To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: http://www.renesas.com

April 1st, 2010
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (http://www.renesas.com)

Send any inquiries to http://www.renesas.com/inquiry.

Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.

   “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.

   “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.

   “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

Application Note

Stepper Motor Control using the µPD78F0714 Microcontroller

Document No. U17733EE1V0AN00
Date Published September 2005

NOTES FOR CMOS DEVICES

① **VOLTAGE APPLICATION WAVEFORM AT INPUT PIN**

Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (MAX) and V_{IH} (MIN) due to noise, etc., the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (MAX) and V_{IH} (MIN).

② **HANDLING OF UNUSED INPUT PINS**

Unconnected CMOS device inputs can be cause of malfunction. If an input pin is unconnected, it is possible that an internal input level may be generated due to noise, etc., causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using pull-up or pull-down circuitry. Each unused pin should be connected to V_{DD} or GND via a resistor if there is a possibility that it will be an output pin. All handling related to unused pins must be judged separately for each device and according to related specifications governing the device.

③ **PRECAUTION AGAINST ESD**

A strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it when it has occurred. Environmental control must be adequate. When it is dry, a humidifier should be used. It is recommended to avoid using insulators that easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors should be grounded. The operator should be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with mounted semiconductor devices.

④ **STATUS BEFORE INITIALIZATION**

Power-on does not necessarily define the initial status of a MOS device. Immediately after the power source is turned ON, devices with reset functions have not yet been initialized. Hence, power-on does not guarantee output pin levels, I/O settings or contents of registers. A device is not initialized until the reset signal is received. A reset operation must be executed immediately after power-on for devices with reset functions.

⑤ **POWER ON/OFF SEQUENCE**

In the case of a device that uses different power supplies for the internal operation and external interface, as a rule, switch on the external power supply after switching on the internal power supply. When switching the power supply off, as a rule, switch off the external power supply and then the internal power supply. Use of the reverse power on/off sequences may result in the application of an overvoltage to the internal elements of the device, causing malfunction and degradation of internal elements due to the passage of an abnormal current.

The correct power on/off sequence must be judged separately for each device and according to related specifications governing the device.

⑥ **INPUT OF SIGNAL DURING POWER OFF STATE**

Do not input signals or an I/O pull-up power supply while the device is not powered. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements.

Input of signals during the power off state must be judged separately for each device and according to related specifications governing the device.

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics America Inc.
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Europe) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 1101
Fax: 0211-65 03 1327

Sucursal en España
Madrid, Spain
Tel: 091- 504 27 87
Fax: 091- 504 28 60

Succursale Française
Vélizy-Villacoublay, France
Tel: 01-30-67 58 00
Fax: 01-30-67 58 99

Filiale Italiana
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

Branch The Netherlands
Eindhoven, The Netherlands
Tel: 040-244 58 45
Fax: 040-244 45 80

Branch Sweden
Taeby, Sweden
Tel: 08-63 80 820
Fax: 08-63 80 388

United Kingdom Branch
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-528-0303
Fax: 02-528-4411

NEC Electronics Singapore Pte. Ltd.
Singapore
Tel: 65-6253-8311
Fax: 65-6250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377
Fax: 02-2719-5951

Table of Contents

List of Figures

List of Tables

Chapter 1 Overview

1.1 Abstract

This application note shows how to implement a controller for a stepper motor using the µPD78F0714 along with a simple analog Drive circuit.
Source code, schematic, bill of material, and board layout files are provided.

1.2 Introduction

Nowadays, stepper motors are used in a wide variety of applications. They are prevalent in consumer office equipment such as printers, scanners, copiers and plotters. They also play an important role in the industry, use in robotics or dashboard indicators, climate control systems in the automotive industry. Purpose of this application note is to show how a stepper motor control is realised on the µPD78F0714 with as few external parts as possible. The software and hardware configurations published here are just examples and are not intend for mass production.

1.3 Overview of µPD78F0714

Table 1-1: Functional Outline

Item		µPD78F0714
Internal memory	Flash memory (self-programming supported)	32 KB
	High-speed RAM	1 KB
Memory space		32 KB
X1 input clock (oscillation frequency)		Ceramic/crystal/external clock oscillation [20 MHz (V_{DD} = 4.0 to 5.5 V)]
Ring-OSC clock (oscillation frequency)		On-chip Ring oscillation (240 kHz (TYP.))
General-purpose registers		8 bits × 32 registers (8 bits × 8 registers × 4 banks)
Minimum instruction execution time		0.1 µs/0.2 µs/0.4 µs/0.8 µs/1.6 µs (X1 input clock: @ f_{XP} = 20 MHz operation)
		8.3 µs/16.6 µs/33.2 µs/66.4 µs/132.8 µs (TYP.) (Ring-OSC clock: @ f_R = 240 kHz (TYP.) operation)
Instruction set		• 16-bit operation • Multiply/divide (8 bits × 8 bits, 16 bits ÷ 8 bits) • Bit manipulate (set, reset, test, and Boolean operation) • BCD adjust, etc.
I/O ports		<u>Total:</u> <u>48</u> CMOS I/O 40 CMOS input 8
Timers		• 10-bit inverter control timer: 1 channel • 16-bit up/down counter: 1 channel • 16-bit timer/event counter: 1 channel • 8-bit timer/event counter: 2 channels • 8-bit timer: 1 channel • Watchdog timer: 1 channel
	Timer outputs	11 (inverter control output: 6)
Clock output		156.25 kHz, 312.5 kHz, 625 kHz, 1.25 MHz, 2.5 MHz, 5 MHz, 10 MHz, 20 MHz (X1 input clock: 20 MHz)
Buzzer output		2.44 kHz, 4.88 kHz, 9.77 kHz, 19.5 kHz (X1 input clock: 20 MHz)
Real-time output ports		• 8 bits × 1 or 4 bits × 2 • 6 bits × 1 or 4 bits × 2
A/D converter		10-bit resolution × 8 channels
Serial interface		• UART mode: 1 channel • 3-wire serial I/O mode: 1 channel
Multiplier/divider		• 16 bits × 16 bits = 32 bits (multiplication) • 32 bits ÷ 16 bits = 32 bits remainder of 16 bits (division)
Vectored interrupt sources	Internal	20
	External	8
Reset		• Reset using $\overline{\text{RESET}}$ pin • Internal reset by watchdog timer • Internal reset by power-on-clear • Internal reset by low-voltage detector
Supply voltage		V_{DD} = 4.0 to 5.5 V
Operating ambient temperature		T_A = -40 to +85°C
Package		64-pin plastic TQFP (fine pitch) (12 × 12)

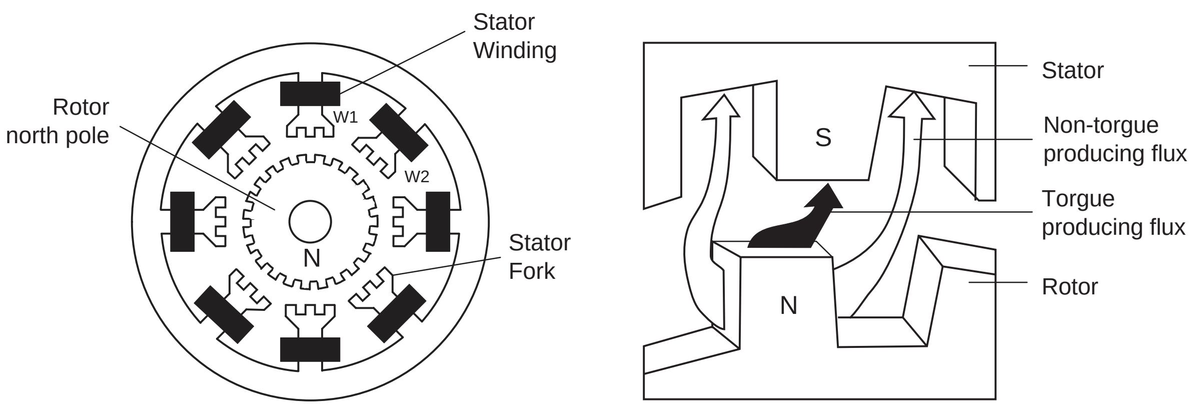
Caution: The operating voltage range may be changed after evaluation of the device.

Chapter 2 Stepper Motor Basics

2.1 Stepper Motor Basics

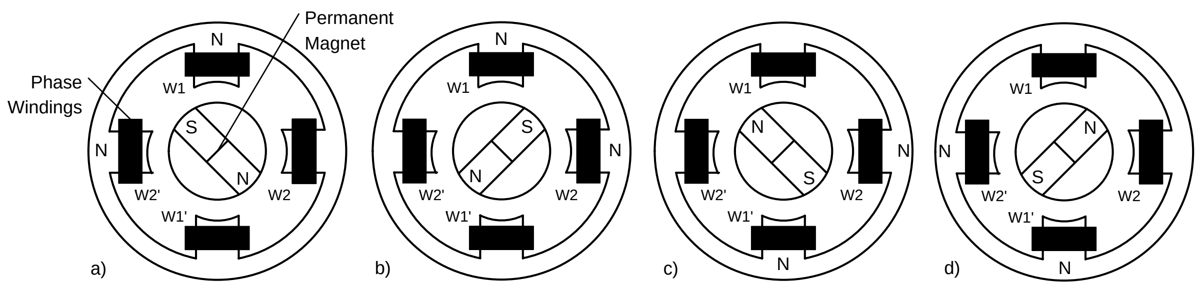
A step motor is an electromagnetic, rotary actuator, which mechanically converts digital pulse inputs to incremental shaft rotation. The rotation has not only a direct relation to the number of input pulses, but its speed is related to the frequency of the pulses.
The motor is able to holds its' position (and its' load) between the steps without the aid of clutches or brakes. Thus a step motor can be precisely controlled so that it rotates a certain number of steps, producing mechanical motion through a specific distance, and then holds its load when it stops. Furthermore, it can repeat the operation at any prescribed number of times.
With the appropriate logic, step motors can be bi-directional, synchronous, provide rapid acceleration, stopping, and reversal, and will interface easily with other digital mechanisms. They are further characterized as having low rotor moment of inertia, no drift, and a non cumulative positioning error.
Generally step motors are operated without feedback in an open-loop manner and often match the performance of more expensive DC Servo positioning Systems.
Stepper motors may be classified by their motor construction, drive topology, and stepping pattern. There are several different types of stepper motor construction. These include variable reluctance, permanent magnet, and hybrid permanent magnet. This reference design is applicable to the permanent magnet and hybrid two phase stepper motors.
The hybrid rotor is constructed using a cylindrical permanent magnet oriented with the north-south polarity along the rotor axis. Two laminated end caps are used with many teeth around the periphery. The north and south teeth are staggered to provide many effective poles using a single permanent magnet. The stator laminates typically have four large forks. Each fork has many teeth. The teeth for the two windings are also staggered to line up with the appropriate teeth on the rotor.

Figure 2-1: Hybrid Stepper Motor



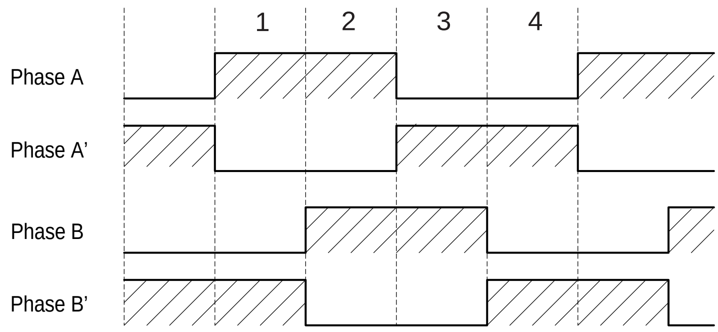
The drive topology of stepper motors is also an important criterion for choosing a motor.
Here are two main topologies to mention, unipolar and bipolar driving.
Unipolar stepping motors are composed of two windings, each with a center tap. The center taps are either brought outside the motor as two separate wires or connected to each other internally and brought outside the motor as one wire. As a result, unipolar motors have 5 or 6 wires. Regardless of the number of wires, unipolar motors are driven in the same way. The center tap wire(s) is tied to a power supply and the ends of the coils are alternately grounded.
Bipolar stepping motors are composed of two windings and have four wires. Unlike unipolar motors, bipolar motors have no center taps. The advantage to not having center taps is that current runs through an entire winding at a time instead of just half of the winding. As a result, bipolar motors produce more torque than unipolar motors of the same size. The draw back of bipolar motors, compared to unipolar motors, is that more complex control circuitry is required by bipolar motors. That is the main reason why in this application a unipolar drive topology is used, to keep the external parts at a minimum.
The basic movement of the motor can be best shown by reducing the defaults to the simplest arrangements. We look at the rotor as one permanent magnet with north-south polarity and the stator comes down to four magnetic poles.

Figure 2-2: 2 Phase Stepper Motor with one pole pair permanent magnet



Unipolar stepping motors operate by attracting the north or south poles of the permanently magnetized rotor to the stator poles. Thus, in these motors, the direction of the current through the stator windings determines which rotor poles will be attracted to which stator poles. Current direction in unipolar motors is dependent on which half of a winding is energized. Physically, the halves of the windings are wound parallel to one another. Therefore, one winding acts as either a north or south pole depending on which half is powered.
In the figure above you can see the four different energizing phases that are necessary to rotate the shaft one time. In the first phase W1 and W2’ are energized, in the second the current flows through W1 and W2 etc. In this application design a variant where two windings are energized at the same time has been chosen to produce more torque. The following figure shows the different energizing states for the example motor above.

Figure 2-3: Energizing States Full-Step

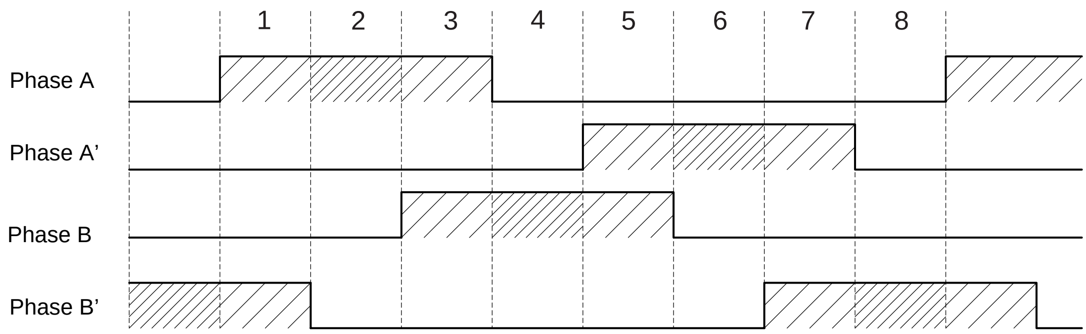


The example motor uses now only four steps for one rotation, in reality the angular resolution is wide spread and depends on the motor data.
The angular resolution that can be executed depends on number of phases p and how many pole pairs m there are:

$$\alpha = \frac{360°}{2 \cdot p \cdot m} \tag{1}$$

The calculated stepper angle applies only with full step-by-step operation mode of the stepping motor. In addition, the so-called half step modus and other step routines are possible. The differences are described in the following.
The full step modus is already shown in the example above, means that you need four steps in the simplest arrangements to obtain one shaft rotation. The angular resolution is 90 degrees.
The difference of the half step operation lies in the fact that the phase coils are not always energized at the same time, but, as the associated figure shows also is switched off. Thus each step is halved, so that for a revolution 8 steps are necessary. The stepper angle halved itself thus on 45 degrees.

Figure 2-4: Energizing States Half-Step

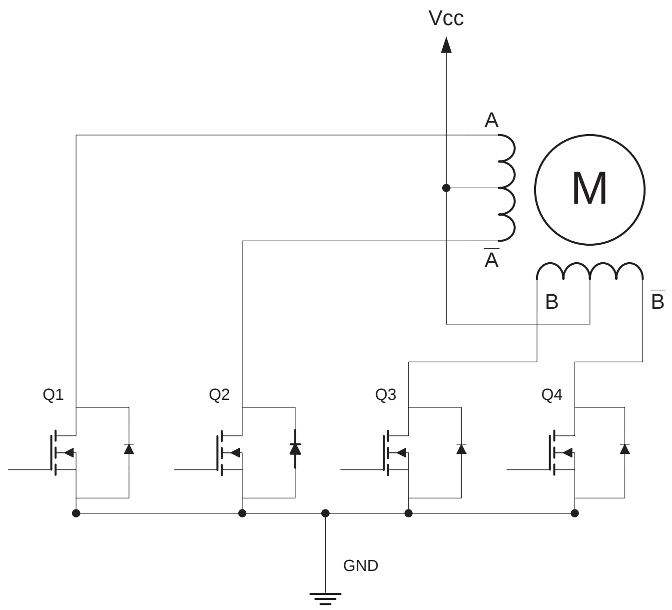


Apart from the described kinds of step modes there is still the mini or micro step operation. By digital tax logic the newest micro step systems reach 250 micro steps for each full step, thus max. 50000 steps for each revolution. This application design doesn't deal with the problems of micro stepping, so the two main drive topologies here are full- and half step operation mode.

2.2 Stepper Motor Control Requirements

As already mentioned this application design deals with an unipolar drive topology and focuses on the full and half-step operation mode. The principal driver design for an unipolar stepper motor is shown below, in Figure 2- 5.

