PERSISTOR® CF2
Programmer's Manual

Release 1.3 – September 2008

Contents

WELCOME TO THE CF2 4

FIRST THINGS FIRST 4
INTENT 4
SLIGHT WARNING 4

GENERAL CF2 PROGRAMMING ISSUES 5

WHAT YOU NEED TO KNOW ABOUT C PROGRAMMING 5
RECOMMENDED READING FOR NEW C PROGRAMMERS 6
ABOUT THE STANDARD ANSI C LIBRARIES 6
THE CF2 OPERATING SYSTEM 7
What is PicoDOS 7
What is the PBM? 7
What is MyPico 8
What is BIOS 8
Building and Running executables on the CF2 9
Using Batch Files 9
USING CAPTURE 10
THE MSP430 – WHAT DOES IT DO? DO YOU CARE? 12
UPDATING THE CF2 TO NEW SOFTWARE 12
PicoDOS on the CF2 12
Library/Linker/Motocross updates for your PC 12

PROGRAMMING THE CF2 HARDWARE 13

HOW TO START A CF2 PROGRAM 13
PROGRAM CONSIDERATIONS 13
Using hardware interrupts 13
Using the Periodic Interrupt Timer 14
USING LOW POWER MODES 15
Fundamental considerations 15
LPStop 16
Suspend Mode 16
SYSTEM RESOURCES 18
Using the Virtual EEPROM 18
VEEPROM Example 19
Using the Real Time Clock Functions 19
WRITING AND READING FILES TO COMPACTFLASH 21
USING PICOZOOM 21
How cool is PicoZOOM? 21
How it works 21
The dangers of PicoZOOM. 22
Using PicoZoom 22
USING A PING-PONG BUFFER 22
CONTROLLING SERIAL COMMUNICATIONS 24
SCI and Console Communications 24
THE TIME PROCESSOR UNIT (TPU) 25
The TPU as software UARTs 25
The TPU as I/O 26
Simple I/O 26

Other TPU Functions *27*
THE LEDS 27
QUEUED SPI (QSPI DEVICES) 29
PROGRAMMING FOR SERIAL A/D AND RECIPECARDS 32
Simple A/D *32*
Multiple Channels (not as simple) *33*
Buffers and Rings (a little more complicated) *35*
Adding a New A/D Converter *37*
PROGRAMMING FOR CF2 ADD ON BOARDS 38
CF2 MEMORY MAP 39

USING THE TABLE DRIVEN COMMAND PROCESSOR 40

OVERVIEW 40
CREATING A TOPICO PROJECT 40
THE COMMAND TABLE 40
ADDING A COMMAND 42
HANDLING SWITCHES 43
SFORMAT 43
THE REST OF THE COMMAND TABLE 44
COMMAND PROCESSOR WITHOUT TOPICO 45

PICODAQ 49

COMMON PROGRAMMING QUESTIONS 53

Welcome to the CF2

Welcome to programming the CF2! Many man-years of effort have gone into producing the CF2 and making it powerful and flexible. We have made every effort to develop a rich set of powerful functions for use in your applications. It is our hope that you agree with this assessment and that you find your experience with the CF2 to be pleasant and rewarding.

First things first

The first thing you need to do before reading any further in this manual is to read the Getting Started Guide (GSG) included with your CF2. Most of the issues covered by the GSG will not be duplicated in this document.

The Getting Started Guide (GSG) covers a range of beginning issues. These include the installation of software: CodeWarrior and PicoDev. It will also walk you through using Motocross and creating your first project.

Once you have been through the Getting Started Guide, you can return to this manual.

Intent

The intent of this manual is to provide guidance for people who are programming the CF2. While it is impractical for us to provide you with an all inclusive-manual, we will walk you through the use of many of the features of the CF2.

If you are an experienced programmer or you are already familiar with development for Persistor products and are only looking for a list of commonly used PicoDOS functions and their descriptions, we would refer you to the CF2 API Reference. There you will find what you need. But even if you are an experienced programmer it is in your best interest to read on.

Slight Warning

Programming in C for the Persistor is very different from desktop coding. First, you're coding programs on one machine that are meant to work on a completely different machine. This means debugging with printf statements and an oscilloscope which is much more tedious than simply dropping into a friendly source level debugger and stepping through the C instructions. Second, it's extremely unlikely you can get away with ignoring the hardware and achieve the goals that prompted you to buy a Persistor.

General CF2 Programming Issues

What you need to know about C programming

C programming can be fun and rewarding or a complete and utter drag. Experienced programmers know this is the truth and may move on to the next section. Anyone who is relatively new to C programming should be afraid…and read on.

There are many problems with programming embedded systems in C. First and foremost is the pointer. If you are new to programming in C then this might sound like something to ignore but **IT IS NOT!** Pointers are a very useful part of programming in C (and in other languages where they are usually called something else). Wild use of pointers in C can cause catastrophic failures. Many CF2 systems are deployed in long-term data logging operations. A crash in the 'field' because a pointer references an area of memory it shouldn't can be fatal to data collection efforts.

Be careful when designing software using pointers. Pay special attention to limits or boundaries of data. A pointer that has 'run amuck' can wipe out whole areas of memory it should normally never touch. When this happens, erratic system behavior and crashes can occur. The source of these crashes becomes very difficult to track down because the symptoms, more often than not, bare no resemblance to the 'disease'.

Most PC programmers assume that in the event of a crash, the user can always re-boot. This may not be intentional, that is, programmers don't **try** to write bad software. But a RESET button is a nice feature when 'all else fails'. RESET may save you when your PC locks up but the ability to press a RESET button is difficult to impossible when the computer may be located at the bottom of the ocean, hanging on an oil pipeline on the frozen Tundra, or suspended from a high-altitude balloon.

Here are (just a few) tips for programming embedded systems:

- Deploying a system that has been coded 'on the dock' before the ship sails is like sending a two year-old out to direct traffic. Software (and hardware) needs to be tested. Test your systems before they go out into the world. This testing should include varying inputs to A/D channels (testing extreme values against your algorithms), temperature testing, vibration, shock and battery 'sag'.
- While testing is essential before deploying any system nothing beats a design that has been thought-through before coding. Patching problems is a common practice in software development but usually indicates problems in design. Too many patches or 'hacks' is a strong indicator that it may be time to re-evaluate the approach.
- All but the simplest applications contain bugs. The longer you test, the less bugs. The shorter the testing period, the greater the likelihood that you will deploy a system that will never be heard from again.
- Make sure that, if you use pointers, you know what your pointers are pointing at. If a pointer is invalid (pointing at nothing or something that it shouldn't be pointing at) then writing to it will clobber some other variable.
- Testing for the boundaries of arrays (for example) can save your system in the field. The computation of indexes and offsets are common in algorithms but make sure you understand the domain and range of those calculations. For example, let's say you declare an array of 10 integers in C like this:

int somearray[10];

You would access the value using somearray[0] to somearray[9]. If you tried to write to somearray[32] C would not complain, even though this element is not in the original declaration. Writing to this element would surely corrupt some other variable with potentially catastrophic results.

- Avoid allocating and de-allocating tiny blocks of HEAP. Memory can easily become fragmented and memory leaks can eventually kill an embedded application.
- Use static storage when you can. An example is that, if you periodically need 256 bytes of memory for string manipulation don't allocate it on the fly every time with alloc or calloc; instead, use a static array of characters. This memory will always be where it is supposed to be and because you know its size you can test for cases where you might overrun the array.
- Use assert statements during debugging to validate values and ranges of values you pass to your functions. It's a great (and easy) way to find places where typos may wreak havoc.

Recommended reading for new C programmers

Read the standard C reference by Kernhigan and Ritchie also referred to as K&R or the White Book. There are many (too many) books that have been written about programming in C. If you are new to C, find a book you like (beside K&R…which is a must) to act as a tutorial. Remember that many books talk about programming in C using PCs or other large computers which have lots of memory and other resources which are not usually in abundance in small embedded systems like the CF2. An example program from a book that allocates memory in megabyte blocks will not work on the CF2.

Embedded systems programming is a different world from programming a PC. Keep looking for references and ask others in the field what they like. Eventually you will collect references that YOU like and which work for YOU.

It is our hope that **this** manual provides sufficient information to make your CF2 programming experience rewarding and beneficial.

About the standard ANSI C Libraries

Persistor Instruments has worked hard at providing functions to access the features of the CF2. This library of functions is buried within PicoDOS and the BIOS of the CF2. Our functions are not all that is needed to program the CF2 in C. The standard ANSI C libraries are available to you in your application and are provided as a part of CodeWarrior. We do not attempt to document the ANSI C libraries but instead refer you to references such as K&R (White Book).

The CF2 Operating System

What is PicoDOS

PicoDOS refers to the collection of program libraries for CF2 program development. It also refers to the 'DOS-like' environment that you use to communicate with the CF2. When you call functions from your program to sample A/Ds, write data to files or use time functions you are using PicoDOS.

We usually refer to the PicoDOS prompt and when we do this means the prompt you get from the CF2 when communicating using Motocross. The prompt is usually:

C:\>

which is reminiscent of the old PC DOS prompt. Using PicoDOS you can execute programs, get information about files on CompactFlash, display or modify the time and many other commands. A list of each of the commands and a short description can be found in the table below. This list can also be displayed from PicoDOS by typing HELP or ?.

```
=========  PicoDOS built-in commands  (plus .PXE and .BAT Files)  =========
APP          run flash app [args...]   ATTRIB       [+ - RASH] [d:][p][name]
BACKROM      [d:][path] [/SAVPI]       BAUD         [newrate] [/Q/P/E/O/N/2]
BOOT         [PICO][PBM][APP]          CAPTURE      [d:][p]fn [/Dx/B/N/E]
CCC          Card Change [delay secs]  CHDIR        [drive:][path]
CHKDSK       [d:][p][fn] [/F][/I]      COPY         source dest [/V]
DUMP         file[start[,end]]         DATE         [mdy[hms[a|p]]] /IEUMCP]
DEL          [drv:][pth][name] [/P]    DIR          [d:][p][fn] [/PWBLV4A:a]
ERASE        [drv:][pth][name] [/P]    FDISK        [/Pnn/M/Sdev/@/F/Rnn/Q]
FORMAT       [drv:][/V[:label]][/Q/E]  GO           args... | addr /A | /Fn
LO           [ofs][;Bx[+]] [;G]        MOUNT        [V:][DEV[-n]][/D/P/N/V/Q]
MKDIR        [drive:][path]            MD           display [range]
MM           modify [address]          ML           disassemble [range]
MR           Memory Read      MR[.bwl] MW           Memory Write     MW[.bwl]
MON          Reset to PBM              PATH         [[d:]path[;...]] [/P]
PBM          Reset to PBM              PROMPT       [text] [/P]
PR           pin read    <1..50>       PC           pin clear   <1..50>
PS           pin set     <1..50>       PT           pin toggle  <1..50>
PM           pin mirror  <1..50>       TIME         [hh:mm:ss [a|p]] [/M/C]
TYPE         [drv:][pth][name]         REN          [d:][p]oldname newname
RESET        (hard reset)              RMDIR        [drive:][path]
SAVE         file[start][end]          SD           sect.dump[d:][range][/C]
SET          [var=[str]] [/SLFE?]      XS           [/X][/C][/Q]file
XR           [/X][/C][/Q][file]        YS           [/G][/Q]file[,file...]
YR           [/G][/Q]                  VER          Firmware versions [/I]
```

What is the PBM?

The Persistor Boot Monitor (PBM) program resides in the bottom 16KB of Flash memory and quietly takes care of running PicoDOS or your application program when the CF2 starts up. Most users will seldom need to interact directly with PBM, and will instead let the more capable PicoDOS handle changes to startup and program execution tasks. Unless you are installing custom software or operating systems that bypass PicoDOS, you can safely skip this entire section.

A tiny portion of PBM is the first bit of software to execute from a power-up or reset. Its primary job is to figure out what program should take over at reset and start it running. Failing that, it performs its secondary job as a monitor program to load and test new programs into the CF2. PBM is a minimalist monitor that's meant to let you load and run more sophisticated software. It accepts standard Motorola S records and MotoCross binary load files which can be stored to RAM or Flash memory, and with the help of PBM, configured to automatically run at startup. PBM also provides a limited set of commands to examine memory locations to help in debugging loads that do not work correctly.

PBM is entirely self-contained and can accomplish all of its maintenance tasks with no references to external driver software. Its terse and unforgiving command set reflects the size limitations and the infrequent need to use a boot monitor. In contrast, software like PicoDOS is generally resident to provide more elegant services.

The PBM can be called directly from the PicoDOS prompt by typing 'PBM'. Here is a list of the commands:

PBM Command	Description
BOOT	Change CF2 boot options. You can boot an application at one of the possible application addresses or you can boot PBM or PicoDOS.
CLEAR	Clear the Flash memory that holds PicoDOS. WARNING: Don't use this unless you are instructed to by Persistor Technical support.
BAUD	Modify the baud rate. This is only used to modify the baud rate during program loads. To change the baud rate permanently, see BAUD under PicoDOS.
CHECK	Checks which areas of Flash memory are erased.
ERR	Displays the last error encountered or None.
G	Go (run a program that has been loaded)
LOAD or LO	Used to load application files. You will normally never use this command as Motocross will perform all the necessary steps for you when you load a file.
MD	Memory display. Used to examine system memory.
WREN	Write Enable the Flash. WARNING: Don't use this unless you are instructed to by Persistor Technical support.
RES	Reset the processor. Causes a hardware Reset of the 68332.
HELP or ?	Displays this list (just the list, not the descriptions).
PICO	Run PicoDOS (back to PicoDOS)

What is MyPico

MyPico is a starter project or Stationery for Metrowerks CodeWarrior that will let you build a custom version of PicoDOS. When you create the project, code for sample commands are added that you can use as 'starters' when creating your own commands.

