Proportional One Axis Position Controller Software Specification and Developer's Guide

Egan McComb

Advisers: Héctor Gamero and Prof. Kazuo Yamazaki

...

Intelligent Manufacturing Systems and Mechatronics Laboratory

Mechanical and Aerospace Engineering Department

University of California, Davis

August 28, 2014

Contents

List of Figures

List of Tables

1 Introduction

This document explains the user operation, conceptual foundation, algorithms, and software implementation of a one axis position controller real time system developed by Professor Kazuo Yamazaki's Intelligent Manufacturing Systems and Mechatronics Laboratory at the University of California, Davis. It is meant to be an update and replacement of a previous document, *One axis postion control software development specification*. The software, to be referred hereafter as "the controller," implements a simple proportional control algorithm in MCS-51 assembly to achieve precise incremental position control of a servomotor. Specifically, it is designed for use with the lab's home-grown cased MORPH board system, which includes an FPGA implementation of the Intel 8052 architecture, useful peripherals and standard man-machine interface components, and a high-quality servo amplifier and encoder. The controller is intended to be used as the axis servo control module (ASC) in conjunction with a free time system on this hardware, operating in parallel as a timer-based interrupt routine for deterministic behavior.

2 Black Box Structure and Operation

To use the controller, the user must call its initialization routine at `0x9F00` from the free time system software to set up the timer-based real time system interrupt. See Sec. 4 for more information on this initialization routine and the interrupt structure of the controller as a whole. After initialization, the controller is ready to use, but will not run until commanded to do so from the free time system software. Table 1 summarizes the inputs and outputs of the controller module for basic operation. They are treated in more detail in the ensuing sections. For an overview of all the internal data used in the controller and their software implementation, see Sec. 5. Specifically, it will be necessary to look up the memory locations of the user data in Table 2 to access them from the free time system.

Table 1: Summary of Black Box Structure: User I/O Data

Inputs		Outputs	
DFX	16-bit servo speed	ABSX	32-bit absolute position
FX	32-bit distance to move	XC	16-bit encoder position
`MCDIR`	1-bit direction flag	CNT	16-bit timing counter
`XGO`	1-bit start flag	`INPOS`	1-bit in position flag

2.1 Inputs

For basic operation, the user may select a speed DFX (16-bit) and distance FX (32-bit) to be moved, as well setting the direction flag `MCDIR` to select clockwise (0) or counterclockwise (1) direction.[1] When the user sets the `XGO` flag, the controller will direct the servomotor the commanded distance and direction at the commanded speed. It is also possible to modify the proportional gains KP (in motion) and KPINP (in position: servomotor locked), but this should not be necessary for normal applications with the MORPH board hardware.

The speed DFX is the position encoder increment per interrupt to be commanded during the interior interval of the motion control. Given a desired feed rate n in RPM, the appropriate value of

[1] This is the sense of the rotor when looking down on it; the mechanism to be driven may have a different convention.

DFX can be calculated as:

$$\text{DFX} = \frac{np}{60000} T_s \tag{1}$$

where p is the resolution of up/down pulses per one rotation of the encoder,[2] and T_s is the sampling period.[3] Note that care must be taken in selecting a feed rate that is appropriate to the mechanism being driven and is low enough to prevent encoder aliasing (see Sec. 6.2).

The distance FX is the number of position encoder pulses to the desired position. Given a distance x in angular units of which there are θ_n in a full rotation, the appropriate value of FX can be calculated as:

$$\text{FX} = x \frac{p}{\theta_n} \tag{2}$$

where p is the resolution of up/down pulses per one rotation of the encoder. For all intents and purposes, FX can be considered unbounded, but in actuality its precision limits it to holding no more than 524288 full revolutions, a limitation shared by the absolute position counter ABSX.

Since the `MCDIR` and `XGO` flags share a byte, it is convenient to set up a variable `motionRegister` in the free time system software to point to their shared memory address. With this setup, it is easy to simultaneously set the direction of the motion control and initiate it with:

```
motionRegister = 1 + 0x10*direction
```

where `direction` is simply a Boolean variable set up prior to select the desired direction.

2.2 Outputs

During operation, the user may read the absolute position ABSX (32-bit) and the hardware encoder position XC (16-bit). Using the former output, the user may measure the unsigned relative position RELX with respect to some stored reference position REFX, taking care to handle cases where ABSX < REFX:

$$\text{RELX} = \begin{cases} \text{ABSX} - \text{REFX} & \text{if ABSX} \geq \text{REFX} \\ 65536 - (\text{REFX} - \text{ABSX}) & \text{otherwise} \end{cases} \tag{3}$$

where the constant 65536 arises from the 16-bit precision of the UDC. Converting to unsigned change in angular units Δx can be done with a relationship like that in Eqn. 2. From there, the simplified angular position θ can be calculated as:

$$\theta = (\theta_0 + dx) \bmod \theta_n \tag{4}$$

where θ_0 is the reference position in the used units that corresponds to REFX, and θ_n is the number of those units in a full rotation. Should the user need to, it is also possible to read the remaining distance to move FX (32-bit), which is updating at the end of each interrupt.

To measure elapsed time in the real time system, a counter CNT (16-bit) is available. It is incremented every interrupt, so it measures 10ms units by default. When the controller achieves the commanded position, the `INPOS` flag is set and the `XGO` flag cleared. Since `INPOS` is in the same byte as `MCDIR` and `XGO`, the same use of `motionRegister` in the free time system software can be useful. With this setup, determining when the controller has achieved the desired position can be done with:

```
motionRegister & 0x02
```

[2] This is 8192 pulses on the MORPH board hardware's UDC.

[3] This sampling period is hard coded in the controller interrupt setup as 10ms.

3 Conceptual Foundation

The position control system used employs the concept of negative feedback, as shown in the block diagram in Fig. 3.

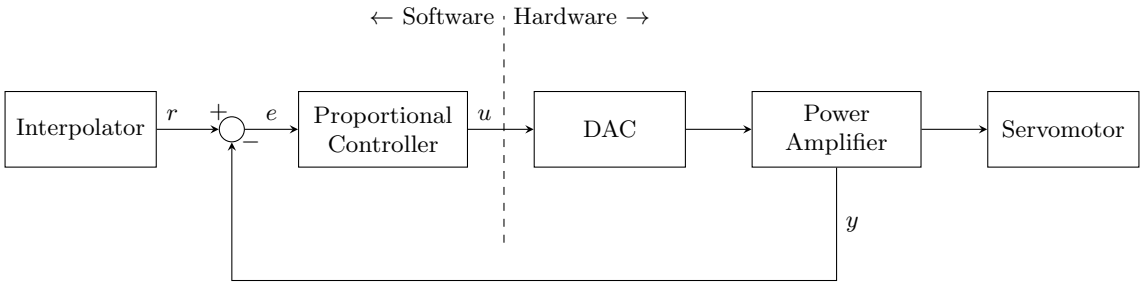

Figure 1: Block Diagram of Position Control Concept

In the cased MORPH board setup, the power amplifier performs the sensing of the servomotor position and also implements sensors and feedback loops for speed and electric current. Because these quantities are controlled by the power amplifier, we must only implement the outer feedback loop to control the servomotor position. Thus interpolation, position error calculation, and proportional control comprise the one axis position controller software.

Because of the discrete nature of computer control, the interpolator is an integral component of the controller design, breaking up the control action into a series of commands that can be achieved during each interrupt cycle in what is known as incremental point-to-point control. The one axis position controller software divides the distance to the destination position into a series of appropriately spaced reference positions. In the interior interval of motion control, these reference positions are equally spaced to achieve smooth, constant velocity movement. As the servomotor nears its destination, the interpolator scales back the reference position increment to avoid overshoot. In this way, quick movement to the destination can be achieved with excellent precision that would otherwise require slow and overly controlled operation. In the controller software implementation, the interpolation is precomputed for the next interrupt cycle rather than for the current one. This is desirable because it allows the interpolation calculations to occur after the output is commanded using the reference position prepopulated by the last interrupt, decreasing latency.

At each interrupt, the position following error is combined from two sources. First, there is the distance between the current position as measured by the power amplifier's UDC and the reference position generated by the interpolator. Ideally, this would be the only error at each step because the servomotor would exactly achieve the commanded positions at every sampling period. The second source arises from the physical behavior of the system: it is either the leftover error from the prior interrupt cycle because the servomotor undershot the reference position, or it is an additional error caused by overshooting the reference position. During the interior interval of motion control, this is the component of the position following error that varies between position samples.

