure data direction to output. */
    MCF_GPIO_DDRTC |= 0xF;
```

```
        /* Turn off TIN0-3. */
        MCF_GPIO_PORTTC &= ~0xF;

        return 1;
}

/* Edit these functions for statistics information. */
void OS_ClockTickHook() {}
void OS_ScheduleHook() {}
void OS_ContextSwitchHook(){}


int main()
{
   unsigned Result = 0;
   Result = InitializeLEDs();

   if( Result == 1)
   {
      /* Initialize the kernel. */
      Result = dk_InitializeKernel();
   }

   if( Result == dk_SUCCESS)
   {
     /* No errors thus far, initialize the tasks. */
     /* No errors thus far, initialize the tasks. */
      dk_InitializeTask(TaskA, READY, 0.0256 );

      dk_InitializeTask(TaskB, READY, 0.0256 );

      dk_InitializeTask(TaskC, READY, 0.0103 );

      dk_InitializeTask(TaskD, READY, 0.0051 );
   }

   /* Start the kernel and never return. */
   dk_StartKernel();

   return 0;
}
```

Set breakpoints on your tasks. Compile and run the project in debug mode by pressing F5.

Proportional Sharing Algorithm Overview

The Dreamcatcher Kernel employs a proportional sharing system to distribute CPU control amongst tasks[1]. This section describes both the implementation and use of this system.

In a traditional proportional sharing implementation, ready tasks are each given a weight over a set duration, where weight is the number of quantums[2] allotted the task divided by the set duration, or total quantums. The running tasks uses up its portion of the duration, releases CPU control, and the next task is given its allotted time share. There is no priority and no task may starve. For example, let task A have a weight of one and task B have a weight of three with a duration or total quantum of 10. If task A gains control of the CPU and enters a running state, its run time may be described by:

1 For purposes of simplification, task, process, and thread will be considered the same within this document.
2 A quantum is the minimum duration a task must be in execution. The length of a quantum is a trade off between allowing finer granularity for task duration and adding context switching overhead.

$$\text{Task A Time Share} = \frac{1}{(1+3)} \cdot 10 = 2.5\, \textit{quantums}$$

Please note that since quantums are indivisible, this fraction must be either rounded up or down. In this example we will round up and consider task A to have a time share of three quantums. The scheduler will interrupt after the first quantum has expired, see that the task has a remaining time share and allow it to continue. This will happen twice more, so that on the third interrupt the task will have no remaining time share and task B will be given control of the CPU for the next seven quantum. Once task B's time share has been expended, assuming no additional tasks exist, task A will receive a new time share of three quantums and the cycle will continue.

Unfortunately, this system invokes a scheduler interrupt unnecessarily as tasks with remaining time shares are never switched. In addition, as quantum are indivisible, task time shares may not be a non-integer number of quantum.

The Dreamcatcher Kernel avoids the heretofore problems[3] by eliminating quantum entirely. Instead, tasks are each given a task duration specified in seconds or clock ticks, permitting precise time allowances. For example, a set duration of ten seconds is chosen. As before, task A is given a weight of one, and task B is given a weight of three, giving task A a duration of:

$$\text{Task A Time Share} = \frac{1}{(1+3)} \cdot 10 = 2.5\, \textit{seconds}$$

The scheduler clock is configured to task A's time share, task A gains controls of the CPU and begins execution. Task A continues uninterrupted until its time share has been expended, the scheduler is configured to task B's time share, and task B begins execution. After 7.5 seconds of execution, task B is interrupted by the scheduler and cycle continues again with task A.

Calculating Time Shares

Determination of a task time share may be found in four steps:

1. Decide on a cycle duration. This is the time it takes to execute every task once and return to the first task. Tasks are given a portion of this duration based on their weight.

2. Decide on the task weight. This is the proportion of time given to a task.

3. Complete step two for all tasks and determine the total task weight. This is the sum of all task weights.