Figure 2-5: Unipolar Stepper Motor drive

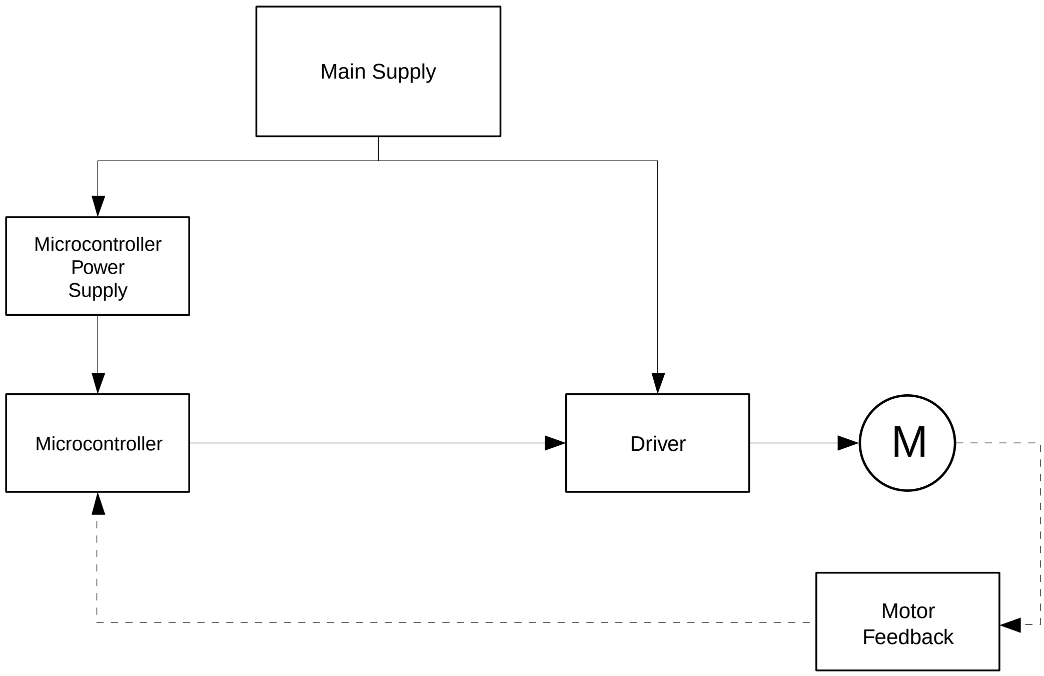


The circuit contains four power mosfets responsible for the current flow through the windings. The center tap of the motor winding is connected to the positive voltage supply. Each coil can be energized in either direction by turning on the appropriate MOSFET. The driving pattern is similar to the energizing states shown in Figure 2-3 and 2-4.

When a motor is operated at a fixed rated voltage its torque output decreases as step rate rises. This is because the increasing back EMF and the rise time of the coil current limits the power actually delivered to the motor. The effect is governed by the motor time constant (L/R). Because of their higher winding resistance unipolar motors have a better L/R ratio than their bipolar equivalents.
The effect can be compensated by either increasing the power supply voltage to maintain constant current as stepping rate increases, or by increasing supply voltage by a fixed
amount and adding series resistors to the circuit.
There is good reason to run a stepping motor at a supply voltage above that needed to push the maximum rated current through the motor windings. Running a motor at higher voltages leads to a faster rise in the current through the windings when they are turned on, and this, in turn, leads to a higher cutoff speed for the motor and higher torques at speeds above the cutoff.
In this application design a PI Regulator is used to maintain current at an average user defined level. The whole PI Regulator is software based to keep external parts as few as possible.

The following figure shows a schematic for general motor control design with a microcontroller.

Figure 2-6: General Motor Control Design



The functions of the components in detail:

Main Supply	Provides circuit energy
Microcontroller Power Supply	Regulates voltage and current for the microcontroller
Microcontroller	Produces the accurate signals for switching the mosfets also contains protection circuit, which ensures on change of the clock frequency that the stepping motor does not loose steps. Microcontroller observes and regulates the current flow through the motor.
Driver	Switches the power necessary for the motor phases.
Current Sensor	Gives continuously information about the current flow through the windings to the microcontroller.

Stepper motor control requirements are summarized below:

- Driver circuit to provide necessary power for the stepper motor
- Current limiting device (software based)
- Interrupted based microcontroller algorithm to produce exact output signals

For the closed loop current control of the motor

- Measurement of the motor current
- PWM Signal to control motor current and power

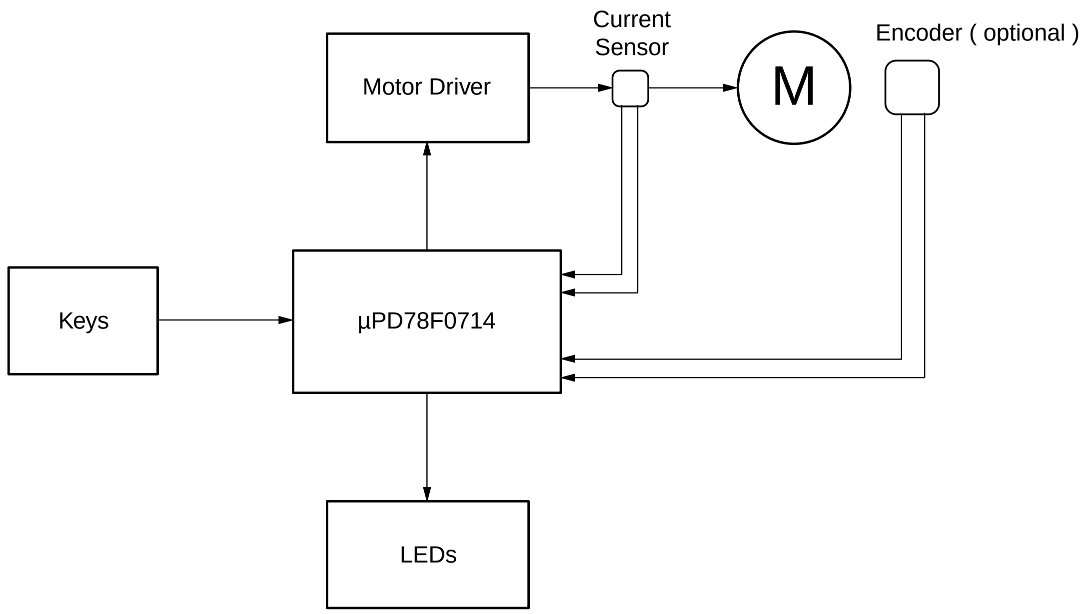
[MEMO]

Chapter 3 System Design Concept

3.1 System Concept

Figure 3-1 shows the principal block diagram of the system concept for the stepper motor.

Figure 3-1: Principal Block Diagram of the System Configuration

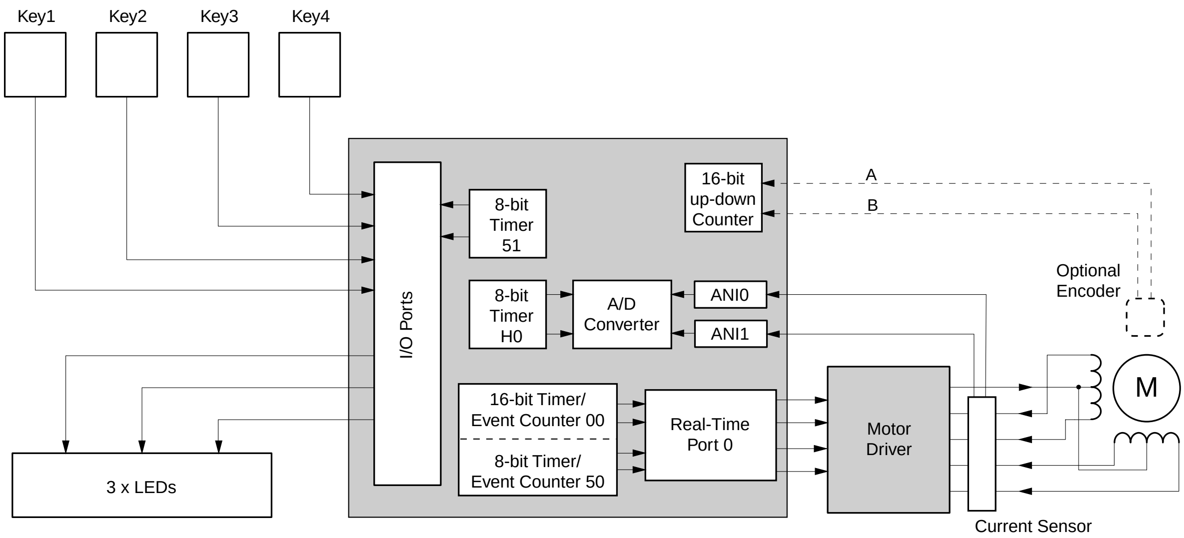


The µPD78F0714 processes the feedback of the current sensor to control the motor driver that supplies the current flow through the windings. An encoder can be optional added to get active feedback of the rotor position.

3.2 System Configuration

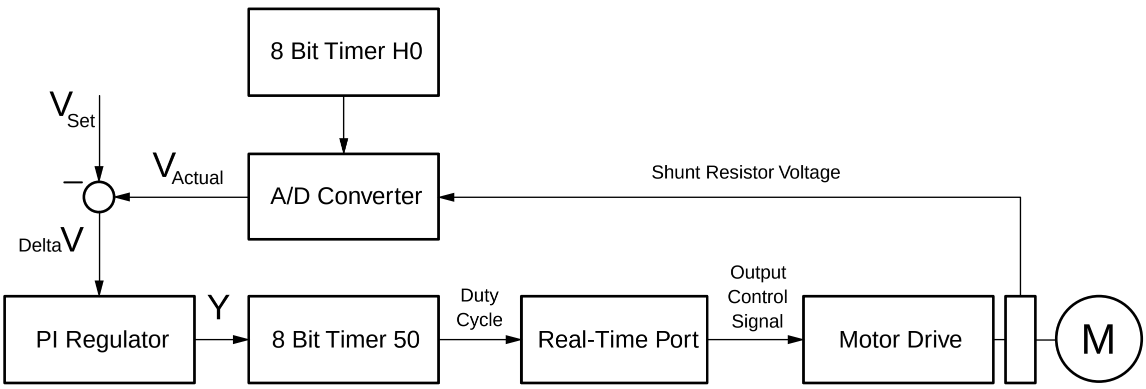
Figure 3-2 shows the system configurations and the peripherals of the µPD78F0714 device used for the stepper motor control.

Figure 3-2: System Configuration with the Peripherals of the µPD78F0714



The Keys are control elements for enabling different running modes for the stepper motor.
The key inputs are sampled by the Interval Timer 51. The 16- Bit Timer/Event Counter 00 and Real–Time Port 0 generate the motor signals for shaft rotation. The actual current flow is detected over extern shunt resistors. The 8- Bit Timer H0 communicates with A/D Converter and defines when to start a conversion. In current dependence Timer 50 generates a PWM with different duty cycle to keep current at the user defined set point. Optional an extern encoder can be implemented to observe the actual position. The encoding of these signals can be realised with the 16- Bit Up/Down Counter. The three LED's visualize the different running modes the stepper motor is in. The function from each peripheral is described in the next chapter. The system topology with the relationship between hardware and software is shown in Figure 3- 3.

Figure 3-3: System Topology and Relationship between the Control Software and the Hardware of the System



Chapter 4 Hardware Configuration

This section describes the hardware requirements for this application example.

4.1 µPD78F0714 Configuration

The µPD78F0714 device is a member of the high performance 78K Family 8-bit microcontrollers, designed specifically for mid-range motor control. The configuration of the device and the operating environment used in this application is listed below:

- CPU: µPD78F0714
- Operating clock: System clock 20 MHz
- Operating Voltage: 5 V
- Internal ROM: 32 Kbytes
- Internal RAM: 1024 bytes
- External expansion memory: not used.

4.2 Peripherals I/O Assignments

Table 4-1 lists all pins of the µPD78F0714 device and the ones that are used in this application are described with their associated function.

Table 4-1: µPD78F0714 Peripherals I/O Assignments (1/2)

Pin No.	Pin Name	Mode Setting	Function
1	AVREF		Extern reference Voltage PIReg.
2	AVSS		Connect to Ground
3	FLMD0	Output	Not used
4	VDD		Power Supply
5	VSS		Ground
6	X1	Input	System Clock
7	X2		System Clock
8	$\overline{\text{RESET}}$	Input	Reset Input
9	INTP3	Output	Not used
10	INTP2	Output	Not used
11	INTP1	Output	Not used
12	INTP0	Output	Not used
13	P30	Output	Not used
14	P31	Output	Not used
15	P32	Output	Not used
16	P33	Output	Not used
17	P50	Output	Not used
18	P51	Output	Not used
19	P52	Output	Not used
20	P53	Output	Not used
21	P54	Output	Not used
22	P55	Output	Not used
23	P56	Output	Not used
24	P57	Output	Not used
25	EVSS		Connect to Ground
26	EVDD		Connect to VDD
27	RTP10	Output	Not used
28	RTP11	Output	Not used
29	RTP12	Output	Not used
30	RTP13	Output	Not used
31	RTP14	Output	Not used
32	RTP15	Output	Not used
33	P10	Output	Not used
34	P11	Output	Not used
35	P12	Output	Not used
36	P13	Output	Not used
37	P14	Output	Not used

Table 4-1: µPD78F0714 Peripherals I/O Assignments (2/2)

Pin No.	Pin Name	Mode Setting	Function
38	P15	Output	Not used
38	P16	Output	Not used
40	P17	Output	Not used
41	RTP00	Output	Phase A
42	RTP01	Output	Phase B
43	RTP02	Output	Phase $\overline{A}$
44	RTP03	Output	Phase $\overline{B}$
45	RTP04	Output	Not used
46	RTP05	Output	Not used
47	RTP06	Output	Not used
48	RTP07	Output	Not used
49	P64	Input	Tracer 1
50	P65	Input	Tracer 2
51	P66	Input	Tracer 3
52	P67	Input	Tracer 4
53	P70	Output	LED red
54	P71	Output	LED yellow
55	P72	Output	LED green
56	P73	Output	Not used
57	ANI7	Output	Not used
58	ANI6	Output	Not used
59	ANI5	Output	Not used
60	ANI4	Output	Connect to Ground
61	ANI3	Output	Connect to VDD
62	ANI2	Output	Not used
63	ANI1	Input	Voltage Shunt 2
64	ANI0	Input	Voltage Shunt 1

4.3 8-bit Timer H0 Function

As shown in the hardware explanation, the time the AD conversion starts is determined by 8-bit timer H0 of the µPD78F0714 device.
The timer has the following operation modes:

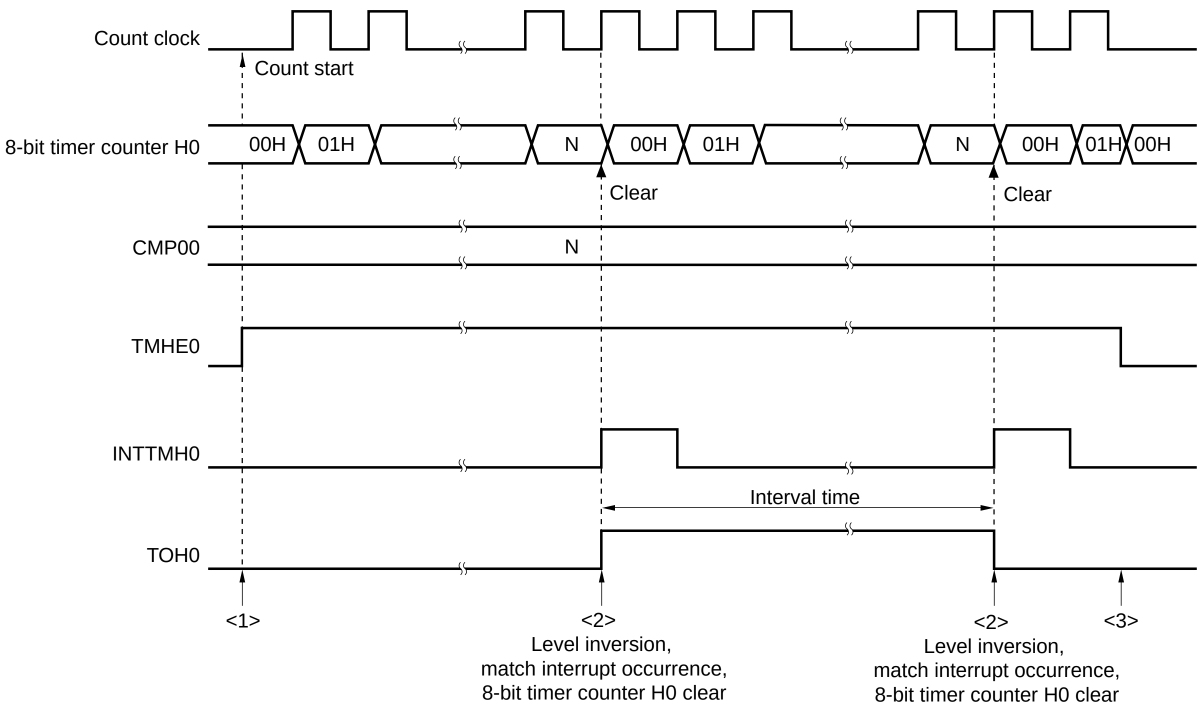
- Interval timer
  - Generates interrupt request at the preset time interval

- PWM output mode
  - A pulse with an arbitrary duty and arbitrary cycle can be output

- Square-wave output
  - Outputs a square wave with any selected frequency.

The interval timer mode was chosen, to define in dependence of 16-bit timer 00 how often an AD conversion occurs in one motor step. An interrupt is generated at the user defined time and the AD conversion is performed.Timer H0 and TM00 work with the same frequency to guarantee synchronously operating.
Figure 4-1 describes the principal flow of TMH0 in the interval timer mode.