What is BIOS

BIOS is an acronym for Basic Input-Output System. The BIOS is a collection of low-level functions for the CF2. The BIOS is the lowest level of software for the CF2. PicoDOS is built on top of the BIOS

Building and Running executables on the CF2

SEE CF2 GETTING STARTED GUIDE

Using Batch Files

PicoDOS supports batch files running from the CompactFlash cards. You can use batch files to display information to a user and optionally launch applications in Flash or run applications stored on the CompactFlash much like the way DOS used to work in older PCs.

PicoDOS supports a limited subset of equivalent DOS batch commands. It also has a few non-standard batch commands. These commands are listed below along with a list of DOS batch commands that are NOT supported on the CF2:

Supported standard batch commands:

ECHO	Displays the text following the ECHO command and controls local echo of commands.
GOTO	Jump to a label in the batch file
IF	Conditional
PAUSE	Halt batch file execution (continue with a key press)
REM	A remark or non-executable notes to the author
SHIFT	Shift input arguments

Supported non-standard batch commands:

GETC	Get a character
KBHIT	Detect when a character is struck on the terminal keyboard

The following are unsupported standard batch commands:

BREAK
CLS
DEVICE
EXIT
INSTALL
LABEL
MEM
MODE
VOL

An example of a batch file would be a text file on the CompactFlash card named AUTOEXEC.BAT. If a file with this special name is present, PicoDOS will open and attempt to execute the commands on power up. One of the things that could be done is to run an executable program stored on the CompactFlash. If, for example, we had an executable program stored call MYPROGRAM.PXE we could execute it at startup by typing this one line in the AUTOEXEC.BAT text file:

MYPROGRAM

USING CAPTURE

Batch files need to be text files stored on the CompactFlash card. One method to create these files is to edit them on a PC using a text editor and transfer them to the CompactFlash via a CompactFlash reader. A possible more convenient way is using the CAPTURE command in PicoDOS. Edit the text of the Batch file on a PC using the text editor of your choice and paste it into a file on the CompactFlash card. You do this at the PicoDOS prompt by type CAPTURE <filename>. PicoDOS will read all text either typed from the keyboard or pasted from the clipboard. When all the text has been typed or pasted, enter a Control-C to terminate the command.

Examples

Echo is a batch command that is used to turn on and off the echoing to the terminal window of executed commands from the batch file. Here is an example of how it works:

```
@echo off
echo You should see the REM VISIBLE lines, but not the REM INVISIBLE lines.
echo on
REM VISIBLE
@echo off
REM INVISIBLE
echo That was the first echo test.
echo on
REM VISIBLE
@REM INVISIBLE
@echo off
echo That's the end of the echo tests.
```

The @ sign prevents the command it precedes from being echoed to the screen.

The following .BAT example will delete a file on the CompactFlash if the file exists. The leading @ symbol force the ‘if’ text to not be displayed. This is a good command to have in an AUTOEXEC.BAT when you have a temporary file that you want to delete before a program starts.

```
@if exist data.txt del data.txt
```

Here is an example of using a menu of sorts to prompt someone over the serial interface to enter a digit 0 to 3. The character typed is echoed to the screen however, these statements could easily be replaced with other batch commands or the names of executable code stored on the CompactFlash card.

```
@echo off
:getcagain
echo Hit a digit key (0-3) or period to end
getc
if not errorlevel 47 goto getcdone
if errorlevel 52 goto getcagain
if not errorlevel 51 goto try2
echo you keyed a '3'
goto getcagain
:try2
if not errorlevel 50 goto try1
```

```
echo you keyed a '2'
goto getcagain
:try1
if not errorlevel 49 goto try0
echo you keyed a '1'
goto getcagain
:try0
if not errorlevel 48 goto try7
echo you keyed a '0'
goto getcagain
:getcdone
echo That's all folks!
echo
```

Here is the last example demonstrating the use of the if command in a Batch file:

```
echo You should see the message "thistext==THISTEXT"
if thistext==THISTEXT echo "thistext==THISTEXT"
echo You should not see the message "thistext==THATTEXT"
if thistext==THATTEXT echo "thistext==THATTEXT"
set testvar1=THISTEXT
set testvar2=THATTEXT
echo You should see the message "thistext==THISTEXT"
if thistext==%testvar1% echo "thistext==THISTEXT"
echo You should not see the message "thistext==THATTEXT"
if thistext==%testvar2% echo "thistext==THATTEXT"
```

The MSP430 – What does it do? Do you care?

On board the CF2 is a Texas Instruments MSP430 microcontroller. The MSP430 performs several key functions:

1. Real-Time Clock – The MSP430 keeps track of time for the CF2. The 68332 keeps track of time while it is running but the MSP430 keeps track of it when the CF2 power is off.

One of the things that gets initialized when the 68332 is powered up is time and the MSP430 provides it. The MSP430 can maintain time while consuming less than 10 uA of current from VBBK.

2. Alarm Clock and Low Power Control/Supervisor – The MSP430 can be programmed to wake up the 68332 like an alarm clock. Using the PwrSuspendSecs function a user program can command the MSP430 to power the 68332 off and wake it at a specific time. This is the lowest of the CF2's low power modes.

3. Compact Flash Monitor – The MSP430 determines when a Compact Flash card is present and delivers this information to the firmware in the CF2.

The MSP430 quietly performs these and other functions in the CF2. Any of its interactions with the CF2 should be transparent to the user.

Updates to the MSP430 firmware are rare, however, there is a mechanism to update the software should a need arise. Any updates to the firmware will likely be distributed along with updates to PicoDOS.

Updating the CF2 to new Software

PicoDOS on the CF2

New versions of PicoDOS may be released from time to time. The form of this update for the CF2 is an APP file: a Flash application (the extension will be .APP). Instructions for updating PicoDOS are distributed with the update. Be sure to read them carefully before attempting to install the update. Release notes are usually distributed with PicoDOS updates. Make sure you read them too because they often contain information about the update and what (and who) it is for.

Library/Linker/Motocross updates for your PC

Normally a distribution disk is sent to the customer when there is an update to PicoDOS. The disk will contain an installer which will automatically load and configure the software on the PC. Follow the instructions included with the disk.

Programming the CF2 hardware

How to start a CF2 Program

See the Getting Started Guide

Program Considerations

Using hardware interrupts

Interrupts are one of the more difficult issues in an application. Interrupts in the 68332 are level sensitive. Before an external signal (presumably from commercial or custom circuitry) asserts an interrupt, the 68332 must have been prepared by the addition of a vector. Which vector you choose depends on which pin you connect to.

There are three possible EXTERNAL interrupts available to the user on the CF2: IRQ2 (pin 41), IRQ5 (pin 39) and IRQ7 (pin 40). The only general purpose IRQ you should use is IRQ2.

IRQ5 is also associated with forcing PBM mode at start. If you use IRQ5 and it can be low at start then you will **ALWAYS** be forced into PBM mode and your program **WILL NOT RUN**.

IRQ7 is non-maskable. If IRQ7 is low at startup then your program **WILL NOT RUN**.

To use an interrupt, for example, IRQ2, you add an interrupt service routine or handler. A function prototype is available for this. You would write something like this:

```
IEV_C_PROTO(IRQHandler);
IEV_C_FUNCT(IRQHandler)
{
        #pragma unused(ievstack)

        //Do something

        //IMPORTANT!!! Clear the source of the interrupt
}
```

To add the handler to the code you have to install the handler like this:

```
IEVInsertCFunct( &IRQHandler, level2InterruptAutovector);
```

The last thing you want to do is change the pin for the interrupt from its default simple I/O configuration to a bus pin. In fact, changing the pin from I/O to bus and back to I/O is possible. Changing the pin back to I/O effectively disables the interrupt. You can always switch back quickly if you need to by changing the pin back to a bus configuration. Here is an example:

```
PIOBusFunct(IRQ2);      //IRQ2 now responds to interrupts.

PIORead(IRQ2)           //IRQ2 is I/O and interrupt function is disabled.
```

Things to Remember

When an interrupt is asserted by external hardware to the CF2 the CPU will respond by saving its context and loading the program counter with the address in the vector table for the interrupt asserted. Lower level interrupts are masked to prevent them from pre-empting your interrupt. When your interrupt service routine has completed and returns **IF THE INTERRUPT IS STILL PRESENT**

then the interrupt service routine **WILL BE CALLED AGAIN**. You **MUST** make sure that you have a guaranteed way in the interrupt service routine to clear the external source of the low signal on the interrupt line!

It is very important to remember that interrupt handler should be very quick. Clear the interrupt and set a flag so the main code can handle any lengthy operations and GET OUT. Keeping interrupt handler code small, quick and to the point will help minimize impact on main program execution time.

Spurious Interrupts

You code must handle spurious interrupts. The CF2 interrupts are level sensitive. If a signal on an interrupt is momentarily low the CPU may call interrupt processing only to discover that no interrupts are low. When this happens (and it usually does) the spurious interrupt handler is called. If you do not have a handler in place for this then a **CRASH IS LIKELY**.

Using the Periodic Interrupt Timer

The Periodic Interrupt Timer on the CF2 can be programmed to run from one of two clock sources. One source can provide a basic clock period of 100 microseconds and the other 51 milliseconds. The PIT can be set up to divide this count by a value in an 8 bit register. So the basic timing interval can be between 100 microseconds and 25.5 milliseconds or 51 milliseconds and 13.005 seconds.

To use the PIT you initialize the PIT and interrupt level (default is 6). Next you set the rate for one of the two basic timing modes (15ms or 100 us). The header files have pre-defined values for the various rates.

Next, you add a chore to the PIT. The chore is a function that is neither passed a value or one that returns a value. A total of 8 chores can be added. When the PIT interval reaches its target value the chores are each called by the PIT management software.

Example:

```
//Your chore declaration and code

void MyChore(void);
void MyChore(void)
{
}

//Then in your main() code…

        PITInit(6);     //At priority 6
        PITSet100usPeriod(PIT100Hz);
        PITAddChore(MyChore,6);

        while(!SCIRxBreak(100))
        ;

        PITSet100usPeriod(PITOff);
        PITRemoveChore(MyChore);
…
```

The chore function above will be called at a rate 100 times per second until a BREAK is sent over the serial interface.

Using Low Power Modes

One of the fundamental features of the CF2 that sets it apart from other embedded solutions are the low power modes of operation that it supports. There are several different low power modes available each with its own advantages and disadvantages. Low power modes can be separated into two groups: CPU power on and CPU power off.

The lowest power consumption mode of the CF2 is Suspend Mode. Suspend Mode is when the 68332 CPU is powered off under the control of the MSP430 Real Time Clock. The CF2 will consume less than 10 microamperes of current while in Suspend Mode.

Fundamental considerations

CLOCK SPEED - CMOS devices like the 68332 in the CF2 consume power really only when they switch logic states. This means that the time spent traversing from high to low or low to high is when significant power is consumed. So, the faster these devices are clocked the more power is consumed.

The 68332 in the CF2 by default is running at a clock rate of 16 MHz. This clock is generated by a PLL driven by a 40 KHz clock. This allows the frequency of the clock to be determined by software. The PicoDOS function for adjusting the speed of the clock is called TMGSetSpeed. The argument that is passed for TMGSetSpeed is a 16 bit (short) representing the new clock speed in Kilohertz. If your application does not need the high clock speed then simply slow it down. You will have to experiment to determine the correct speed for your application.

PERIPHERALS – Another power saving idea involves peripherals. Many peripherals in the CF2 can be disabled if not needed or not needed all the time. For example, if no serial I/O will be needed while your application is running then you can turn off the MAX3222 driver:

```
EIAForceOff(true);//    Force RS232 transmitters off
EIAEnableRx(false);//   Enable RS232 receivers
```

Other peripherals are internal subsystems of the 68332. Many of these can be turned off as well. Here is a list from the LPSTOP example:

PITSet100usPeriod(PITOff)	Turns the PIT off
PITSet51msPeriod(PITOff)	Turns the PIT off
TPURun(false)	Turns off the TPU (ALL TPU FUNCTIONS)
EIAForceOff(true)	MAX3222 driver
QSMStop()	Queued Serial Module

LPStop

Another way to save power is to use LPStop. LPStop will terminate execution of instructions. An interrupt can wake the CF2 from LPStop. When executing the LPStop you specify what needs to be running.

```
// LPStopCSE that modifies SYNCR with bits specifying:
//      bit 6 (0x40) = 1 is just CPU off, other submodules clocked - 68338 only
//      bit 1 (0x02) = 1 is VCO running and driving SIMCLK
//      bit 0 (0x01) = 1 is external clock driven as determined by STSIM

//              lowest power            fast IRQ response       submodules running
enum {      FullStop = 0x00,    FastStop = 0x02,        CPUStop = 0x42 };
```

An example of using LPStop would be to use the PIT to make the CF2 execute a periodic chore. When the PIT determines that the chore needs to be executed the PIT interrupt will force the CF2 out of LPStop. Assuming that a chore has been setup to be called at a given rate here is a sample of how to use LPStop:

```
LPStopCSE(FastStop);                    // we're here until interrupted
```

To really save power you can turn off things you are not using while stopped and start them back up when the LPStop is interrupted. In the examples supplied with the Persistor CF2 development environment there is a great example of using LPStop for very low power operation. Look under the examples for a file names lpstop.c.