4. Determine the time share for each task with the following formula:

$$\text{Time Share} = \frac{\text{Task Weight}}{\text{Total Task Weight}} \cdot \text{Cycle Duration}$$

For example, a cycle duration of one fifteenth of a second is chosen. Four tasks, A, B, C, and D, are made to have weights of 25, 25, 10, and five, respectively. Summing the task time weights, we find the total task weight to be 65. In review:

1. We arbitrarily chose a cycle duration of one fifteenth of a second. This is the time it will take to execute tasks A, B, C, and D, each once.

2. We decided that task A should have a large portion of CPU time and gave it a weight of 25; we decided that task B should have an equally large share and gave it a weight of 25 as well; we decided that task C should be given a lesser weight of 10; finally, we decided task D should have a weight of five.

3. We determined the total weight by summing each task's individual weight:

$$\text{Total Task Weight} = \text{Weight A} + \text{Weight B} + \text{Weight C} + \text{Weight D} = 25+25+10+5 = 65$$

3 For completeness, let it be noted that in the situation where only the idle task is ready or existent, needless scheduler interrupts may occur in both methods presented.

We now take the final step and determine the time share for each task:

$$\text{Share A} = \frac{\text{Weight A}}{\text{Total Task Weight}} \cdot \text{Cycle Duration} = \frac{25}{65} \cdot (\frac{1}{15}) \approx 0.0256\, seconds$$

$$\text{Share B} = \frac{\text{Weight B}}{\text{Total Task Weight}} \cdot \text{Cycle Duration} = \frac{25}{65} \cdot (\frac{1}{15}) \approx 0.0256\, seconds$$

$$\text{Share C} = \frac{\text{Weight C}}{\text{Total Task Weight}} \cdot \text{Cycle Duration} = \frac{10}{65} \cdot (\frac{1}{15}) \approx 0.0103\, seconds$$

$$\text{Share D} = \frac{\text{Weight D}}{\text{Total Task Weight}} \cdot \text{Cycle Duration} = \frac{5}{65} \cdot (\frac{1}{15}) \approx 0.0051\, seconds$$

As a precaution, we check our work by summing the time shares to see that they indeed equal one fifteenth of a second:

$$\frac{1}{15} \approx 0.06667\, seconds$$

$$\text{Share A} + \text{Share B} + \text{Share C} + \text{Share D} = 0.0256 + 0.0256 + 0.0103 + 0.0051 = 0.0666\, seconds$$

Other than a slight rounding error, we have verified our math is correct and finished successfully computing the time shares for tasks A, B, C, and D.

Calculating Prescale and Modulo Manually

Users not wishing to call dk_CalculatePrescaleAndModulo may calculate the scheduler clock's prescale and modulo manually.

Interrupt frequency may be defined by the following formula:

$$\text{Interrupt Frequency} = \frac{\frac{\text{Clock Ticks}}{\text{Second}}}{[2^{Prescale} \cdot 4 \cdot \text{Modulo}]} = \frac{\text{Clock Frequency}}{[2^{Prescale} \cdot 4 \cdot \text{Modulo}]}$$

From this, a task time share may be defined:

$$\text{Task Time Share} = \frac{1}{\text{Interrupt Frequency}} = \frac{[2^{Prescale} \cdot 4 \cdot \text{Modulo}]}{\text{Clock Frequency}}$$

Following the steps outlined in the *Calculating Time Shares* section, the task time share may be found. A prescale may then be tried, and a modulo found.

There are some constraints. A prescale may be no less than zero and no larger than 15; smaller prescale's yield more accurate timings. If a modulo is greater than the capacity of an unsigned short, 65 535, then a higher prescale must be chosen and a new modulo calculated. Finally, attempting to give a task a time share that results in a modulo of zero is undefined.

For example, after following the steps described in the *Calculating Time Shares* section, we find task A to have a time share of 0.0256 seconds. We find our clock frequency to be 60 MHz, 60 000 000 Hz. We multiply our original formula through by our constants for simplification:

$$\text{Task Time Share} \cdot \frac{\text{Clock Frequency}}{4} = \frac{[2^{Prescale} \cdot 4 \cdot \text{Modulo}]}{\text{Clock Frequency}} \cdot \frac{\text{Clock Frequency}}{4}$$

Filling in 60 000 000 Hz for our clock frequency and allowing cancellations:

$$\text{Task Time Share} \cdot \frac{60000000}{4} = \text{Task Time Share} \cdot 15000000 = 2^{Prescale} \cdot \text{Modulo}$$

Now that the original formula has been simplified to solve for the modulo, we fill in for our time share and obtain:

$$\text{Share A} \cdot 15000000 = 0.0256 \cdot 15000000 = 384000 = 2^{Prescale} \cdot \text{Modulo}$$

Solving for the modulo yields:

$$\text{Modulo} = \frac{384000}{2^{Prescale}}$$

We try out a prescale of zero:

$$\text{Modulo} = \frac{384000}{2^{Prescale}} = \frac{384000}{2^0} = \frac{384000}{1} = 384000$$

We realize that this modulo value is way too large, it must be less than or equal to 65 535 to fit. We try steadily increasing prescale's until we arrive at a modulo value less than or equal to the 65 535 constraint, producing the following table:

$$\text{Modulo} = \frac{384000}{2^{Prescale}}$$

Prescale	Modulo
0	384000
1	192000
2	96000
3	48000

We find the ideal prescale and modulo, three and 48 000, respectively.

Interrupts

The scheduler clock is defined as a level six interrupt. All user defined interrupts should be of level less than six.

User Symbol, Macro, and Data Type Reference

Symbolics defined or used by the user.

dk_MAXIMUM_TASKS
User definable. Specifies the maximum number of tasks that will be in existence at any point in time. Used to determine TCB allocation quantity and individual task stack size. Must be greater than or equal to 1. This number includes the idle task.

dk_TaskState
Possible task states enumeration. Only ready and running tasks are processed by the scheduler; dead tasks are inactive and may be initialized to a new task; other task states are the responsibility of the user. The following states exist:

DEAD This task is inactive and its resources are deallocated.
READY This task is not being executed, but the scheduler is allowed to select it for execution.
RUNNING This task is currently being executed.
BLOCKED This task cannot be executed because it is awaiting access to some resource.
WAITING This task is being forced to wait.
DORMANT This task should be ignored and is not processed by the task scheduler.

dk_TaskAddress
A pointer to a task typedef. Tasks should have a signature of void Task(void).

dk_EnableInterrupts();
dk_DisableInterrupts();
Atomic macros for disabling or enabling interrupts in critical sections. These macros do not stop the scheduler clock, but do disable the scheduler from asserting itself. In addition, the following macros erase any values found in the condition code register, which contains various arithmetic and other flags. Use with care.

M52233DEMO
Must be defined. Specifies that the current development environment is the MCF52233DEMO.

TRUE
Must be defined to 1.

FALSE
Must be defined to 0.

dk_SUCCESS
Standard function success return value.

User Function Reference

Functions that may be called by the user.

unsigned dk_InitializeKernel()
Initializes kernel for use.

return dk_SUCCESS if successful.

void dk_StartKernel()
Starts kernel execution. This function contains a critical section that is not exited until the idle task has finished initializing. This function will never return.

unsigned dk_CalculatePrescaleAndModulo(double Duration,
unsigned char * pPrescale,
unsigned short * pModulo)
Determines the scheduler clock prescale and modulo for the length of time specified in seconds by Duration. The final result is truncated down to integer form to fit in a short. Duration must be greater than zero and no greater than 143 seconds.

Duration Length of time to calculate prescale and modulo for.
pPrescale The storage location for the prescale result.
pModulo The storage location for the modulo result.
return dk_SUCCESS if successful.

unsigned dk_ConfigureSchedulerClock(unsigned char Prescale,
unsigned short Modulo)

Modifies the scheduler clock to interrupt at a specified time from the next start. As a side-effect, this function clears out the scheduler clock interrupt flag. It is recommended this function only be called within a critical, uninterruptable section. Callers should call dk_StopSchedulerClock prior to calling this function. Callers should obtain the appropriate prescale and modulo values by calling dk_CalculateSchedulerClockPrescaleAndModulo or by deriving it themselves using the manual method described in the manual.

Prescale	Clock prescale value.
Modulo	Clock rollover modulo.
return	dk_SUCCESS if successful.

unsigned dk_InvokeScheduler()

Invokes the scheduler immediately and resets the scheduler clock. The calling task forfeits any remaining time share.

return dk_SUCCESS if successful.