The controller module is proportional, simply multiplying the position following error by an appropriately selected positive gain to calculate the velocity command output to the DAC, reacting to both of the described error components. In the ideal case, the controller would command

constant velocity operation during the interior interval of motion control, slowing as the servomotor approached the destination and the interpolation step decreased. With the nonideal error, the controller reacts to disturbance in the planned motion process by slightly reducing or increasing the commanded velocity to counter overshoot and undershoot, respectively. In conjunction with the interpolator behavior, this simple behavior achieves very precise control without needing integral or derivative terms, greatly simplifying the required software implementation of the controller.

4 Initialization and Interrupt Structure

4.1 Initialization

The initial entrance point of the program is `0x9F00`, to be called by the free time system. There, the interrupt vector is set up so that overflows in the microcontroller's Timer1 call the main real time system instructions at `0x9F50`. Timer1 is set up as a 16-bit timer with an initial value of `0xBEE5`, which corresponds to an overflow period of 10ms at a oscillator frequency of 20MHz. To calculate the required value for a different overflow period, take into account that each machine cycle consists of 12 oscillator pulses, and that the timers always count up. Thus the initial timer value i_0 for an overflow period of T with an n-bit timer can be calculated as follows:

$$i_0 = \left\lfloor 2^n - \frac{Tf}{p} \right\rfloor \tag{5}$$

where f is the oscillator frequency and p is the number of pulses per machine cycle. A conceptual schematic of the interrupt mechanism is shown in Fig. 2. The address `0xFA1B` corresponds to the Timer1 vector table as implemented by the MORPH board's monitor program.

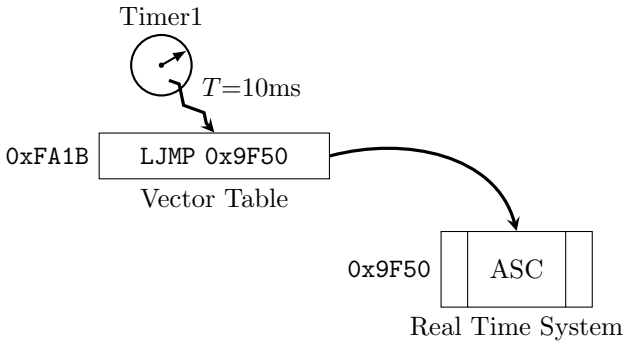

Figure 2: Schematic of the Timer Interrupt Jump

After populating the timer interrupt vector table, the hardware UDC is cleared with the `ClearCounter` routine and the DAC is instructed to convert the value `0x8000`[4] with the `DAOut` routine. These two hardware interface routines are documented in Sec. 4.3. The `INITVAR` routine (see Sec. 5.2) is then called to initialize values of the controller's internal data. At this point, Timer1 is started and interrupts therefrom are enabled on the microcontroller's IE register.

[4]This value corresponds to little or no movement of the servomotor (see Sec. 6.6).

4.2 Interrupt Structure

On each interrupt, Timer1 is repopulated with its start value for the next overflow cycle. From there, each of the modules of the real time system are called in turn, as shown in Fig. 3. Each of these modules will be treated in depth in Sec. 6.

1. `ReadUDCounter`:
   Read the current encoder position from the hardware UDC.
2. `PosDiffCAL`:
   Calculate the position change since the last interrupt and update absolute position for the software counter.
3. `PEXUpdate`:
   Calculate the position following error, taking into account the subsequent motion command selected during the last interrupt.
4. `INPOSJudge`:
   Determine the required state of the motion control, setting the servo lock flag if the destination position will be reached during the current interrupt.
5. `PPCAL`:
   Perform the proportional control calculation to produce a velocity command based on the state of the motion control.
6. `DAOUTVEL`:
   Offset and output the commanded velocity to the hardware DAC for conversion.
7. `INTPAndFXCheck`:
   Interpolate the position change during the next interrupt, select an appropriate motion command, and update the distance to the destination position.

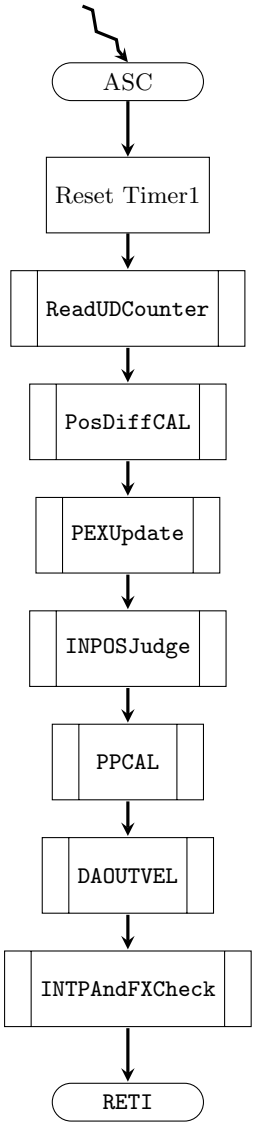

Figure 3: Real Time System Processing Flow

4.3 Hardware Interface

4.3.1 ClearCounter — Reset Up/Down Counter

```
subroutine ClearCounter
Clear U/D Counter1
```

This routine simply sends 0x00 to _UDCNT1_HIGH, instructing the UDC to reset its value to 0.

4.3.2 DAOut — Write to DAC and Convert

```
subroutine DAOut
Output to Digital-Analog Converter

inputs: b, acc = D
```

This routine writes the 16-bit value held in b, acc to _DA1_HIGH, _DA1_LOW and instructs the DAC to perform the conversion by writing to _DA1_CNVT.

5 Internal Data and Variable Definitions

5.1 Memory Allocation

Table 2 shows the data names used in the controller, their memory addresses, and their descriptions. The byte order convention for 16- and 32-bit data names are $X = {}^*\mathtt{H},{}^*\mathtt{L}$, and $X = {}^*\mathtt{4},{}^*\mathtt{3},{}^*\mathtt{2},{}^*\mathtt{1}$, respectively. A few data names are disused in the current controller software, but are kept for their potential historical uses.

Table 2: Internal Data Locations and Semantics

System flags:		
XGO	0x21.0	Motion start/stop flag.
INPOS	0x21.1	Servo lock flag.
DATSAV	0x21.3	Unknown use.
MCDIR	0x21.4	Motion direction flag.
MSIGN0	0x21.5	Multiplicand sign flag.
MSIGN1	0x21.6	Multiplier sign flag.
MSIGNALL	0x21.7	Product sign flag.
Timing counter value (16-bit):		
CNTL	0x23	
CNTH	0x24	
Buffer address value (16-bit, unknown use):		
BUFF_ADRL	0x4A	
BUFF_ADRH	0x4B	
Commanded velocity value VELX (16-bit):		
VELXH	0x4C	

VELXL	0x4D	
Measured position increment DXC (16-bit):		
DXCH	0x4E	
DXCL	0x4F	
Measured hardware UDC value XC (16-bit):		
XCH	0x50	
XCL	0x51	
Position interpolation increment DFX (16-bit):		
DFXH	0x53	
DFXL	0x52	
Position following error PEX (16-bit):		
PEXH	0x54	
PEXL	0x55	
Residual distance to destination position FX (32-bit):		
FX1	0x56	
FX2	0x57	
FX3	0x58	
FX4	0x59	
Proportional gains:		
KP	0x5A	During motion.
KPINP	0x5B	Servo locked (in position).
Commanded position increment DX (16-bit):		
DXH	0x5E	
DXL	0x5F	
Absolute position counter value ABSX (32-bit):		
ABSX1	0x60	
ABSX2	0x61	
ABSX3	0x62	
ABSX4	0x63	
Miscellaneous Flags:		
RDABSEN	0x64	Flag to disable ABSX updating.
INTCNT	0x65	Unknown use.
Temporary variable for position increment calculation (16-bit):		
DABSH	0x66	
DABSL	0x67	
Digital-Analog conversion addresses:		
_DA1_LOW	0xFE18	

_DA1_HIGH	0xFE19
_DA_CNVT	0xFE1F
Up/Down counter addresses:	
_UDCNT1_LOW	0xFF05
_UDCNT1_HIGH	0xFF06

5.2 INITVAR — Initialization of Internal Data

```
subroutine INITVAR
Set Up Initial Values of Variables
```

Initialization of the controller's internal data is performed by this routine. It sets the values of ABSX, PEX, DX, XC, and CNT to zero, as well as INTCNT, RDABSEN, and DABS. The state of the motion control is set to stopped (XGO = 0) with locked servo (INPOS = 1), and DATSAV is turned off.

6 Real Time System Modules

The interdependency of the real time modules and the paths through which data flow are more complex than Fig. 3 would suggest. Figure 4 displays a graph of the interdependencies and information flow in the real time system. The quoted documentation headers in the following sections are directly copied from their definitions in the source code.

