02-1 Pulse Width Modulation

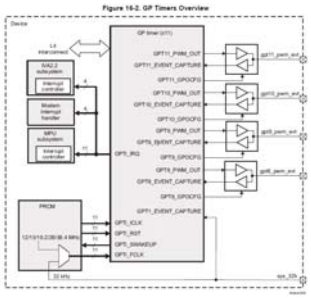
Controlling an output pin without using the CPU

Pulse Width Modulation

- Using the CPU to toggle an IO pin is a poor use of the CPU
- A 1 GHz process can only toggle at about
  - 100 Hz using the shell, or
  - _____ using a C program
- Many applications could use such a signal
  - at a higher frequency
  - without using so much of the CPU
- Use PWM hardware

General Purpose Timers

- The DM3730 has 11 general purpose timers
- 4 of which (gpt8-gpt11) can be brought out of the chip and used for pulse width modulation (DM3730TRM page 2689)



Pin MUXing

- Problem: DM3730 has more internal lines than hardware IO pins.
- Solution: IO pins run though a MUX
  - …selects which internal lines appear on IO pins
- A pin can have 1 from as many as 8 lines assigned to it
- MUXes are set at boot time*
  - must be set when the kernel boots, or
  - in u-boot
- I couldn't set them during kernel boot with the 2.6.32 kernel, so I used u-boot.

Interfacing with timers

- Standard way to interface with the outside world in Linux is through Kernel Drivers (**`/sys`**)
- No standard PWM drivers for the Beagle, though couple have been proposed ([1], [2] and [3])
- A more traditional MCU approach by accessing the memory mapped PWM registers directly using **mmap** in a C program
- Although this approach works, it is really transitional until a standard can be established

Mini Project

- Work up notes on how to do PWM from a shell command by using devmem2 to write to the memory mapped registers from a command line

PWM via mmap (dm3730-pwm-demo.c)

Line 59

```
main(int argc, char **argv) {
    int frequency;
    if (argc < 2) {
        printf("Usage: %s <frequency>\n", argv[0]);
        printf("Sets up a PWM signal on gpio 145 and 146 at the given frequency\n");
        printf("and sweeps the duty-cycle up and down once\n");
            exit(-1);
        }

    frequency = atoi(argv[1]);
    int mem_fd = pwm_open_devmem();
    if (mem_fd == -1) {
        g_error("Unable to open /dev/mem, are you root?: %s",
                g_strerror(errno));
    }
```

PWM via mmap (dm3730-pwm.c)

Line 188

```
int
pwm_open_devmem(void) {
        check_pagesize();
        return open("/dev/mem", O_RDWR | O_SYNC);
}
```

```
// The default Linux page size is 4k and the GP timer register
// blocks are aligned to 4k.  Therefore it is convenient to just
// assume that pages are aligned there for the purposes of mmap()
// (since mmap only maps aligned pages).  This function checks
// that assumption and aborts if it is untrue.
static void
check_pagesize(void) {
    if (getpagesize() != 4096) {
        g_error("The page size is %d.  Must be 4096.", getpagesize());
    }
}
```

Line 93

dm3730-pwm-demo.c

Line 75

```
// Set instances 10 and 11 to use the 13 Mhz clock
    pwm_config_clock(mem_fd, TRUE, TRUE);
    guint8 *gpt10 = pwm_mmap_instance(mem_fd, 10);
    guint8 *gpt11 = pwm_mmap_instance(mem_fd, 11);

    // Get the resolution for 20 kHz PWM
    guint32 resolution = pwm_calc_resolution(frequency,
                            PWM_FREQUENCY_13MHZ);
    printf("Resolution = %d\n", resolution);
```

dm3730-pwm.c

```
// Configure the clocks for GPTIMER10 and GPTIMER11, which can be set to
// use the 13 MHz system clock (otherwise they use the 32 kHz clock like
// the rest of the timers).  Return -1 on failure, with errno set.
int
pwm_config_clock(int mem_fd,gboolean gptimer10_13mhz,
                            gboolean gptimer11_13mhz)
{
    int page_addr    = CM_CLKSEL_CORE & 0xfffff000;
    int offset       = CM_CLKSEL_CORE & 0xfff;

    guint8 *registers = mmap(NULL, 4096, PROT_READ |
       PROT_WRITE, MAP_SHARED, mem_fd, page_addr);
    if (registers == MAP_FAILED) {
        return -1;
    }
…
}
```