unsigned dk_StartScheduler()

Enables the scheduler, allowing it to assert itself once the current time share expires.

return dk_SUCCESS if successful.

unsigned dk_StopScheduler()

Disables the scheduler from asserting itself and preserves the current time share.

return dk_SUCCESS if successful.

unsigned dk_InitializeTask(dk_TaskAddress Task,
dk_TaskState State,
double Duration)

Registers a task with the scheduler and initializes a task control block. This function contains a critical section.

Task	A pointer to the the task address.
State	State to initialize task to.
Duration	Time share to initialize task to.
return	Task identity if successful (non-zero), 0 if there are no task control blocks available or an error occurred.

unsigned dk_ConfigureTaskState(unsigned char Identity,
dk_TaskState NewState)

Modifies a specified task's state to NewState. Tasks made ready will be registered with the scheduler if previously not ready, tasks changed from ready to an unready state will be de-registered with the scheduler. The running task's time share will not be adjusted, even if it's state has been changed, until the next scheduler interrupt. However, the running task will be affected by the processing time of the state change. Users wishing to maintain the time share of the running task should first call dk_StopScheduler prior to calling this function. This function contains a critical section.

TaskIdentity	The identity of the task whose state is to be modified.
NewTaskState	The state to place the specified task in.
return	dk_SUCCESS if successful.

unsigned char dk_GetRunningTaskIdentity()

Returns the currently running tasks identity.

return Identity of the task currently running.

Kernel Symbol, Macro, and Data Type Reference

Symbolics that users should not use or define.

dk_TCB
Task control block structure. Used to maintain task specific data for reference by the scheduler.

unsigned StackPointer	Contains the task's stack pointer.
unsigned short Modulo	Contains the task's scheduler clock modulo.
unsigned char Prescale	Contains the task's scheduler clock prescale.
unsigned char Identity	Contains the task's identity.
unsigned char State	Contains the task state.
dk_TCB * Next;	Contains a pointer to the next ready task when in ready list.
dk_TCB * Prev;	Contains a pointer to the previous ready task when in ready list.

Kernel Function Reference

Functions that should not be called by the user.

unsigned dk_InitializeTCBSegment()
Initializes the task control block vector.

return dk_SUCCESS if successful.

__interrupt__ void dk_ISR_SchedulerClock()
An assembly ISR to save registers and perform other low level work before control is given back to the compiler generated code. This ISR will never perform the standard return from exception command (RTE), instead passing the buck to the scheduler who will pass the buck to the dispatcher, which shall execute an RTE when it finishes. This function is a critical section.

unsigned dk_InitializeSchedulerClock()
Initializes the scheduler clock. Called by dk_InitializeScheduler.

return dk_SUCCESS if successful.

unsigned dk_InitializeScheduler()
Initializes the scheduler and its resources. Called by dk_InitializeKernel.

return dk_SUCCESS if successful.

void dk_Scheduler()
Determines the next task to run, manages the schedule, and calls the dispatcher. This function will not return, and instead invoke dk_ISR_Dispatcher which will end in a return from interrupt / exception to the newly selected process. This function is a critical section.

__interrupt__ void dk_ISR_Dispatcher()
Loads the next task's context. Returns to new dispatched task instead of caller, dk_Scheduler. This function is a critical section.

unsigned dk_RegisterTask(unsigned char Identity)

Registers a task with the scheduler. This function should only be called within within a critical section.

return dk_SUCCESS if successful.

unsigned dk_DeregisterTask(unsigned char Identity)

De-registers a task with the scheduler. This function should only be called within a critical section.

return dk_SUCCESS if successful.

void dk_IdleTask()

A special default task that is always ready and is only given CPU control at kernel start and when no other task is ready or existent. This function has an initialization section that occurs once at startup that contains a very brief critical section.