so we cannot just reset them to inputs in software. However, because of the flexibility of the PIO, we can easily hack some other signals on the Applications board to get some digital inputs.

The Application board has four PCB pads (plated holes in the PCB board) designated for use with serial communications or as general purpose extra I/O. These pads are visible in Figure 6 as the rightmost four of the six white dots just above the connector “Analog In Bank 1”. We have attached wires and brought them out to BNC jacks and a loose wire on your clear front panel labeled Digital Input 0 and 1. **Note that since these signals connect direct to the Sam7 pins, you must not put more than 3.3 volts onto them or you could easily burn out the Make Controller ($69).**

The signals we are using are

- Pad TX / GPIO0 (BNC Jack “Tx”)
- Pad Rx / GPIO1 (BNC Jack “Rx”)
- Pad RTS / GPIO2 (Black Wire)
- Pad CTS / CPIO3 (White Wire)

Using the procedure above, you can configure them as inputs and read data from them or configure them as outputs. They can also be set up to generate interrupts from external signals.

### 9.3 Using Digital I/O with the Make Controller API

The lowest level of the Make Controller API is `io.c` / `io.h` in the “Core” modules. The functions in `io.c` perform the same tasks described in Section 9.2 and in addition, provide an optional locking mechanism to prevent different modules of software from accessing pins at the same time.

To use the Make API to set up and operate pin PA02 as an output, as we did in Section 9.2, we would do:

```c
#include io.h
#define 472_OUTPUT 1
#define 472_INPUT 0

//
// Initialization
//
Io_SetPio(IO_PA02, true);  // Allow the PIO to control the PA02 pin
Io_SetDirection(IO_PA02, 472_OUTPUT)  // Set PA02 as an output

...

//
// example use within a task to generate a 100ms pulse.
//
Io_SetValue(IO_PA02, 1);  // set the output to 1, 3.3V
Sleep(100);  // delay 100 ms.
Io_SetValue(IO_PA02, 0);  // set the output to 0.
```

The constant `IO_PA02` is a `#define` in `io.h` which evaluates to 2, the index (not the bit mask!) of bit PA02. Here we have not used the locking mechanism. In EE472 we will not need the locking mechanism because a single hardware device will be hooked up to each I/O pin that we use, and only one task will be accessing that pin. For those interested, the functions to use the locking mechanism are `Io_GetActive`, `Io_Start`, `Io_Stop`.

### 10 Timers

Timers are basically digital counters. Logic is provided to give the counters lots of capabilities. The logic
• Allows different signals to be used as the counter's clock including clock oscillators of different frequencies, and external signals which you might want to count.

• Provides flexible modes for checking when the count has reached one or more thresholds.

There are several timers within the SAM7

• the Real Time Timer (RTT) (Chapter 15)
• Periodic Interval Timer (PT) (Chapter 16)
• Watchdog Timer (WDT) (Chapter 17)
• Timer Counter (TC) (Chapter 33)
• Pulse Width Modulation Controller (PWM) (Chapter 34)

10.1 Low Level Programming of Timers

10.2 Timers in the Sam7

This section will describe how to use the three Timer/Counters (TC) in the Atmel Sam7 microcontroller. The timers are extensively documented in Chapter 33 of AT91SAM7X_doc6120.pdf.

Perhaps the first and most basic use for a timer/counter is to measure an interval of time. We want the input to the counter to come from a regular clock signal at constant frequency. Then the value of the counter will be our measure of time. The TC hardware can compare this value to our desired time interval. As with digital I/O, our strategy will first be to set up the control registers so that the TC system does what we want, and then interact with the timer to set and monitor its operation.

We will set the timer to run with an internal clock and start it with a software trigger (when we set a bit). Then it will run up until it hits the value stored in Register C. At that point we will configure it to set the TIOA bit, and reset. Our software will have to detect this bit (by reading the Timer Status register).