Suspend Mode

The lowest power consumption in a CF2 is obtained when the CF2 is powered off under the control of the MSP430 Real Time Clock. While suspended, the current to the CF2 can be as low as 10 uA.

PWRSuspendSecs is the function used to initiate Suspend Mode. The first argument for PWRSuspendSecs is the seconds to suspend as a long value. The next argument is a Boolean value which determines if the program execution should continue after the PWRSuspendSecs function call or if the CF2 should be RESET. The last argument is an enumerated value which states what events can cause the MSP430 to wake the CF2. The possible values are:

WakeOnTimeout	Wake on programmed time in seconds.
WakeTmtOrWAKEFall	Wake on programmed time in seconds OR on the falling edge of the WAKE pin.
WakeTmtOrCFChg	Wake on programmed time in seconds OR on indication of a change on the CompactFlash slot (card inserted or removed).
WakeTmtWAKECFChg	Wake on programmed time in seconds OR on the falling edge of the WAKE pin OR on indication of a change on the CompactFlash slot (card inserted or removed).

Below is an example of suspending the CF2 and determining what woke it.

```
cprintf("\n\nSleeping...\n\n");
switch(what = PWRSuspendSecs(10L,       true, WakeTmtOrWAKEFall))
{
```

```
        case WokeFromTimeout:
                cprintf("\nWokeFromTimeout\n");
        break;
        case WokeTmtOrWAKEFall:
                cprintf("\nWokeTmtOrWAKEFall\n");
        break;
        case WokeTmtOrCFChg:
                cprintf("\nWokeTmtOrCFChg\n");
        break;
        case WokeTmtWAKECFChg:
                cprintf("\nWokeTmtWAKECFChg\n");
        break;
        case WokeNeverSuspending:
                cprintf("\nWokeNeverSuspending\n");
        break;
}

cprintf("\n\nAwake!\n\n");
```

System Resources

Using the Virtual EEPROM

VEE stands for Virtual EEPROM. It's 7.5KB of persistent storage that can be used to hold small amounts of any kind of binary or text data (usually configuration data). This data is stored in Flash memory and therefore stays with the CF2 which eliminates the need to save similar data in files on the Compact Flash.

The 'virtual' in Virtual EEPROM is used in the context of the familiar small serial EEPROM chips that are often associated with little embedded controllers. The EEPROM part of the name is electrically erasable read-only memory, albeit two reserved 8KB sectors of the big flash array.

The VEEPROM can store three types of standard data (longs, floats, and C strings) as well as arbitrary binary data for custom types and structures. Each entry is stored along with a 16 bit CRC, and the entire VEEPROM is verified when PicoDOS initializes. Each fetch of a VEE entry also re-verifies the CRC.

VEEPROM lookup is considerable slower than conventional memory accesses, so a wise strategy for frequently accessed data is to fetch it once at the start of your program and keep a local copy in RAM for repeat access.

PicoDOS manages the 7.5KB VEEPROM data efficiently by minimizing the writes and erases performed by using a pair of 8KB flash memory sectors (the missing 512 bytes (x2) is reserved by PBM for controlling the boot process).

From your programs, you use API calls like VEEFetch(), VEEStore(), and VEEDelete() to get data into and out of the virtual EEPROM, and from the PicoDOS command shell you can access the string-type VEEPROM data with the DOS-like SET command.

A quick example helps shed light on how and why VEEPROM is useful: by default, the CF2 performs all of its communications at 9600 BAUD, though it's easily capable of talking at any rate from 50 to 230,000 BAUD. This behavior is used so that you can hook the CF2 to any 9600 BAUD terminal program and be assured that if the CF2 speaks, you'll be able to hear it. Very nice, unless your application embeds the CF2 in a system where everything else communicates at 38,400 BAUD. You can of course write your CF2 program to change the baud rate as one of its first tasks, but it's sometime tedious and error prone to have to switch baud rates for some reason such as maintenance.

With PicoDOS, you can tell the CF2 to use a custom baud rate using the agreed upon VEEPROM variable called "SYS.BAUD" (though PBM still uses 9600 baud for emergencies). To set this VEEPROM variable from PicoDOS you would simply type "SET SYS.BAUD=38400", or alternatively, from within your program with the call to VEEStore("SYS.BAUD", "38400"). To see the current setting, from PicoDOS you could type "SET" (viewing all of the VEE variables), or from within your program with the call VEEFetch("SYS.BAUD").

Though PicoDOS does its best to minimize writes and erases, using VEEPROM does place additional wear and tear on the flash memory. In typical use, such as less than 1KB data in less than 50 variables, you're probably good for about a million VEEPROM changes before the flash fatigues (just the VEEPROM sectors would be fatigued). For its intended purpose, this is indeed overkill. Nevertheless, it is possible to code nasty loops into your applications that could burn through the flash in less than a day. For this reason, the VEEPROM limits sector erasures to 4 per reset (about a

thousand VEEPROM changes). There is also an API function named VEELock() that prohibits any future VEE writes until the next full reset cycle.

VEEPROM Example

Here is an example of reading and writing VEEPROM data.

```
//      Looking for a string stored in VEEPROM called MYSTUFF.STR
        printf("VEEFetchStr returned [%s]\n", VEEFetchStr("MYSTUFF.STR", "Fallback string"));

//      Looking for a long integer stored in VEEPROM called MYSTUFF.LNG
        printf("VEEFetchLong returned [%ld]\n",VEEFetchLong("MYSTUFF.LNG", 123L));

//      Looking for a floating point number stored in VEEPROM called MYSTUFF.FLT
        printf("VEEFetchFloat returned [%f]\n",VEEFetchFloat("MYSTUFF.FLT", (float) 2.718281828));

//      Store a string back in the VEEPROM
        if(VEEStoreStr("MYSTUFF.STR","HELLO DOLLY!"))
                printf("String variable stored!\n");
        else
                printf("Error storing string variable!\n");


//      Store a long integer in the VEEPROM
        lvar = 1234567890L;
        if(VEEStoreLong("MYSTUFF.LNG", lvar))
                printf("Long variable stored!\n");
        else
                printf("Error storing long variable!\n");

//      Store a floating point value in the VEEPROM
        fvar = (float) 3.141592654;
        if(VEEStoreFloat("MYSTUFF.FLT", fvar))
                printf("Float variable stored!\n");
        else
                printf("Error storing float variable!\n");

//      See what we get when we try to read them back now.
        printf("VEEFetchStr returned [%s]\n", VEEFetchStr("MYSTUFF.STR", "Fallback string"));
        printf("VEEFetchLong returned [%ld]\n",VEEFetchLong("MYSTUFF.LNG", 123L));
        printf("VEEFetchFloat returned [%f]\n",VEEFetchFloat("MYSTUFF.FLT", (float) 2.718281828));
```

Using the Real Time Clock Functions

This section describes some of the uses of the Real Time Clock or RTC functions. The RTC functions are implemented using a combination of the TPU and the RTC.

Time

Time is kept in the CF2 the same way it is in most computer systems: as a number of seconds since a particular moment in time. The CF2 uses the Unix Epoch of Midnight January 1, 1970 as this moment. All time references this moment.

To get the time in the CF2 you make a call to RTCGetTime:

RTCGetTime(&secs, &ticks);

where secs is defined as an unsigned long or ulong and ticks is defined as a ushort or unsigned short. When RTCGetTime returns secs and ticks will hold the value of the current time in seconds and ticks which is an interval counter between the seconds and has a range of 0 to 39999.

Another way to call RTCGetTime is to pass zero for both argument and just look at the return value. RTCGetTime returns a ulong representing the time in seconds. Sometimes this is more convenient that passing arguments especially when you do not care about the value of ticks. Here is the call:

RTCGetTime(0,0);

To set the time in your program you call RTCSetTime. You pass to RTCSetTime the current time in seconds and ticks.

RTCSetTime(secs, ticks);

Wasting Time

Often programs need to make use of delays. The CF2 provides RTC functions that can accomplish this. RTCDelayMicroSeconds uses the RTC to implement a hard delay. The value of the delay is a ulong value representing the desired delay in microseconds.

NOTE: There is an alternate to using the RTC functions for delays. The following functions can be used to implement delays WITHOUT using the RTC functions (and therefore, the TPU). These delays use a feature of the CPU called Loop Mode. Here is a list of the functions:

Delay1us()	Delay50us()
Delay2us()	Delay100us()
Delay5us()	Delay200us()
Delay10us()	Delay500us()
Delay20us()	Delay1ms()

Counting Down

The RTC Countdown is used to watch for a specific period of time. These are basically hourglass functions. They operate using a structure of type RTCTimer.

You call a function to initialize them passing it the address of your RTCTimer structure and a target time value in microseconds:

RTCCountdownTimerSetup(&MyRTCTimer, 10000000L); //10 seconds

Your program can now check to see if the 'hourglass' has run out of sand (time has expired) by calling RTCCountdownTimeout passing it the address of your RTCTimer structure. RTCCountdownTimeout returns true when time has run out and false otherwise.

Stopwatch

The CF2 has a function that can be used to time an interval. The so called Stop Watch functions are called in a way similar to the Countdown functions. Begin by initializing the RTCTimer by making a call to RTCElapsedTimerSetup:

RTCElapsedTimerSetup(&MyRTCTimer);

You may now compute the elapsed time by making a call to RTCElapsedTime passing it the address of your RTCTimer variable. RTCElapsedTime returns the elapsed time in microseconds:

Timeus = RTCElapsedTime(&MyRTCTimer);

Because RTCTimer watches time in microseconds using a ulong, then the maximum value that can be measured is 4294967295 microseconds or about 71 minutes.

Writing and Reading Files to CompactFlash

Writing and reading of files to the CompactFlash is performed using standard C file I/O calls. There are however, special considerations when using files, especially when you are trying to write lots of data as fast as you can.

FAT Corruption - If you open a file and write to it continuously without closing the file you run the risk of loosing data. The FAT (File Allocation Table) will not be updated completely until the file is closed. If you remove power before your application can close the file(s) then you run the risk of loosing data.

SPEED ISSUES –CompactFlash cards are achieving higher and higher densities. They are also very fast to write to however, only under special circumstances.

Small writes to the CompactFlash are penalized because of tiny writes to update the FAT. You **can** write fast but only when using large block of data that are multiples of 512. Usually a larger block of data is best e.g. 32K or 64K. Use a buffer (like a Ping-Pong buffer) to write large blocks of data. Speed will also be improved when using PicoZoom (see below).

If you are trying to write to a file quickly then avoid writing using fprintf. It is far faster to write binary data in large blocks using fwrite functions than it is to try and print ASCII values using fprintf. Many people would prefer writing ASCII on the CompactFlash card however, ASCII takes up gobs of extra space and the conversion of data to ASCII also takes up time. If you are converting temperature or some other quantity at a very slow rate (like 10 Hz) then converting to ASCII is fine. If, on the other hand, you are trying to write 8 channels of 16 bit A/D data to CompactFlash at 500 Hz, use PingPong buffers and fwrites.

Example: Slow writes using fprintf:

```
fp = fopen("data.txt","w");
if(fp != NULL)
{
        for(i=0;i<NUMCHANNELS;i++)
        {
                f = (float) adchan[i];
                fprintf(fp,"%d, %u, %f\n", i, adchan[i], f*CONV/2);
        }
}
fclose(fp);
```

Using PicoZOOM

How cool is PicoZOOM?

PicoZoom can speed up the file write times for CompactFlash. Its use is simple but its benefits are substantial. The difference between write speeds with and without PicoZoom, while dependent on CompactFlash card technology, are amazing.

How it works

PicoZoom creates a version of the File Allocation Table in RAM. The FAT, which resides on the CompactFlash, needs to be accessed and written during file writes. The nature of these writes is that

they involve many small changes. CompactFlash cards work best (fastest) when large blocks of data are written. By keeping the FAT in RAM, the small changes needed are significantly sped up. When files have been closed and PicoZoom is no longer needed it can be released and the FAT data in RAM can be written back to the CompactFlash.

The dangers of PicoZOOM.

While the FAT is held in RAM, the CompactFlash and its data are vulnerable. Should power be lost before PicoZoom can be shut down in an orderly way (and FAT data written back to the card) then data loss will result. Because the data was written to the card it may be possible, in some circumstances, to recover the data. It is for this reason that you should provide a way in your application to ensure that files are closed and PicoZoom is shut down properly. Let's face it, you wouldn't just pull the plug on your PC so why would you pull the plug on your CF2 application?

Using PicoZoom

To use PicoZoom call PZCacheSetup:

```
PZCacheSetup('C'-'A', calloc, free);
```

This gives PicoZoom access to our memory allocation functions and sets up the FAT in RAM for the logical drive given.

Now, all subsequent file operation use RAM on the CF2 as the FAT. Data writes actually go to the card but knowledge of it stay in RAM where it is fast.