Figure 4-1: Timing of TMH0 Interval Timer Operation



Interval time = (N +1)/f_{CNT}

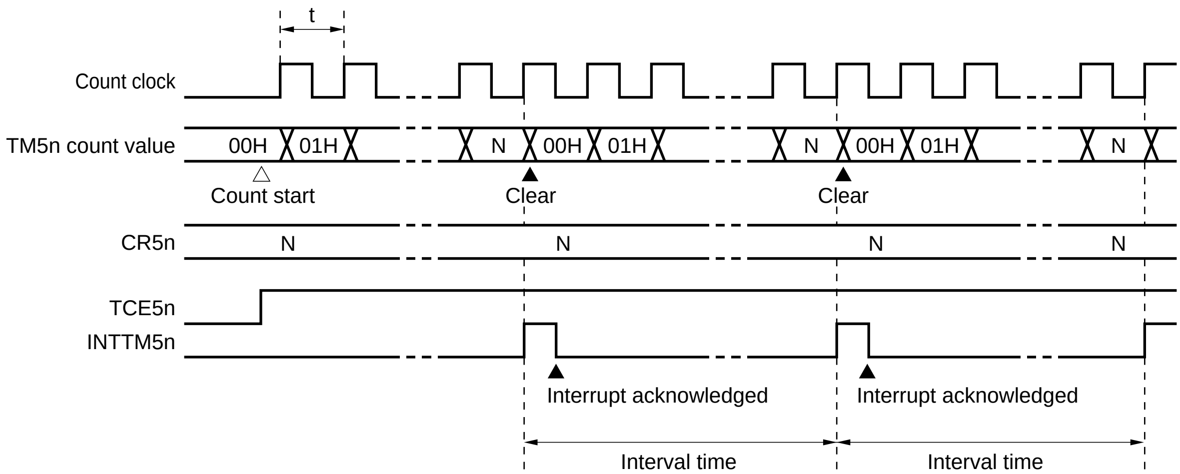
4.4 8-bit Timer 51 Function

TM 51 has different operation modes:

- Interval timer
  - Generates interrupt request at the preset time interval
- External event counter
  - Counts number of external clock pulses to be input to the TI51 pin
- Square-wave output
  - Outputs a square wave with any selected frequency
- PWM output
  - A pulse with an arbitrary duty and arbitrary cycle can be output

Timer 51 is run in the interval mode to continuously check if a tracer is pressed, also handles the bouncing control.
Following figure demonstrates the basic timing diagram for the interval mode.

Figure 4-2: Timing of TM51 Interval Timer Operation

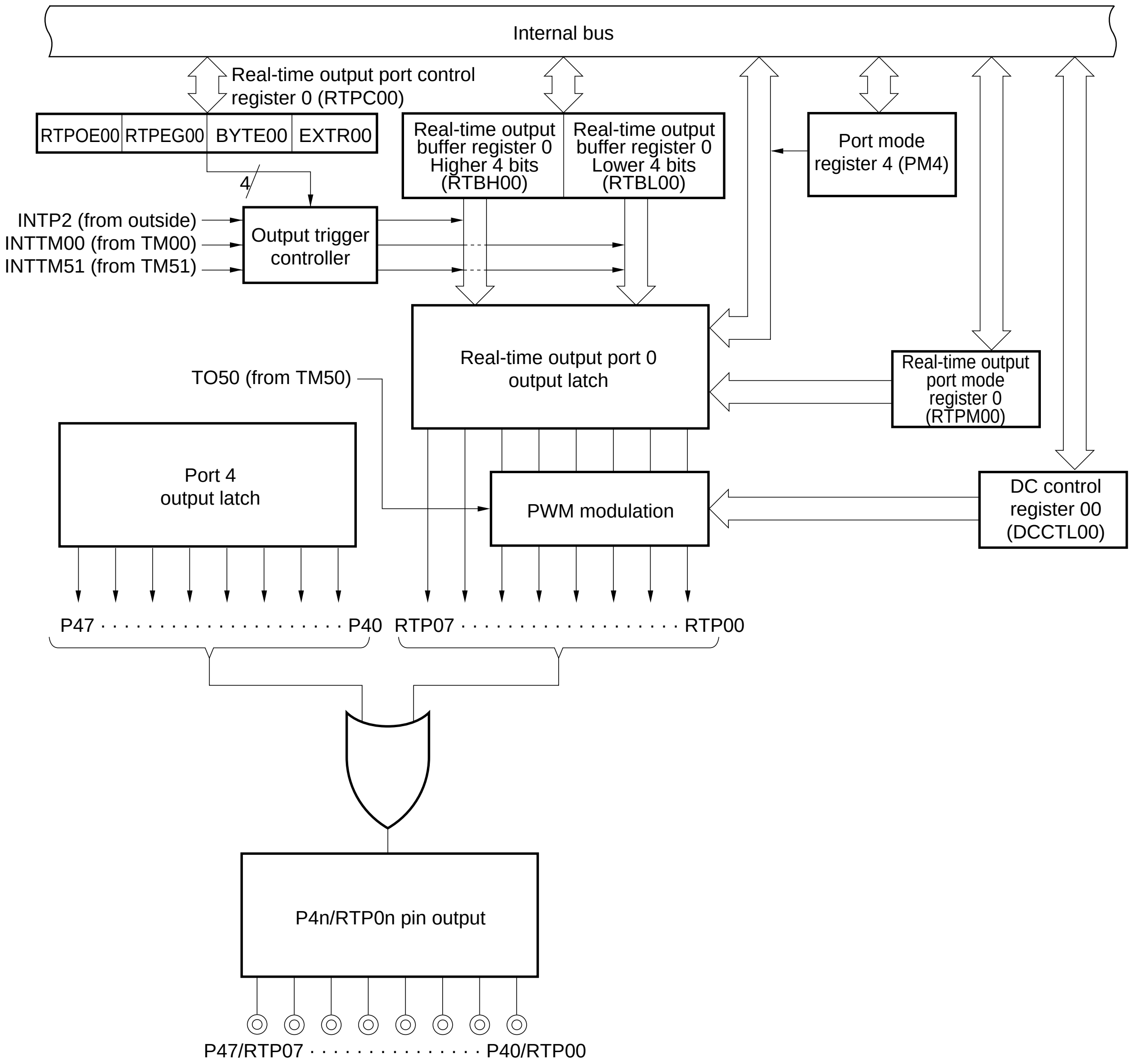


Remark: Interval time = (N + 1) × t
N = 00H to FFH
n = 0, 1

4.5 Real Time Port 0 Function

The Real -Time output Port (RTP) transfers previously set data in the real-time buffer register to the output latch by hardware. The transfer is controlled with timer interrupts or external interrupt request generation. It is also possible to perform PWM modulation of a special pin with output pattern that can be specified in one bit unit.
The µPD7F0714 has 2 channels of real-time output ports on chip. The RTP0 port is shared with Port 4 and RTP1 is shared with inverter control timer. The real-time port used in this application is the RTP0 port. Therefore the function of the RTP0 port will be described in detail.

Figure 4-3: Block Diagram of Real-Time Output Port RTP0



Remark: n = 0 to 7

Figure 4-3 shows the block diagram of the real-time output port RTP0 that shares the output with P4.

The real-time output buffer register 0 (RTBH00, RTBL00) is the register that holds the data in advance. It is specified in entirely 8 bits that can be select either as 1 channel x 8 bits or 2 channels x 4 bits. The real time output mode is set with the port mode register RTPM00 that allows 1-bit units selection. The real-time output port control register RTPC00 sets the operating mode, enables/disables the operation of the real-time output port. The DC control register DCCTL00 controls the PWM modulation, enabling/ disabling of the output waveform inversion.
The relationship between the register settings of the real-time output port and the effects on the output is described in the Table 4-2 below:

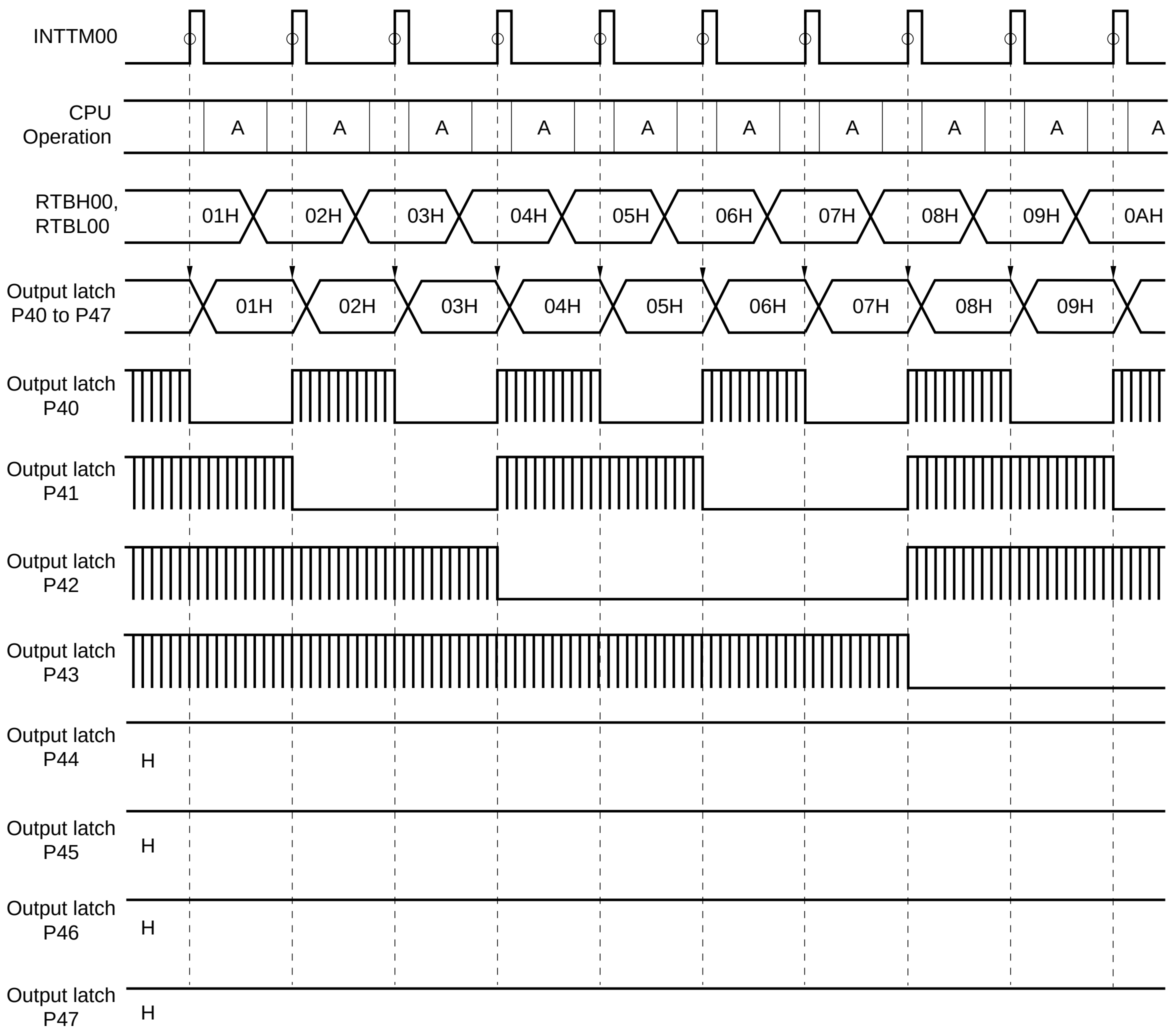
Table 4-2: Relationship Between Settings of Each Bit of Control Register and Real-Time Output

<table>
<tr><th>PM4n</th><th>P4n</th><th>DCEN00</th><th>INV00</th><th>PWMCH00/
PWMCL00</th><th>RTPOE00</th><th>RTPM00n</th><th>RTBH00m/
RTBL00m</th><th>Pin P4n Status</th></tr>
<tr><td>1</td><td>×</td><td>×</td><td>×</td><td>×</td><td>×</td><td>×</td><td>×</td><td>Input port</td></tr>
<tr><td rowspan="21">0</td><td>1</td><td>×</td><td>×</td><td>×</td><td>×</td><td>×</td><td>×</td><td>“high” output</td></tr>
<tr><td rowspan="20">0</td><td rowspan="4">0</td><td rowspan="4">×</td><td rowspan="4">×</td><td>0</td><td>×</td><td>×</td><td>“low” output</td></tr>
<tr><td rowspan="3">1</td><td>0</td><td>×</td><td>“low” output</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>“low” output</td></tr>
<tr><td>1</td><td>“high” output</td></tr>
<tr><td rowspan="16">1</td><td rowspan="8">0</td><td rowspan="4">0</td><td>0</td><td>×</td><td>×</td><td>“low” output</td></tr>
<tr><td rowspan="3">1</td><td>0</td><td>×</td><td>“low” output</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>“low” output</td></tr>
<tr><td>1</td><td>“high” output</td></tr>
<tr><td rowspan="4">1</td><td>0</td><td>×</td><td>×</td><td>“TO50” output</td></tr>
<tr><td rowspan="3">1</td><td>0</td><td>×</td><td>“TO50” output</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>“TO50” output</td></tr>
<tr><td>1</td><td>“high” output</td></tr>
<tr><td rowspan="8">1</td><td rowspan="4">0</td><td>0</td><td>×</td><td>×</td><td>“high” output</td></tr>
<tr><td rowspan="3">1</td><td>0</td><td>×</td><td>“high” output</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>“high” output</td></tr>
<tr><td>1</td><td>“low” output</td></tr>
<tr><td rowspan="4">1</td><td>0</td><td>×</td><td>×</td><td>“TO50” output</td></tr>
<tr><td rowspan="3">1</td><td>0</td><td>×</td><td>“TO50” output</td></tr>
<tr><td rowspan="2">1</td><td>0</td><td>“TO50” output</td></tr>
<tr><td>0</td><td>“low” output</td></tr>
</table>

The interaction between the generated signal from the 16-bit timer and the modulation of it with the real time output port makes generation of a wide range of signal wave forms possible. The solution of the signal generation for the control of the stepper motor drive circuit will be described in Chapter 5, where the software will also be introduced and described. As already mentioned TM00 is responsible for the general shape of the curve, where TM 50 generates the PWM. Following figure shows the connection between these two timers.

Figure 4-4: Real-Time Output Port Operation Timing Example

8 bits × 1 channel, inverted output enabled, PWM modulation
(EXTR00 = 0, BYTE00 = 1, INV00 = 1, PWMCH00 = 1, PWMCL00 = 1)



Remark: A: INTTM00 software processing (RTBH00, RTBL00 write)

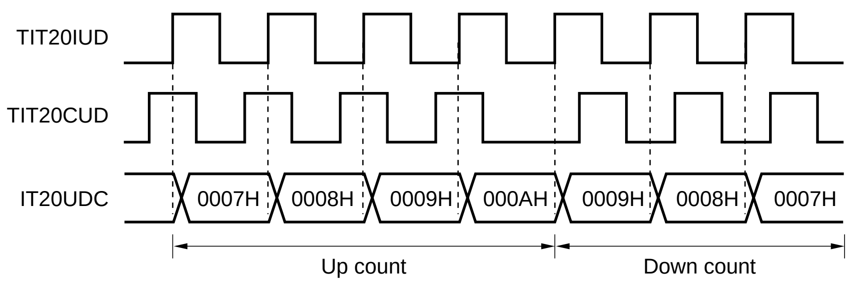
4.6 16-bit Up/Down Counter Function

As an additionally option an extern quadrature encoder can be added. To work with these signals the 16-bit up/down timer can be used. This timer can work with 2-phase extern encoder signals as the count clock of the timer/counter via extern input pins. The following modes can be achieved with this timer.

- Mode 1
  - Counts the input pulses of the count pulse input pin. Up down is specified by the level of the other input pin.
- Mode 2
  - Counts up/down using the respective input pulses of the up count pulse input pin and down count pulse input pin.
- Mode 3
  - Counts up/down using the phase relationship of the pulses input to the 2 pins
- Mode 4
  - Counts up/down using the phase relationship of the pulses input to the 2 pins. Counting is done using the respective rising and falling edges of the pulses.

Mode 3 was chosen to keep exact track of the signal. Figure 4-5 shows basic working condition of the timer for mode 3, TIT20IUD and TIT20CUD represent the two input pins.

Figure 4-5: Timing for Up/Down Counter in Mode 3



4.7 Motor Specification

The specification of the stepper motor used in this application note is as follows:

- Related Voltage 5.7 V
- Current per Phase 1 A
- Resistance per Phase 5.7 Ω
- Inductance 5.4 mH

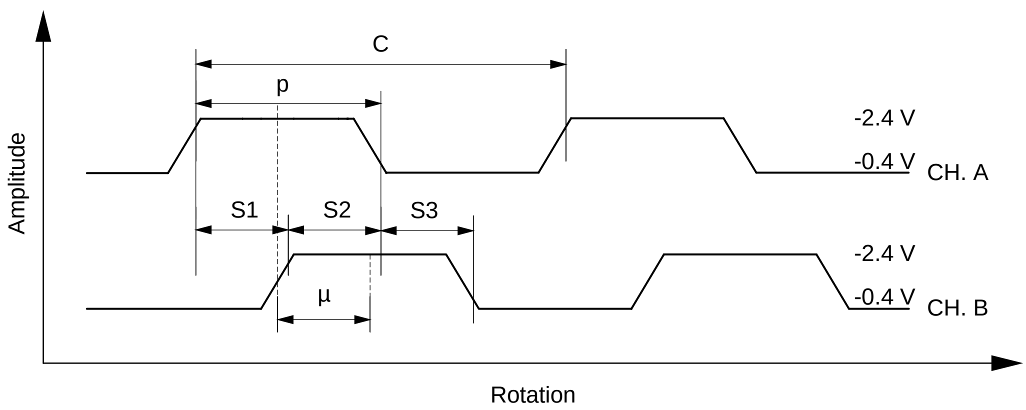
All motors can be used that are able to work in the unipolar mode, the only difference must be made in the user defined values, to adjust the software to the particular stepper motor.
In this application design the Oriental Motor PK264-01A stepper motor is used. Oriental Motors provides also a solution where the quadrature encoder, that fully satisfies the requirements, is already included in the motor.

