Dreamcatcher Kernel

Postmortem

Stephen Niedzielski

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

April 18, 2007

Introduction

This paper is an overview of the kernel and application I wrote in Spring, 2007 for EEL 270. I present my progress linearly from assignment three and conclude with my kernel application.

* * *

Assignment Three

Assignment three should have been suffixed "make the kernel," as it was a pretty big jump from the previous lab and really changed the previous program from a loop with interrupts to a kernel. At completion of assignment three, my kernel was built with Standard C and assembly, running on Freescale's MCF52233DEMO, and did little more than manage tasks[1] using a proportional sharing algorithm, though technically also featured a semaphore system.

Since I have yet to do any research on the prop share algorithm and won't start now, I'll just rip Chris' description of the algorithm without permission for a good quick explanation:

> "Proportional share scheduling: The READY tasks are each given a weight over a set duration. The RUNNING task continues to run until it uses up its portion of the duration. Let's say I have task A with a weight of 1 and task B with a weight of 3 with duration of 10 ticks. If task A is running, then it will continue to run over the next 1/(1 + 3) * 10 = 3 ticks at which point task B will run for the next 7 ticks. There is no inherent priority of task A over task B within a given duration but you could choose to give precedence to A as the shorter weighted task."

Put more simply, prop share is round robin with variable task times. Once a task is initialized, there is no further or dynamic sorting that needs to be performed for scheduling, meaning scheduling is literally loading the next member of the task linked list. Since tasks are guaranteed their CPU allocation, a quantum[2] based system, where the scheduler asserts itself at regular intervals to check the remaining quantum allocation of a task, would be wasteful. As such I implemented a system that would only interrupt when a context switch was known to be necessary[3]. This is a great benefit as unnecessary context switching is quite taxing of the CPU.

MCF52233DEMO 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

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.
3 Except in the case of the idle task being the only task available and in the case of only one task being available.

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.

Proportional Sharing Algorithm Overview

In a traditional proportional sharing implementation, ready tasks are each given a weight over a set duration, where weight is the number of quantums 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:

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

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 implementation avoids the heretofore problems[4] 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 Proportional 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:

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

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

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

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

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

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

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

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 Prescaler and Modulo Manually

Although a function is provided to automate prescaler and modulo calculation, users may calculate the scheduler clock's prescaler and modulo manually.

For the MCF52233DEMO, interrupt frequency may be defined by the following formula:

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

From this, a task time share may be defined:

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

Following the steps outlined in the *Calculating Proportional 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 Proportional 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:

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

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

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

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

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

Solving for the modulo yields:

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

We try out a prescale of zero:

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

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 prescaler's until we arrive at a modulo value less than or equal to the 65 535 constraint, producing the following table:

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

Prescaler	Modulo
0	384000
1	192000
2	96000
3	48000

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

* * *

Porting the Kernel

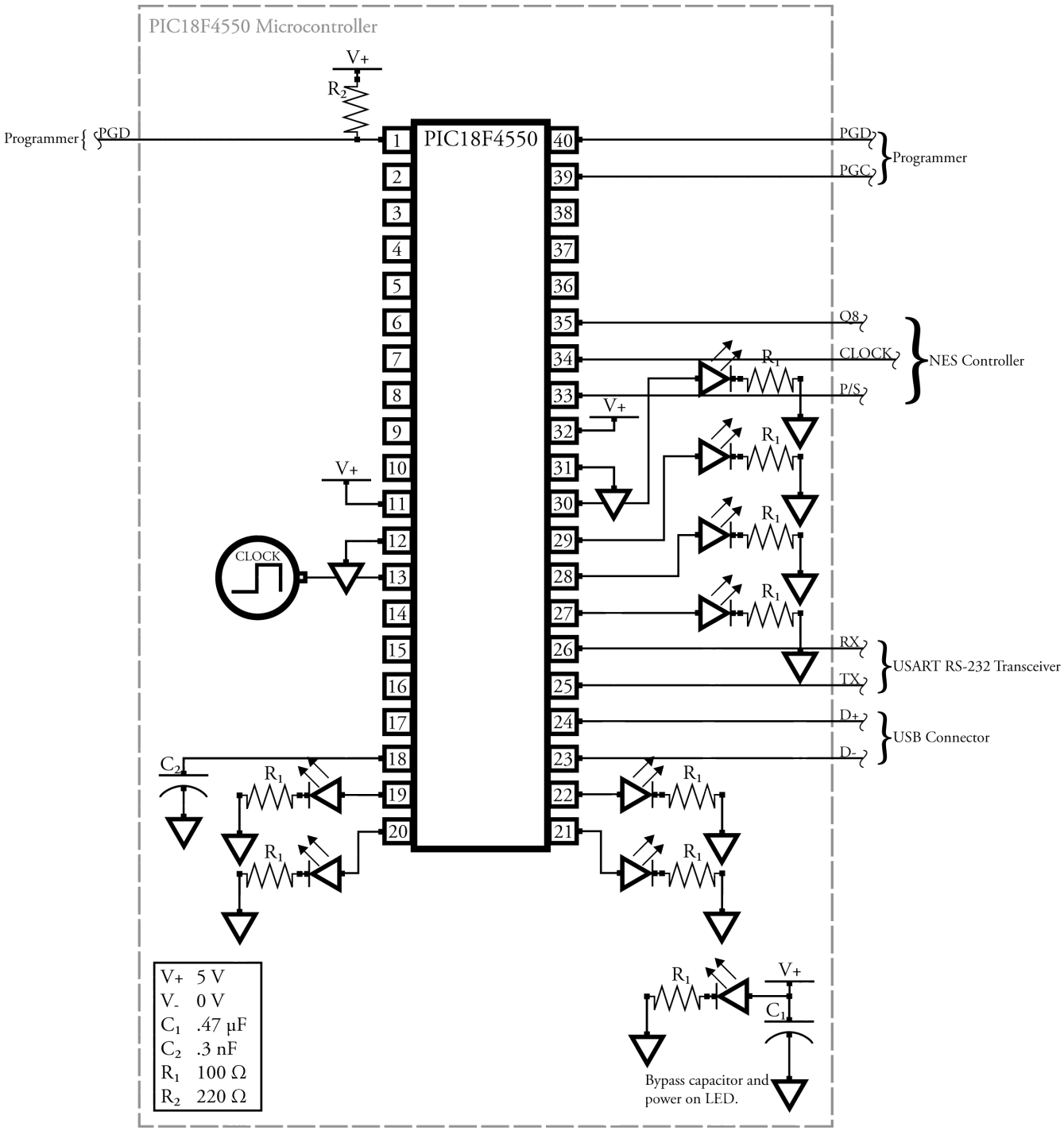
I did not make the decision to port my kernel lightly. The pursuit of a USB application led to a