To close PicoZoom all you need to do is call PZCacheRelease:

```
PZCacheRelease('C'-'A');
```

If the need arises to flush the FAT to the card without closing PicoZoom then call PZCacheFlush:

```
PZCacheFlush('C'-'A');
```

Using a ping-pong buffer

In many logging applications you will need to continuously log data while simultaneously write to CompactFlash. Writing to CompactFlash is fastest when writing large contiguous blocks of data that are binary multiples of 512 bytes (See Writing and Reading Files to CompactFlash). The question that always comes up is: how do I write to a buffer while reading from the buffer at the same time and how can I guarantee I can have access to that data in large contiguous binary chunks? The answer is: use a Ping Pong buffer.

A Ping Pong buffer is one big buffer in memory treated by PicoDOS Ping Pong buffer functions as two buffers side-by-side: one for reading and one for writing. The Ping Pong buffer functions provide a mechanism to identify when one half of the Ping Pong buffer is full. When one half of the Ping Pong buffer is full, you can get a pointer to it and do whatever you need to do with it: write it, copy it, or move it. The Ping Pong buffer functions know which half of the buffer is for writing and which is next to be read. The 'Pinging' and 'Ponging' of the buffer, i.e. the changing of status from read to write, is handled internally by the Ping Pong buffer functions (making your life easier).

When you allocate memory for a Ping Pong buffer make it a large contiguous block that is a binary multiple of 512 e.g. 16284, 32678 or 65536. This ensures that when you fill half of the Ping Pong buffer you will be writing one large contiguous chunk of data to the CompactFlash and thereby helping to ensure a fast write speed.

Data collection (e.g. A/D) is usually performed by interrupt routines. These interrupt routines can be triggered via the PIT or the QSPI or an external signal or a combination of these. The Ping Pong buffer is written to in the interrupt routine or 'background' while the determination of full Ping Pong buffer 'halves' is done in the main loop of the program or the 'foreground'. The foreground process keeps writing to the CompactFlash while the short but periodic data collection happens in the 'background'. This is how data collection and storage appear to occur simultaneously.

Here is an example of using a Ping Pong buffer. First, let's allocate the buffer:

```
RAMPingPongBuf = calloc(PINGPONGBUFFSIZE, 1);
RAMppb = PPBOpen(PINGPONGBUFFSIZE, RAMPingPongBuf, 0, 0, 0)
```

We allocate memory for a buffer and then call PPBOpen to create the Ping Pong buffer from this raw data storage. PPBOpen returns a pointer which we will use in all subsequent PPB calls.

To write to the buffer we do the following:

```
PPBWrite(RAMppb, buff, size);
```

To determine if we have a full half-buffer we make a call like this:

```
if((ppavail = PPBCheckRdAvail(RAMppb)) >= PINGPONGBUFFSIZE / 2)
```

If half of the buffer is available PPBCheckRdAvail will return a value that is equal to half of the original size of the PingPong buffer.

Now we can get a pointer to the beginning of the ping-ponged data like this:

```
ppbuf = PPBGetMemBuf(RAMppb, &ppavail, true);        // and flush it
```

Now we can store the data or dump it to another device or whatever we need to do with it.

Controlling Serial Communications

SCI and Console Communications

Serial communications on the CF2 is provided by the 68332 SCI module. PicoDOS provides the programmer with a number of functions to access and utilize the SCI.

SCITxPutChar and SCIRxGetChar will send and get characters on the SCI port. There is also a function SCIRxGetCharWithTimeout which is useful when you need to wait for a character but not wait forever.

SCITxPutByte and SCIRxGetByte will send and receive characters on the SCI but have an optional block argument which will allow program execution to continue if the interface is not ready or if characters are not available.

Buffered Operation

SCITxSetBuffered and SCIRxSetBuffered are functions that are used to enable or disable buffered operation using the SCI. When buffered operation is enable a 2048 byte buffer is used for receive and a 512 byte buffer is used for transmit.

SCIRxQueuedCount and SCITxQueuedCount are used to determine the number of characters waiting in the buffers. SCITxFlush and SCIRxFlush will clear the queue counts effectively flushing any data waiting in the buffers.

SCITxWaitCompletion will wait for characters to be sent and the output buffer to be empty.

CTS/RTS and Flow Control

Hardware flow control is implemented on the SCI in the CF2 but not by default. If your application requires hardware flow control then enable it by making calls to SCIRxHandshake and SCITxHandshake. You can specify hardware flow control or XON/XOFF software control by passing arguments. Values for these arguments are spelled out in the *cfx*sercomm.h header file.

Example: simplest communications logger

```
short   i;
FILE *fp;
uchar c;

fp = fopen("serial.txt","w");

if(fp != NULL)
{
        while(!SCIRxBreak(10))      //Wait for a BREAK to exit
        {
                if(cgetq())
                {
                        c = SCIRxGetChar();
                        fwrite(&c, sizeof(uchar), 1, fp);
                        SCITxPutByte((ushort) c, false);
                }
        }
}
fclose(fp);
```

The Time Processor Unit (TPU)

The TPU as software UARTs

The TPU in the 68332 has the ability to perform the function of an asynchronous serial transmitter or receiver. You can also pair two TPU pins and use them together as a serial port. Functions are provided to access the capabilities of the TPU UART.

A couple of words of warning about using TPU UARTs: Flow Control and Loading.

Flow Control

There is no hardware flow control for a TPU UART. You may implement a software scheme by sending XON and XOFF characters if you are ambitious. But, be warned, if the equipment that you are connected to is limited in its ability to sustain the baud rate you are using then you will surely suffer a loss of data.

Loading

The Motorola document Asynchronous Serial Interface TPU Function (UART_AsyncSerial_tpup07r1.pdf) can be found here in C:\Program Files\Persistor\MotoCross Support\CFX\Docs\pdf\TPU. This document contains all of the information about the TPU UARTs. A question that is often asked is something like, “How many channels can I transmit /receive /both transmit AND receive at the same time”. The best information is on page 9 of the above-mentioned document. Here is the excerpt:

> **All examples assume that only the UART function is running. Examples are for absolute worst case, e.g. all receivers receive a stop bit at the same time, since this is the longest state.**
>
> **When only transmitters are running, maximum baud rate for all channels combined is 360 kbaud. This can be one transmitter with 360 kbaud, or nine with 38.4 kbaud, or any other combination.**
>
> **When only receivers are running, maximum baud rate is 233 kbaud. This can be one receiver with 233 kbaud, or six with 38.4 kbaud, or any other combination.**
>
> **When both receivers and transmitters are running, maximum baud rate is 142 kbaud. This can be one pair running at 142 kbaud, or three pair at 38.4 kbaud, or seven pair at 19.2 kbaud, or any other com-bination.**

Again, for the sake of clarity, “All examples assume that only the UART function is running”. This is almost never the case. Your CF2 program will probably be doing a lot of other tasks besides servicing the TPU UARTs. This means that the maximum number of channels at some mix of baud rates that YOU will be able to achieve will depend on what you are doing. TEST, TEST, TEST!

Structures

The TUChParams structure is defined in cfxpico.h. This structure holds parameters for initializing a TPU UART port.

```
typedef struct
{
short bits; // data bits exclusive of start, stop, parity
short parity; // parity: 'o','O','e','E', all else is none
```

```
short autobaud; // automatically adjust baud when clock changes
long baud; // baud rate
short rxpri; // receive channel TPUPriority
short txpri; // transmit channel TPUPriority
short rxqsz; // receive channel queue buffer size
short txqsz; // transmit channel queue buffer size
short tpfbsz; // transmit channel printf buffer size
} TUChParams
```

Example: TUAltConsole in C:\Program Files\Persistor\MotoCross Support\CFX\Examples\TPU\UARTs

The TPU as I/O

The TPU pins can all be used for simple I/O however, because the TPU is a separate computer subsystem, special considerations need to be made.

Sometimes you may need to toggle or clock a device with the TPU. For example:

```
while(!SCIRxBreak(10))
{
      PinSet(25);
      PinClear(25);
}
```

Be warned that placing pin control statements in a loop like the code above may give strange results. The reason is that when you issue multiple requests to change the state of TPU pins, the previous request may not have been completed before the next request is made. The result may be that the output does not change the number of times requested.

If you need to generate high speed signals using I/O, consider using one of the TPU's internal functions (like QOM). If your application can tolerate waiting for the TPU pin state request to occur BEFORE sending another request, try this:

```
while(!SCIRxBreak(10))
{
      PinSet(25);
      TPUHostServiceCheckComplete(TPUChanFromPin(25),true);
      PinClear(25);
      TPUHostServiceCheckComplete(TPUChanFromPin(25),true);
}
```

The 'true' in the TPUHostServiceCheckComplete call means wait.

Simple I/O

All pins on the CF2 that can be used as I/O can be controlled with two different sets of commands. Commands that begin with PIO will set the direction of the pin AND set the logic level. These commands take more time to execute because they set the direction first. If the direction has previously been set with a PIO function then a simple Pin function can be used to change or read the logic level of the specified pin. Here are some examples:

PIOSet(25); //Make pin 25 an output and set its logic level to one.

PIOClear(25); //Make pin 25 an output and set its logic level to zero.

PinSet(25); //Assume pin 25 is already an output and set it's logic level to one.

PinClear(25); //Assume pin 25 is already an output and set it's logic level to zero.

PIORead(25); Make pin 25 an input and return its logic level.

Other TPU Functions

TO BE WRITTEN

The LEDs

There are two LEDs on the front edge of the CF2. One is to the left of the CompactFlash header and the other is to the right. Each LED can be programmed ON or OFF with two colors: Red and Green. The CF2 also has functions to toggle the LEDs and to alternate the LEDs in such as way as to make them appear to orbit around clockwise and counterclockwise.

```
#define WAITKEY while(!kbhit()) continue; cgetc()

        LEDInit();

//      Return the current LED state
        cprintf("LEDGetState(LEDleft) = %d\n",LEDGetState(LEDleft));
        cprintf("LEDGetState(LEDright) = %d\n",LEDGetState(LEDright));

        WAITKEY;

//      Set the LED state
        LEDSetState(LEDleft, LEDred);
        cprintf("\nThe left LED is RED\n");
        WAITKEY;

        LEDSetState(LEDleft, LEDgreen);
        cprintf("\nThe left LED is GREEN\n");
        WAITKEY;

        LEDSetState(LEDleft, LEDoff);
        cprintf("\nThe left LED is OFF\n");
        WAITKEY;

        LEDSetState(LEDright, LEDred);
        cprintf("\nThe right LED is RED\n");
        WAITKEY;

        LEDSetState(LEDright, LEDgreen);
        cprintf("\nThe right LED is GREEN\n");
        WAITKEY;

        LEDSetState(LEDright, LEDoff);

//      Toggle LED between Red and Green
        LEDToggleRG(LEDright);
        cprintf("\nThe right LED is toggled\n");
        WAITKEY;

        LEDToggleRG(LEDright);
        cprintf("\nThe right LED is toggled\n");
```

```
        WAITKEY;

        LEDSetState(LEDright, LEDoff);

//      Toggle LED between Red, Green, and Off
        LEDToggleRGOff(LEDright);
        cprintf("\nThe right LED is RED\n");
        WAITKEY;

        LEDToggleRGOff(LEDright);
        cprintf("\nThe right LED is GREEN\n");
        WAITKEY;

        LEDToggleRGOff(LEDright);
        cprintf("\nThe right LED is OFF\n");
        WAITKEY;


//      Orbit the LEDs on each call
        cprintf("\nOrbit one way\n");
        while(!kbhit())
        {
                LEDOrbit(true);
                RTCDelayMicroSeconds(500000L);
        }
        cgetc();

        cprintf("\nOrbit the other way\n");
        while(!kbhit())
        {
                LEDOrbit(false);
                RTCDelayMicroSeconds(500000L);
        }
        cgetc();

        LEDInit();
        cprintf("\nDone!\n");
```

Queued SPI (QSPI devices)

The QSPI is the Queued Serial Peripheral Interface of the 68332. The QSPI is a souped-up SPI port. With a queue of up to 16 words the QSPI can command and transfer data from external SPI devices independent of the CPU.

For this section we will assume that a user has experience with SPI however, for the uninitiated, SPI is serial hardware used to transfer data between a master and a slave. The master and the slave basically exchange data by shifting data through paired shift registers with input, output and common clock lines. A single chip select is asserted to the slave in order to change common output lines from tri-state to driven logic levels.

Queued Pico Bus or QPB are a set of functions to make the QSPI easier to use. The model used for QPB to manage SPI peripherals is the device. Below is a structure, QPBDev, which QPB uses to define a device (or SPI peripheral):

```
static QPBDev MyDevStructure1 =
{
        NMPCS0,         // qslot        our qspi slot (non-multiplexed connection to PCS0)
        "MAX147\0",     // devName      C string with device name (15 max): can be anything.
        2000000,        // maxBaud      maximum baud rate in Hz for device
        16,             // bits         Bits Per Transfer
        iaHighSCK,      // clockPolar   SPI Clock Polarity (iaHighSCK,iaLowSCK)
        captFall,       // clockPhase   SPI Clock Phase (captLead/captFall)
        false,          // contCSMulti  Continue CS assert between mult xfrs
        true,           // autoTiming   Auto adjust timing to clock flag
        1000000,        // psDelaySCK   Min. Delay Before SCK (picoSecs). A million is a microsecond
        1000000,        // psDelayTXFR  Min. Delay After Transfer
        0,              // *rcvData     pointer to received data buffer (filled in by QPBInitSlot)
        0               // xfrCount     words transferred
};
```

QSLOT

There are 4 dedicated chip selects associated with QSPI: PCS0-PCS3. The slot (qslot) in the QPBDev structure is a value that is written to control registers for PCS level control. The predefined symbol NMPCS0 used in the example above equates to a value of 14 decimal or, expressed as four bits starting from PCS3 to PCS0, 1110. This means that when data is being transferred using QSPI PCS0 will be the only PCS line low. The SPI peripheral connected to PCS0 is selected to transfer data (in this case).