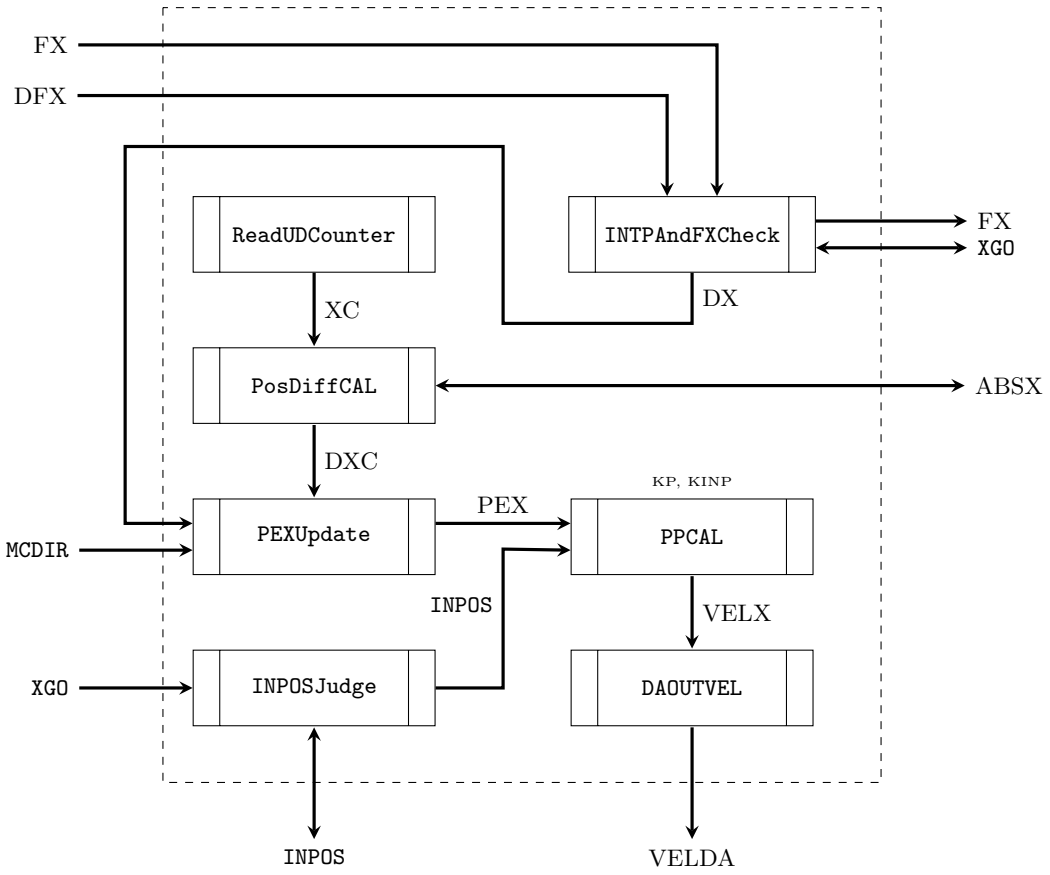

Figure 4: Real Time System Module Interdependency Graph

6.1 `ReadUDCounter` — Read Up/Down Counter

```
subroutine ReadUDCounter
1. Read Up/Down counter.

output: XCH, XCL = UDC counter value (16-bit)
        r3, r2 = XCH_{i-1}, XCL_{i-1}
```

The UDC is read by sending it the start read enable command `0x00` at `_UDCNT1_LOW`. The old encoder position is saved in `r3, r2` for use in `PosDiffCAL` and the current encoder position `XCH, XCL` is then read from `_UDCNT1_HIGH, _UDCNT1_LOW`.

6.2 `PosDiffCAL` — Position Change and Absolute Counter Update

```
subroutine PosDiffCAL
2. Motion Inc/Decrement Calculation and Absolute Counter Update

inputs: XCH, XCL = XC_i; r3, r2 = XC_{i-1};
        ABSX4, ABSX3, ABSX2, ABSX1 = ABSX

output: DXCH, DXCL = DXC_i = XC_i - XC_{i-1}
        If RDABSEN is not set:
        ABSX4, ABSX3, ABSX2, ABSX1 = ABSX_i = ABSX_{i-1} + DXC_i

alters: DABSH, DABSL, register bank 3
```

The hardware UDC is 16-bit, divided such that the range of measurement is 8 full rotations. In order to keep track of motion beyond this, a software counter is used for absolute position measurement. This module calculates the position change $\mathrm{XC}_i - \mathrm{XC}_{i-1}$ of the UDC value read by `ReadUDCounter`, accumulating the result on the absolute position ABSX if `RDABSEN` is not set.

In order to avoid aliasing of the UDC values in adjacent sampling periods, it is necessary to ensure that the distance between adjacent position samples is always less than half of the full range of the hardware UDC, *i.e.* 4 revolutions. This constraint can be expressed in the relationship:

$$\left(\frac{np}{60}T_s < 2^{m-1}\right) \wedge \left(-2^{m-1} \leq \mathrm{DXC}_i < 2^{m-1} - 1\right) \tag{6}$$

where n is the maximum rotation speed in RPM, p is the resolution of up/down pulses per one rotation of the encoder, T_s is the sampling period, and m is the number of bits in the UDC. From an application perspective, this is effectively a limitation on the rotational speed of operations.

With the possibility of aliasing eliminated, there are 4 possible cases to be handled, two of them trivial. Figure 5 shows these 4 cases, from left to right: 1) simple increase, 2) increase with overflow, 3) decrease with overflow, 4) simple decrease. Using 2's complement arithmetic, those cases with counter overflow can be handled automatically. In this way, the module calculates the position change DXC from the sampled values of XC. This position change is temporarily stored in DABS before it is cast to 32-bit precision and added to ABSX, thereby updating the software counter.

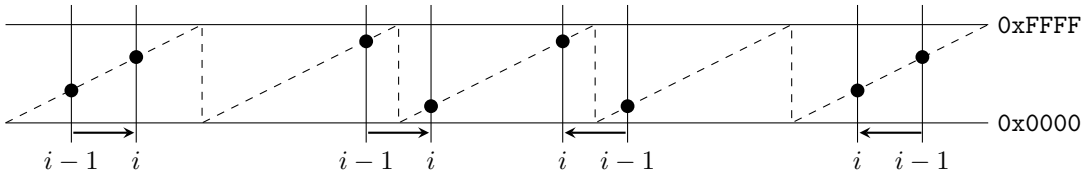

Figure 5: Schematic of Adjacent Position Sample Cases

6.3 PEXUpdate — Update Position Following Error

```
subroutine PEXUpdate
3. Update Position Error

inputs: DXCH, DXCL = DXC; DXH, DXL = DX; PEXH, PEXL = PEX_{i-1}

output: PEXH, PEXL = PEX_i = PEX_{i-1} - DXC +/- DX

alters: register bank 3
```

The position following error PEX is calculated depending on the direction of motion control defined by MCDIR:

$$\mathrm{PEX}_i = \begin{cases} \mathrm{PEX}_{i-1} - \mathrm{DXC} + \mathrm{DX} & \text{if MCDIR} = 0 \\ \mathrm{PEX}_{i-1} - \mathrm{DXC} - \mathrm{DX} & \text{if MCDIR} = 1 \end{cases} \tag{7}$$

It carries over the prior error PEX_{i-1} less the distance traveled during the last motion period DXC. The actual update comes from the addition/subtraction of the distance motion command DX determined by INTPAndFXCheck at the end of the last interrupt (see Sec. 6.7).

6.4 INPOSJudge — Determine Required Motion Control State

```
subroutine INPOSJudge
4. Determine Required State of Motion Control

inputs: XGO; INPOS

output: INPOS
```

This module determines the required state of motion control by checking the motion control flag XGO and the servo lock flag INPOS. The state discrimination flow is shown in Fig. 6.