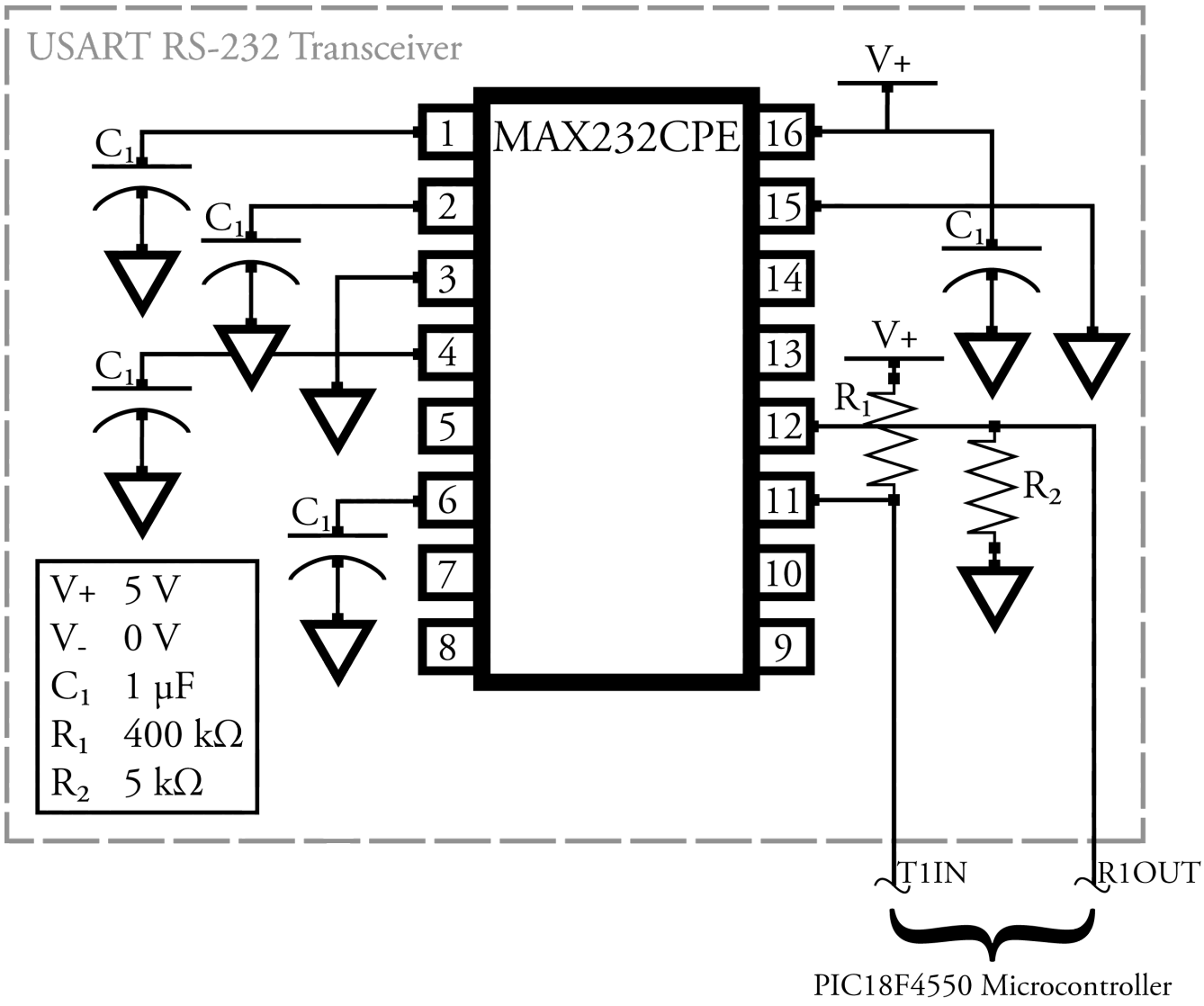
slew of difficulties that primarily centered on whether or not the MCF52233DEMO was capable of USB 2.0 full speed. Some time later, I was still inconclusive about whether the MCF52233DEMO was capable of USB 2.0 full speed, but at any rate, it was lacking an important USB component known as a serial interface engine (SIE). As it seemed that constructing a SIE was beyond the scope of the semester and that emulating it in software, if possible, would lead to an RTOS devoted to just servicing USB, and for other hardware reasons, I chose to port the kernel to Microchip's PIC18F4550.