DEVNAME

devName is just a name for the device. It is not used at all for anything more than an identifier for your target device, should you need it. You are limited to 15 characters.

MAXBAUD

maxBaud is the maximum desired baud rate for the SPI device. QPB will not exceed this rate but it also may not be able to achieve it. If, for example, you used 2 MHz as a value for maxBaud but you set the CPU clock rate to 1.25 MHz then the actual baud rate will be lower.

BITS

Bits is the number of bits to transfer. This will usually be 8 or 16 but can be anything *from* 8 to 16. If you have a need for 9 bits transfers then you are in luck: the QSPI is flexible.

CLOCKPOLAR and CLOCKPHASE

These two bits define the SPI mode used. SPI has modes 0 through 3. Set these values to match your device.

CONTCSMULTI

This is a Boolean value which, if true, will force the QSPI to hold the chip select for the given device low between transactions. This is great for devices that support it but not for other who need to see CS go high before some internal action occurs. Check your device before using.

AUTOTIMING

If true this parameter will cause the QSPI to automatically set the timing of the device according to the levels seen on the first clock edge.

PSDELAYSCK

This is the desired minimum delay from CS that you want the QSPI to insert before SCK is active (in picoseconds).

PSDELAYTRANSFER

This is the desired minimum delay from after SCK is inactive until CS goes high (in picoseconds).

RCVDATA

A pointer to the data transferred. This is allocated by the QPB when you initialize your device using QPBInitSlot.

XFRCOUNT

A short value which contains the number of words transferred.

Initializing QPB

First, you need a pointer to a QPB structure. In this example we will call it QPB_Slot1. To initialize the QPB call QPBInitSlot like this:

```
QPB_Slot1 = QPBInitSlot(&MyDevStructure1);
```

Remember to check the return value to ensure that it is not NULL. If it is not NULL then you are ready to use the SPI device with the QSPI.

Using QPB

Before you should initiate a transaction using QPB the data we wish to send to the SPI device must be initialized. In this example we have declared an array of ushorts called usCommand1, which will hold the commands we will send to a hypothetical SPI device. Fill usCommand1 with four commands like this:

```
usCommand1[0] = 0xAAAA;
usCommand1[1] = 0x5555;
usCommand1[2] = 0xFFFF;
usCommand1[3] = 0x0000;
```

Assign a value to the structure for the number of words to send:

```
QPB_Slot1->dev->xfrCount = 4;
```

You can now initiate a transfer to a QPSI device using QPB by calling QPBTransact.

```
QPBTransact(QPB_Slot1, 0, 4, usCommand1);
```

As you can see you pass QPBTransact the pointer to QPB_Slot1, a zero (which will be explained further down), a count of the number of words to transfer, and a pointer to the array of data to be sent.

The data received by the master (the CF2) will be in rcvData which can be accesses through the QPBDev structure like this:

```
cprintf("QPB_Slot1->dev->rcvData[0] = [%u]\n", QPB_Slot1->dev->rcvData[0]);
cprintf("QPB_Slot1->dev->rcvData[1] = [%u]\n", QPB_Slot1->dev->rcvData[1]);
cprintf("QPB_Slot1->dev->rcvData[2] = [%u]\n", QPB_Slot1->dev->rcvData[2]);
cprintf("QPB_Slot1->dev->rcvData[3] = [%u]\n", QPB_Slot1->dev->rcvData[3]);
```

It is a good idea to move the data to another location so that it is not overwritten by a subsequent transaction.

Automatic Transfer

The second argument passed to QPBTransact above was a zero. If you write a function whose job it is to copy the data from QPBDev to some other structure you can pass it as an argument in place of the zero. One example of what might be done in such a routine is to transfer the data harvested by the QSPI Master to a queue of some kind (perhaps, a Ping Pong Buffer).

It is also possible to run the QSPI automatically in that, when a transfer is completed the ISR can remove any data collected and automatically force a restart of the QPBTransact using the same parameters used initially. To restart the QSPI in this way you make a call to QPBRepeatAsync.

Multiple Devices

You can initialize other QSPI QPB devices that you will need. Call QPBTransact with different arguments to communicate with different devices:

```
QPBTransact(QPB_Slot1, 0, 4, usCommand1);     //Device 1

QPBTransact(QPB_Slot2, 0, 2, usCommand2);     //Device 2

QPBTransact(QPB_Slot3, 0, 4, usCommand3);     //Device 3
```

Expanding the Limits of QSPI

Many people wish to expand the CF2 and QSPI beyond the four basic chip selects. The value for qslot can be any value desired so, it is therefore possible to decode the chip selects themselves to expand the number of chip selects the QSPI could use.

Lets assume that you want to expand the QSPI to use 15 devices. Qslot has 16 different possible values in four bits. Simply use one value as a holder for the inactive state and the rest of the qslots will be unique. So, if 15 (1111) was the inactive case, then decoding the bits for the other states yields 15 active chip selects (using an external decoder to decide the individual states into active low chip selects). A pair of HC138's could do this job nicely.

Programming for Serial A/D and RecipeCards

The Persistor Instruments Recipe Cards are available with two different synchronous serial A/D converters: the Maxim 12 bit MAX146 and the Burr-Brown 16 bit ADS8344. What follows is a description and examples of how to use these two devices. Following that we will demonstrate adding a different serial A/D converter using the same foundations.

The following material will make references to, and will assume that you have read, the previous section **Queued SPI (QSPI devices)**.

Simple A/D

First, create a blank PicoDOS program. This blank PicoDOS program will have one source file: cfxmain.c. For details on doing this we refer you to the CF2 Getting Started Guide.

To use the MAX146 A/D converter in this simple example, we will add the following information to the top of cfxmain.c:

```
#define     ADSLOT      NMPCS3          // The QPB slot number (0..14)
#define     ADTYPE      ADisMAX146      // The A-D selector from <cf1AD.h>
#define     VREF  2.5                   // A-D reference voltage
#include <ADExamples.h>

CFxAD adbuf, *ad;
```

Here is what it all means:

- ADTYPE refers to the actual A/D converter we intend to use.
- ADSLOT corresponds to the QSPI chip select we will use to communicate with the A/D. NMPCS3 will give us a low on PCS3 (Pin 17) on the CF2.
- VREF is the reference voltage. For 12 bit measurements and a 2.5 volt reference, a ratiometric conversion of an input voltage of 1.25 would return to us an A/D reading of approximately 2048 decimal.
- ADExamples.h has most of the definitions we need to use the A/D converters.
- We will use adbuf to contain all the information and pointers to functions to access the A/D converters. The pointer *ad will point to adbuf and we will use it in all the functions that require a pointer to the information we create in adbuf. Later, in our advanced material, we will use CFxAD to add another A/D converter.

The next thing we will do is to add to the body of main() in cfxmain.c the following:

```
ad = CFxADInit(&adbuf, NMPCS3, ADInitFunction);
```

CFxADInit has three arguments. They are, a pointer to storage of type CFxAD, the chip select for the QSPI and a pointer to the function responsible for initialization of the A/D converter. The pointer to the storage for the CFxAD info is returned and we assign it to our CFxAD pointer.

Now that we have initialized the A/D all we need to do to sample a single channel is to call the function CFxADSample like this:

```
addata = CFxADSample(ad, 0, true, true, false);
```

addata is an unsigned 16 bit value (that we declared in main()) where we would like the A/D result placed. When we call CFxADSample we pass it the pointer to our CFxAD data, the channel number (0-7) and three Boolean values. The Boolean values meanings, in order, are:

- unipolar (true) or bipolar (false)
- single-ended (true) or differential (false)
- powerdown (true) no powerdown (false)

Please refer to the MAX146 data sheet for an explanation of the meaning of these. For this simple example we asked for unipolar, single-ended measurements and we leave the A/D converter powered-up after the conversion has completed.

After a call to CFxADSample, addata contains the ratiometric value of the voltage on A/D channel zero. Here is the code:

```
#define ADSLOT  NMPCS3                  // The QPB slot number (0..14)
#define ADTYPE  ADisMAX146              // The A-D selector from <cf1AD.h>
#define VREF    2.5                             // A-D reference voltage

#include <ADExamples.h>

CFxAD   adbuf, *ad;

/*****************************************************************************\
**      main
\*****************************************************************************/
int main(int argc, char **argv)
        {
        short   result = 0;             // no errors so far
        ushort addata;

        if(argc)        //Keeps it quiet
                argv;

        ad = CFxADInit(&adbuf, NMPCS3, ADInitFunction);

        addata = CFxADSample(ad, 0, true, true, false);

        cprintf("\naddata = %04X\n",addata);

        return result;

        }       //____ main() ____//
```

Multiple Channels (not as simple)

We can expand the previous example for single channel A/D measurements to perform multiple samples. The function we will use is called **CFxADSampleBlock**. Using it is a little different from using CFxADSample. Here is an example call to the function and an explanation of the different arguments:

```
CFxADSampleBlock(ad, 0, 8, &THandler, true, true, false);
```

Following the CFxAD pointer, ad, are two values. These values are **first** and **count**. The zero we use for **first** means we will start sampling with channel zero. The value of 8 for **count** means we want to sample 8 sequential channels following our **first** channel 0.

The next argument after **count** is **asyncf** or THandler. This is a pointer to a function that will be responsible for gathering all the data from the **CFxADSampleBlock** function call.

We are using the QSPI to control the A/D converter. Whenever a QSPI transaction occurs, the results of the transaction i.e. data from the A/D converter destined for our application, are left in QSPI RAM (QRR). After we call **CFxADSampleBlock** the **asyncf** function, THandler, will automatically move the A/D values from the QRR to some place we can hold, modify or store it permanently. We will add to this multiple channel example a structure definition that we will use to hold the A/D data. Here is that structure:

typedef struct { short adata[8]; } RawData; // allows structure copy

This structure will allow our **asyncf** handler, THandler, to copy the data from QRR fast. In the code (see below) all it needs to do is clear the interrupt for the QSPI or, using the syntax of the QPB, call QPBClearInterrupt. Following this the data is copied from QRR to the RawData structure in the code, pdata.

```
typedef struct { short adata[8]; }    RawData;       // allows structure copy
RawData ADDataBuf;

#define ADSLOT  NMPCS3                 // The QPB slot number (0..14)
#define ADTYPE  ADisMAX146             // The A-D selector from <cf1AD.h>
#define VREF    2.5                            // A-D reference voltage

#include <ADExamples.h>

CFxAD   adbuf, *ad;

IEV_C_PROTO(THandler);
IEV_C_FUNCT(THandler)  // implied (IEVStack *ievstack:__a0) parameter
{
        #pragma unused(ievstack)

        QPBClearInterrupt();

        ADDataBuf = * (RawData *) CFxADQueueToArray(ad, (void *) QRR, 8);

}

/*****************************************************************************\
**      main
\*****************************************************************************/
int main(int argc, char **argv)
        {
        short   i;
        short   result = 0;            // no errors so far

        if(argc)       //Keeps it quiet
                argv;

        ad = CFxADInit(&adbuf, NMPCS3, ADInitFunction);

        CFxADSampleBlock(ad, 0, 8, &THandler, true, true, false);

        for(i=0;i<8;i++)
                cprintf("\nADDataBuf.adata[%d] = %04X\n",i,ADDataBuf.adata[i]);


// Various exit strategies
//      BIOSReset();                   // full hardware reset
//      BIOSResetToPBM();              // full reset, but stay in Pesistor Boot Monitor
//      BIOSResetToPicoDOS();  // full reset, but jump to 0xE10000 (PicoDOS)
        return result;

        }       //____ main() ____//
```

In C, a function can only return a single value. This is what makes multiple channels more difficult than single channel sampling with **CFxADSample**. Of course, you could just call **CFxADSample** eight times to sample all 8 channels of the A/D. This will work just fine however, if your application calls for faster sampling then **CFxADSampleBlock** is both faster and more efficient than multiple calls to **CFxADSample**.