4.8 Encoder Specification

The optional included encoder should generally have following specification and provide the signals shown in Figure 4-6.

- Supplied Voltage 5 V
- Resolution per Step Up to 1024 Counts
- Load Capacity max. 100 pF
- TTL Compatible

Figure 4-6: General Signal Process of the Encoder



In order to shorten the rise time of the output pulse channel, the outputs be pulled up with a resistance of 2.7 kΩ.

4.9 Stepper Motor Driving Circuit and User Interface Circuit

Figure 4-7 is a schematic of the motor driver and user interface used in the development of this application note.

Complete part list for Figure 4-7:

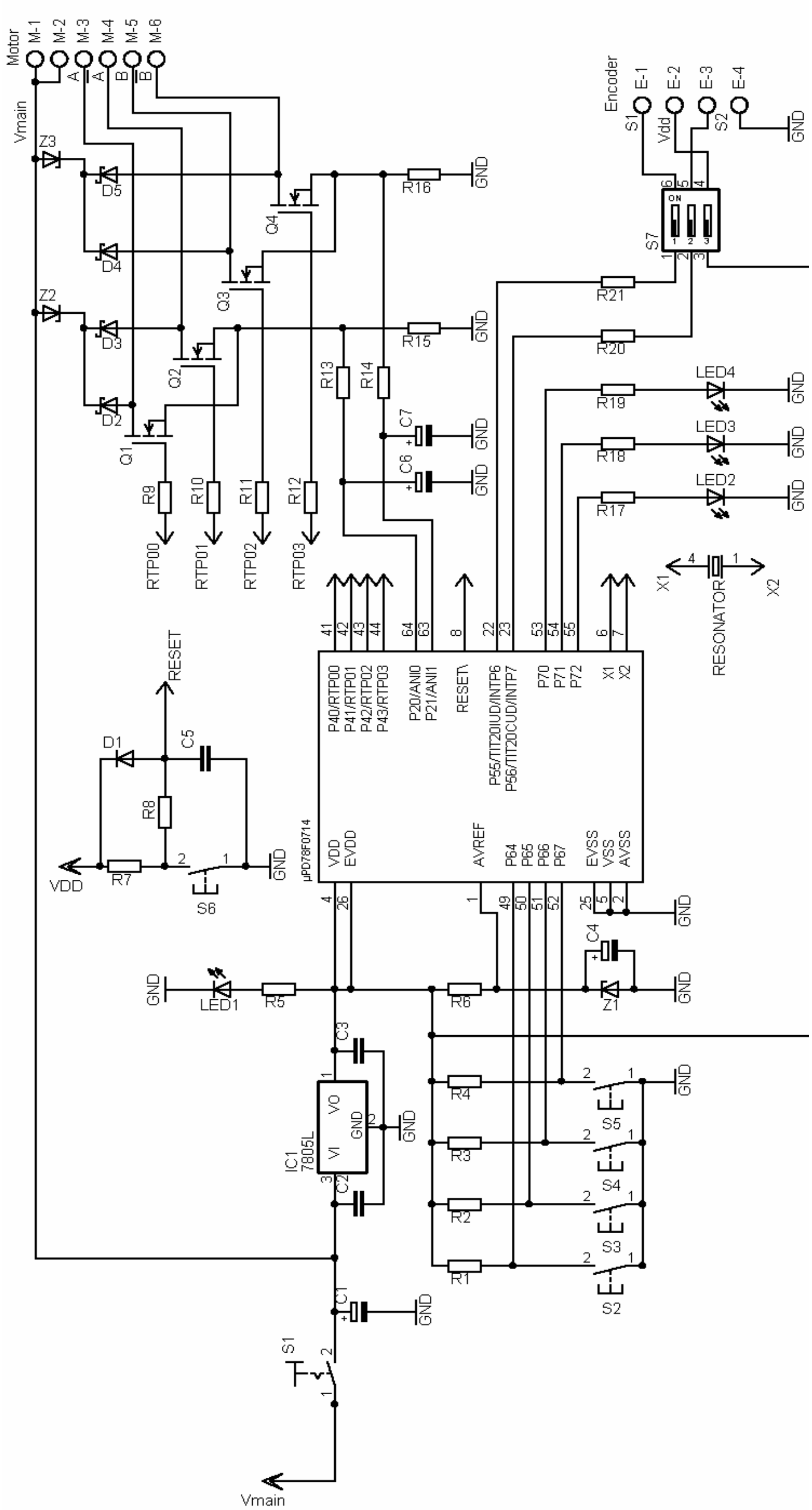
Resistors	
R1...R4, R13, R14	10 kΩ
R5, R17, R18, R19	330 Ω
R6, R9...R12	100 Ω
R7	1 kΩ
R8	10 Ω
R15,R16	0.25 Ω
R20, R21	2.7 kΩ

Capacitors	
C1	470 µF
C2	0.33 µF
C3	0.1 µF (ceramic)
C4	0.1 µF
C5	1 µF
C6, C7	0.47 µF

Diodes	
Zener Diodes	
Z1	4.7 V
Z2, Z3	23 V
Shottky Diodes	
D2... D5	1N58190210
D1	1N4148

Mosfets	
Q1... Q4	NEC NP36N055HLE

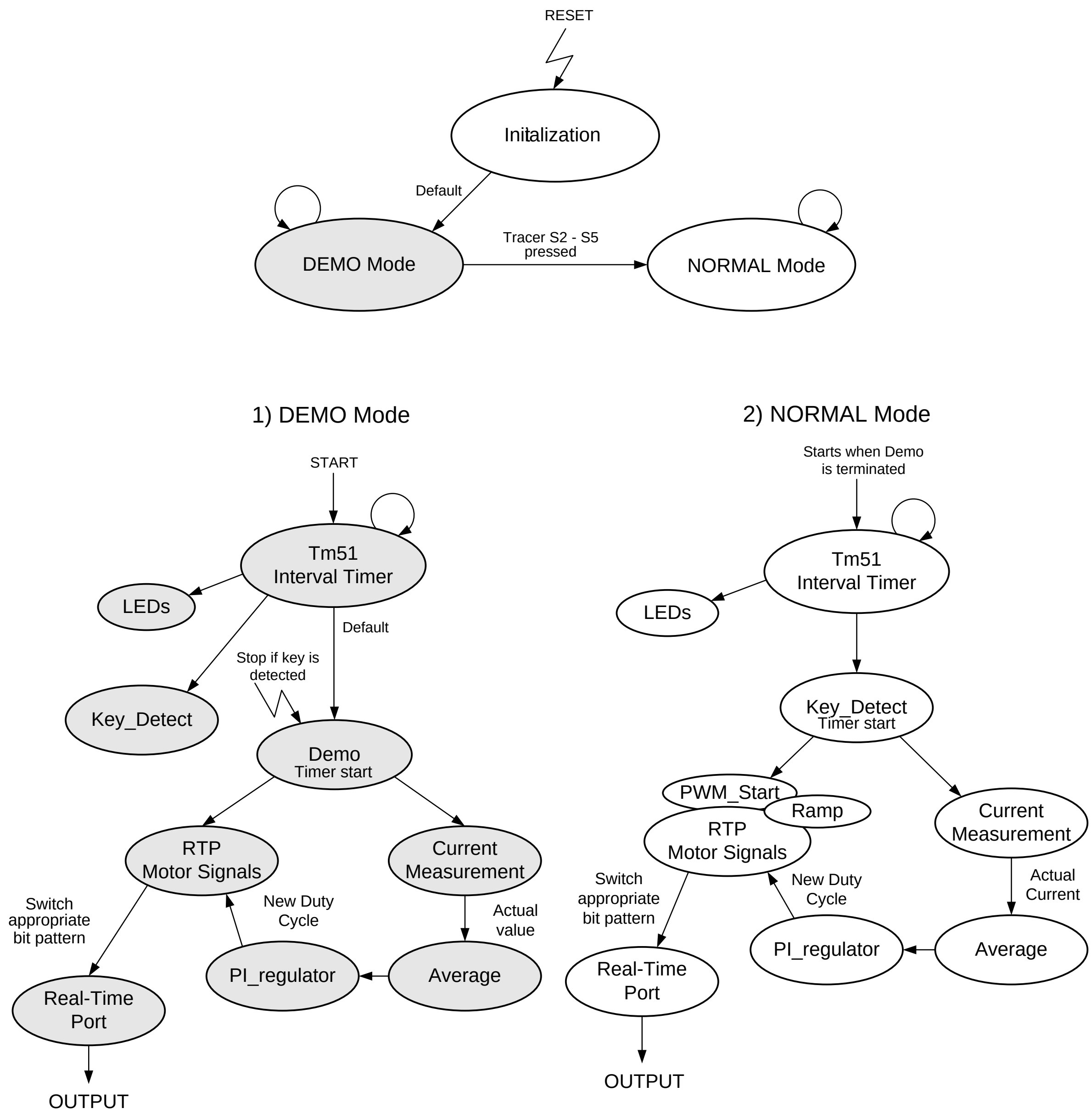
Figure 4-7: Motor Driver and User Interface for Stepper Motor Control



Chapter 5 Software Process Description

This section describes the software implementation for this application example. Figure 5-1 shows the principal data flow diagram and the relationship between the software modules and hardware peripherals that are involved in the control of the motor.

Figure 5-1: Principal Data Flow Diagram



The software can be separated into two main phases, the first one occurs right after initialization, the application runs into demo mode, where a certain demo program is executed. The second phase is run into as soon as an external switch is pressed, this terminates the demo program and leads to the normal working mode of the motor. From here on the motor can be driven in different pattern that are described in the following chapter. The functions of the system shown in Figure 5-1 are sequential and implemented and executed in the main endless loop of the software.

5.1 Demo Mode

In the demo mode the stepper motor can present the different working steps the motor is able to operate in. The mode is entered directly after switch on. The motor follows the user defined program as long as no tracer is pressed. In this application design the motor rotates 90 degrees with a rotation speed of 1 RPM in detectable single steps and 270 degrees with a speed of 168 RPM in a continues way. Once a tracer is pressed the demo mode is terminated and the motor stops.

5.2 Normal Mode

This is the mode where the motor is usual been driven. Four switches, S2 through S5, control how the Stepper Motor Controller board operates. S2 controls direction. Each time S2 is pressed the motor changes its direction of rotation. S3 controls how the motor is stepped. Each time S3 is pressed for less than 1 second, the motor toggles between continuous mode and single step mode, this can only be performed when the motor is standing still. Holding S3 down for more than 1 second toggles the stepping sequence between full-stepping and half-stepping. Pressing S2 and S3 starts the motor in the continuous mode, stopping the motor in this mode can be achieved by pressing switches S4 and S5. The green LED3 is illuminated while operating in half-stepping mode. Otherwise, LED3 is off. Yellow LED2 is illuminated while in single step mode, flashing while the motor rotates left, permanent if spinning right. While operating in continuous mode, pressing S3 increases the motor's stepping rate, S4 decreases it. For single step mode, the motor steps as long as S3 is pressed. S4 advances the motor one step each time it is pressed.
The red LED1 shows that the motor is running and shuts off as soon as the motor stands.

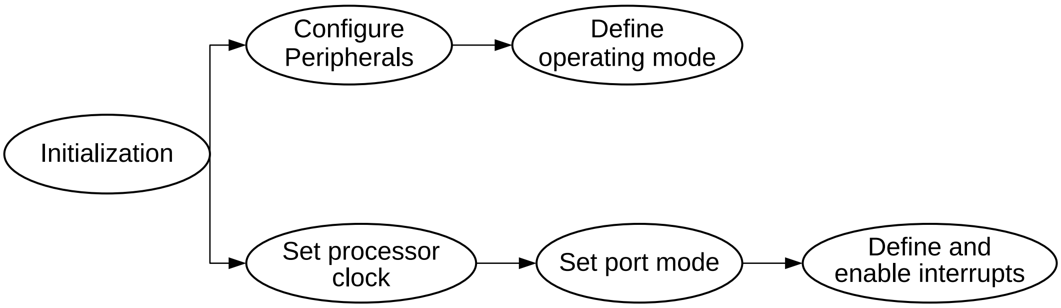
Table 5-1: Switch Operation

Switch	Operation
S2	Toggles Motor Direction
S2 & S3	Starts motor in the continuous mode
S4 & S5	Stops motor in the continuous mode
S3	Toggles between continuous and step mode (less than 1 second pressed)
S3	Toggles between full-stepping and half-stepping mode (more than 1 second pressed)
S4	Increases motor’s speed (continuous mode) / steps motor as long as pressed (step mode)
S5	Decreases motor’s speed (continuous mode) / single steps motor (step mode)

5.3 Initialization

The initialization process is responsible for the initializing the µPD78F0714 device after a system reset. It configures the basic clock settings of the device, initializes the peripherals that are used for the motor control application and disables/ enables interrupts. The initialization contains two parts as shown in Figure 5-2, the first part that initializes the configuration of the device and the second part initialize the peripherals with their operating mode.

Figure 5-2: Initialization Process



5.4 TM51 Interval Timer

The timer TM51 is used to realize an interval timer function. It is used to generate an interrupt request at the preset time interval. The interval time length is set to the period of T = 10.2 ms. Port six, the input port for the switches, is masked in the ISR and checked every 10.2 ms. If a signal stands low for 20.4 ms, means that the ISR has occurred two times, the actual switch is considered pressed and returned to the main loop. Different variables are generated. The timer is also responsible to keep a counting variable for time variant applications, for instance the flashing of a led.

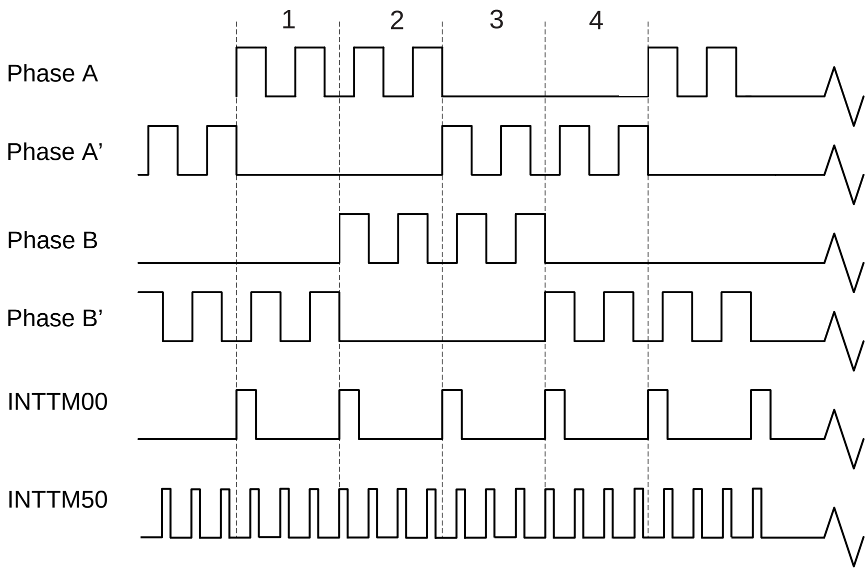
5.5 Key_Detect

This function detects the key inputs and serves different system running modes depending on the key input. The Key_Detect function is event controlled and it is executed only when a key entry is recognized. The sample time of the key entry is defined with the elapse time of the TM51 ISR.

5.6 RTP Motor Signals

This process is responsible to generate the motor signals. It consists of three main functions. First function defines the real time output port, including port settings and enable pulse width modulation. The second function contributes the bit patterns to drive the stepper motor. The timer TM00 is realised as an interval timer function to generate an interrupt at a present time. Every time the ISR is executed a different bit pattern is load to the real-time output buffer register 0, responsible to drive the mosfets in the right order.
This function has to guarantee the exact bit pattern for every situation, including direction changes, half-full step and start- stop mode. The position of the stepper is continuous tracked by a variable counting how often the ISR is entered.
To generate the PWM for the real time output port the third function is used. TM50 works as an interval timer. The timer works with a steady 5 MHz frequency, the duty cycle can differ from 0 to 100%. Value of compare register CR50 modulates the duty cycle.
Figure 5-3 shows connection between TM00, TM50 and the generated output motor signals.