Overall, the port worked out great. Unfortunately, it took longer than expected, as I had some difficulties with the architectural differences between the 4550 and the 52235. Of the challenges faced when porting, three seem foremost to me: the 4550 has one accumulator, which the compiler developer made do with by creating many global variables that also need to be saved during a context switch; the 4550 has both a software and hardware stack, in addition, I was never able to get the hardware stack viewer working, making debugging rather difficult; the best way to structure cross-platform source.

PIC18F4550 Hardware Setup

As the hardware part of the project was also a burden, I document it. Yes, I know, put a little EE into EEL 270? Blasphemous! I included an RS-232 transceiver for `printf` capabilities to HyperTerminal. RS-232 helped at times for debugging, but other times created problems due to the overhead of printf. Also, the C_2 cap in PIC18F4550 microcontroller schematic is necessary.
The USB eye test feature of the PIC also helped for debugging some of the early hardware issues.


PIC18F4550 Microcontroller
PIC18F4550
Programmer
PGD
PGC
NES Controller
Q8
CLOCK
P/S
USART RS-232 Transceiver
RX
TX
USB Connector
D+
D-
V+
R2
R1
C2
C1
CLOCK
V+ 5 V
V- 0 V
C1 .47 µF
C2 .3 nF
R1 100 Ω
R2 220 Ω
Bypass capacitor and power on LED.



* * *

USB Implementation

After completing assignment three, we were told that our TA, that's teaching assistant, had abandoned us and that we should pursue things on our own. Specifically, to pursue additional kernel features and application development At the beginning of the course I talked to said TA about pursing a USB 2.0 full speed slave application, which he said was a good idea. So I did. I would later find out that, apparently, USB much like porting a kernel, is not a good idea for EEL 270 where students are left to their own devices. So, in completing the port of my kernel and the hardware necessary for its operation, I pursued my reason for doing so, USB.

An important part of implementing USB was the tools I used to debug, SnoopyPro, Total Phase Beagle Protocol Analyzer, and printf. Each tool had its own pros and cons. SnoopyPro was free and worked well until it started blue screening my computer, the Beagle worked well to capture mouse USB setup packet data, and catches some of the packets in my implementation, and printf works well in sections of code where timing isn't an issue.

The most helpful thing I had for implementing USB was a variety of examples. Particularly, Microchip has a very good example available on their website, and so does Olin colleges Principles of Engineering course (suggested by Chris).

USB Co-requisite and Additional Documentation

Universal Serial Bus Revision 2.0 Specification
http://www.usb.org/developers/docs/usb_20_040907.zip

Universal Serial Bus Class Definitions for Communication Devices
http://www.usb.org/developers/devclass_docs/usbcdc11.pdf

Language Identifiers
http://www.usb.org/developers/docs/USB_LANGIDs.pdf

USB Complete: Everything You Need to Develop Custom USB Peripherals by Jan Axelson.

Migrating Applications to USB from RS-232 UART with Minimal Impact on PC Software.
http://ww1.microchip.com/downloads/en/AppNotes/00956b.pdf

Microchip PICDEM FS USB Demonstration Board User's Guide.
http://ww1.microchip.com/downloads/en/devicedoc/51526a.pdf

USB in a Nutshell: Making Sense of the USB Standard by Craig Peacock.
http://www.beyondlogic.org/usbnutshell/usb-in-a-nutshell.pdf

USB Mass Storage Device Using a PIC MCU.
http://ww1.microchip.com/downloads/en/AppNotes/01003a.pdf

USB Design by Example: A Practical Guide to Building I/O Devices by John Hyde.

USB Mass Storage: Designing and Programming Devices and Embedded Hosts by Jan Axelson.