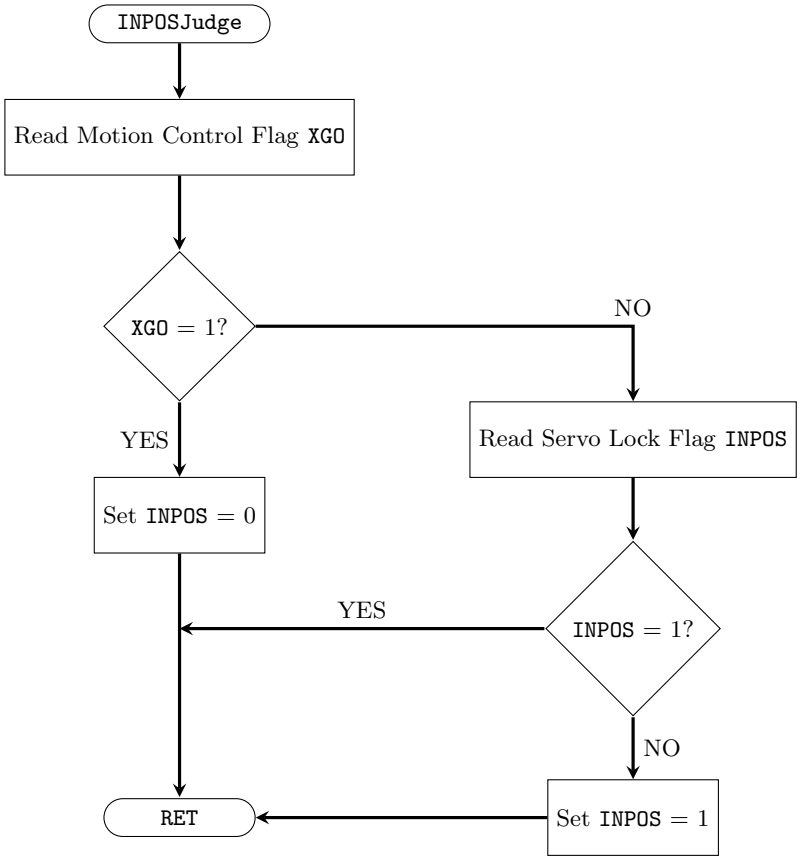

Figure 6: Motion Control State Discrimination Flow

6.5 PPCAL — Proportional Control Calculation

```
subroutine PPCAL
5. Proportional Control Algorithm

inputs: PEXH, PEXL = PEX; KP; KPINP; INPOS

output: VELXH, VELXL = VELX = KP * PEX if INPOS = 0
                            = KPINP * PEX if INPOS = 1

alters: acc, register bank 3
```

This module performs the proportional control based on the position following error PEX, outputting a velocity VELX. It uses different gains depending on whether the servo lock flag INPOS is set.

$$\text{VELX} = \begin{cases} \text{KP} \cdot \text{PEX} & \text{if INPOS} = 0 \\ \text{KPINP} \cdot \text{PEX} & \text{if INPOS} = 1 \end{cases} \tag{8}$$

6.6 DAOUTVEL — Offset Correction for DAC

```
subroutine DAOUTVEL
6. Offset Correction for DAC Output

inputs: VELXH, VELXL = VELX

output: digital velocity command value VELDA

alters: acc, b, register bank 3
```

The digital value corresponding to approximately zero output velocity command from the DAC is halfway through the range of its 16-bit container format, corresponding to a value of `0x7FFF`. For this reason, the velocity command value VELX produced by the proportional control module needs an offset correction by this amount. The sign of VELX is stored in `MSIGNALL`, so this is checked to see whether this `0x7FFF` should be added (if positive) or subtracted (if negative) therefrom. This corrected value VELDA is then written to the DAC using the `DAOut` routine.

6.7 INTPAndFXCheck — Interpolation and Final Position Check

```
subroutine INTPAndFXCheck
7. Motion Command Interpolation & Final Position Distance Update

inputs: DFXH, DFXL = DFX; FX4, FX3, FX2, FX1 = FX_i; XGO

output: FX4, FX3, FX2, FX1 = FX_{i+1}
        DXH, DXL = 0 if XGO = 0; else:
                 = DFX if FX_{i+1} >= 0
                 = FX_i otherwise
        XGO = 0 if FX_{i+1} < 0
            = 1 otherwise

alters: register bank 3
```

The next distance motion command DX should be selected so that the servo maintains a constant speed except when such a command would cause an overshoot of the destination position. This module achieves this by extrapolating the planned motion DFX to the next sampling period and determining if the distance to the destination position FX becomes negative. In this case, an overshoot would occur so the module clears the `XGO` flag and selects the current FX as the distance motion command instead of the user-selected DFX used in the interior interval of motion control. This logical flow is displayed in Fig. 7

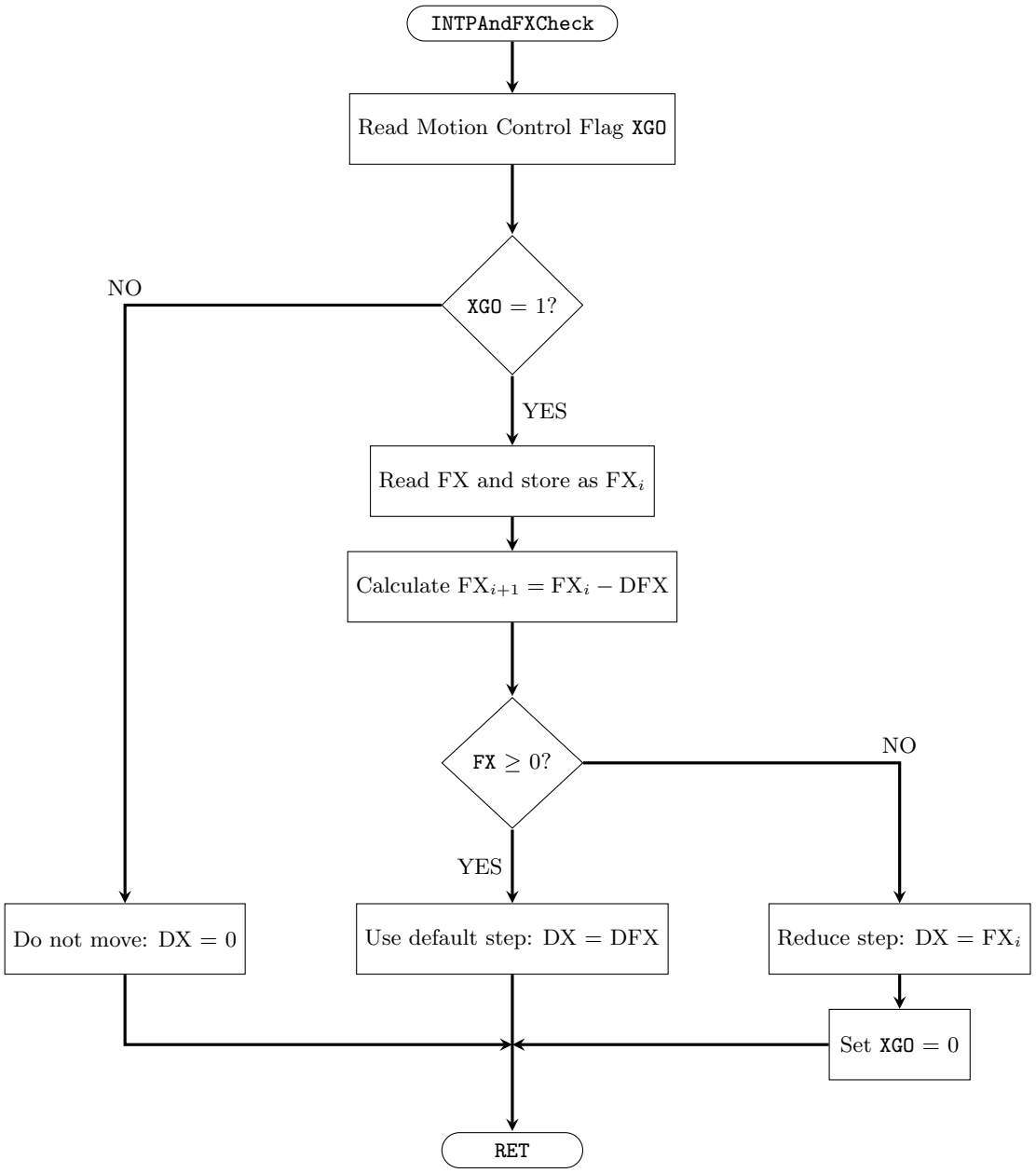

Figure 7: Distance Motion Command Selection Flow

7 Data Representation and Arithmetic

This application requires a greater range and precision of numerical data than that afforded by the native instructions of the Intel 8052, an 8-bit architecture. Throughout the one axis position controller code, data are represented in 8-, 16-, 24-, and 32-bit 2's complement format as necessary. For convenience, a collection of arithmetic routines have been implemented to perform addition, subtraction, multiplication, and precision conversion among the supported formats.

Each of the implemented routines acts on little-endian arguments in register bank 3, leaving the results therein. A few subroutines relating to multiplication also set single-bit sign flags. The quoted documentation headers included herein are directly copied from their definitions in the source code.

7.1 16-Bit Arithmetic Routines

Each of the 16-bit arithmetic routines act on two 16-bit 2's complement operands (X, Y) to be loaded by the user as $X =$ `r1,r0` and $Y =$ `r3,r2`. Each uses `OV` to indicate that the result was out of range.

