Ajeco ANDI-SERVO Motion Controller Device Driver

Mark Dennehy

July 7, 2010

Abstract

The ANDI-SERVO is a motion controller board for DC Servo motors. This document describes a Linux device driver written for this board, giving a brief introduction to the board itself, the internal structure of the driver and the user-space interfaces provided.

Part I
ANDI-SERVO Motion Control Board

The ANDI-SERVO is a 8-bit PC104 board which can control two DC servo motors. It uses two LM629 motion controllers and two H-Bridges to do so, one of each per motor. The LM629s take either a velocity or a position requirement and generate the appropriate PWM signal, with encoder inputs to create a control feedback loop. Acceleration, velocity profiling and so forth are handled in hardware by the LM629s, thus alleviating the workload on the host CPU (Dagda in this case). Thus the driver's tasks are to communicate with the LM629 chips and provide them with data concerning the PID filter used in the control loop, the trajectory you wish the motor to follow (comprising position, velocity or acceleration), and initialisation of the board. The driver also attempts to track dagda's heading by monitoring the encoder counts for the left and right wheels. Dagda does not use the on-board H-Bridges as we require more power than they can provide so we take the PWM signals off-board from connector P2 to external H-Bridges.

Part II
Internal Driver Structure

The driver structure is based around four data structures. Two of these are derived from the data to be sent to the LM629 chips to control the PID filter and the trajectory followed by the LM629 :

```
/*-------------------------------------------------------------------+
 |    Structure definition for PID filter adjustment                 |
 +-------------------------------------------------------------------*/

struct LM629_Filter
{
    int dterm;                  /* Derivative sampling interval factor  */
    int kp;                     /* Proportional parameter               */
    int ki;                     /* Integrating parameter                */
    int kd;                     /* Derivating parameter                 */
    int il;                     /* Integration limit                    */
};

/*-------------------------------------------------------------------+
 |    Structure definition containing all trajectory parameters      |
 +-------------------------------------------------------------------*/

struct LM629_Trajectory
{
    BOOLEAN forward_dir;        /* TRUE|FALSE (velocity mode only)      */
    BOOLEAN velocity_mode;      /* TRUE|FALSE (velocity|position mode)  */
    BOOLEAN stop_smooth;        /* TRUE|FALSE (smooth|no stop)          */
    BOOLEAN stop_abrupt;        /* TRUE|FALSE (abrupt|no stop)          */
    BOOLEAN motor_off;          /* TRUE|FALSE (motor off|on)            */
    BOOLEAN load_acc;           /* TRUE|FALSE (load acceleration|don't) */
    BOOLEAN load_vel;           /* TRUE|FALSE (load velocity|don't)     */
    BOOLEAN load_pos;           /* TRUE|FALSE (load position|don't)     */
    BOOLEAN acc_relative;       /* TRUE|FALSE (relative acc|absolute)   */
    BOOLEAN vel_relative;       /* TRUE|FALSE (relative vel|absolute)   */
    BOOLEAN pos_relative;       /* TRUE|FALSE (relative pos|absolute)   */
    long acc;                   /* Acceleration, range: 0..MAXRANGE     */
    long velocity;              /* Position, range: -MAXRANGE..MAXRANGE */
    long position;              /* Position, range: -MAXRANGE..MAXRANGE */
};
```

These should be self-explanatory. The third and fourth data structures are used to maintain data about the LM629s and keep a model of sorts of the chips. This is done as there are no instructions to extract some information from the LM629, e.g. the current trajectory.

```
/*-------------------------------------------------------------------+
 |    Structure definition for Motion controller channel             |
 +-------------------------------------------------------------------*/

struct LM629
{
    struct LM629_Filter *Filter;
    struct LM629_Filter *NewFilter;
    struct LM629_Trajectory *Trajectory;
    struct LM629_Trajectory *NewTrajectory;
    BOOLEAN filter_updated;
    BOOLEAN trajectory_started;
    BOOLEAN trajectory_complete;
    BOOLEAN pwm_brake;
    int position_error;
};

/*-------------------------------------------------------------------+
 |    Structure definition for ANDI-SERVO board                      |
 +-------------------------------------------------------------------*/

struct andi_servo
{
    struct LM629 *Channel0;
    struct LM629 *Channel1;
    BOOLEAN FaultLED;
```

```
    int base_address;
};
```