Line 140

dm3730-pwm.c

```
pwm_config_clock(int mem_fd, gboolean
gptimer10_13mhz, gboolean gptimer11_13mhz)
{
…
    guint32 value = *REG32_PTR(registers, offset);
    value &= ~(    CLKSEL_GPT10_MASK |
                   CLKSEL_GPT11_MASK);
    if (gptimer10_13mhz) value |= CLKSEL_GPT10_MASK;
    if (gptimer11_13mhz) value |= CLKSEL_GPT11_MASK;
    *REG32_PTR(registers, offset) = value;

    return munmap(registers, 4096);
}
```

Line 140

dm3730-pwm.c

```
// Clock configuration registers (TRM p. 470)
#define CM_CLKSEL_CORE 0x48004A40
#define CLKSEL_GPT10_MASK (1 << 6)
#define CLKSEL_GPT11_MASK (1 << 7)

// GPTIMER register offsets
#define GPT_REG_TCLR 0x024
#define GPT_REG_TCRR 0x028
#define GPT_REG_TLDR 0x02c
#define GPT_REG_TMAR 0x038

// Get a guint32 pointer to the register in block `instance` at byte
// offset `offset`.
#define REG32_PTR(instance, offset) ((volatile guint32*) (instance + offset))


// General purpose timer instances.  Not all of these can actually be
// used for PWM --- see the TRM for more information.
static guint32 gpt_instance_addrs[] = {
    0x4903e000, // GPTIMER8
    0x49040000, // GPTIMER9
    0x48086000, // GPTIMER10
    0x48088000, // GPTIMER11
};
```

Line 79

Page 460 of TRM

Table 3-134. CORE_CM Register Summary (continued)

Register Name	Type	Register Width (Bits)	Address Offset	Physical Address	Reset Type
CM_FCLKEN3_CORE	RW	32	0x0000 0008	0x4800 4A08	W
CM_ICLKEN1_CORE	RW	32	0x0000 0010	0x4800 4A10	W
Reserved for non-GP devices.	RW	32	0x0000 0014	0x4800 4A14	W
CM_ICLKEN3_CORE	RW	32	0x0000 0018	0x4800 4A18	W
CM_IDLEST1_CORE	R	32	0x0000 0020	0x4800 4A20	C
Reserved for non-GP devices.	R	32	0x0000 0024	0x4800 4A24	C
CM_IDLEST3_CORE	R	32	0x0000 0028	0x4800 4A28	C
CM_AUTOIDLE1_CORE	RW	32	0x0000 0030	0x4800 4A30	W
Reserved for non-GP devices.	RW	32	0x0000 0034	0x4800 4A34	W
CM_AUTOIDLE3_CORE	RW	32	0x0000 0038	0x4800 4A38	W
CM_CLKSEL_CORE	RW	32	0x0000 0040	0x4800 4A40	W
CM_CLKSTCTRL_CORE	RW	32	0x0000 0048	0x4800 4A48	W
CM_CLKSTST_CORE	R	32	0x0000 004C	0x4800 4A4C	C

Page 471 of TRM

Table 3-151. CM_CLKSEL_CORE

Address Offset	0x0000 0040
Physical Address	0x4800 4A40 Instance CORE_CM
Description	CORE modules clock selection.
Type	RW

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6	5 4	3 2	1 0
RESERVED		CLKSEL_96M RESERVED	CLKSEL_GPT11 CLKSEL_GPT10	RESERVED	CLKSEL_L4	CLKSEL_L3

Bits	Field Name	Description	Type	Reset
31:14	RESERVED	Write 0s for future compatibility. Read returns 0.	R	0x00000
13:12	CLKSEL_96M	Selects the 96 MHz clock; Other enums: Reserved. 0x1: The clock 96 MHz is the DPLL4_M2_CLK divided by 1 0x2: The clock 96 MHz is the DPLL4_M2_CLK divided by 2	RW	0x1
11:8	RESERVED	Write the reset value, read returns reset value.	RW	0x1

dm3730-pwm.c

```
// Clock configuration registers (TRM p. 470)
#define CM_CLKSEL_CORE 0x48004A40
#define CLKSEL_GPT10_MASK (1 << 6)
#define CLKSEL_GPT11_MASK (1 << 7)

// GPTIMER register offsets
#define GPT_REG_TCLR 0x024      // Optional Features
#define GPT_REG_TCRR 0x028      // Current time value
#define GPT_REG_TLDR 0x02c      // Load Register
#define GPT_REG_TMAR 0x038      // Match Register

// Get a guint32 pointer to the register in block `instance` at byte
// offset `offset`.
#define REG32_PTR(instance, offset) ((volatile guint32*) (instance + offset))


// General purpose timer instances.  Not all of these can actually be
// used for PWM --- see the TRM for more information.
static guint32 gpt_instance_addrs[] = {
    0x4903e000, // GPTIMER8
    0x49040000, // GPTIMER9
    0x48086000, // GPTIMER10
    0x48088000, // GPTIMER11
};
```

Line 79

Page 2710 of TRM

16.3 General-Purpose Timers Register Manual

16.3.1 GP Timer Register Map

16.3.1.1 Instance Summary

Table 16-12 lists the base address and block size for the GP timer module instances. All timers are memory mapped to the L4 peripheral bus memory space.