7.1.1 `CADD16` — 2's Complement Addition

```
subroutine CADD16
16-Bit Signed (2's Complement) Addition

inputs: r1, r0 = X
        r3, r2 = Y

output: r1, r0 = signed sum S = X + Y

alters: acc, C, OV, register bank 3
```

7.1.2 `CSUB16` — 2's Complement Subtraction

```
subroutine CSUB16
16-Bit Signed (2's Complement) Subtraction

inputs: r1, r0 = X
        r3, r2 = Y

output: r1, r0 = signed difference D = X - Y

alters: acc, C, OV, register bank 3
```

7.1.3 MUL16 — Signed Multiplication

```
subroutine MUL16
16-Bit Signed (2's Complement) Multiplication to 32-Bit Product
with Sign Flag

inputs: r1, r0 = X
        r3, r2 = Y

output: r3, r2, r1, r0 = product magnitude P = |X x Y|
        MSIGNALL = sign(P): 1 if negative

calls: UMUL16, Cr0r1, Cr2r3, Mr0r3

alters: acc, C, OV, MSIGN0, MSIGN1, MSIGNALL, register bank 3
```

Unlike the other 16-bit arithmetic routines, this one returns an unsigned result; the signs are handled by helper subroutines that store them in single-bit flags, with 1 representing negative. Additionally, owing to the properties of multiplication, the product is necessarily double the precision of its inputs.

UMUL16 — Unsigned Multiplication

```
subroutine UMUL16
16-Bit Unsigned Multiplication to 32-Bit Product

inputs: r1, r0 = |X|
        r3, r2 = |Y|

output: r3, r2, r1, r0 = product P = |X| x |Y|

alters: acc, C, OV, register bank 3
```

This subroutine performs the actual multiplication of the multiplicand and multiplier after their magnitudes have been calculated by the other helper subroutines.

Cr0r1 — Multiplicand Magnitude and Sign

```
subroutine Cr0r1
16-Bit 2's Complement Magnitude and Sign

inputs: r1, r0 = X

output: r1, r0 = |X|
        MSIGN1 = sign(X): 1 if negative

alters: acc, C, MSIGN1, register bank 3
```

This subroutine determines and stores the sign of the multiplicand and performs 2's complement magnitude calculation if it is negative. **N.B.:** the sign of this first operand is stored in MSIGN1, contrary to what might be expected.

`Cr2r3` — **Multiplier Magnitude and Sign**

```
subroutine Cr2r3
16-Bit 2's Complement Magnitude and Sign

inputs: r3, r2 = Y

output: r3, r2 = |Y|
        MSIGN0 = sign(X): 1 if negative

alters: acc, C, MSIGN0, register bank 3
```

This subroutine determines and stores the sign of the multiplier and performs 2's complement magnitude calculation if it is negative. **N.B.:** the sign of this second operand is stored in `MSIGN0`, contrary to what might be expected.

`Mr0r3` — **Product Sign**

```
subroutine Mr0r3
Sign Bit Multiplication.

inputs: MSIGN1 = sign(X): 1 if negative
        MSIGN0 = sign(Y)

output: MSIGNALL = sign(X x Y)

alters: acc, C, MSIGNALL
```

This subroutine determines and stores the sign of the product based on the `MSIGN1` and `MSIGN0` flags set by the previous helper subroutines, but *does not* perform 2's complement conversion thereon.

7.2 16-Bit Conversion Routines

Each of the 16-bit conversion routines act on one 16-bit 2's complement operand X to be loaded by the user as $X = \texttt{r1}, \texttt{r0}$.

7.2.1 `C2Comp` — 2's Complement Conversion

```
subroutine C2Comp
16-Bit 2's Complement Operation

inputs: r1, r0 = X

output: r1, r0 = 2's(X)


alters: acc, C, register bank 3
```

7.2.2 C16toC24 — Cast to 24-Bit 2's Complement

```
subroutine C16toC24
16-Bit 2's Complement Cast to 24-Bit 2's Complement

inputs: r1, r0 = X: 16-bit signed (2's complement)

output: r2, r1, r0 = X: 24-bit signed (2's complement)

alters: acc, C
```

7.2.3 C16toC32 — Cast to 32-Bit 2's Complement

```
subroutine C16toC32
16-Bit 2's Complement Cast to 32-Bit 2's Complement

inputs: r1, r0 = X: 16-bit signed (2's complement)

output: r3, r2, r1, r0 = X: 32-bit signed (2's complement)

alters: acc, C
```

7.3 24-Bit Arithmetic Routines

Each of the 24-bit arithmetic routines act on two 24-bit 2's complement operands (X, Y) to be loaded by the user at $X =$ `r2`, `r1`, `r0` and $Y =$ `r6`, `r5`, `r4`. **N.B.:** the fourth register `r3` is unused in these routines so take care to load the operands correctly. Each sets `C` in addition to `OV` to indicate that the result was out of range.

7.3.1 CADD24 — 2's Complement Addition

```
subroutine CADD24
24-Bit Signed (2's Complement) Addition

inputs: r2, r1, r0 = X
        r6, r5, r4 = Y

output: r2, r1, r0 = signed sum S = X + Y
        C is set if overflow occurs

alters: acc, C, OV, register bank 3
```

7.3.2 CSUB24 — 2's Complement Subtraction

```
subroutine CSUB24
24-Bit Signed (2's Complement) Subtraction

inputs: r6, r5, r4 = X
        r2, r1, r0 = Y

output: r2, r1, r0 = signed difference D = X - Y
        C is set if overflow occurs

alters: acc, C, OV, register bank 3
```

7.4 32-Bit Arithmetic Routines

Each of the 32-bit arithmetic routines act on two 32-bit 2's complement operands (X, Y). Each sets `C` in addition to `OV` to indicate that the result was out of range.

7.4.1 CADD32 — 2's Complement Addition

```
subroutine CADD32
32-Bit Signed (2's Complement) Addition

inputs: r3, r2, r1, r0 = X
        r7, r7, r5, r4 = Y

output: r3, r2, r1, r0 = signed sum S = X + Y
        C is set if overflow occurs

alters: acc, C, OV, register bank 3
```

7.4.2 CSUB32 — 2's Complement Subtraction

```
subroutine CSUB32
32-Bit Signed (2's Complement) Subtraction

inputs: r7, r6, r5, r4 = X
        r3, r2, r1, r0 = Y

output: r3, r2, r1, r0 = signed difference D = X - Y
        C is set if overflow occurs

alters: acc, C, OV, register bank 3
```

N.B.: the register convention for this routine is reversed from the others.

A Code Style

For consistency and ease of editing, the assembly code adheres to a style outlined here.

A.1 Capitalization

- All standard operation and operand mnemonics as well as byte-addressed registers use lower case.
- All bit-addressed registers use upper case.
- All user-defined variable names use upper case.
- User-defined routines use a combination of upper case and camel case, but all references thereto must use the same case.
- Overall, all references to any name must be done in the same case to facilitate searching and ease of editing.

A.2 Comments

- All informational comments include a space before the text: `; Example`.
- All unused code to be commented out does not include the space: `;lcall BADCODE`.
- Inline comments are aligned within their code section or subroutine using tabs. They are written in sentence case with a period.
- Informational comments occur ahead of the operation to which they refer, where the data is loaded in preparation.
- Code sections are indicated at their start with a double-lined, closed banner. Each banner is of the same width, and has a title in uppercase and a description in title case. Alignment is done with spaces only.
- Subroutines are indicated at their start with a single-lined, right-open banner. Each banner is of the same width, has the subroutine name, its description in title case, its inputs, outputs, and side effects. Alignment is done with spaces only.

A.3 Constants

- All constants except 0 and 1 in arithmetic operations should be written in hexadecimal format of the appropriate precision using the `0x` notation with uppercase high numerals `A`–`F`.

A.4 Spacing

- Every line begins at the first column, with no indent.
- All operands are separated by a space after the comma: `mov b, a`.
- Labels not immediately following block comment have an additional newline before them. The first instruction following a label occurs on a new line.