All functions in the driver get passed a pointer to an andi_servo struct. From this all necessary information can be accessed easily. The LM629 chips are controlled by sending one of a set of instructions as listed here :

```
/*-----------------------------------------------------------------+
 |    LM629 command Mnemonics                                      |
 +-----------------------------------------------------------------*/

#define RESET    0x0          /* Soft RESET command */
#define DFH      0x2          /* DeFine Home */
#define SIP      0x3          /* Set Index Position */
#define LPEI     0x1b         /* Interrupt on excessive position error */
#define LPES     0x1a         /* Stop on excessive error */
#define SBPA     0x20         /* Set BreakPoint, Absolute */
#define SBPR     0x21         /* Set BreakPoint, Relative */
#define MSKI     0x1c         /* MaSK Interrupts */
#define RSTI     0x1d         /* ReSeT Interrupts */
#define LFIL     0x1e         /* Load PID FILter parameters */
#define UDF      0x4          /* UpDate PID Filter */
#define LTRJ     0x1f         /* Load TRaJectory parameters */
#define STT      0x1          /* StarT Trajectory */
#define RDSIGS   0xc          /* ReaD SIGnalS register */
#define RDIP     0x9          /* ReaD Index Position */
#define RDDP     0x8          /* ReaD Desired Position */
#define RDRP     0xa          /* ReaD Real Position */
#define RDDV     0x7          /* ReaD Desired Velocity */
#define RDRV     0xb          /* ReaD Real Velocity */
#define RDSUM    0xd          /* ReaD integration SUM */
```

This is done by a set of functions listed in andi_servo.h, and called from servo.h. The user never sees these functions and shouldn't attempt to call them as that could corrupt the internal model of what has happened on the LM629.

Part III
User-Space Driver Interfaces

There are three levels of interface to the driver from user-space. These are low-level access to the PID filters and trajectories; mid-level access to commands pertaining to a single motor axis; and high-level heading and position setting. They are accessed in the normal unix manner, through device files :

/dev/andi_servo/filter[0,1] Low-level access to PID filters.

/dev/andi_servo/trajectory[0,1] Low-level access to trajectories.

/dev/andi_servo/channel[0,1] Mid-level access to motor axes.

/dev/andi_servo/board High-level access to board.

The various read()/write()/ioctl() functions differ in effect from file to file as listed here :

/dev/andi_servo/filter[0,1]

read()	*write()*	*ioctl()*
Current PID filter	Load new PID filter	Update filter Is filter updated ?

/dev/andi_servo/trajectory[0,1]

read()	*write()*	*ioctl()*
Current Trajectory	Load new Trajectory	Start new Trajectory Trajectory Started ? Trajectory Completed ? Register for notification on completion of trajectory

/dev/andi_servo/channel[0,1]

read()	*write()*	*ioctl()*
Get desired position Get actual position Get desired velocity Get actual velocity	Set desired position Get desired velocity	Position/Velocity mode Desired/Actual feedback Smooth stop Abrupt stop Motor off Get/Set PWM brake Get/Set breakpoint Get/Set acceleration Get/Set signals Get/Set status Get/Set position error threshold Get/Set IRQ mask Define home position

/dev/andi_servo/board

read()	*write()*	*ioctl()*
Get Status, Signals, Position, Heading	Set desired position, heading	Hard Reset Soft Reset Set/Get PWM Brakes Smooth stop Abrupt stop Motor off Set Fault LED Enable IRQs Get Interrupt source

The formats written or read from the files also differs :

/dev/andi_servo/filter[0,1]
Read/Write a LM629_Filter struct in binary.

/dev/andi_servo/trajectory[0,1]
Read/write a LM629_Trajectory struct in binary.

/dev/andi_servo/channel[0,1]
Read/write in ascii.

/dev/andi_servo/board Read/write in ascii.

The language chosen to write control applications is not relavent as long as it understands the binary format used to write the filter and trajectory data. In C and C++ and Objective C, this is done natively. In Perl and Python it is easy to arrange, and also in Java through the use of bitvectors.

The structures used in the driver, as well as the ioctl() commands used are contained in a public header file *andi.h*. This should be used by all C, C++ and Objective C programs. A similar file has yet to be written for other languages.