The TC contains three identical timer counter units called “channels”. Because these channels have so many powerful features, their setup and control is a bit complex. By focusing on just a couple of features we will cut down this complexity. Unfortunately, there is no substitute for reading up on every bit in every register so that we can set them up right. So grab a latte, open up AT91SAM7X_doc6120.pdf to page 387 and read the Overview (33.1) and study Figures 33-1, 33-2, 33-3, and 33-5 (we will not use “Capture mode” at this time).

For naming, we refer to AT91SAM7X256.h which contains the following constants for the addresses of the registers of TC0 and the two block registers which control all three timers:

```c
// ========= Register definition for TC0 peripheral =========
#define AT91C_TC0_SR ((AT91_REG *) 0xFFFA0020) // (TC0) Status Register
#define AT91C_TC0_RC ((AT91_REG *) 0xFFFA001C) // (TC0) Register C
#define AT91C_TC0_RB ((AT91_REG *) 0xFFFA0018) // (TC0) Register B
#define AT91C_TC0_CCR ((AT91_REG *) 0xFFFA0000) // (TC0) Channel Control Register
#define AT91C_TC0_IER ((AT91_REG *) 0xFFFA0024) // (TC0) Interrupt Enable Register
#define AT91C_TC0_RA ((AT91_REG *) 0xFFFA0014) // (TC0) Register A
#define AT91C_TC0_IDR ((AT91_REG *) 0xFFFA0000) // (TC0) Interrupt Disable Register
#define AT91C_TC0_CV ((AT91_REG *) 0xFFFA0001) // (TC0) Counter Value
#define AT91C_TC0_IMR ((AT91_REG *) 0xFFFA002C) // (TC0) Interrupt Mask Register
...

// ========= Register definition for TCB peripheral =========
#define AT91C_TCB_BMR ((AT91_REG *) 0xFFFA00C4) // (TCB) TC Block Mode Register
```
Now let’s look at the registers one-by-one starting on page 402:

There are two classes of registers, two “Block” registers which control all three timers, and 10 Timer Control registers which are specific to one of the three timers.

### Block Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer Counter Block Control Register</td>
<td>AT91C.TCB.BCR</td>
<td>Only 1 bit is used, to trigger all three timers simultaneously.</td>
</tr>
<tr>
<td>Timer Counter Block Mode Register</td>
<td>AT91C.TCB.BMR</td>
<td>Determines which external pins might be connected as clock inputs.</td>
</tr>
</tbody>
</table>

### Timer 0 Control Registers

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel Control Register</td>
<td>AT91C.TC0.CCR</td>
<td>Controls some features of the timer/counter</td>
</tr>
<tr>
<td>Channel Mode Register</td>
<td>AT91C.TC0.CMR</td>
<td>Selects a large set of features.</td>
</tr>
<tr>
<td>Counter Value Register</td>
<td>AT91C.TC0.CV</td>
<td>Value of the 16 bit counter</td>
</tr>
<tr>
<td>Register A</td>
<td>AT91C.TC0.RA</td>
<td>First compare value.</td>
</tr>
<tr>
<td>Register B</td>
<td>AT91C.TC0.RB</td>
<td>Second compare value.</td>
</tr>
<tr>
<td>Register C</td>
<td>AT91C.TC0.RC</td>
<td>Third compare value.</td>
</tr>
<tr>
<td>Status Register</td>
<td>AT91C.TC0.SR</td>
<td>Status bits</td>
</tr>
<tr>
<td>Interrupt Enable Register</td>
<td>AT91C.TC0.IER</td>
<td>Enable interrupts from timer signals.</td>
</tr>
<tr>
<td>Interrupt Disable Register</td>
<td>AT91C.TC0.IDR</td>
<td>Disable interrupts from timer signals.</td>
</tr>
<tr>
<td>Interrupt Mask Register</td>
<td>AT91C.TC0.IMR</td>
<td>Interrupt Enable status of each signal</td>
</tr>
</tbody>
</table>

(replace the ‘0’ in ‘TC0’ to switch to timer 1 or 2.) We will use `#defines` to access each bit in these (or any) registers based on:

```c
#define Bit0 1
#define Bit1 2
#define Bit2 4
#define Bit3 8
#define Bit4 16
...
#define Bit31 1073741824
#define Bit32 2147483648
```