B Referenced Code Revision

Included below is the latest revision of the referenced code as of compiling this document.

```
;=====================================================================;
;---------------------------------------------------------------------;
; INTERNAL DATA SETUP: Stack bottom must be set to 0x70 in main       ;
;                      program to prevent conflicts.                  ;
;---------------------------------------------------------------------;
;=====================================================================;

; System flags.
XGO def 0x21.0      ; Motion start/stop flag.
INPOS def 0x21.1    ; Servo lock flag.
DATSAV def 0x21.3
MCDIR def 0x21.4    ; Motion direction flag.
MSIGN0 def 0x21.5   ; Multiplicand sign flag.
MSIGN1 def 0x21.6   ; Multiplier sign flag.
MSIGNALL def 0x21.7 ; Product sign flag.

; Timing counter value (16-bit).
CNTL epz 0x23
CNTH epz 0x24

; XXX: Buffer address value (16-bit).
BUFF_ADRL def 0x4A
BUFF_ADRH def 0x4B

; Velocity command value VELX (16-bit).
VELXH epz 0x4C
VELXL epz 0x4D

; Position increment/decrement DXC (16-bit).
DXCH epz 0x4E
DXCL epz 0x4F

; Up/Down counter value XC (16-bit).
XCH epz 0x50
XCL epz 0x51

; Position interpolation increment DFX (16-bit).
DFXH epz 0x53
DFXL epz 0x52

; Position following error PEX (16-bit).
PEXH epz 0x54
PEXL epz 0x55

; Residual distance to end position FX (32-bit).
FX1 epz 0x56
FX2 epz 0x57
FX3 epz 0x58
FX4 epz 0x59

; Proportional gains.
KP epz 0x5A     ; During motion.
KPINP epz 0x5B  ; Servo locked (in position).

; Distance motion command per sampling period DX (16-bit).
DXH epz 0x5E
DXL epz 0x5F

; Absolute position counter value ABSX (32-bit).
ABSX1 epz 0x60
ABSX2 epz 0x61
ABSX3 epz 0x62
ABSX4 epz 0x63
```

```
; Miscellaneous Flags.
RDABSEN epz 0x64    ; Enable updating software absolute position counter.
INTCNT epz 0x65

; Temporary variable (16-bit) for position inc/decrement calculation.
DABSH epz 0x66
DABSL epz 0x67

; Digital-Analog conversion and control addresses.
_DA1_LOW equ 0xFE18
_DA1_HIGH equ 0xFE19
_DA_CNVT equ 0xFE1F

; Up/Down counter control addresses.
_UDCNT1_LOW equ 0xFF05
_UDCNT1_HIGH equ 0xFF06

;=====================================================================;
;---------------------------------------------------------------------;
;         INITIALIZATION: Setup on initial timer interrupt.           ;
;---------------------------------------------------------------------;
;=====================================================================;
org 0x9F00
push psw            ; Save main PSW to stack.
orl psw, #0x18      ; Use register bank 3 from now on.
push acc            ; Save main arithmetic registers to stack.
push b
push dpl            ; Save main data pointer registers to stack.
push dph
mov dptr, #0xFA1B   ; Set Timer1 interrupt vector address (0xFA1B).
mov a, #0x02        ; Setup interrupt vector: LJMP 0x9F50.
movx @dptr, a
inc dptr
mov a, #0x9F
movx @dptr, a
inc dptr
mov a, #0x50
movx @dptr, a

; At 20MHz, 1ms => 0xF97D, 3ms => 0xEC78, 10ms => 0xBEE5
mov tmod, #0x10     ; Setup Timer1 as 16-bit timer.
mov tl1, #0xE5      ; Setup Timer1 overflow period (10ms).
mov th1, #0xBE

lcall ClearCounter  ; Clear the U/D counter.
mov a, #0x00        ; Start DAC with value 0x8000.
mov b, #0x80
lcall DAOut
lcall INITVAR       ; Setup initial values of variables.
setb TR1            ; Start Timer1.
orl ie, #0x88       ; Enable interrupts from Timer1 overflow.

pop dph             ; Restore stacked registers.
pop dpl
pop b
pop acc
pop psw
ret                 ; Done.


;=====================================================================;
;---------------------------------------------------------------------;
;         REAL TIME SYSTEM: Main code for timer interrupt.            ;
;---------------------------------------------------------------------;
;=====================================================================;
org 0x9F50
push b                  ; Save main arithmetic registers to stack.
push acc
```

```
orl psw, #0x18          ; Use register bank 3 from now on.
mov tl1, #0xE5          ; Setup Timer 1 overflow period (10ms).
mov th1, #0xBE
push dpl                ; Save data pointer registers to stack.
push dph

;mov dptr, #0xFF00
;mov a, #0xFF
;movx @dptr, a

mov dph, CNTH           ; Increment timing counter.
mov dpl, CNTL
inc dptr
mov CNTH, dph
mov CNTL, dpl

; Call each of the modules in turn.
lcall ReadUDCounter     ; Module 1.
lcall PosDiffCAL        ; Module 2.
lcall PEXUpdate         ; Module 3.
lcall INPOSJudge        ; Module 4.
lcall PPCAL             ; Module 5.
lcall DAOUTVEL          ; Module 6.
lcall INTPAndFXCheck    ; Modules 7 and 8.

;mov dptr, #0xFF00
;mov a, #0x00
;movx @dptr, a

done:
pop dph                 ; Restore stacked registers.
pop dpl
pop acc
pop b
pop psw                 ; PSW automatically pushed by monitor.
reti

;=====================================================================
; subroutine INITVAR
; Set Up Initial Values of Variables
;=====================================================================
INITVAR:
mov ABSX4, #0x00    ; Start absolute position ABSX at 0.
mov ABSX3, #0x00
mov ABSX2, #0x00
mov ABSX1, #0x00
mov PEXH, #0x00     ; Start position error PEX at 0.
mov PEXL, #0x00
mov DXH, #0x00      ; Start distance motion command at 0.
mov DXL, #0x00
mov XCH, #0x00      ; Start UDC value at 0.
mov XCL, #0x00
setb INPOS          ; Lock servo in position.
clr DATSAV
clr XGO             ; Stop motion control.
mov INTCNT, #0x00
mov RDABSEN, #0x00
mov DABSH, #0x00    ; Start temporary distance variables at 0.
mov DABSL, #0x00
mov CNTH, #0x00     ; Start the timer at 0.
mov CNTL, #0x00     ;
ret

;=====================================================================
; subroutine ClearCounter
; Clear U/D Counter1
;=====================================================================
ClearCounter:
```

```
mov dptr, #_UDCNT1_HIGH ; Send reset command to UDC.
mov a, #0x00
movx @dptr, a
ret

;=====================================================================
; subroutine DAOut
; Output to Digital-Analog Converter
;
; inputs: b, acc = D
;=====================================================================
DAOut:
mov dptr, #_DA1_LOW     ; Push D low byte to DAC.
movx @dptr, a
nop                     ; Wait two computation cycles.
nop
mov a, b                ; Push D high byte to DAC
mov dptr, #_DA1_HIGH
movx @dptr, a
nop                     ; Wait two computation cycles.
nop
mov dptr, #_DA_CNVT     ; Start D/A conversion.
movx @dptr, a
ret

;=====================================================================
; subroutine ReadUDCounter
; 1. Read Up/Down counter.
;
; output: XCH, XCL = UDC counter value (16-bit)
;         r3, r2 = XCH_{i-1}, XCL_{i-1}
;=====================================================================
ReadUDCounter:
mov r3, XCH             ; Store value at last period.
mov r2, XCL
mov dptr, #_UDCNT1_LOW  ; Send start read enable command to UDC.
mov a, #0x00
movx @dptr, a
mov dptr, #_UDCNT1_LOW  ; Read low byte XCL.
movx a, @dptr
mov XCL, a
mov dptr, #_UDCNT1_HIGH ; Read high byte XCH.
movx a, @dptr
mov XCH, a
ret

;=====================================================================
; subroutine PosDiffCAL
; 2. Motion Inc/Decrement Calculation and Absolute Counter Update
;
; inputs: XCH, XCL = XC_i; r3, r2 = XC_{i-1};
;         ABSX4, ABSX3, ABSX2, ABSX1 = ABSX
;
; output: DXCH, DXCL = DXC_i = XC_i - XC_{i-1}
;         If RDABSEN is not set:
;         ABSX4, ABSX3, ABSX2, ABSX1 = ABSX_i = ABSX_{i-1} + DXC_i
;
; alters: DABSH, DABSL, register bank 3
;=====================================================================
PosDiffCAL:
; Calculate position encoder inc/decrement.
mov r1, XCH         ; Subtract last XC (loaded into registers by
mov r0, XCL         ; ReadUDCounter) from current XC.
lcall CSUB16
mov DXCH,r1         ; Store result in DXC.
mov DXCL,r0
; Calculate absolute position inc/decrement.
mov r3, DABSH       ; Add DXC to DABS.