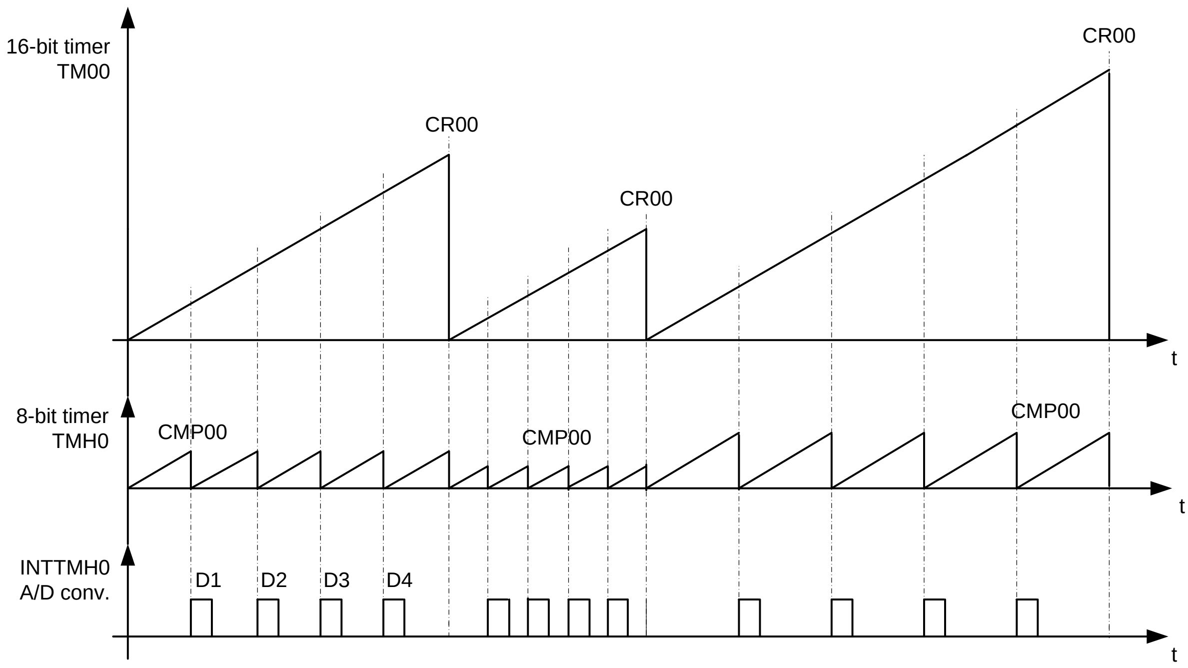
Figure 5-3: Connection between TM00, TM50 and Motor Signals



5.7 Current Measurement

This process is responsible for measuring the voltage over the shunt resistors, which gives actual information about the current flow through the windings. The measured value is used as a feedback for the closed current loop control.
The timer TMH0 is used to realize an interval timer function to generate an interrupt request at the preset time interval. The interval time length is set to a period four times faster as TM00 frequency, to ensure synchronously working both timers run with the same frequency.
The ADC function is executed every time TMH0 interrupt service routine is active.
The interrupt request flag of the ADC function is polled and an AD conversion is executed each time the interrupt request flag is detected high.
This procedure guarantees that an AD conversion only occurs at the defined time.
Figure 5-4 shows the basic working of TM00, TMH0 and the AD conversion.

Figure 5-4: Connection between TM00, TMH0 and AD conversion



5.8 Average

This function sequentially builds an average value over the actual value delivered by the current measuring process. Purpose of this function is to filter out voltage ripples.
The Average function is event controlled and it is executed only when an AD conversion is recognized. It also sets an overflow flag if the set point for regulation is exceeded.

5.9 PWM_Start

This small function limits the switch on current of the motor. Starting with a PWM duty cycle of 20% and increasing it to 100% or the desired regulation set point.

5.10 Ramp

Stepper motors can not start, stop or change direction above there maximum start- stop frequency without loosing steps. This frequency differs from motor to motor. To ensure the accurate steps, Ramp function implements a speed ramp.
This function is also responsible for increasing and decreasing the motor speed if external tracer S4 or S5 is pressed.

5.11 Stall_Detect

Stall_Detect recognizes loosing steps if the motor is running too fast. As result of this detection, the function slows the motor down with the goal to avoid step losses of the motor.

5.12 PI-Regulator

The PI-Regulator used is the classical Proportional Integral (PI) control method in the closed loop current control of the stepper motor.
The regulator is based on the recursive PI algorithm known also as the speed algorithm and takes the form of:

$$G(s) = K_p + K_i * \frac{1}{S}$$

transformed into a discrete form:

$$K_p * X_d + K_i * \left(\sum X_d\right)$$

$$X_d = X(n) - X(n-1)$$

where:

K_p	presents the proportional gain
K_i	present the integral gain
X_d	presents the voltage error
$\sum X_d$	presents the accumulated voltage error

The coefficients K_p and K_i were derived empirically and optimized based on system behaviour produced by disturbances during the system testing.
The sample time of the regulator depends on the rotation speed of the motor. The function also normalise the value and transforms the calculated regulated quantity into the duty cycle of the PWM signal.

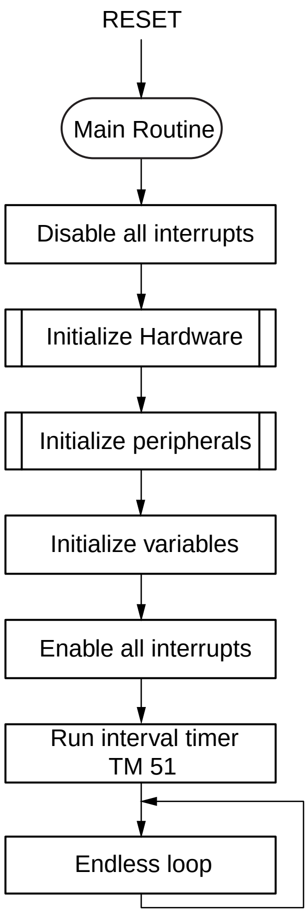
Chapter 6 Software Flowcharts

This chapter describes the important functions used in the system of the stepper motor control application. The functions that are responsible for the key input and the menu points are not included in this chapter. Please refer in the software source codes if more information about these functions is needed.

6.1 Concept and Main Flowchart

Figure 6-1 shows the main program flowchart.

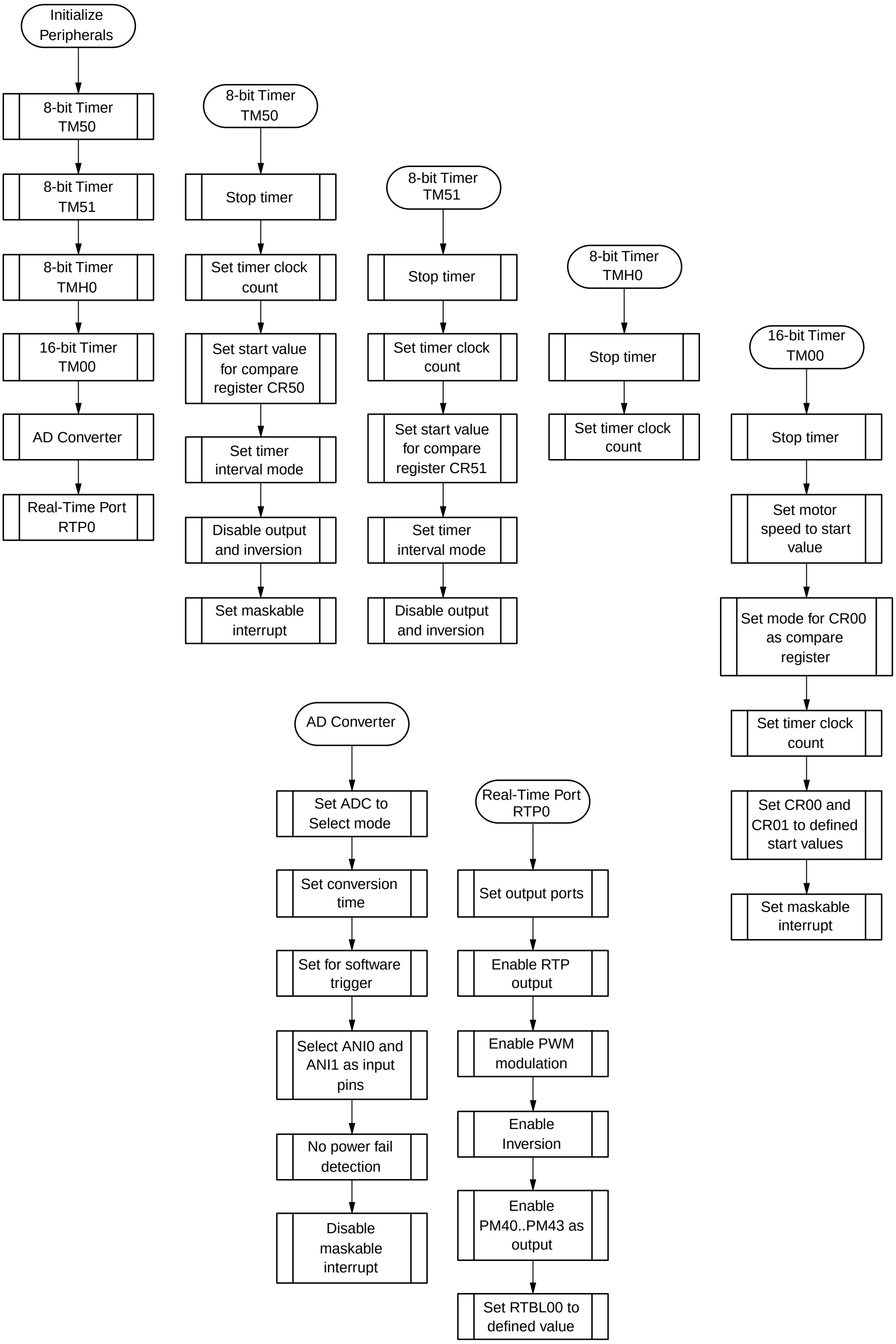
Figure 6-1: Main Program Flowchart



6.2 Peripherals Initialization

Figure 6-2 shows the initialization of the used hardware peripherals of the µPD78F0714 device with their operation mode in this application.

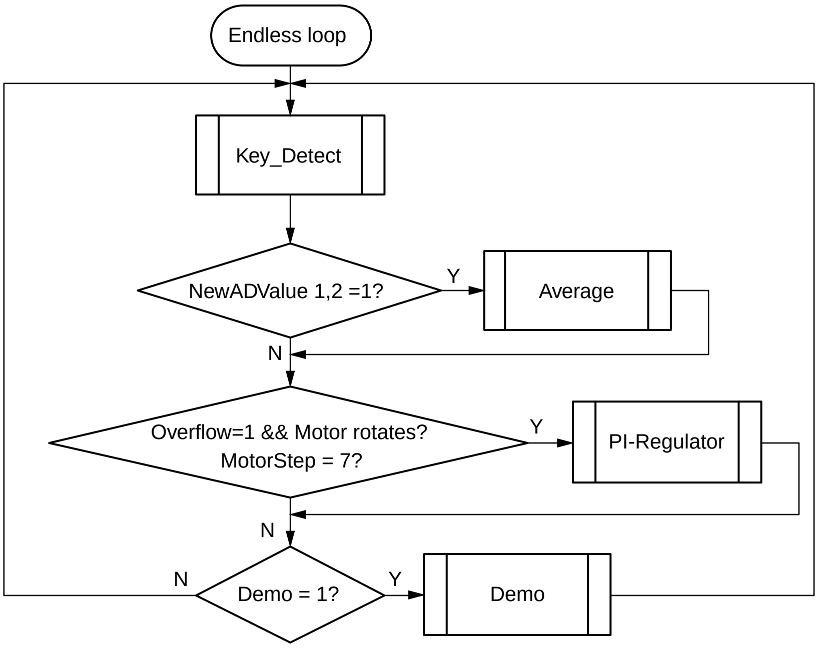
Figure 6-2: Peripherals Initialization



6.3 Main Concept

Figure 6-3 shows the endless loop of the main program used in this application.

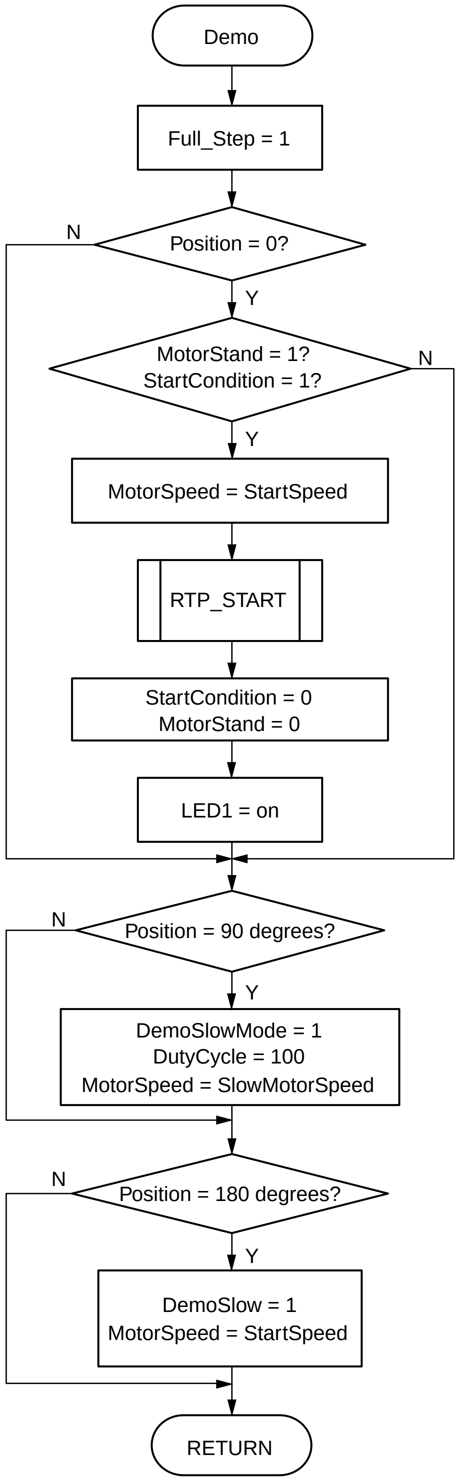
Figure 6-3: Endless Loop Function Flow



6.4 Demo Concept

Figure 6-4 shows the flow chart of the demo program used in this application.

Figure 6-4: Demo Function Flow



6.5 RTP Motor Signals Concept

Figures 6-5 to 6-7 show the basic concept flow for generating the motor signals. Especially TM00 and TM50 interrupt service routines are responsible for generating the motor signal pattern. The function TMH0_START is starting the AD converter and will be explained in the next chapter.

Figure 6-5: RTP_START Flowchart

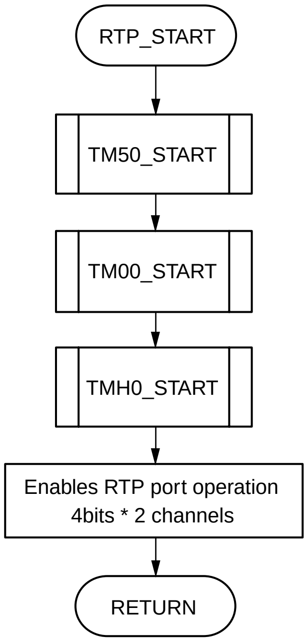


Figure 6-6: TM50_ISR Flowchart

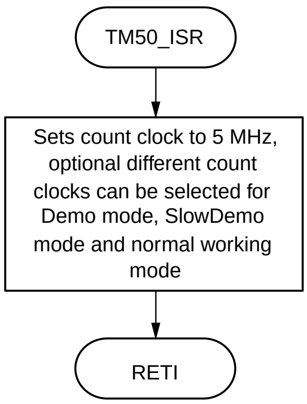
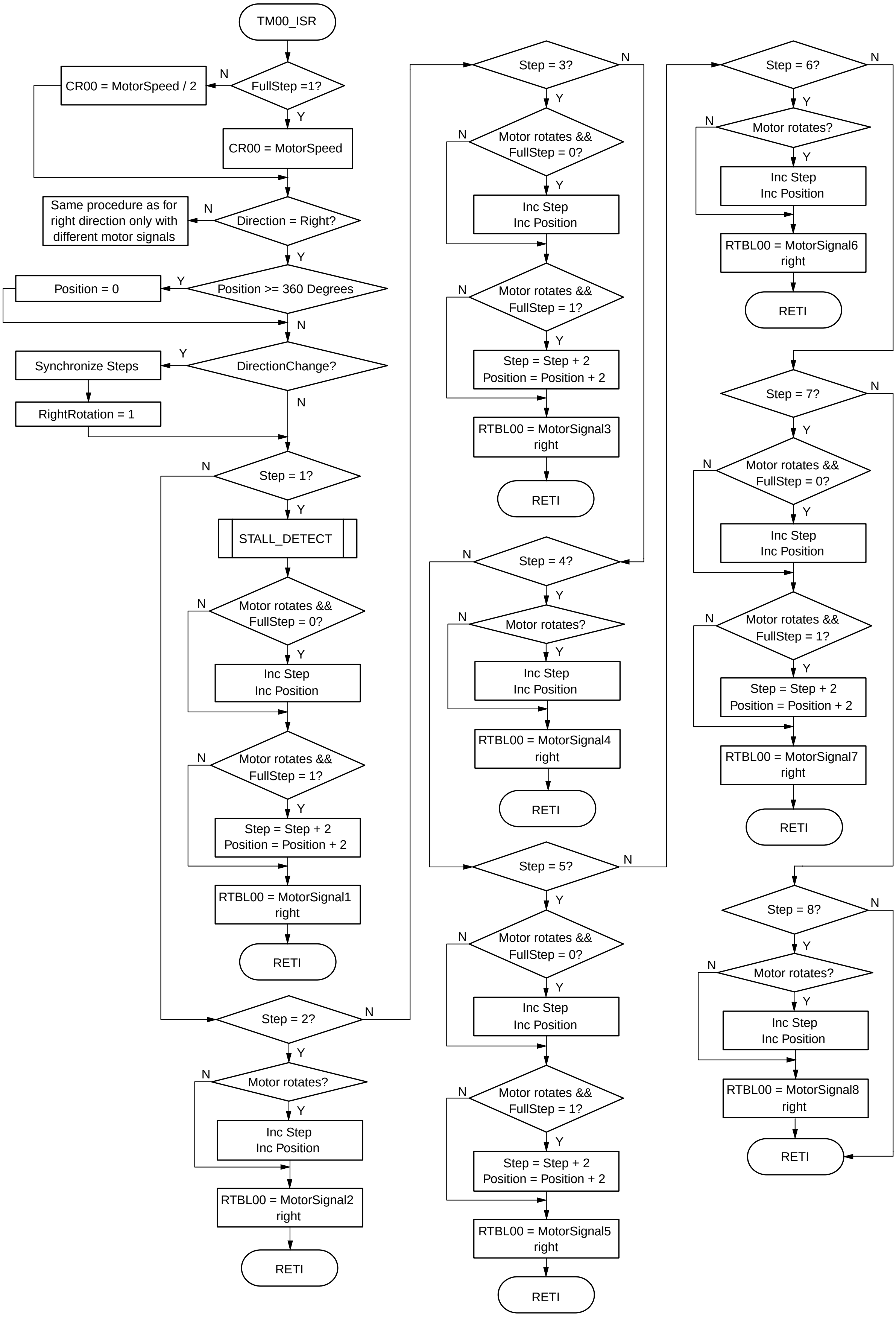


Figure 6-7: TM00_ISR Flowchart



6.6 Current Measurement

As mentioned TMH0 starts the AD conversion every time the timer ISR is entered. Figure 6-8 and 6-9 show the connection between the interrupt service routine and the AD polling.