First, the timer must be turned on. To allow power savings, the Sam7 has a Power Management Controller (PMC) which can turn off some of the peripherals if they are not in use. By “Turn Off” we mean just blocking the clock signal to that peripheral. Most of the power is consumed when bits change state at clock transitions (not in between clock edges when the transistors are off). The PMC controls power by just gating the clocks to each peripheral. The PMC Peripheral Clock Enable Register (`AT91C_PMC_PMER`) is a write-only register which contains a bit for each peripheral. If you set that bit, the clock for that peripheral is enabled. The bits for the 3 TCs are

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit12, TC 0</td>
<td></td>
</tr>
<tr>
<td>Bit13, TC 1</td>
<td></td>
</tr>
<tr>
<td>Bit14, TC 2</td>
<td></td>
</tr>
</tbody>
</table>

Since we have plenty of power, lets just turn them all on:

```c
#include "AT91SAM7X256.h"

*AT91C_PMC_PMER = Bit12|Bit13|Bit14;       // Turn on all three timer/counters
```

Note that since `PMC_PMER` is an “enable” register, setting our three bits turns on the three timers but does NOT turn off other devices. To turn off a device use the disable register: `PMC_PCDR`. 
### 10.3 Timer Register Setup

Now, let’s go through the registers one by one and see what we have to do to set them up (which bits we have to set). To get specific, we’ll use as an example a 1ms clock generated in Waveform mode.

1. **AT91C_TCB_BCR** Nothing needed here.

2. **AT91C_TCB_BMR** We need to set each timer to \{0,1\} because we will not use external signals. The value we will write to this register is therefore: Bit0|Bit2|Bit4.

3. **AT91C_TCO_CMR** Here’s the big one: We have the following fields and their settings:

<table>
<thead>
<tr>
<th>Bit Range</th>
<th>Name</th>
<th>values</th>
<th>bits</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCCLKS</td>
<td>0,1,0</td>
<td>(0,1,2)</td>
<td></td>
<td>select timer clock 3 (668ns)</td>
</tr>
<tr>
<td>CLK1</td>
<td>0</td>
<td>(3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BURST</td>
<td>0,0</td>
<td>(4,5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPCSTOP</td>
<td>0</td>
<td>(6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPCDIS</td>
<td>0</td>
<td>(7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EEVTEDG</td>
<td>0,0</td>
<td>(8,9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EEVT</td>
<td>0,0</td>
<td>(10,11)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENETRG</td>
<td>0</td>
<td>(12)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WAVESEL</td>
<td>0,1</td>
<td>(13,14)</td>
<td></td>
<td>Up mode, with autotrig on RC</td>
</tr>
<tr>
<td>WAVE</td>
<td>1</td>
<td>(15)</td>
<td></td>
<td>Use counter in Waveform Mode.</td>
</tr>
<tr>
<td>ACPA</td>
<td>1,0</td>
<td>(16,17)</td>
<td></td>
<td>set TIOA bit on RA value reached.</td>
</tr>
<tr>
<td>ACPC</td>
<td>0,1</td>
<td>(18,19)</td>
<td></td>
<td>clear TIOA bit on RC value reached.</td>
</tr>
<tr>
<td>AEEVT</td>
<td>0,0</td>
<td>(20,21)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASWTRG</td>
<td>0,0</td>
<td>(22,23)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCPB</td>
<td>0,0</td>
<td>(24,25)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BCPC</td>
<td>0,0</td>
<td>(26,27)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEEVT</td>
<td>0,0</td>
<td>(28,29)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSWTRG</td>
<td>0,0</td>
<td>(30,31)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Because there are only a few ones, we have a fairly simple bitmask of: Bit1|Bit14|Bit16|Bit19.