```

```
mov r2, DABSL
lcall CADD16
mov DABSH, r1       ; And store result in DABS.
mov DABSL, r0
mov a, RDABSEN      ; Is RDABSEN 0?
xrl a, #0x01
jnz ABXUpdate       ; If RDABSEN == 0, continue.
ret                 ; Else return.

ABXUpdate:
mov r1, DABSH       ; Cast DABS to 24-bit format.
mov r0, DABSL
lcall C16toC24
mov r7, ABSX4       ; Add DABS to ABSX.
mov r6, ABSX3
mov r5, ABSX2
mov r4, ABSX1
lcall CADD24
mov ABSX4, r3       ; And store result in ABSX.
mov ABSX3, r2
mov ABSX2, r1
mov ABSX1, r0
mov DABSH, #0x00    ; Reset DABS.
mov DABSL, #0x00
ret

;=====================================================================
; subroutine PEXUpdate
; 3. Update Position Error
;
; inputs: DXCH, DXCL = DXC; DXH, DXL = DX; PEXH, PEXL = PEX_{i-1}
;
; output: PEXH, PEXL = PEX_i = PEX_{i-1} - DXC +/- DX
;
; alters: register bank 3
;=====================================================================
PEXUpdate:
mov r1, PEXH        ; Subtract current DXC from last PEX.
mov r0, PEXL
mov r3, DXCH
mov r2, DXCL
lcall CSUB16
mov r3, DXH
mov r2, DXL
jnb MCDIR, PlusDir  ; Is commanded direction positive?
lcall CSUB16        ; If MCDIR == 1, subtract DX from difference.
sjmp OutPEX         ; Finish.

PlusDir:
lcall CADD16        ; If MCDIR == 0, add DX to difference.

OutPEX:
mov PEXH, r1        ; Store result in PEX.
mov PEXL, r0
ret

;=====================================================================
; subroutine INPOSJudge
; 4. Determine Required State of Motion Control
;
; inputs: XGO; INPOS
;
; output: INPOS
;=====================================================================
INPOSJudge:
jnb XGO, MCStop     ; Is motion control active?
clr INPOS           ; Then unlock servo.
ret                 ; And finish.
```

```
MCStop:             ; Otherwise:
jnb INPOS, SetINPOS ; Is the servo locked?
ret                 ; Then finish.

SetINPOS:           ; If servo not locked,
setb INPOS          ; Lock servo.
ret

;=====================================================================
; subroutine PPCAL
; 5. Proportional Control Algorithm
;
; inputs: PEXH, PEXL = PEX; KP; KPINP; INPOS

; output: VELXH, VELXL = VELX = KP * PEX if INPOS = 0
;                             = KPINP * PEX if INPOS = 1
;
; alters: acc, register bank 3
;*********************************************************************
PPCAL:
mov a, KP           ; Default to in-motion gain KP.
jnb INPOS, CALPP    ; Is servo locked?
mov a, KPINP        ; If INPOS = 1, use servo locked gain KPINP.

CALPP:
mov r2, a           ; Pad gain to 16-bit precision.
mov r3, #0x00
mov r1, PEXH        ; And multiply with PEX.
mov r0, PEXL
lcall MUL16
mov VELXL, r0       ; Store result in VELX.
mov VELXH, r1
ret

;=====================================================================
; subroutine DAOUTVEL
; 6. Offset Correction for DAC Output
;
; inputs: VELXH, VELXL = VELX
;
; output: digital velocity command value VELDA
;
; alters: acc, b, register bank 3
;=====================================================================
DAOUTVEL:
jb MSIGNALL, VELNEG ; Is commanded velocity negative?
mov r0, #0xFF       ; If positive, add 0x7FFF (offset of 0)
mov r1, #0x7F
mov r2, VELXL
mov r3, VELXH
lcall CADD16
sjmp WROutput       ; Done.

VELNEG:
mov r0, #0xFF       ; If negative, subtract 0x7FFF.
mov r1, #0x7F
mov r2, VELXL
mov r3, VELXH
lcall CSUB16

WROutput:           ; Write offset value to DAC.
mov a, r0
mov b, r1
lcall DAOut
ret

;=====================================================================
```

```
; subroutine INTPAndFXCheck
; 7. Motion Command Interpolation & Final Position Distance Update
;
; inputs: DFXH, DFXL = DFX; FX4, FX3, FX2, FX1 = FX_i; XGO
;
; output: FX4, FX3, FX2, FX1 = FX_{i+1}
;         DXH, DXL = 0 if XGO = 0; else:
;                  = DFX if FX_{i+1} >= 0
;                  = FX_i otherwise
;         XGO = 0 if FX_{i+1} < 0
;             = 1 otherwise
;
; alters: register bank 3
;=====================================================================
INTPAndFXCheck:
jnb XGO, StopINTP   ; If motion control is inactive, skip.
mov r1, DFXH        ; Otherwise, convert DFX to 32-bit precision.
mov r0, DFXL
lcall C16toC32
mov r7, FX4         ; Subtract DFX from FX.
mov r6, FX3
mov r5, FX2
mov r4, FX1
lcall CSUB32
mov FX4, r3         ; Store result in FX.
mov FX3, r2
mov FX2, r1
mov FX1, r0
sjmp FXCheckXGO

FXCheckXGO:
mov a, FX3          ; Is predicted FX negative?
jb acc.7, CLRXGO    ; If yes, we will pass destination.
mov DXH, DFXH       ; If not, use standard DX DFX.
mov DXL, DFXL
ret                 ; Finish.

CLRXGO:
mov DXH, r5         ; If will pass, DX is low bytes of FX... XXX?
mov DXL, r4
clr XGO             ; Will be done next interrupt.
ret                 ; Finish.

StopINTP:
mov DXH, #0x00      ; Motion control inactive => no motion command.
mov DXL, #0x00
ret

;=====================================================================;
;---------------------------------------------------------------------;
;       ARITHMETIC ROUTINES: 16-, 24-, 32-Bit 2's Complement          ;
;---------------------------------------------------------------------;
;=====================================================================;

;=====================================================================
; subroutine CADD16
; 16-Bit Signed (2's Complement) Addition
;
; inputs: r1, r0 = X
;         r3, r2 = Y
;
; output: r1, r0 = signed sum S = X + Y
;
; alters: acc, C, OV, register bank 3
;=====================================================================
CADD16:
orl psw, #0x18
mov a, r0   ; Add low bytes together.
```

```
add a, r2
mov r0, a   ; Save result in S low byte.
mov a, r1   ; Add high bytes together with carry.
addc a, r3
mov r1, a   ; Save result in S high byte.
clr C
ret

;=====================================================================
; subroutine CSUB16
; 16-Bit Signed (2's Complement) Subtraction
;
; inputs: r1, r0 = X
;         r3, r2 = Y
;
; output: r1, r0 = signed difference D = X - Y
;
; alters: acc, C, OV, register bank 3
;=====================================================================
CSUB16:
orl psw, #0x18
clr C
mov a, r0   ; Subtract low bytes.
subb a, r2
mov r0, a   ; Save result in D low byte.
mov a, r1   ; Subtract high bytes with borrow.
subb a, r3
mov r1, a   ; Save result in D high byte.
clr C
ret

;=====================================================================
; subroutine MUL16
; 16-Bit Signed (2's Complement) Multiplication to 32-Bit Product
; with Sign Flag
;
; inputs: r1, r0 = X
;         r3, r2 = Y
;
; output: r3, r2, r1, r0 = product magnitude P = |X x Y|
;         MSIGNALL = sign(P): 1 if negative
;
; calls: UMUL16, Cr0r1, Cr2r3, Mr0r3
;
; alters: acc, C, OV, MSIGN0, MSIGN1, MSIGNALL, register bank 3
;=====================================================================
MUL16:
orl psw, #0x18
clr MSIGNALL    ; Reset product sign flag.
lcall Cr0r1     ; Find magnitude and sign of multiplicand X.
lcall Cr2r3     ; Find magnitude and sign of multiplier Y.
lcall UMUL16    ; Perform unsigned multiplication of |X| x |Y|.
lcall Mr0r3     ; Find sign of product P.
ret

;=====================================================================
; subroutine UMUL16
; 16-Bit Unsigned Multiplication to 32-Bit Product
;
; inputs: r1, r0 = |X|
;         r3, r2 = |Y|
;
; output: r3, r2, r1, r0 = product P = |X| x |Y|
;
; alters: acc, C, OV, register bank 3
;=====================================================================
UMUL16:
orl psw, #0x18
```

```
push b          ; Save extra registers we will use to stack.
push dpl
mov a, r0       ; Multiply XL x YL.
mov b, r2
mul ab
push acc        ; Stack result low byte.
push b          ; Stack result high byte.
mov a, r0       ; Multiply XL x YH.
mov b, r3
mul ab
pop 0x18        ; Pop to R0.
add a, r0       ; Add result low byte to first result high byte.
mov r0, a
clr a
addc a, b       ; Add carry to result high byte.
mov dpl, a      ; Store carried result high byte in DPL.
mov a, r2       ; Multiple XH x YL.