Buffers and Rings (a little more complicated)

When looking at the THandler code, notice that pdata could be a pointer to an array or RawData structures. This would provide a way to store block samples in a buffer. For example:

Ring[element++] = * (RawData *) CFxADQueueToArray(ad, (void *) QRR, 8);

where Ring is declared as:

RawData *Ring[RINGSIZE];

This may seems more complicated and it is. It is also very powerful. If a PITChore was used to start the A/D sampling then the whole operation could happen in the 'background'. A 'foreground' application could watch to see when data was available and write it to a file on the CompactFlash. Here is an example of such an application:

```
void PITHandler(void);
IEV_C_PROTO(THandler);

/*
**      Change if you are using a different A/D or chip select
*/
#define ADSLOT  NMPCS3                  // The QPB slot number (0..14)
#define ADTYPE  ADisMAX146              // The A-D selector from <cf1AD.h>
#define VREF    2.5                             // A-D reference voltage

#include <ADExamples.h>

CFxAD   adbuf, *ad;

#define ADFAST_RAW_DATA_BUF_SIZE      16384L          //Change to suit needs
#define ADFAST_RAW_DATA_BUF_MASK      ((ADFAST_RAW_DATA_BUF_SIZE / (8 * 2)) - 1)

typedef struct { short adata[8]; }    RawData;        // allows structure copy
RawData *PLADDataBuf;                                           // dynamically allocated at run time
ushort  PLRawHead, PLRawTail;                                   // head and tail indexes

/*****************************************************************************\
**      main
\*****************************************************************************/
int main(int argc, char **argv)
        {
        short   result = 0;             // no errors so far
        FILE    *fp;
        char    *mybuff = NULL;

        if(argc)
                argv;

/*
**      We'll use this pin to watch the interrupts on an oscilloscope
*/
        PIOSet(1);

/*
**      Open the file for output
*/
        fp = fopen("C:\\ADDATA.DAT","w");     //This will overwrite any old ones
        if(fp == NULL)
        {
                cprintf("\nCannot open file...aborting\n");
                return(0);
        }
        else
        {
```

```
                cprintf("\nFile opened!!!\n");
        }

/*
**      Allocate space for the buffer for A/D data.  This is where data will be written
**      from the ISR for the QSPI.  This data will then be written to the BigIDEA using
**      fwrites.  The ISR writes at the head and the forground process writes from the tail.
**      The tail chases the head.  The only error can occur when the head 'passes' the tail.
*/
        PLADDataBuf = malloc(ADFAST_RAW_DATA_BUF_SIZE);
        if(PLADDataBuf == NULL)
        {
                cprintf("\nCannot allocate PLADDataBuf\n");
                goto end;
        }

/*
**      Head and tail are equal to start.
*/
        PLRawHead = PLRawTail = 0;

/*
**      Get the QSPI ready for the A/D, lock the slot and take a single
**      block sample to set up the QSPI registers.
*/
        ad = CFxADInit(&adbuf, NMPCS3, ADInitFunction);
        CFxADSampleBlock(ad, 0, 8, &THandler, true, true, false);

/*
**      Head and tail are equal to start (reset)
*/
        PLRawHead = PLRawTail = 0;

/*
**      Start the PIT
*/
        PITInit(5);
        PITAddChore(PITHandler,5);
        PITSet100usPeriod(PIT100Hz);  //100 Samples per second

        cprintf("Writing data!\n");

/*
** Until we see a break (F8 under Motocross)
*/
        while(!SCIRxBreak(100))
        {
                //Any data waiting to be written to the file?
                if(PLRawHead!=PLRawTail)
                {
                        fwrite(&PLADDataBuf[PLRawTail], sizeof(RawData), 1, fp);
                        PLRawTail += 1;
                        PLRawTail &= ADFAST_RAW_DATA_BUF_MASK;
                }
        }

        PITSet100usPeriod(PITOff);
        PITRemoveChore(PITHandler);

end:

/*
**      Close the file
*/
        if(fp)
                fclose(fp);

// Various exit strategies
//      BIOSReset();                    // full hardware reset
//      BIOSResetToPBM();               // full reset, but stay in Pesistor Boot Monitor
        BIOSResetToPicoDOS();  // full reset, but jump to 0xE10000 (PicoDOS)
        return result;

        }       //____ main() ____//


void PITHandler(void)
{
        QPBRepeatAsync();
}
```

```
IEV_C_FUNCT(THandler)  // implied (IEVStack *ievstack:__a0) parameter
{
        #pragma unused(ievstack)

        QPBClearInterrupt();
        PinClear(1);

/*
**      This check can be removed when testing is complete.  It just prints a '.' (period) out the serial
**      port if the head is about to pass the tail.
*/
        if(PLRawHead == (PLRawTail - 1))
                cprintf(".");

/*
** Grab the data
*/
        PLADDataBuf[PLRawHead++] = * (RawData *) CFxADQueueToArray(ad, (void *) QRR, 8);
        PLRawHead &= ADFAST_RAW_DATA_BUF_MASK;        //Mask

        PinSet(1);

}
```

The examples can also be found on the Pii web site http://www.persistor.com under Support->Programming Examples.

Adding a New A/D Converter

TO BE DONE

Programming for CF2 Add On Boards

TO BE DONE

CF2 Memory Map

512B	FFFFFFFF FFFFFE00	TPU	MCU
512B	FFFFFDFF FFFFFC00	QSM	
256B	FFFFFBFF FFFFFB00	SBRAM	
256B	FFFFFAFF FFFFFA00	SIML	
2.5KB	FFFFF9FF FFFFF000	TPU RAM	

2KB	FFFFE7FF FFFFE000	/CS3 /CS4 /CS5	CF
8KB	FFFFDFFF FFFFC000	/CS8 /CS10	XPND
16KB	FFFFBFFF FFFF8000	/CS8 /CS10	SC XPND

16KB	FFFF3FFF FFFF0000	/CS8 /CS10	SC XPND

256KB	00EFFFFF 00EC0000	APP	1MB Flash /CSBOOT /CS7
256KB	00EBFFFF 00E80000	APP	
256KB	00E7FFFF 00E40000	APP	
192KB	00E3FFFF 00E10000	PicoDOS	
32KB	00E0FFFF 00E08000	BIOS	
8KB	00E07FFF 00E06000	VEE2	
8KB	00E05FFF 00E04000	VEE1	
16KB	00E03FFF 00E00000	PBM	
14MB	00DFFFFF 00100000	/CS8 /CS10	XPND

64KB	0007FFFF 00070000	Pico	512 KB SRAM /CS0 /CS1 /CS2 /CS9
384KB	0006FFFF 00010000	APP-HEAP	
16KB	0000FFFF 0000C000	APP-STACK	
4KB	0000BFFF 0000B000	Pico-STACK	
28KB	0000AFFF 00004000	Pico-DATA	
15KB	00003FFF 00000400	BIOS-RAM	
1KB	000003FF 00000000	VBR	

Using the Table Driven Command Processor

Overview

Included in the software provided by Persistor Instruments Inc. for use with the CF2 is a simple, table-driven command line interface that you can use in your own programs. There are two basic ways to use the Command Processor:

One way is to use it to create your own custom version of PicoDOS. You can create your own custom commands with options of your own choosing. You can also use built-in PicoDOS commands as a part of your 'custom' PicoDOS or remove any of them that you don't want. For example, if you were creating a custom command set for a user you might choose to NOT include the FORMAT command so as to prevent accidental formatting of CompactFlash cards in the field. We will discuss the Command Processor in detail by describing the ToPico stationery that provides a working skeleton from which you can base your own custom PicoDOS.

The other way to use the Command Processor is as a parser for commands and command switches in a standard PicoDOS application. This beats trying to hack together a command line interface of your own (always fraught with danger).

Creating a ToPico Project

It is assumed at this point that you have already gone through the CF2 Getting Started Guide and know how to create a project using CodeWarrior. If you need a refresher then go to page 13 of the CF2 Getting Started Guide. If you know how to create a blank project then all you need to do to create a ToPico project is to pick ToPico when you are selecting Project Stationery. I will assume that you have done this at this point.

Now that we have the ToPico project open we will want to give the target a name and also to name the output file. I will name mine MyPico. I will also name the C source file MyPico.c. I will use those names when referring to this example.

The Command Table

Notice that near the top of the source file around line 58 is the following definition CmdTable ToPicoCmdTable[]. This is the table that will drive the Command Processor. It will contain the names of the commands we will create and point to the actual code for those functions. The table will also contain other essential parameters needed by the Command Processor.

Let's learn how to create a command by looking at one of the simple commands provided in ToPico. We will skip some information in the Command Table for now but we will come back to it later. Skip down in the MyPico source file and look for a line that begins with the word SIMPLE in all capital letters (around line 92). It should look something like this:

```
,"SIMPLE"             , SimpleCmd,                  0,      0,      0,      0,      "Simple test command"
```

The quoted string "SIMPLE" will be the name of the command itself. This is what you will type at the prompt under MyPico. When the Command processor has recognized this as the command "SIMPLE" it will call a function that corresponds to that command. The function that it will call is SimpleCmd which is the next entry in the table. Let's look at SimpleCmd to see what it does. Skip down in the MyPico source file until you find SimpleCmd (around line 228). An easy way to do thus under CodeWarrior is to use the Functions Button on the Toolbar.

SimpleCmd looks like this:

```
char *SimpleCmd(CmdInfoPtr cip)
      {
      enum  { cmd, arg1, arg2, arg3, arg4 };

      cprintf("\n%s\n", cip->argv[cmd].str);    // just prints the command string

      return 0;

      }     //____ SimpleCmd() ____//
```

Notice that this and all custom commands are declared the same way. Each returns a pointer to char and each is passed one and only one variable: a pointer to cip of the type CmdInfoPtr. CmdInfoPtr is a structure that contains information that the Command Processor has parsed for us. We can even further parse the command line that called us to reveal switches and other options.

Here is the definition of CmdInfoPtr found in cfxpico.h:

```
typedef struct CmdInfo
      {
      CmdTablePtr    ctp;                  // working command table
      CmdTablePtr    altctp;               // secondary command table
      char           privLevel;            // working privilege level
      char           justCR;               // last was just carriage return
      char           abbrevOk;             // accept abbreviated commands (default)
      char           justCROk;             // accept CR repeat commands (default)
      char           optDelims;            // default as #defined
      char           lineDelim;            // default as #defined
      char           repChar;              // default as #defined
      char           rangeChar;            // default as #defined
//    char           quiet;                // no messages or prompts
      short          argc;                 // number of arguments (incl. command
#define              ARGS    (cip->argc-1) // convenient argument accessor
      CmdParam       argv[CMDMAXARGS];     // arguments
      char           line[CMDLINELEN];     // working command line buffer
//    char           text[CMDLINELEN];     // verbatim copy of input line
      char           *options;             // command modifiers (typ. CMD.mmm)
      char           *errmes;              // last error message
      long           repeat;               // number of times to execute
      getsfptr       altgets;              // zero defaults to stdlib gets()
      }      CmdInfo, *CmdInfoPtr
```

Don't be too alarmed by all of the things you see in CmdInfoPtr. You may not need to access it directly but by using powerful PicoDOS functions you too will be able to harness the power it provides. In fact, look back at SimpleCmd. Notice that all it does is print the name of the command that invoked it, namely, SIMPLE. It accesses this information from the command line using the pointer to the CmdInfo structure:

```
cprintf("\n%s\n", cip->argv[cmd].str);       // just prints the command string
```

We enum offsets to each of the various arguments on the command line. To see what we can do with SimpleCmd, lets add a line to print the first argument, if it is present. The code you write should look something like this:

```
if( cip->argc>1)
       cprintf("\n%s\n", cip->argv[arg1].str);      // prints the first argument...if present
```

To try this program all we have to do is compile and load and run MyPico.RUN using MotoCross. When the program runs all you need to do is to call SimpleCmd and pass it an argument. Anything will work but for this example we will choose the word 'HELLO' (for historical reasons ala K&R). So my command line will be just **SIMPLE HELLO** and the output will look like this:

```
TOP)C:\>SIMPLE HELLO
SIMPLE

HELLO

TOP)C:\>
```

This may not look too exciting but we have demonstrated how to pass a simple argument to a command of our own using the Command Processor and PicoDOS.

Now for something a little harder. Read on.

Adding a Command

We can add a command to our MyPico program easily. First, we need to write a function to add. To do this, copy the function SwitchesCmd which is found near the bottom of the MyPico project. Copy the entire function and paste it at the end of the file. Rename the function SFormat. Copy the declaration and move it to the top of the source file for MyPico and place it under the other function declarations just above the declaration of our Command Table ToPicoCmdTable (don't forget the semicolon at the end of the declaration of SFormat).

Now that we have a function that we can add to the Command Table, go to the command table and copy the following line and paste a copy right below it:

```
,"SWITCH"              , SwitchesCmd,              0,      0,      0,      10,     "Demos arg switches"
```

Change SWITCH, the first entry, to SFORMAT. This will be the name of our new command. Now change SwitchesCmd to the name of the function to call, in this case, SFormat.

The next four numbers in the command table need some explanation. The first number is the minimum number of required arguments for this new command. We will require no additional arguments for the command we will write so this number can stay zero. The next number is the privilege level. This is left for a possible future enhancement of the command processor and so can just be left as zero. The next number is whether or not a carriage return (CR) at a subsequent prompt should allow the command to just repeat as is either zero or one for false and true. For the command we will write leave this value as zero. The next number is the base of the number system used for optional arguments. This can be left as 10. Lastly, change the text at the end of the table entry to "Special Format". This will be displayed in a listing of the available commands.