Figure 6-8: TMH0_ISR Flowchart

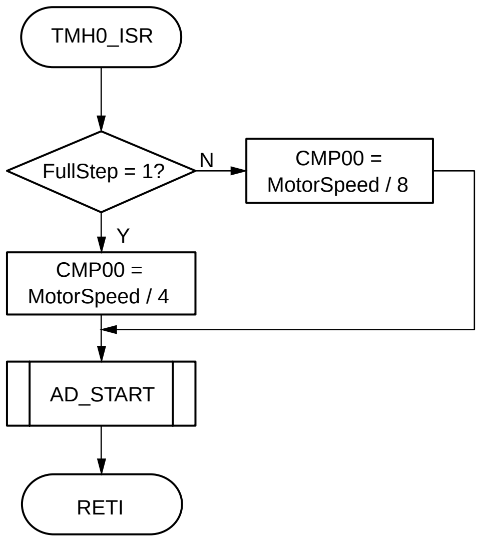
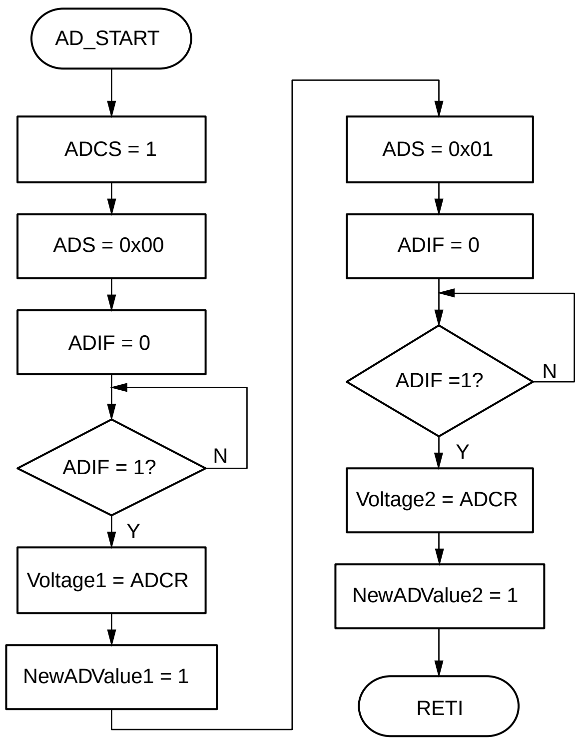


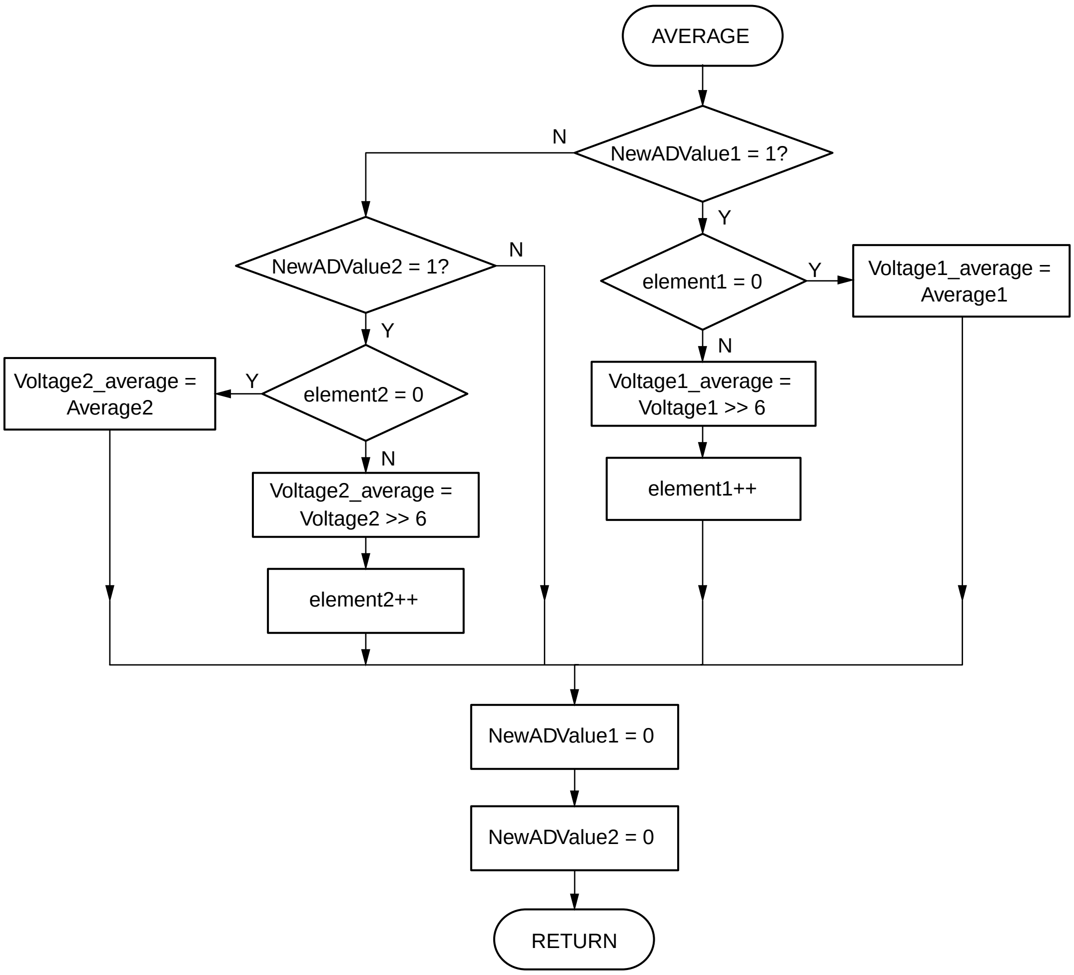
Figure 6-9: AD_START Function Flowchart



6.7 Average

Figure 6-10 shows flow of the Average function.

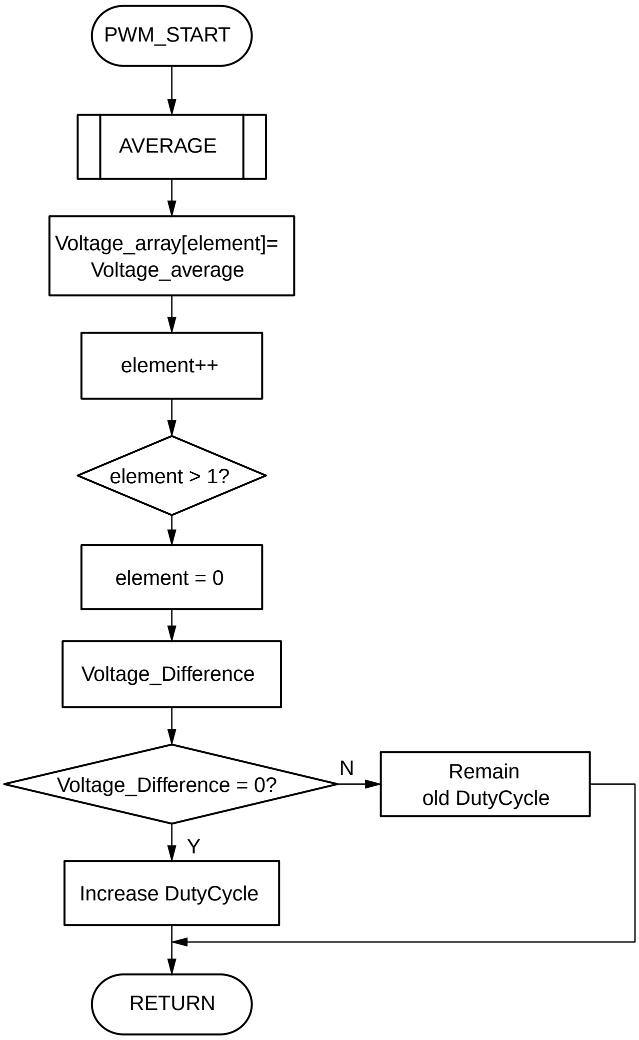
Figure 6-10: Average Function Flowchart



6.8 PWM Start

Figure 6-11 shows the concept flow of PWM Start.

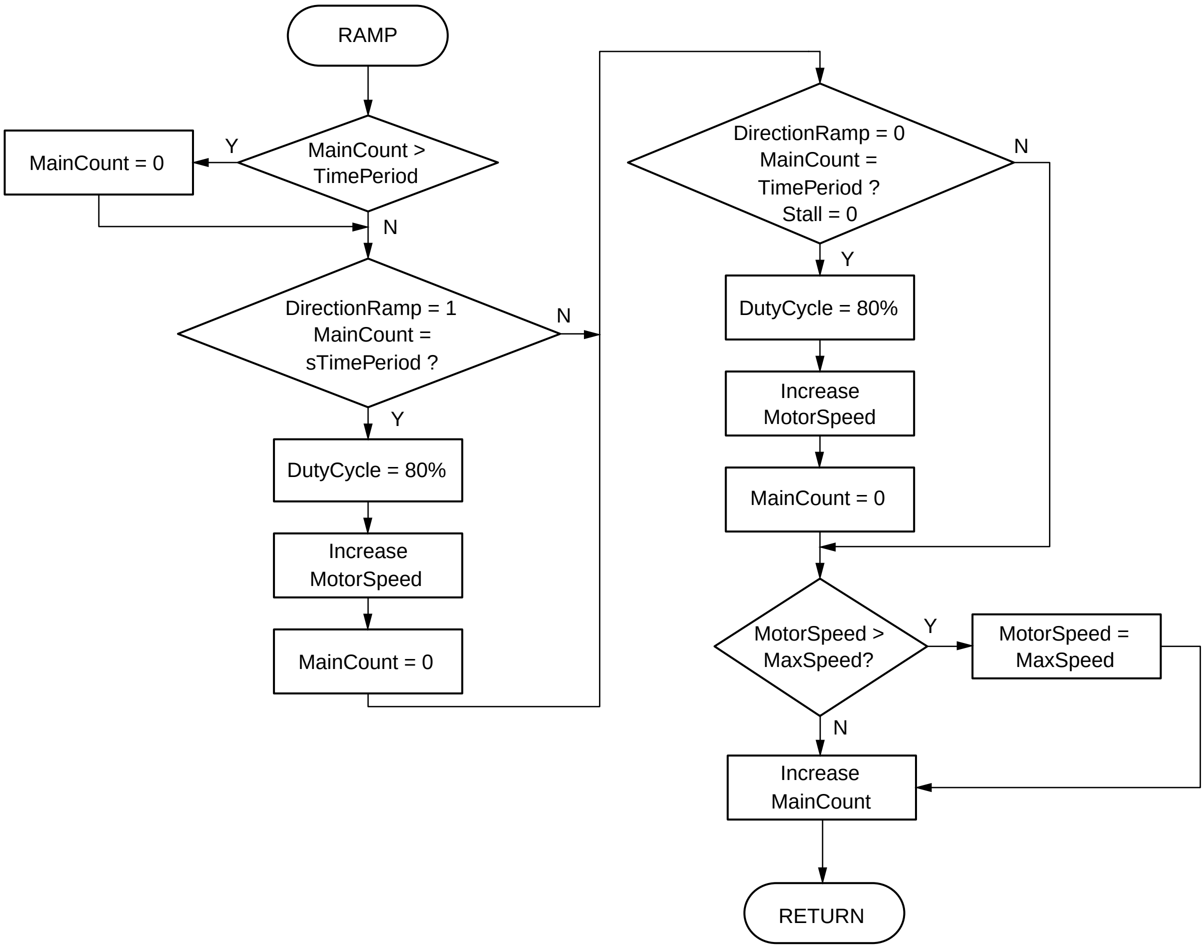
Figure 6-11: PWM_START Flowchart



6.9 Ramp

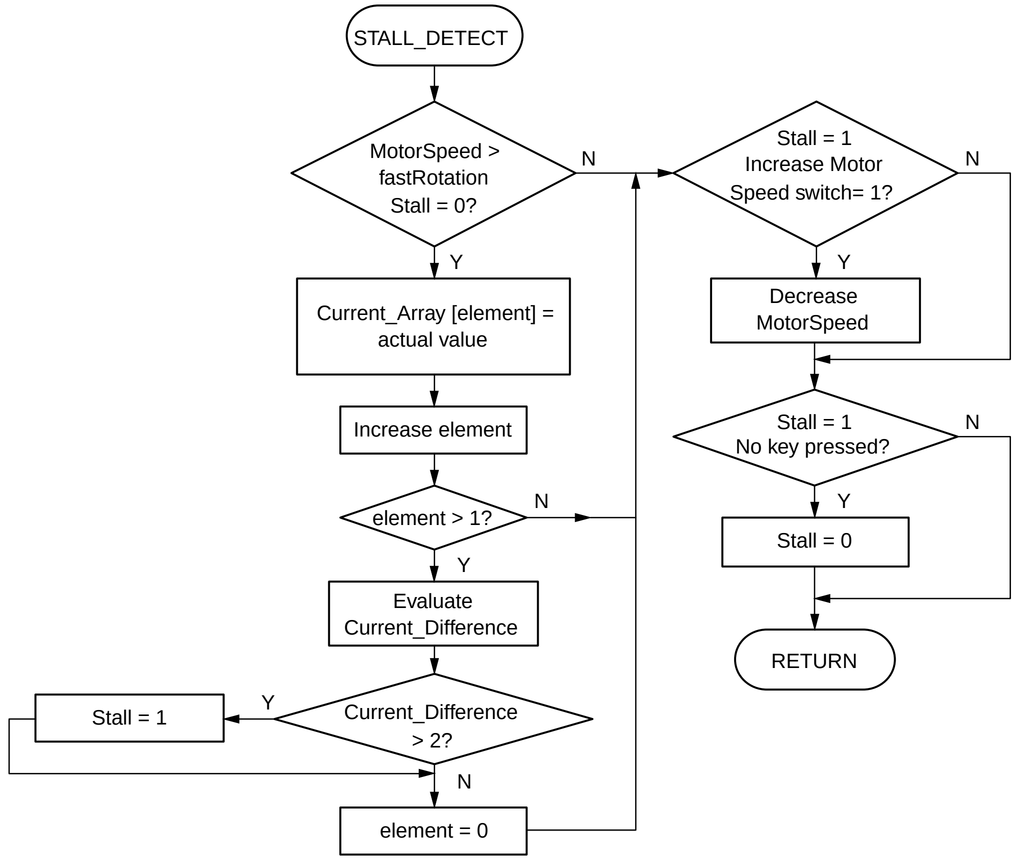
The Ramp function consists of two sub functions, RAMP_UP and RAMP_DOWN.
Figure 6-12 shows the process flow for the RAMP_UP function, RAMP_DOWN is not further described, because it is very similar to the RAMP_UP function.

Figure 6-12: RAMP_UP Flowchart



6.10 Stall Detect

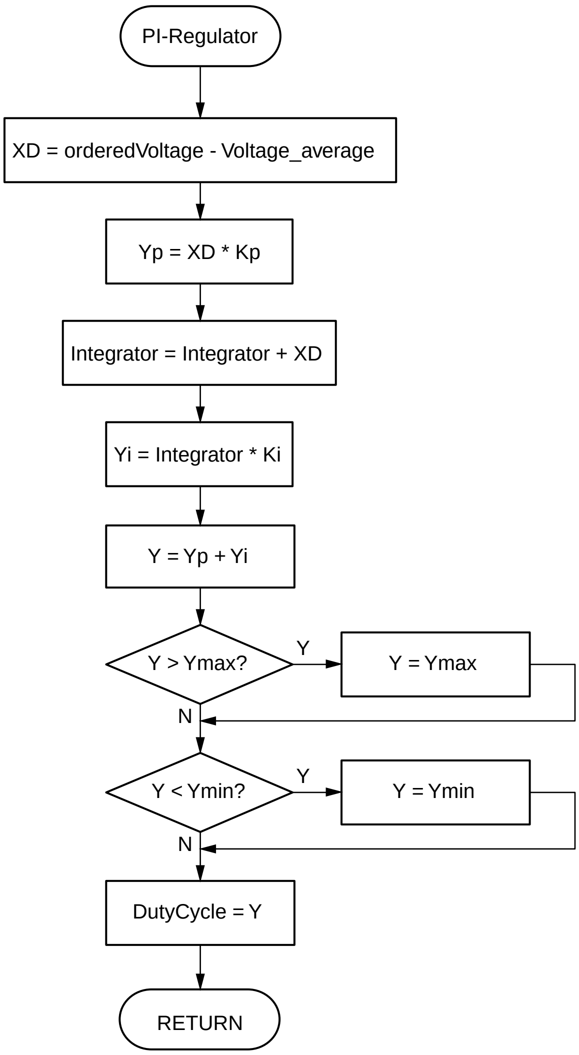
Figure 6-13: STALL_DETECT Flowchart



6.11 PI-Regulator

Figure 6-14 shows the function flow of the implemented PI-Regulator in the system.