Each timer can use one of five internal clocks. They are documented in the Make Controller API as follows:

*The Master Clock frequency (MCK) is 47923200*

- DIV1: A tick MCK/2 times a second (every 41.73ns)
- DIV2: A tick MCK/8 times a second (every 167ns)
- DIV3: A tick MCK/32 times a second (every 668ns)
- DIV4: A tick MCK/128 times a second (every 2.671us)
- DIV5: A tick MCK/1024 times a second (every 21.368us)

4. **AT91C_TCO_CV** read-only.

5. **AT91C_TCO_RC** We’ll load this with our period value, PERIOD\_VALUE. Since we are using clock3 (DIV3: above), compute the number of ticks of clock 3 in 1ms and define PERIOD\_VALUE to your answer.

6. **AT91C_TCO_RA** set this to PERIOD\_VALUE−2 so that we get a short pulse at end of interval.

7. **AT91C_TCO_RB** not needed.

8. **AT91C_TCO_SR** This is read-only. We’ll read this during operation to check if the RC counter has set the MTIOA bit (BIT17).

9. **AT91C_TCO_IER** not used until we use interrupts later.
10. AT91C_TC0_IDR let’s set all bits to '1' to make sure no interrupts are generated: Bit0|Bit1 ... Bit7.

11. AT91C_TC0_IMR not needed.

12. AT91C_TC0_CCR Now that everything is set up, we set a couple of bits in this register to enable the clock and trigger the start of counting: Bit0|Bit2.

So, here’s the code fragment which sets up the timers as described above:

```c
#include "AT91SAM7X256.h"
#define PERIOD_VALUE <<YOUR RESULT HERE>>

// note that you should comment these bits or use appropriate symbolic constants
// from the .h file above
*AT91C_TCB_BMR = Bit0|Bit2|Bit4;
*AT91C_TCO_CMRI = Bit1|Bit14|Bit15|Bit16|Bit19;
*AT91C_TCO_RC = PERIOD_VALUE; // (we don’t know this number yet!)
*AT91C_TCO_RA = PERIOD_VALUE-2;
*AT91C_TCO_IDR = 0xFF; // disable all timer interrupts.

// set enable to turn the thing on and trigger counting:
*AT91C_TCO_CCR = Bit2|Bit0;
```

Now the timer is set up. If we want to wait for a time interval to be over, we can go into an infinite loop checking Bit17 of AT91C_TCO_SR. Bit 17 is the bit which is set and cleared by the timer values.

## 11 Control Concepts

### 11.1 Pulse Width Modulation (PWM)

Pulse width modulation means varying the width of a periodic digital pulse to approximate a continuously variable output. PWM waveforms are illustrated in Figure 7.

The period of each waveform is the same but the fraction of the time the waveform is at its high value varies. For example, at 50% duty cycle, the waveform is on for half of the period and off for the other half. Also shown is a dashed horizontal line which gives the average value of each PWM waveform. Of course, a signal which is 0 all the time has a 0% duty-cycle and one that is high all the time has a 100% duty cycle.

Using such a pulse-width modulated signal, if the switching period is much shorter than the time constant of the motor, we can control the voltage applied to a DC motor, because the PWM signal is effectively averaged by the motor. Another way to look at it is in the frequency domain. The Fourier Transform of the PWM signal (which we will skip here) contains a DC term equal to the duty cycle times the amplitude of the digital waveform. The higher frequency components are at multiples of the switching frequency — above the bandwidth of the motor — so they are greatly attenuated. The motor “sees” just the average DC value of the PWM waveform. To run the motor at full power we output a digital signal with a 100% duty-cycle, to run the motor at half power we output a digital signal with a 50% duty-cycle, etc. Note that motor speed does not follow a 1:1 relationship with applied voltage or PWM value. That is because the speed also depends on the load (including the motor’s own inertia and friction).

We can use a timer (see Section 10) to create a PWM signal with a digital output port from our microcontroller. For example, suppose that we want to implement a PWM signal with a 75% duty-cycle and we have chosen to use 100ms for the signal’s period. We can implement the signal as follows: we turn a digital output on and wait for 75ms using our timer. Then, we turn the digital output off and pause for 25ms. We then repeat this process over and over to generate the 10Hz, 75% duty-cycle PWM signal.