Handling Switches

We will use some special functions that will allow us to parse information from the command line. We will need to declare some special functions to allow us to specify what options are possible with the new command we will write. The type of structure we will use is called DosSwitch. The declaration of DosSwitch is in cfxpico.h and looks like this:

```
typedef struct
        {
        const char      *swset;         // standard is "/"
        const char      idch;           // switch ID character ('V' for /V)
        char            pos;            // was specified, at argv position n
        const char      *scfmt;         // nonzero sscanf format to attempt scan
        const bool      skpspc;         // accept space between switch and argument
        bool            hasv;           // value was found and successfully sscanf'd
        union
                {
                long    lv;             // long or unsigned long value
                float   fv;             // float (32 bit) value
                char    *sp;            // >2.30, save string pointer for "%s" scfmt
                };
        } DosSwitch
```

The way we use a DosSwitch structure is to load it with some parameters which indicate the acceptable command line switches for the command we will write. We can then use them and the CmdInfoPtr passed to our function to determine which options are present on the line that we type.

Before we determine what we need for switches, let's look at the command we will add.

SFormat

SFormat is a program and stands for Special Format. Often when using a CF2 in the field you may need to format CompactFlash cards. The problem with format is that it erases all the data on the card. Sometimes you may have an AUTOEXEC.BAT file and/or some helper .PXE functions on the card that you would like to keep. SFormat was written to allow formatting of the card while keeping optional files already stored on the card. The way it works is that SFormat will allocate RAM and move as many files specified on the command line into RAM for safe keeping. SFormat will then format the card and then move all of the files from RAM back to the CompactFlash card. Because CF2 users tend to have AUTOEXEC.BAT files stored on their CompactFlash cards, we will make SFormat look for and always save an AUTOEXEC.BAT file if it is present. We may need to override this feature sometimes so we will need a command switch to tell SFormat to ignore the AUTOEXEC.BAT file and erase it anyways.

The syntax of the SFormat command will be:

SFORMAT file1, file2, file3, ... fileN –A

where –A means ignore AUTOEXEC.BAT

If any of the files specified on the command line are not present SFormat will not continue and will display an error message. Another error message is generate when SFormat runs out of memory for all of the specified files.

The declaration of the DosSwitch that we will use for the –A option is this:

```
DosSwitch     asw = { "+",  'A', 0, 0 };
```

Then we will initialize the structure with a minus sign inside of double quotes to indicate that the switch is not valid on a command line unless it is preceded with a minus sign. The switch ID will be 'A' and we will zero the next two fields: pos and scfmt.

The function that we use to gather the switch data is CmdExtractCIDosSwitches. We will call it and pass it cip () a string with a placeholder for each DosSwitch to be scanned and a pointer to the DosSwitch structure. We use a placeholder of 1 but it could be a letter too…the only requirement is that there is one placeholder for each DosSwitch to be scanned and that there is a corresponding pointer passed for it.

When CmdExtractCIDosSwitches returns asw.pos will be zero if the switch is NOT present and non-zero if it is. If it is non-zero then its value is the position within all the other tokens on the command line. The code for using the switch is this:

```
CmdExtractCIDosSwitches(cip, "1", &asw);
if(asw.pos==0)
{
//    Look for a file at the root called autoexec.bat
.
.
.
}
```

The code for SFormat is available for download at the Persistor Instruments web site at http://www.persistor.com.

The Rest of the Command Table

The members of the first entry in the command table are treated with special meanings. The first entry is the prompt. This follows much of the same convention that old DOS prompts did. We precede our prompt with "TOP)" to identify this as ToPico and not actual PicoDOS. You can of course change it to whatever you want.

```
"TOP)$P$G"          // Your custom command prompt using DOS prompt translation
, PDCCmdStdPicoRun,      // The default interactive run handler
0,                       // 1 for two column help, 0 for single column
100,                     // Default privilege level at start
1,                  // Flag to accept carriage-return as repeat last command
0,                       // Flag to accept abbreviated commands
//    Text to display before listing all of the individual commands help text
"\n~~~~~~~~~~~~~~~~~~~~  ToPico custom commands   ~~~~~~~~~~~~~~~~~~~"
```

The next entry is the run handler. Leave this with the default value of PDCCmdStdPicoRun.

Help in ToPico can be displayed in two or one columns. If the help text that you enter for one of your commands is too long to display in a two-column listing it will be truncated. If you need to use a longer help string but would like to make sure it is displayed in its entirety then use a one column listing, i.e. make the entry in the command table equal to one.

The next entry is privilege level. Leave this as 100.

The overriding flag to allow individual commands to be repeated with a carriage-return is the next parameter. Making this a zero will prevent ANY command from being repeated with just a carriage-return. Leaving this as a one is a good idea so that commands like MD (memory display) can be repeated easily with a carriage-return.

If you would like to make abbreviated commands possible change the value of the next field to a one. There can sometime be confusion on commands which start with the same first few letters though. The last special entry is the text string which is displayed above the help listing for custom commands. You can change this to anything you would like.

Command Processor without ToPico

The command processor can be used without using ToPico stationery. This is nice if you have written your own program which allows a user to type in commands; maybe in a special 'mode' when some other control and/or logging operation is normally running. In any case, you can use the command processors ability to look for switches with optional string or numeric parameters in a normal PicoDOS application. Here are the essential things you need to do:

1. Initialize the CmdInfo structure that you have declared by calling CmdStdSetup.

2. Cram a string of characters you wish to process into some area of memory and get a pointer to the beginning of that memory (we'll call ours *str).
3. Copy that string of characters to the CmdInfo structure line element:

```
CmdInfo MyCmd;
strcpy(MyCmdPtr->line, str);
```

4. Parse the text:

```
CmdParse(&MyCmd);
```

5. Extract the switches:

```
CmdExtractCIDosSwitches(&MyCmd, "1", &nsw);
```

That's all there is to it. Now you can look at the pos element of the DosSwitch structures to see if any of the switches are present. If pos is non-zero then the switch was found.

The example program we will write converts decimal arguments to hexadecimal and, optionally, to octal. This is rather simple but is just meant to demonstrate the use of the command processor outside of ToPico.

Example

This is an example of creating simple program to convert a decimal argument into a hexadecimal value and optionally to octal.

The syntax of the command we will add is this:

D2B /V NNNNN [/O]

where NNNNN is a decimal integer. The decimal value and its hexadecimal value are printed. If the /O switch is found then the Octal value is printed as well. Here is the code:

```
/*****************************************************************************\
**      cmdalone.c
**
**      Initial Release:                November 25, 2002
**
*****************************************************************************
**
**      COPYRIGHT (C) 2002 PERSISTOR INSTRUMENTS INC., ALL RIGHTS RESERVED
**
**      Developed by: Thomas P. Sullivan for Persistor Instruments Inc.
**      254-J Shore Road, Bourne, MA 02532  USA
**      tpsully@persistor.com - http://www.persistor.com
**
**      Copyright (C) 2002 Persistor Instruments Inc.
**      All rights reserved.
**
*****************************************************************************
**
**      Copyright and License Information
**
**      Persistor Instruments Inc. (hereafter, PII) grants you (hereafter,
**      Licensee) a non-exclusive, non-transferable license to use the software
**      source code contained in this single source file with hardware products
**      sold by PII. Licensee may distribute binary derivative works using this
**      software and running on PII hardware products to third parties without
**      fee or other restrictions.
**
**      PII MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE
**      SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
**      IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE,
**      OR NON-INFRINGEMENT. PII SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY
**      LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE OR
**      ITS DERIVATIVES.
**
**      By using or copying this Software, Licensee agrees to abide by the
**      copyright law and all other applicable laws of the U.S. including, but
**      not limited to, export control laws, and the terms of this license. PII
**      shall have the right to terminate this license immediately by written
**      notice upon Licensee's breach of, or non-compliance with, any of its
**      terms. Licensee may be held legally responsible for any copyright
**      infringement or damages resulting from Licensee's failure to abide by
**      the terms of this license.
**
\*****************************************************************************/

#include    <cfxbios.h>         // Persistor BIOS and I/O Definitions
#include    <cfxpico.h>         // Persistor PicoDOS Definitions

#include    <assert.h>
#include    <ctype.h>
#include    <errno.h>
#include    <float.h>
#include    <limits.h>
#include    <locale.h>
```

```
#include	<math.h>
#include	<setjmp.h>
#include	<signal.h>
#include	<stdarg.h>
#include	<stddef.h>
#include	<stdio.h>
#include	<stdlib.h>
#include	<string.h>
#include	<time.h>

#include	<dirent.h>		// PicoDOS POSIX-like Directory Access Defines
#include	<dosdrive.h>		// PicoDOS DOS Drive and Directory Definitions
#include	<fcntl.h>		// PicoDOS POSIX-like File Access Definitions
#include	<stat.h>		// PicoDOS POSIX-like File Status Definitions
#include	<termios.h>		// PicoDOS POSIX-like Terminal I/O Definitions
#include	<unistd.h>		// PicoDOS POSIX-like UNIX Function Definitions

#define MAX_STR	256

/*****************************************************************************\
**	main
\*****************************************************************************/
int main(int argc, char **argv)
	{
	short	i;
	short	result = 0;		// no errors so far
	uchar	*ptr = NULL;
	uchar	str[MAX_STR];

	DosSwitch	hsw = {0}, hsw_ = { "/-", 'H', 0, 0 };
	DosSwitch	osw = {0}, osw_ = { "/-", 'O', 0, 0 };
	DosSwitch	vsw = {0}, vsw_ = { "/-", 'V', 0, "%ld", true };

	CmdInfo		MyCmd = {0};
//	CmdInfoPtr	MyCmdPtr = &MyCmd;

	// Identify the progam and build
	printf("\nProgram: %s: %s %s \n", __FILE__, __DATE__, __TIME__);
	// Identify the device and its firmware
	printf("Persistor CF%d SN:%ld  BIOS:%d.%d  PicoDOS:%d.%d\n", CFX,
		BIOSGVT.CFxSerNum, BIOSGVT.BIOSVersion, BIOSGVT.BIOSRelease,
		BIOSGVT.PICOVersion, BIOSGVT.PICORelease);
	// Identify the arguments
	printf("\n%d Arguments:\n", argc);
	for (i = 0; i < argc; i++)
		printf("  argv[%d] = \"%s\"\n", i, argv[i]);

	while(QRstring("\n\nEnter Command: ", "%s", false, str, MAX_STR))
	{

		//Initilize DOS Switches from template
		memcpy(&hsw, &hsw_, sizeof(DosSwitch));
		memcpy(&osw, &osw_, sizeof(DosSwitch));
		memcpy(&vsw, &vsw_, sizeof(DosSwitch));
```

```
            //Setup our structure
            CmdStdSetup(&MyCmd,0,0);

            //Copy over the text we have grabbed
            strcpy(MyCmd.line, str);

            //Parse
        CmdParse(&MyCmd);

            //Extract the switches
            CmdExtractCIDosSwitches(&MyCmd, "VHO", &vsw, &hsw, &osw);

            //Process Commands here
            if(strcmp(MyCmd.argv[0].str,"D2H\0")==0)
            {
                    if(vsw.pos)             //If pos is non-zero then the switch was found (/V means
we have a value)
                    {
                            cprintf("\nThe value %ld (decimal) is %lX
(hexadecimal)",vsw.lv,vsw.lv);
                            if(osw.pos)
                            {
                                    cprintf(" and %lo (Octal)",vsw.lv);
                            }
                            cdrain();
                            cprintf("\n\n");
                    }
            }
    }


// Various exit strategies
//      BIOSReset();                // full hardware reset
//      BIOSResetToPBM();           // full reset, but stay in Pesistor Boot Monitor
//      BIOSResetToPicoDOS();       // full reset, but jump to 0xE10000 (PicoDOS)
        return result;

        }       //____ main() ____//
```

PicoDAQ

Introduction

PicoDAQ is a data logging program for the Persistor Instruments CF2 that provides a simple method for Scientists and Engineers to sample and store data. PicoDAQ uses a simple command line interface which can be tailored to the needs of the user. You do not have to be a computer programmer to use PicoDAQ, in fact, PicoDAQ was written for non-programmers to make data acquisition easier. All that is needed is to add sensors and interface circuits and PicoDAQ is ready to go!

Using PicoDAQ

The CF2 has no on-board A/D but Persistor Instruments makes Recipe Cards which contain A/D converters. A recipe card can have a 12 bit or a 16 bit A/D converter. The A/Ds have 8 inputs. This allows PicoDAQ to make measurements with up to 8 channels. Each channel of analog data is converted to a 16 bit binary value (even for 12 bit A/Ds). This data is then stored to Compact Flash for later offloading and processing.

Command

Standard PicoDAQ can be run from the command prompt under PicoDOS. The format of the command is:

-FFILENAME -R|H< ratelist> -D<delay> -S< samplems> -I -P -2 -B -M -A -L -V -?

FILENAME		an 8 character filename (no space between –F and filename)
-R|H	<ratelist>	Channel Rate Divisors
-D	<delay>	Start Delay (0-65535 milliseconds)
-S	<samplems>	Sample Duration (1-2147483647 milliseconds)
-I		IRQ5 (pin 39) is start/stop pin
-P		Bipolar Conversions
-2		Differential Sampling
-B		Break Detect Start/Stop
-M		Motorola Format
-A		ASCII Format
-L		LED Off (defaults to on)
-Q		Single Binary File
-V		Verbose diagnostics
-?		Display usage

Filenames

Files are created with an extension that indicates what kind of data is stored in the card. Up to 8 characters can be used for a filename. The extension of the files that will be stored can be Pan, PLn, or ALL. The ‘n’ is the number representing the A/D channel 0-7.