Figure 6-14: PI-Regulator Flowchart



Chapter 7 Program Listing

```
/*===================================================================================
** PROJECT          = Stepper motor control
** MODULE           = Hardware Initialisation
** VERSION          = V0.1
** DATE             = 07.07.2005
** LAST CHANGE      =
**
** =================================================================================
** Description: Hardware initialization
**
** =================================================================================
** Environment:     Device:         uPD78F0714
**                  Assembler:      A78000          Version X.XXX
**                  C-Compiler:     ICC78000        Version X.XXX
**                  Linker:         XLINK           Version X.XXX
** =================================================================================
** By:              NEC Electronics (Europe) GmbH
**                  Oberrather Strasse 4
**                  D-40472 Duesseldorf
**
**
** ===============================================================================*/
// INCLUDE the HEADER FOR THE 78F0714 Device!!!!!!

#include <io78f0714.h>
#include <intrinsics.h>
#include <migration.h>

/********************************************/
/****        H/W UPD INIT               ****/
/********************************************/
void vHardwareInit(void)
{

// port latch

P0   = 0x00;         // set output latch to 0
P1   = 0x00;         // set output latch to 0
P2   = 0x00;         // set output latch to 0
P3   = 0x00;         // set output latch to 0
P4   = 0x00;         // set output latch to 0
P5   = 0x00;         // set output latch to 0
P6   = 0x00;         // set output latch to 0
P7   = 0x00;         // set output latch to 0

// port mode
PM0  = 0x00;         // port 0.1, 0.2, 0.3 are input for Key 1,2 and 3
PM1  = 0x00;         // port 1 output
PM2  = 0x00;         // port 2 = Input only
PM3  = 0x00;         // port 3 = output
PM4  = 0x00;         // port 4 = output
PM5  = 0x00;         // port 5 = output
PM6  = 0xFF;         // port 6 = input
PM7  = 0xF0;         // port 7 = output
```

```
// pull up resistors
/*
PU0 = 0x00;          // no pull up-resistors
PU2 = 0x00;          // no pull up-resistors
PU3 = 0x00;          // no pull up-resistors
PU4 = 0x00;          // no pull up-resistors
PU5 = 0x00;          // no pull up-resistors
PU6 = 0x00;          // no pull up-resistors
*/
// interrupt definition
IF0L = 0x00;         // clear INT request
IF0H = 0x00;         // clear INT request
IF1L = 0x00;         // clear INT request

// 7 6 5 4 3 2 1 0 Bit Number
MK0L = 0xFF;            // 1 1 1 1 1 1 1 1
// |_|_|_|_|_|_|_|_INTLVI disabled
// |_|_|_|_|_|_|___INTP0  disabled
// |_|_|_|_|_|_____INTP1  enabled
// |_|_|_|_|_______INTP2  enabled
// |_|_|_|_________INTP3  enabled
// |_|_____________INTP4  disabled
// |_|_____________INTP5  disabled
// |_______________INTP6  disabled

// 7 6 5 4 3 2 1 0 Bit Number
MK0H = 0xFF;//0xE1;     // 1 1 1 1 1 1 1 1
// |_|_|_|_|_|_|_|_INTP7     disabled
// |_|_|_|_|_|_|___INTTW0UD  enabled
// |_|_|_|_|_|_____INTTW0CM3 disabled
// |_|_|_|_|_______INTTW0CM4 disabled
// |_|_|_|_________INTTW0CM5 disabled
// |_|_____________INTCM10   disabled
// |_|_____________INTCM11   disabled
// |_______________INTCC10   disabled

// 7 6 5 4 3 2 1 0 Bit Number
MK1L = 0xFF;            // 1 1 1 1 1 1 1 1
// |_|_|_|_|_|_|_|_INTCC11   disabled
// |_|_|_|_|_|_|___INTBEMF0  enabled
// |_|_|_|_|_|_____INTTM0    disabled
// |_|_|_|_|_______INTTM01   disabled
// |_|_|_|_________INTSRE00  disabled
// |_|_____________INTSR00   disabled

// |_|_____________INTST00   disabled
// |_______________INTTM50   disabled

// 7 6 5 4 3 2 1 0 Bit Number
MK1H = 0xFF;                // 1 1 1 1 1 1 1 1
// |_|_|_|_|_|_|_|_INTTM51   disabled DF
// |_|_|_|_|_|_|___INTTMH0   disabled
// |_|_|_|_|_|_____INTCSI10  disabled
// |_|_|_|_|_______INTDMU    disabled
// |_|_|_|_________INTAD     disabled
// |_|_____________Not USED  disabled Read Only
// |_|_____________Not USED  disabled After Reset Value is 0!
// |_______________Not USED  disabled Read Only
```

```
PR0L = 0xFF;         // INT low priority
PR0H = 0xFF;         // INT low priority
PR1L = 0xFF;         // INT low priority
PR1H = 0xFF;         // INT low priority


// 7 6 5 4 3 2 1 0 Bit Number
EGP = 0x00;          // 0 0 0 0 0 0 0 0
EGN = 0x00;          // 0 0 0 0 0 0 0 0
//         |_|_|_|_disabled ext. INT. TW0TOFFP Security Shut Off TMW Outputs
//         |_|_|___INTP1 enabled Key 1 Input (rising Edge)
//         |_|_____INTP2 enabled Key 2 Input (rising Edge)
//         |_______INTP3 enabled Key 3 Input (rising Edge)

//CLOCK Settings

OSTS = 0x05;         // Set Stabilization Time to 3.27 ms

while(!OSTC_bit.no0) // Get acknowledgment for the stab. time
{
_NOP();
}

// processor clock
PCC  = 0x00;         // with max Freq.

MOC  = 0x00;         // X1 Oscillator operating
MCM  = 0x01;         // X1 Input Clock

                     //Check if the X1 operates
while(!MCM_bit.no1)
{
MOC  = 0x00;         // X1 Oscillator operating
MCM  = 0x01;         // X1 Input Clock
}

RCM  = 0x01;         // Ring-OSC Stopped

//VSWC = 0x02;       // Insert Two wait state

WDTM = 0x77;         // STOP Watchdog TIMER!
}
```

```
/*===================================================================================
** PROJECT          = Stepper Motor
** MODULE           = Global Variables
** VERSION          = V0.1
** DATE             = 07.07.2005
** LAST CHANGE      =
**
** ================================================================================*/


//==================================================================================
// Global variables
//==================================================================================

/* User defined variables */
unsigned int StartSpeed = 139;    // sets the starting speed of the motor
unsigned char DutyStart = 80;     // sets starting frequency for the PWM
unsigned int orderedVoltage = 18; // sets the voltage value for the PIRegulator

/* ADC variables */
unsigned int Voltage1;
unsigned int Voltage2;
unsigned char NewADValue1;
unsigned char NewADValue2;

/* Average variables */
unsigned int Voltage1_average = 0;
unsigned int Voltage2_average = 0;
unsigned char element1 = 0;
unsigned char element2 = 0;
unsigned char Overflow = 0;       // indicates Overflow and starts regulator
unsigned char Voltage_array[2];

/* Demo variables */
unsigned char Demo = 1;
unsigned char DemoSlowMode = 0;

/* KeyDetect variables */
unsigned int Speed_old;
unsigned char Cycle_old;
unsigned char StartCondition = 1;
unsigned char MotorStand = 1;
unsigned char Direction_Ramp;
unsigned char SingleStep = 0;
unsigned char FullStep = 1;
unsigned char Continues = 1;

/* PI variables */
int XD=0;                         // Delta X
long Y=0;
long Yp=0;                        // Y proportional part
long Yi=0;                        // Y integral part
long Integrator=0;
unsigned char Y_max = 235;        // Y_max value set to 235, value is chosen
unsigned char Y_min = 0;          // to keep AD Converter time in range

/* Ramp variables */
unsigned int MainCount = 0;
```

```
/* RTP variables */
unsigned char Step = 1;
unsigned char Direction = 0;
unsigned int MotorSpeed;
unsigned char DutyCycle = 0;
unsigned char RightRotation = 1;
unsigned int Position = 0;

/* Tm51 Key variables */
unsigned int i = 0;
unsigned char Key;
unsigned char Tracer;
unsigned char pressed;
unsigned char LED_Counter = 0;

/* Stall variables */
unsigned char Stall = 0;
signed char Current_Difference;
unsigned char Current_Array[2] = {0, 0};
unsigned char elementCurrent = 0;
signed char Current_Difference;


/* PWMStart variables */
unsigned char element;
unsigned char Voltage_Difference;
```

```
/*===================================================================================
** PROJECT          = Stepper Motor
** MODULE           = A/D Converter Initialisation
** VERSION          = V0.1
** DATE             = 07.07.2005
** LAST CHANGE      =
**
** =================================================================================
** Description: Function for AD conversion by flag polling
**
** =================================================================================
** Environment:     Device:         uPD78F0714
**                  Assembler:      A78000          Version X.XXX
**                  C-Compiler:     ICC78000        Version X.XXX
**                  Linker:         XLINK           Version X.XXX
** =================================================================================
** By:              NEC Electronics (Europe) GmbH
**                  Oberrather Strasse 4
**                  D-40472 Duesseldorf
**
**
** ===============================================================================*/
// INCLUDE the HEADER FOR THE uPD78F0714 Device!!!!!!

#include <io78f0714.h>
#include <intrinsics.h>
#include <migration.h>
#include “variables.h”

void AD_INIT(void);
void AD_START(void);
void AD_STOP(void);


/*===================================================================================
** Initfunction of ADConverter
**
**================================================================================*/


void AD_INIT(void)
{
ADM = 0x1A;          // sets Ad Converter to select mode and 4.8 us conversion time
ADS = 0x11;          // sets for software trigger, time trigger and ANI0 and ANI1
                     // input channel
PFM = 0x00;          // sets the whole PFM register
MK1H_bit.no4 = 1;    // disable ADC ISR maskable interrupt
}
```

```
/*================================================================================
** Startfunction of ADConverter
**
**==============================================================================*/

void AD_START(void)
{
ADCS = 1;               // starts AD Conversion
ADS = 0x00;             // ANI0 Inputchannel, Selectmode
ADIF = 0;
while(!ADIF)
             ;
Voltage1 = ADCR;        // Voltage1 gets first AD value
NewADValue1 = 1;        // NewADValue1 Flag is set
ADS = 0x01;             // ANI1 Inputchannel, Selectmode
ADIF = 0;
while (!ADIF)
             ;
Voltage2 = ADCR;        // Voltage2 gets second AD value
NewADValue2 = 1;        // NewADValue2 Flag is set
}

/*================================================================================
** Stopfunction of ADConverter
**
**==============================================================================*/


void AD_STOP(void)
{
ADCS = 0;               // stops the AD Converter
Voltage1 = 0;           // variables are set to defined values
Voltage2 = 0;
element1 = 0;
element2 = 0;
}
```

```
/*==================================================================================
** PROJECT           = Stepper Motor
** MODULE            = AD average value evaluation, Stall detection
** VERSION           = V0.1
** DATE              = 07.06.2005
** LAST CHANGE       =
**
** ================================================================================*/

#include <io78f0714.h>
#include <intrinsics.h>
#include <migration.h>
#include “variables.h”


void AVERAGE(void);
void STALL_DETECT(void);


/*==================================================================================
** Function to build average value of all incoming current values and to detect overflow
**
**=================================================================================*/


void AVERAGE(void)
{
if (NewADValue1)     // for each winding a separate current is evaluated
{
   if(element1 == 0)
   {
          Voltage1_average =(Voltage1 >> 6);
          element1++;
   }
   else
          Voltage1_average = (Voltage1_average+(Voltage1 >> 6))/2;
}
if (NewADValue2)
{
   if(element2 == 0)
   {
          Voltage2_average =(Voltage2 >> 6);
          element2++;
   }
   else
          Voltage2_average = (Voltage2_average+(Voltage2 >> 6))/2;
}
if (Voltage2_average > orderedVoltage)
   Overflow = 1;     // Overflow Flag is set when setpoint is reached
if ((Voltage2_average < orderedVoltage - 3) && DutyCycle == 0)
   Overflow = 0;     // resets the Overflow Flag
NewADValue1 = 0;
NewADValue2 = 0;
}
```

```
/*==================================================================================
** Function to detect if the motor runs into critical area
**
**================================================================================*/

void STALL_DETECT(void)
{
if ((MotorSpeed < 44 &&!Stall))
                // sets the speed from where on the stall detection becomes active
{
   Current_Array[elementCurrent] = Voltage2_average;
   elementCurrent++;
                // Current Array is filled with average values from different steps
   if (elementCurrent > 1)
   {
         Current_Difference=(Current_Array[0] - Current_Array[1]);
         if ((Current_Difference > 2) || (Current_Difference < -2))
            Stall = 1;          // Stall flag is set when there is a difference in
                                // the two values
         elementCurrent = 0;
   }
   if (Stall && (Tracer == T3))
         MotorSpeed = MotorSpeed + 2;
                                // slows the motor down to leave critical area
}
if (Stall &&!Tracer)            // clears the flag, so the motor can react to
                                // upcoming changes
   Stall = 0;
}
```

```
/*==================================================================================
** PROJECT           = Stepper Motor
** MODULE            = Demoprogram
** VERSION           = V0.1
** DATE              = 07.07.2005
** LAST CHANGE       =
**
** ================================================================================
** Description: Little Demo program that starts if you turn on the board
**
** ================================================================================
** Environment:      Device:        uPD78F0714
**                   Assembler:     A78000         Version X.XXX
**                   C-Compiler:    ICC78000       Version X.XXX
**                   Linker:        XLINK          Version X.XXX
** ================================================================================
** By:               NEC Electronics (Europe) GmbH
**                   Oberrather Strasse 4
**                   D-40472 Duesseldorf
**
**
** ==============================================================================*/

#include <io78f0714.h>
#include <intrinsics.h>
#include <migration.h>
#include “variables.h”

#define STAND           0x00
#define LED1            P70
#define LED2            P71
#define LED3            P72
#define Degr90          100
#define Degr180         200
#define SlowMotorSpeed  19999
#define on              1
```

```
void DEMO(void);

extern void RTP_START(void);


/*==================================================================================
** Little demo-program that lets the steppermotor turn 270 degrees in fast continues
** and 90 in slow single step mode
**================================================================================*/


void DEMO(void)
{
FullStep = 1;
if (Position == 0)              // sets the start position
{
   if (StartCondition && MotorStand)
   {
         MotorSpeed = StartSpeed;// sets the motorspeed to the user defined
         startspeed
         RTP_START();           // starts the motor
         MotorStand = 0;
         StartCondition = 0;
         LED1 = on;
   }
}
if (Position == Degr90)         // motorposition = 90 degrees from startposition
{
   DemoSlowMode = 1;            // sets DemoSlowMode flag, to manipulate the PWM signal
   DutyCycle = 100;
   MotorSpeed = SlowMotorSpeed; // sets the motorspeed to a very slow rotating speed
}
if (Position == Degr180)        // when motor reaches 180 degrees it goes back to
                                // normal mode
{
   DemoSlowMode = 0;
   MotorSpeed = StartSpeed;
}
}
```

```
/*=================================================================================
** PROJECT          = Stepper Motor
** MODULE           = PIRegulator
** VERSION          = V0.1
** DATE             = 07.07.2005
** LAST CHANGE      =
**
** ================================================================================
** Description: Closed loop regulation with an PIRegulator
**
** ================================================================================
** Environment:     Device:         uPD78F0714
**                  Assembler:      A78000         Version X.XXX
**                  C-Compiler:     ICC78000       Version X.XXX
**                  Linker:         XLINK          Version X.XXX
** ================================================================================
** By:              NEC Electronics (Europe) GmbH
**                  Oberrather Strasse 4
**                  D-40472 Duesseldorf
**
**
** ==============================================================================*/

#include <io78f0714.h>
#include <intrinsics.h>
#include <migration.h>
#include “variables.h”


/*PI coefficients*/
#define KP(a) (((a) * 200)/100)   // Kp value of the PI Regulator
#define KI(a) (((a) * 600)/1000)  // Ki value of the PI Regulator


void PIRegulator(void);

/*=================================================================================
** PIRegulator
**
**===============================================================================*/

void PIRegulator(void)
{
XD =- (orderedVoltage - Voltage2_average);
                                  // calculate XD
Yp = KP(XD);                      // calculate Yp = XD * Kp;
Integrator = (Integrator + XD);   // calculate Yi
Yi = KI(Integrator);              // update integrator Y(n) = Y(n-1) + Ki*XD(n)*T
Y = ((Yp + Yi)*254)/1024;         // scale to PWM CYCLE

if (Y > Y_max)                    // limit Y
   Y = Y_max;
else
{
   if (Y < Y_min)
          Y = Y_min;
}

DutyCycle = Y;                   // DutyCycle gets new evaluated value
}
```

```
/*=================================================================================
** PROJECT          = Stepper Motor
** MODULE           = PWM Starting Ramp
** VERSION          = V0.1
** DATE             = 30.03.2005
** LAST CHANGE      =
**
** ================================================================================
** Description: Limits the switch on current
**
** ================================================================================
** Environment:     Device:         uPD78F0714
**                  Assembler:      A78000          Version X.XXX
**                  C-Compiler:     ICC78000        Version X.XXX
**                  Linker:         XLINK           Version X.XXX
** ================================================================================
** By:              NEC Electronics (Europe) GmbH
**                  Oberrather Strasse 4
**                  D-40472 Duesseldorf
**
** ==============================================================================*/

#include <io78f0714.h>
#include <intrinsics.h>
#include <migration.h>
#include “variables.h”

extern void AVERAGE(void);
void PWM_START(void);


/*=================================================================================