Table 16-12. GP Timer Instance Summary

Module Name	Base Address	Size
GPTIMER1	0x4831 8000	4K bytes
GPTIMER2	0x4903 2000	4K bytes
GPTIMER3	0x4903 4000	4K bytes
GPTIMER4	0x4903 6000	4K bytes
GPTIMER5	0x4903 8000	4K bytes
GPTIMER6	0x4903 A000	4K bytes
GPTIMER7	0x4903 C000	4K bytes
GPTIMER8	0x4903 E000	4K bytes
GPTIMER9	0x4904 0000	4K bytes
GPTIMER10	0x4808 6000	4K bytes
GPTIMER11	0x4808 8000	4K bytes

Page 2713 of TRM

Table 16-15. GPTIMER9 to GPTIMER11 Register Summary

Register Name	Type	Register Width (Bits)	Address Offset	Physical Address (GPTIMER9)	Physical Address (GPTIMER10)	Physical Address (GPTIMER11)
TIDR	R	32	0x000	0x4904 0000	0x4808 6000	0x4808 8000
TIOCP_CFG	RW	32	0x010	0x4904 0010	0x4808 6010	0x4808 8010
TISTAT	R	32	0x014	0x4904 0014	0x4808 6014	0x4808 8014
TISR	RW	32	0x018	0x4904 0018	0x4808 6018	0x4808 8018
TIER	RW	32	0x01C	0x4904 001C	0x4808 601C	0x4808 801C
TWER	RW	32	0x020	0x4904 0020	0x4808 6020	0x4808 8020
TCLR	RW	32	0x024	0x4904 0024	0x4808 6024	0x4808 8024
TCRR	RW	32	0x028	0x4904 0028	0x4808 6028	0x4808 8028
TLDR	RW	32	0x02C	0x4904 002C	0x4808 602C	0x4808 802C
TTGR	RW	32	0x030	0x4904 0030	0x4808 6030	0x4808 8030
TWPS	R	32	0x034	0x4904 0034	0x4808 6034	0x4808 8034
TMAR	RW	32	0x038	0x4904 0038	0x4808 6038	0x4808 8038
TCAR1	R	32	0x03C	0x4904 003C	0x4808 603C	0x4808 803C
TSICR	RW	32	0x040	0x4904 0040	0x4808 6040	0x4808 8040
TCAR2	R	32	0x044	0x4904 0044	0x4808 6044	0x4808 8044
TPIR	RW	32	0x048	-	0x4808 6048	-
TNIR	RW	32	0x04C	-	0x4808 604C	-
TCVR	RW	32	0x050	-	0x4808 6050	-
TOCR	RW	32	0x054	-	0x4808 6054	-
TOWR	RW	32	0x058	-	0x4808 6058	-

dm3730-pwm-demo.c

Line 75

```
// Set instances 10 and 11 to use the 13 Mhz clock
   pwm_config_clock(mem_fd, TRUE, TRUE);
   guint8 *gpt10 = pwm_mmap_instance(mem_fd, 10);
   guint8 *gpt11 = pwm_mmap_instance(mem_fd, 11);

   // Get the resolution for 20 kHz PWM
   guint32 resolution = pwm_calc_resolution(frequency,
                                PWM_FREQUENCY_13MHZ);
   printf("Resolution = %d\n", resolution);
```

dm3730-pwm.c

```
// Simply a wrapper around mmap that passes the correct arguments
// for mapping a register block.  `instance_number` must be between
// 1 and 12, or errno will be set to EDOM and MAP_FAILED returned.
// Otherwise the return value is that of `mmap()`.
guint8*
pwm_mmap_instance(int mem_fd, int instance_number) {
    if (instance_number < 8 || instance_number > 11)
{
        errno = EDOM;
        return MAP_FAILED;
    }
    int instance_addr =
      gpt_instance_addrs[instance_number - 8];
    return mmap(NULL, 4096, PROT_READ | PROT_WRITE,
      MAP_SHARED, mem_fd, instance_addr);
}
```