Rates

No matter how many channels are selected or at what rate of sampling is requested, all 8 channels are sampled at 1000 samples per second. When a user is entering rate specifications for a channel you are really specifying how often A/D data is stored NOT how often it is sampled.

Rates can be entered as desired samples per second (-R) or as a skip count (-H). A skip count of 2 means to store every other sample, or, put another way, to wait until 2 samples are collected before storing one. This means that a skip count of 2 means to sample at 500 samples per second. A skip count of 1 means to store every sample for the given channel. This would be a sample rate of 1 kHz.

If the rate is entered as frequency the user must be aware of a reality of skip counts: odd sample rate will not be allowed. For example, a skip count of 2 is 500 Hz and a skip count of 1 is 1000 Hz. There is no skip count that will yield a sample rate of 750 Hz. If 750 Hz is entered as the sampling frequency, PicoDAQ will round up to the next higher available frequency, in this case, 1 kHz. See the table in Appendix C for skip counts and frequencies.

Start Delay

A Start Delay can be indicated on the command line. The format is –D and the delay from 0 to 65535 milliseconds.

Controlling Collection

There are several ways to control the start and end of data collection. Duration for logging can be specified on the command line. The syntax is –S and the duration in milliseconds (1-2147483647).

Another way to control logging is with a BREAK on the serial port. A BREAK sent to PicoDAQ will both start and stop data collection. Controlling logging with BREAK is indicated on the command line with a –B.

The last way to control data collection is with pin 39 or IRQ5. This is also the same pin used to get into PBM mode at power up. If you are using a Persistor RecipeCard there is a button available for this making it convenient to use. Controlling logging with pin 39/IRQ5 is indicated on the command line with a –I.

Binary vs. ASCII

PicoDAQ can store data as binary data (the default) and as ASCII (-A). ASCII may be the preferred format for importing into programs such as Excel. There are some commercial programs that can import the binary data directly. If you have the need to read these files yourself then have a look at Appendix A for an explanation of the binary storage format. There is also an application available on the Persistor Instruments website that will convert binary to ASCII (www.persistor.com).

Big Endian or Little Endian

Two bytes form a 16 bit A/D value. When this data is stored it can be in two different formats: most significant byte first or, last. Engineers designing computer systems had a choice to store high byte first or low bytes first. The order is usually dictated by the architects of the machine. Intel machines have usually stored the data with the low or least significant byte first. Motorola machines usually store the data with the high or most significant byte first. When the high byte is stored first it is referred to as Big Endian and when it is stored low byte first it is referred to as Little Endian (this naming convention is from the book Gulliver's Travels).

What does Big and Little Endian have to do with logging decisions? Well, the CF2 is a Motorola machine and so therefore it stores the data Big Endian. When you move the data to a PC based on an Intel processor (not a Mac but a IBM PC or compatible) the data will be backwards. In order to prevent this you can ask PicoDAQ to flip the bytes over so they are stored on the Compact Flash as Little Endian. The story doesn't end here…

Time in the CF2 is at a premium in a fast logging application. In order to save time inside the CF2 you'll want save the data in its native Big Endian format. Later, you can have the conversion software flip the bytes around on the PC when there is time, memory and power available to do it.

Single File Mode

In order to speed up data acquisition you can specify that all data be stored in a single file instead of individual files. Handling multiple files takes extra memory and time. This takes valuable CPU cycles which would otherwise be available for data acquisition. Selecting single file storage can increase the speed and/or the number of channels that can be sampled.

To indicate that a single file is desired a –Q is entered on the command line.

Differential Sampling or Bipolar

Differential Sampling is indicated on the command line with -2. Bipolar measurements are indicated with –P.

Diagnostics

Some diagnostic information is available when using PicoDAQ. To print these extra messages during program execution include a –V on the command line.

Binary Single File Format

For data stored as a single file, the format of the data is binary. This is because it is faster to store data in binary format rather than convert the binary data to ASCII.

When writing data to a single file, the data can be stored in Motorola or Intel format (big endian and little endian respectively). It is faster to store the data in Motorola format because that is the native format of the 68332; a Motorola processor. When storing in Intel format, the processor must flip the bytes before storing. While this may make reading files slightly easier on the PC end it is at the expense of processor cycles. It is far easier to flip the bytes on the PC end because the PC has the luxury of speed and time to do it. This of course is a design decision left to you, the designer.

Another consideration when storing data in a single file format relates to the rates of data collection. If all the channels are recorded at the same rate then the data will be stored contiguously in the file. Table 1 shows the sequence of bytes stored in a single file where all eight A/D channels are sampled at the same rate. The actual rate does not matter, what matters is that the rates for all channels are the same. The channels are all stored in Motorola format: high byte first.

Hex Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Channel 0		Channel 1		Channel 2		Channel 3		Channel 4		Channel 5		Channel 6		Channel 7	
	high	low	high	low	high	low	high	low	high	low	high	low	high	low	high	low
10	Channel 0		Channel 1		Channel 2		Channel 3		Channel 4		Channel 5		Channel 6		Channel 7	
	high	low	high	low	high	low	high	low	high	low	high	low	high	low	high	low
20	Channel 0		Channel 1		Channel 2		Channel 3		Channel 4		Channel 5		Channel 6		Channel 7	
	high	low	high	low	high	low	high	low	high	low	high	low	high	low	high	low
30	Channel 0		Channel 1		Channel 2		Channel 3		Channel 4		Channel 5		Channel 6		Channel 7	
	high	low	high	low	high	low	high	low	high	low	high	low	high	low	high	low

Table 1

When the rates of the channels are not the same but are all written to one file, the format of the file is different than when the rates are the same. First, all channels that are active are sampled and stored for the first time. After that, the A/D data is written to the file in an order that is dictated by the specified rates.

The PicoDAQ program in the CF2 samples all channels at a rate of 1 KHz, i.e. it takes a sample of each of the eight channels once every millisecond. PicoDAQ keeps track of which values to store and when by keeping a skip-counter. A skip-counter value of 1 means to store every sample taken for that channel which is a sample rate of 1 KHz. A skip-counter value of 2 means to store every other sample taken for that channel for a rate of 500 Hz. A skip-counter value of 1000 means to store every one-thousandth sample and because PicoDAQ is sampling at 1000 times a second then a skip-counter value is a sample rate of once per second or 1 Hz.

Table 2 represents the output file if channel 0 sampled at 4 Hz, channel 1 at 2 Hz, channel 2 @ 1 Hz. All the samples are stored in Motorola format or high byte first (big endian).

Hex Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	Channel 0		Channel 0		Channel 1		Channel 0		Channel 0		Channel 1		Channel 2		Channel 0	
	high	low	high	low	high	low	high	low	high	low	high	low	high	low	high	low
10	Channel 0		Channel 1		Channel 0		Channel 0		Channel 1		Channel 2		Channel 0		Channel 0	
	high	low	high	low	high	low	high	low	high	low	high	low	high	low	high	low
20	Channel 1		Channel 0		Channel 0		Channel 1		Channel 2		Channel 0		Channel 0		Channel 1	
	high	low	high	low	high	low	high	low	high	low	high	low	high	low	high	low
30	Channel 0		Channel 0		Channel 1		Channel 2		Channel 0		Channel 0		Channel 1		Channel 0	
	high	low	high	low	high	low	high	low	high	low	high	low	high	low	high	low

Table 2

Common Programming Questions

Why are there .APP and .RUN files?

APP files are executable programs which target FLASH. FLASH applications are permanent and the CF2 can be programmed to load and run the application from FLASH on power up.

How do I load my .APP and have it run on power up?

Load the .APP file using Motocross (Transfer->Load or F7). After the application has loaded and, at the PicoDOS prompt, type BOOT APP <ENTER>. The next time power is re-applied or the CF2 is RESET your APP will be executed.

How do I run my .APP from PicoDOS

If you have an APP loaded into FLASH and you would like to run it from the PicoDOS prompt just type APP <ENTER>.

How can I load and run a .RUN program stored on a CompactFlash Card?

Load the .RUN file using Motocross (Transfer->Load or F7). After the file has loaded type SAVE filename <ENTER> or GS filename <ENTER>. Next, type DIR and you will see your file on the CompactFlash with the extension .PXE. Now, to run the program, just type the name of the program and press <ENTER>.

How do I get back to PicoDOS from my program?

One way to get back to PicoDOS is to just execute a return from main in your program. You can also execute functions to perform a full RESET of the CF2. Still another way is to force a RESET and return to either PBM or PicoDOS.

The following is from a PicoDOS program created from Persistor Stationery.

```
// Various exit strategies
//      BIOSReset();              // full hardware reset
//      BIOSResetToPBM();         // full reset, but stay in Pesistor Boot Monitor
//      BIOSResetToPicoDOS();     // full reset, but jump to 0xE10000 (PicoDOS)
        return result;
```

I changed my baud rate and now and can't see the PicoDOS prompt at ANY baud Rate. What's up with that?

It is possible to change the baud rate on the CF2 to a rate that is non-standard. Because of this you may not be able to see the characters being produced by the CF2 in Motocross or another terminal communications program. Power the CF2 off and connect a ground lead to pin 39 (or hold PBM on if you are using a Persistor Recipe Card) and re-apply the power. Keep pin 39 grounded and type PICO <ENTER> at the PBM prompt. You should be in PicoDOS now at 9600 bps. Remove the ground on pin 39. Change the baud rate to something else your terminal program can handle.

How can I tell if my CF2 is working properly?

A new CF2 plugged into a Recipe card should power up to the PicoDOS prompt at 9600 bps 8N1. The current will vary a small amount but will go from an initial 20 mA or so to a low value of 1 or 2 mA at the PicoDOS prompt.

If you cannot get the PicoDOS prompt then power the CF2 off and connect a ground lead to pin 39 (or hold PBM on if you are using a Persistor Recipe Card) and re-apply the power. You should now be at the PBM prompt at 9600 bps 8N1. Type BOOT PICO <ENTER>. The next time you RESET or cycle power you should see the PicoDOS prompt.

I can't get a PicoDOS prompt and I tried forcing PBM mode but that didn't work. What do I do now?

Indications of a dead CF2 are not being able to force the PicoDOS or PBM prompt and high current draw (which could be anything from 20 mA to some gigantic value). If you have a red LED lit on the left side of the CompactFlash header and cannot get a PicoDOS or PBM prompt then you have a dead CF2. Many, many things can cause this including mis-wiring of power or static electricity damage. Call Persistor Instruments for instructions on what to do next.

I programmed my application in Flash but now I want to load a new version of it and I cannot. I know I need to get to PicoDOS but I don't know how. What do I do?

Buried within the CF2 is software that sits below PicoDOS. This is the Persistor Boot Monitor or PBM. You can get to PBM when you apply power to the CF2. This can be accomplished by grounding pin 39 (or depressing the PBM button on most RecipeCards) before applying power. Once power is applied you will jump to the PBM. A 60 second countdown will start. You can ignore the countdown for now. Keep pin 39 grounded and type WREN at the PBM prompt and hit Enter. You will then be asked by the PBM to release pin 39 and confirm. Disconnect the ground on pin 39 and type 'Y' and Enter. Now that PBM writes are enabled (WREN), you can ask the PBM to boot to PicoDOS. Type BOOT and hit Enter. You will be given a list of boot addresses with numbers. Type '9' and hit enter. This instructs the PBM to boot PicoDOS at power-up and NOT the application in Flash. The next time you apply power you should see the PicoDOS prompt.

I don't want to load my program into Flash. I would like to have it available to run as a file on the Compact Flash card kind of like DOS program used to run. Can I do this with PicoDOS?

Yes. Sometimes you may have a need for keeping several CF2 applications stored on a Compact Flash card. You would then be able to run them by just typing their names. Load a .RUN file version of your application using Motocross. When the load is complete you will see a 'G' appear on the screen Now add an 'S' to the 'G' so it forms 'GS' and type one space and enter an 8-character filename. Hit Enter when done. If you type 'DIR' and Enter you will get a directory listing. In the list you should see the name you typed with an extension of .PXE (for PicoDOS eXEcutable). To run your application, just type the filename without the extension and hit Enter.

I loaded my program into Flash. How do I make it run whenever I turn on my CF2?

At the PicoDOS prompt type BOOT APP. This instructs the CF2 to run your application out of Flash whenever power is applied. I loaded my program into Flash. How can I tell PicoDOS to run the program in Flash? If you want to just run a program in Flash from PicoDOS, all you have to do is type APP at the PicoDOS prompt

Should I format my Compact Flash on my PC or should I format using PicoDOS?

Because of some nuances with formatting on PCs today, we recommend formatting Compact Flash cards using PicoDOS.

When I apply power to my CF2, I can't get a PicoDOS prompt to appear. How do I fix this?

This could be one of several different problems. Make sure that you are running a communications program (we will assume Motocross) and that the CF2 is connected to an available COM port on the PC. Make sure that Motocross is using the correct COM port (the one the CF2 is connected to). Make sure that the baud rates are the same for the CF2 and Motocross (CF2 default baud rate is 9600 8N1). Check to see that power is getting to the CF2. You may have a program running in the CF2 that prevents PicoDOS from running. The solution to this is explained in an earlier problem/solution.

What voltage should I use for CF2 development?

Keeping the voltage low whenever you are developing is a good way to help minimize the damage that can occur if you were to momentarily short a pin while you probe the board. We develop with a current limited supply. We keep the voltage at 4 volts and the current limited to about 100-150 mA.