mov b, r1
mul ab
add a, r0       ; Add result low byte to R0.
mov r0, a
mov a, dpl
addc a, b       ; Add result high byte to DPL.
mov dpl, a
clr a
addc a, #0      ; Carry and save to R4.
mov r4, acc
mov a, r3       ; Multiply XH x YH.
mov b, r1
mul ab
add a, dpl      ; Add result low byte to DPL.
mov r2, a       ; Save product 3rd byte.
mov acc, r4     ; Retrieve carry and add to result high byte.
addc a, b
mov r3, a       ; Save product high byte.
mov r1, 0x18    ; Save product second byte from R0.
pop 0x18           ; Retrieve product low byte.
pop dpl         ; Restore registers.
pop b
ret

;=====================================================================
; subroutine Cr0r1
; 16-Bit 2's Complement Magnitude and Sign
;
; inputs: r1, r0 = X
;
; output: r1, r0 = |X|
;         MSIGN1 = sign(X): 1 if negative
;
; alters: acc, C, MSIGN1, register bank 3
;=====================================================================
Cr0r1:
orl psw, #0x18
mov a, r1
jb acc.7, c0a   ; Negative if high byte bit 7 is 1.
clr MSIGN1      ; Clear sign bit if positive.
ret             ; Done.

c0a:
setb MSIGN1     ; Set sign bit because negative.
mov a, r0       ; 1's complement low byte.
cpl a
add a, #1       ; And add 1 to do 2's complement.
mov r0, a       ; Save result low byte.
mov a, r1       ; 2's complement high byte.
cpl a
addc a, #0      ; With carry from low byte.
```

```
mov r1, a       ; Save result high byte.
ret

;=====================================================================
; subroutine Cr2r3
; 16-Bit 2's Complement Magnitude and Sign
;
; inputs: r3, r2 = Y
;
; output: r3, r2 = |Y|
;         MSIGN0 = sign(X): 1 if negative
;
; alters: acc, C, MSIGN0, register bank 3
;=====================================================================
Cr2r3:
orl psw, #0x18
mov a, r3
jb acc.7, c1a   ; Negative if high byte bit 7 is 1.
clr MSIGN0      ; Clear sign bit if positive.
ret             ; Done.

c1a:
setb MSIGN0     ; Set sign bit because positive.
mov a, r2       ; 1's complement low byte.
cpl a
add a, #1       ; And add 1 to do 2's complement.
mov r2, a       ; Save result low byte.
mov a, r3       ; 2's complement high byte.
cpl a
addc a, #0      ; With carry from low byte.
mov r3, a       ; Save result high byte.
ret

;=====================================================================
; subroutine C2Comp
; 16-Bit 2's Complement Operation
;
; inputs: r1, r0 = X
;
; output: r1, r0 = 2's(X)
;
;
; alters: acc, C, register bank 3
;=====================================================================
C2Comp:
orl psw, #0x18
mov a, r0   ; 1's complement low byte.
cpl a
add a, #1   ; And add 1 to do 2's complement.
mov r0, a   ; Save result low byte.
mov a, r1   ; 2's complement high byte.
cpl a
addc a, #0  ; With carry from low byte.
mov r1, a   ; Save result high byte.
ret

;=====================================================================
; subroutine Mr0r3
; Sign Bit Multiplication.
;
; inputs: MSIGN1 = sign(X): 1 if negative
;         MSIGN0 = sign(Y)
;
; output: MSIGNALL = sign(X x Y)
;
; alters: acc, C, MSIGNALL
;=====================================================================
Mr0r3:
```

```
jb MSIGN1, Mr0r3b   ; Test if X or Y are negative.
jb MSIGN0, Mr0r3a
ret                 ; Done if X and Y are positive.

Mr0r3b:
jnb MSIGN0, Mr0r3a  ; Test if Y is negative.
ret                 ; Done if X and Y are negative.

Mr0r3a:
setb MSIGNALL       ; Set sign bit if X xor Y are negative.
ret

;=====================================================================
; subroutine C16toC32
; 16-Bit 2's Complement Cast to 32-Bit 2's Complement
;
; inputs: r1, r0 = X: 16-bit signed (2's complement)
;
; output: r3, r2, r1, r0 = X: 32-bit signed (2's complement)
;
; alters: acc, C
;=====================================================================
C16toC32:
mov r3, #0x00       ; Pad high bytes with zeros.
mov r2, #0x00
mov a, r1           ; Check sign of input.
jnb acc.7, CC32End
mov r3, #0xFF       ; Pad high bytes with ones because negative.
mov r2, #0xFF

CC32End:            ; Done because positive.
ret

;=====================================================================
; subroutine CADD32
; 32-Bit Signed (2's Complement) Addition
;
; inputs: r3, r2, r1, r0 = X
;         r7, r7, r5, r4 = Y
;
; output: r3, r2, r1, r0 = signed sum S = X + Y
;         C is set if overflow occurs
;
; alters: acc, C, OV, register bank 3
;=====================================================================
CADD32:
orl psw, #0x18
mov a, r0   ; Add low bytes together.
add a, r4
mov r0, a   ; Save result in S low byte.
mov a, r1   ; Add 2nd bytes together with carry.
addc a, r5
mov r1, a   ; Save result in S 2nd byte.
mov a, r2   ; Add 3rd bytes together with carry.
addc a, r6
mov r2, a   ; Save result in S 3rd byte.
mov a, r3   ; Add high bytes together with carry.
addc a, r7
mov r3, a   ; Save result in S high byte.
mov C, OV   ; Set carry if overflow occurred.
ret

;=====================================================================
; subroutine CSUB32
; 32-Bit Signed (2's Complement) Subtraction
;
; inputs: r7, r6, r5, r4 = X
;         r3, r2, r1, r0 = Y
```

```
;
; output: r3, r2, r1, r0 = signed difference D = X - Y
;         C is set if overflow occurs
;
; alters: acc, C, OV, register bank 3
;=========================================================================
CSUB32:
orl psw, #0x18
clr C
mov a, r4   ; Subtract low bytes.
subb a, r0
mov r0, a   ; Save result in D low byte.
mov a, r5   ; Subtract 2nd bytes with borrow.
subb a, r1
mov r1, a   ; Save result in D 2nd byte.
mov a, r6   ; Subtract 3rd bytes with borrow.
subb a, r2
mov r2, a   ; Save result in D 3rd byte.
mov a, r7   ; Subtract high bytes.
subb a, r3
mov r3, a   ; Save result in D high byte.
mov C, OV   ; Set carry if overflow occurred.
ret

;=========================================================================
; subroutine C16toC24
; 16-Bit 2's Complement Cast to 24-Bit 2's Complement
;
; inputs: r1, r0 = X: 16-bit signed (2's complement)
;
; output: r2, r1, r0 = X: 24-bit signed (2's complement)
;
; alters: acc, C
;=========================================================================
C16toC24:
orl psw, #0x18
mov r2, #0x00       ; Pad high byte with zeros.
mov a, r1
jnb acc.7, CC24End  ; Check sign of input.
mov r2, #0xFF       ; Pad high byte with ones because negative.

CC24End:            ; Done because positive.
ret

;=========================================================================
; subroutine CADD24
; 24-Bit Signed (2's Complement) Addition
;
; inputs: r2, r1, r0 = X
;         r6, r5, r4 = Y
;
; output: r2, r1, r0 = signed sum S = X + Y
;         C is set if overflow occurs
;
; alters: acc, C, OV, register bank 3
;=========================================================================
CADD24:
orl psw, #0x18
clr C
mov a, r0   ; Add low bytes together.
add a, r4
mov r0, a   ; Save result in S low byte.
mov a, r1   ; Add middle bytes together with carry.
addc a, r5
mov r1, a   ; Save result in S middle byte.
mov a, r2   ; Add high bytes together.
addc a, r6
mov r2, a   ; Save result in S high byte.
```

```
mov C, OV   ; Set carry if overflow occurred.
ret

;=====================================================================
; subroutine CSUB24
; 24-Bit Signed (2's Complement) Subtraction
;
; inputs: r6, r5, r4 = X
;         r2, r1, r0 = Y
;
; output: r2, r1, r0 = signed difference D = X - Y
;         C is set if overflow occurs
;
; alters: acc, C, OV, register bank 3
;=====================================================================
CSUB24:
orl psw, #0x18
clr C
mov a, r4   ; Subtract low bytes.
subb a, r0
mov r0, a   ; Save result in D low byte.
mov a, r5   ; Subtract middle bytes with borrow.
subb a, r1
mov r1, a   ; Save result in D middle byte.
mov a, r6   ; Subtract high bytes with borrow.
subb a, r2
mov r2, a   ; Save result in D high byte.
mov C, OV   ; Set carry if overflow occurred.
ret

; vim: filetype=tasm: tabstop=4:
```