Line 108

dm3730-pwm-demo.c

Line 75

```
// Set instances 10 and 11 to use the 13 Mhz clock
   pwm_config_clock(mem_fd, TRUE, TRUE);
   guint8 *gpt10 = pwm_mmap_instance(mem_fd, 10);
   guint8 *gpt11 = pwm_mmap_instance(mem_fd, 11);

   // Get the resolution for 20 kHz PWM
   guint32 resolution = pwm_calc_resolution(frequency,
                          PWM_FREQUENCY_13MHZ);
   printf("Resolution = %d\n", resolution);
```

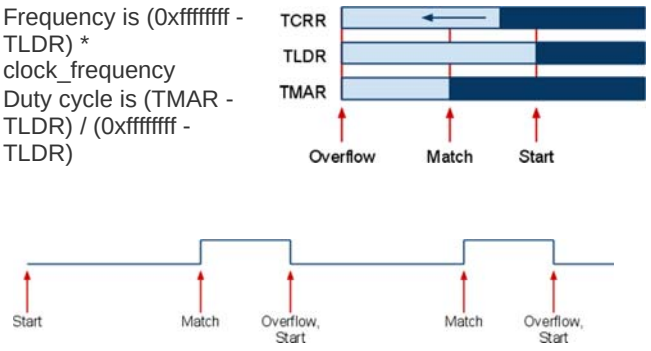
dm3730-pwm.c

```
// Calculate the resolution of the PWM (the number of clock ticks
// in the period), which is passed to `pwm_config_timer()`.
guint32
pwm_calc_resolution(int pwm_frequency, int clock_frequency)
{
    float pwm_period = 1.0 / pwm_frequency;
    float clock_period = 1.0 / clock_frequency;
    return (guint32) (pwm_period / clock_period);
}
```

Line 150

PWM Continued

- Frequency is (0xffffffff - TLDR) * clock_frequency
- Duty cycle is (TMAR - TLDR) / (0xffffffff - TLDR)



dm3730-pwm-demo.c

Line 84

```
  // Ramp up and down a bit
    int i;
    for (i = 0; i <= 100; i++) {
        g_print("%3d\n", i);
        pwm_config_timer(gpt10, resolution, i / 100.0);
        pwm_config_timer(gpt11, resolution, i / 100.0);
        usleep(100000);
    }
    sleep(1);
…

    pwm_munmap_instance(gpt10);
    pwm_close_devmem(mem_fd);
}
```

dm3730-pwm.c

Line 152

```
// Initialize the control registers of the specified timer
// instance for PWM at the specified resolution.
void
pwm_config_timer(guint8 *instance, guint32 resolution,
                 float duty_cycle) {
    guint32 counter_start = 0xffffffff - resolution;
    guint32 dc = 0xffffffff - ((guint32) (resolution * duty_cycle));

   // Edge condition: the duty cycle is set within two units of the overflow
   // value.  Loading the register with this value shouldn't be done (TRM
16.2.4.6).
    if (0xffffffff - dc <= 2) {
        dc = 0xffffffff - 2;
    }
```

dm3730-pwm.c

```
    // Edge condition: TMAR will be set to within two units of the overflow
    // value.  This means that the resolution is extremely low, which doesn't
    // really make sense, but whatever.
    if (0xffffffff - counter_start <= 2) {
        counter_start = 0xffffffff - 2;
    }

    *REG32_PTR(instance, GPT_REG_TCLR) = 0; // Turn off
    *REG32_PTR(instance, GPT_REG_TCRR) = counter_start;
    *REG32_PTR(instance, GPT_REG_TLDR) = counter_start;
    *REG32_PTR(instance, GPT_REG_TMAR) = dc;
    *REG32_PTR(instance, GPT_REG_TCLR) = (
        (1 << 0)  | // ST -- enable counter
        (1 << 1)  | // AR -- autoreload on overflow
        (1 << 6)  | // CE -- compare enabled
        (1 << 7)  | // SCPWM -- invert pulse
        (2 << 10) | // TRG -- overflow and match trigger
        (1 << 12)   // PT -- toggle PWM mode
    );
}
```

Line 185

dm3730-pwm-demo.c

Line 84

```
  // Ramp up and down a bit
    int i;
    for (i = 0; i <= 100; i++) {
        g_print("%3d\n", i);
        pwm_config_timer(gpt10, resolution, i / 100.0);
        pwm_config_timer(gpt11, resolution, i / 100.0);
        usleep(100000);
    }
    sleep(1);
…
 pwm_munmap_instance(gpt10);
 pwm_munmap_instance(gpt11);
 pwm_close_devmem(mem_fd);
}
```