** PWM Starting Ramp
** function for limiting the “switch on current” of the motor
**===============================================================================*/

void PWM_START(void)
{
if (MainCount >= 10)
   MainCount = 0;
MainCount++;
AVERAGE();
Voltage_array[element]=Voltage2_average;
                                  // Array is used and the two values are compared,
                                  // to make sure that you aren't
element++;                        // in the rising time of the curve
if (element > 1)                  // if motor is in static mode, current is regulated
{
   element = 0;
   Voltage_Difference=Voltage_array[0]-Voltage_array[1];
}
if (!Voltage_Difference && MainCount == 10)
{
   DutyStart = DutyStart - 1;
   DutyCycle = DutyStart;
}
else
   DutyCycle = DutyStart;
}
```

```
/*==================================================================================
** PROJECT          = Stepper Motor
** MODULE           = SpeedRamp
** VERSION          = V0.1
** DATE             = 07.07.2005
** LAST CHANGE      =
**
** ================================================================================*/


/*==================================================================================
** Ramp functions to lower and rise motor speed for direction change or start, stop
**
**=================================================================================*/


void RAMP_DOWN(void)
{
if (MainCount>1700)
   MainCount = 0;
if (Direction_Ramp)         // fast slowdown ramp for the direction change mode
{
   if (MainCount == 15)     // MainCount value determines static behaviour to change
   {                        // the upward gradient change value; one Maincount ~
          DutyCycle = 29;
          MainCount = 0;
          MotorSpeed = (MotorSpeed + 1);
   }
}
if (!Direction_Ramp && MainCount == 1700)
{                           // mode to slow down motor by pressing the tracers on the board
          MainCount = 0;
          MotorSpeed = MotorSpeed + 1;
}
if (MotorSpeed > Speed_min) // limits speed to user determined values
   MotorSpeed = Speed_min;
else
   MainCount++;
}
```

```
void RAMP_UP(void)
{
if (MainCount>1700)
MainCount = 0;
if (Direction_Ramp)         // fast speed-up ramp for the direction change mode
{
   if (MainCount == 15)
   {
          DutyCycle = 29;
          MainCount = 0;
          MotorSpeed = (MotorSpeed - 1);
   }
}
if (!Direction_Ramp && MainCount == 1700 &&!Stall)
{                           // mode to speed up motor by pressing the tracers on the board
   MainCount = 0;
   MotorSpeed = MotorSpeed - 1;
}
if (MotorSpeed < 110)       // automatic shift into HalfStep to reach higher motor speed
{
   FullStep = 0;
   LED3 = 1;
}
if (MotorSpeed <= Speed_max)// limits motor speed
   MotorSpeed = Speed_max;
else
   MainCount++;
}
```

```
/*==============================================================================
** PROJECT          = Stepper Motor
** MODULE           = TM00, TM50, RTP Initialisation and driving application
** VERSION          = V0.1
** DATE             = 30.03.2005
** LAST CHANGE      =
**
** =============================================================================*/

/*==============================================================================
** ISR of TM50 Timer
**
**==============================================================================*/

#pragma vector = INTTM50_vect     // 8-bit Timer/event counters 50 ISR
#pragma bank = 2                  // Register Bank 2
__interrupt void Timer50(void)    // Interrupt
{
if (MotorStand || DemoSlowMode)   // sets count clock to 5 MHz, you can set different
                                  // clocks for different modes
   TCL50 = 0x03;                  // DemoSlowMode

if (!MotorStand && Demo && (!DemoSlowMode && Step == 1))
                                  // count clock = 5 MHz for TM 50
   TCL50 = 0x03;                  // Demo Mode normal speed
if (!MotorStand &&!Demo)          // count clock = 5 MHz for TM 50
   TCL50 = 0x03;                  // Normal working phase
CR50 = DutyCycle;                 // CR50 as compare register with compare value of
                                  // duty cycle
}

/*==============================================================================
** ISR of TM00 Timer
**
**==============================================================================*/

#pragma vector = INTTM00_vect     // 16-bit Timer00 ISR
#pragma bank = 2                  // Register Bank 2
__interrupt void Timer00(void)    // Interrupt
{
if (FullStep)
   CR00 = MotorSpeed;             // sets frequency for the FullStep mode
if (!FullStep)
   CR00 = MotorSpeed / 2;         // doubles the frequency for the HalfStep mode so
                                  // no step loss occurs
if (!Direction)                   // Direction flag shows rotating direction of the motor
          {
   if (Position >= Degr360)
          Position = 0;
```

```
if (!RightRotation &&!Continues)
                              // different patterns must be used if motor
                              // changes direction
{                             // in the continues mode or single step mode
   switch (Step)              // RightRotation flag show if a direction change
                              // has occurred
   {
   case 1: Step = 7;
   break;
   case 2: Step = 6;
   break;
   case 3: Step = 5;
   break;
   case 4: Step = 4;
   break;
   case 5: Step = 3;
   break;
   case 6: Step = 2;
   break;
   case 7: Step = 1;
   break;
   case 8: Step = 8;
   break;
   }
   RightRotation = 1;
}
if (!SingleStep && Step > 8)
      Step = 1;
if (!RightRotation && Continues)
{
   switch (Step)
   {
   case 1: Step = 8;
   break;
   case 2: Step = 7;
   break;
   case 3: Step = 6;
   break;
   case 4: Step = 5;
   break;
   case 5: Step = 4;
   break;
   case 6: Step = 3;
   break;
   case 7: Step = 2;
   break;
   case 8: Step = 1
   break;
   }
   RightRotation = 1;
}
```

```
switch (Step)
{
case 1:
   STALL_DETECT();
                  // Important to call this function only once, to get right
                  // voltage values

   if (!MotorStand &&!FullStep)
                  // in HalfStep mode, Step and Position are increased every
                  // runtrough by one
   {
      Step++;
      Position++;
   }
   if (!MotorStand && FullStep)
                  // in FullStep mode, Step and Position are increased every
                  // runtrough by two
   {
      Step = Step + 2;
      Position = Position + 2;
   }
   RTBL00 = 0x09;
                  // suited value is written to RTBL00 register and transferred
                  // to ReatTimePort
   break;
case 2:
   if (!MotorStand)
   {
   Step++;
   Position++;
   }
   RTBL00 = 0x0D;
   break;
case 3:
   if (!MotorStand &&!FullStep)
   {
   Step++;
   Position++;
   }
   if (!MotorStand && FullStep)
   {
   Step = Step + 2;
   Position = Position + 2;
   }
   RTBL00 = 0x0C;
   break;
case 4:
   if (!MotorStand)
   {
   Step++;
   Position++;
   }
   RTBL00 = 0x0E;
   break;
```

```
    case 5:
        if (!MotorStand &&!FullStep)
        {
        Step++;
        Position++;
        }
        if (!MotorStand && FullStep)
        {
        Step = Step + 2;
        Position = Position + 2;
        }
        RTBL00 = 0x06;
        break;
    case 6:
        if (!MotorStand)
        {
        Step++;
        Position++;
        }
        RTBL00 = 0x07;
        break;
    case 7:
        if (!MotorStand &&!FullStep)
        {
        Step++;
        Position++;
        }
        if (!MotorStand && FullStep)
        {
        Step = Step + 2;
        Position = Position + 2;
        }
        RTBL00 = 0x03;
        break;
    case 8:
        if (!MotorStand)
        {
        Step++;
        Position++;
        }
        RTBL00 = 0x0B;
        break;

    }
}
```

```
else
{
       if (Position <= 0)
          Position = 400;
       if (RightRotation &&!Continues)
       {
              switch (Step)
              {
              case 1: Step = 7;
                break;
              case 2: Step = 6;
                break;
              case 3: Step = 5;
                break;
              case 4: Step = 4;
                break;
              case 5: Step = 3;
                break;
              case 6: Step = 2;
                break;
              case 7: Step = 1;
                break;
              case 8: Step = 8;
                break;
              }
              RightRotation = 0;
       }
       if (!SingleStep && Step > 8)
          Step = 1;
       if (RightRotation && Continues)
       {
              switch (Step)
              {
              case 1: Step = 8;
                break;
              case 2: Step = 7;
                break;
              case 3: Step = 6;
                break;
              case 4: Step = 5;
                break;
              case 5: Step = 4;
                break;
              case 6: Step = 3;
                break;
              case 7: Step = 2;
                break;
              case 8: Step = 1;
                break;
              }
              RightRotation = 0;
       }
```

```
switch (Step)
{
case 1
       STALL_DETECT();              // Important to call this function only
                                    // once, to get right voltage values
       if (!MotorStand &&!FullStep) // in HalfStep mode, Step is increased,
                                    // Position decreased every runtrough by one
       {
         Step++;
         Position--;
       }
       if (!MotorStand && FullStep) // in HalfStep mode, Step is increased,
                                    // Position decreased every runtrough by two
       {
         Step = Step + 2;
         Position = Position - 2;
       }
       RTBL00 = 0x03;
       break;

case 2:
        if (!MotorStand)
       {
         Step++;
         Position--;
       }
       RTBL00 = 0x07;
       break;

case 3:
       if (!MotorStand &&!FullStep)
       {
         Step++;
         Position--;
       }
       if (!MotorStand && FullStep)
       {
         Step = Step + 2;
         Position = Position - 2;
       }
       RTBL00 = 0x06;
       break;

case 4:
       if (!MotorStand)
       {
         Step++;
         Position--;
       }
       RTBL00 = 0x0E;
       break;
```

```
        case 5:
                if (!MotorStand &&!FullStep)
                {
                   Step++;
                   Position--;
                }
                if (!MotorStand && FullStep)
                {
                   Step = Step + 2;
                   Position = Position - 2;
                }
                RTBL00 = 0x0C;
                break;

        case 6:
                if (!MotorStand)
                {
                   Step++;
                   Position--;
                }
                RTBL00 = 0x0D;
                break;

        case 7:
                if (!MotorStand &&!FullStep)
                {
                   Step++;
                   Position--;
                }
                if (!MotorStand && FullStep)
                {
                   Step = Step + 2;
                   Position = Position - 2;
                }
                RTBL00 = 0x09;
                break;

        case 8:
                if (!MotorStand)
                {
                   Step++;
                   Position--;
                }
                RTBL00 = 0x0B;
                break;
                }
        }
}
```

```
/*=================================================================================
** Definition of TM50
** Interval Timer for PWM Modulation
**===============================================================================*/

void TM50_INIT(void)
{
TCE50 = 0;           // stops TM50
TCL50 = 0x03;        // count clock = 5 MHz for TM 50
CR50  = DutyStart;   // CR50 as compare register with compare value = DutyStart 80
TMC50 = 0x40;        // inversion disabled, F/F no change,
                     // Timer output reset ->F/F set to zero & output disabled
MK1L_bit.no7 = 0;    // Enable ISR mask Interrupt
}

void TM50_START(void)
{
CR50  = DutyStart;   // CR50 as compare register with compare value = DutyStart
TCE50 = 1;           // starts TM51, if the value TM51 and CR51 match, INTTM51 is
                     // generated
}

void TM50_STOP(void)
{
TCE50 = 0;           // stops TM50
}


/
*=================================================================================
** Definition of TM00
** Interval Timer for Motor signals
**===============================================================================*/

void TM00_INIT(void)
{
TMC00 = 0x00;        // stops TM00
MotorSpeed  = StartSpeed;
CRC00  = 0x00;
PRM00  = 0x02;       // Prescaler Port register, count clock is set to 78.125 kHz
CR01 = 0x00;         // sets unused CompareRegister CR01 to defined values
CR00 = StartSpeed;   // sets user defined startspeed
MK1L_bit.no2 = 0;    // Enable ISR mask Interrupt
}

void TM00_START(void)
{
MotorSpeed = StartSpeed;
TMC00 = 0x0C;        // starts Timer, generates interrupt on match between TM00 and CR00
}

void TM00_STOP(void)
{
TMC00 = 0x00;        // stops TM00
}
```

```
/*=================================================================================
** Definition of RTPs
**
**===============================================================================*/

void RTP_INIT(void)
{
PM4       = 0xF0;    // set ports P40 - P43 as output ports
DCCTL00   = 0xE0;    // RTP output, PWM enabled, Inversion Enabled
RTPM00    = 0x0F;    // set RTPM00 - RTPM03 as real time output ports
RTBL00    = 0x0F;
}

void RTP_START(void)
{
TM50_START();
TM00_START();
TMH0_START();
RTPC00    = 0x80;    // enables operation, operation mode 4bits * 2channels
INV00     = 1;       // enables inversion to fit the PWM into the motor driving curve
}

void RTP_STOP(void)
{
RTPC00    = 0x00;    // disables operation, operation mode 4bits * 2channels
INV00     = 0;       // disables inversion
TMH0_STOP();
TM00_STOP();
TM50_STOP();
}
```

```
/*=================================================================================
** PROJECT          = Stepper Motor
** MODULE           = Main
** VERSION          = V0.1
** DATE             = 07.07.2005
** LAST CHANGE      =
**
** ==============================================================================*/


#pragma language = extended
/* ================================================================================
** include
** ==============================================================================*/

#include <io78f0714.h>
#include <intrinsics.h>
#include <migration.h>
#include “variables.h”

#pragma constseg=OPTBYTE
__root const char option = 0x00;
#pragma constseg=default

#define TRUE  1
#define FALSE 0
#define LED1  P70
#define LED2  P71
#define LED3  P72
#define STAND 0x00
#define T1    1
#define T1L   10
#define T2    2
#define T2L   20
#define T3    3
#define T3L   30
#define T4    4
#define T4L   40
#define T12   12
#define T34   34
```

```
/*==============================================================================
** MAIN
**
**==============================================================================*/

void main(void)
{

_DI();                  // Disable all interrupts

/* uPD init */
vHardwareInit();        // initialise the hardware

TM50_INIT();
TM51_INIT();
TMH0_INIT();
TM00_INIT();
AD_INIT();
RTP_INIT();

_EI();
// Enable all Interrupts
TM51_START();           // starts TM51 in charge of tracer detection
while (1)
{
if (Demo)               // starts Demo program
DEMO();
Key_Detect();
if (NewADValue1 || NewADValue2)
       AVERAGE();
if ((Overflow) && (!SingleStep) &&(!StartCondition ||!MotorStand) && (Step == 7)
&&!DemoSlowMode)
       PIRegulator(),
//LED3 ^= 1;            // Test LED

}                       //End of Endless While Loop
}
```

```
/*=================================================================================
** PROJECT           = Stepper Motor
** MODULE            = 8-bit TMH0 Initialisation
** VERSION           = V0.1
** DATE              = 07.07.2005
** LAST CHANGE       =
**
** ================================================================================
** Description: Determines AD conversion time
**
** ================================================================================
** Environment:      Device:         uPD78F0714
**                   Assembler:      A78000         Version X.XXX
**                   C-Compiler:     ICC78000       Version X.XXX
**                   Linker:         XLINK          Version X.XXX
** ================================================================================
** By:               NEC Electronics (Europe) GmbH
**                   Oberrather Strasse 4
**                   D-40472 Duesseldorf
**
** ==============================================================================*/


/*=================================================================================
** ISR of TMH0 Timer
** Interval Timer
**===============================================================================*/

#pragma vector = INTTMH0_vect     // 8-bit Timer/event counters 51 ISR
#pragma bank = 3                  // Register Bank 3
__interrupt void TimerH0(void)    // Interrupt
{
if (FullStep)
       CMP00 = (MotorSpeed / 4);  // sets the frequency for the TMH0, how often the
                                  // AD converter is active
else
       CMP00 = (MotorSpeed / 8);
AD_START();
//LED3 ^= 1;                      // Test LED
}



/*=================================================================================
** Definition of TMH0 Timer
** Interval Timer
**===============================================================================*/
void TMH0_INIT(void)
{
TMHMD0 = 0x30;                    // set CountsClock to 78.125 kHz
CMP00 = (StartSpeed / 4);         // sets CMP00 as compare register with compare
                                  // value = StartSpeed / 4
}
```

```
/*==================================================================================
** Start of TM51 Timer
** Interval Timer
**================================================================================*/
void TMH0_START(void)
{
CMP00 = 0x00;
MK1H_bit.no1 = 0;    // enables maskable Interrupt
TMHE0 = 1;           // starts TMH0, if the value CMP00 and TMH0 match, INTTMH0 is
                     // generated
}



/*==================================================================================
** Stop of TM51 Timer
** Interval Timer
**================================================================================*/
void TMH0_STOP(void)
{
TMHE0 = 0;           // stops TMH0
}
```

NEC

Facsimile Message

From:

Name

Company

Tel. FAX

Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

Thank you for your kind support.

North America
NEC Electronics America Inc.
Corporate Communications Dept.
Fax: 1-800-729-9288
1-408-588-6130

Europe
NEC Electronics (Europe) GmbH
Market Communication Dept.
Fax: +49(0)-211-6503-1344

Hong Kong, Philippines, Oceania
NEC Electronics Hong Kong Ltd.
Fax: +852-2886-9022/9044

Korea
NEC Electronics Hong Kong Ltd.
Seoul Branch
Fax: 02-528-4411

Taiwan
NEC Electronics Taiwan Ltd.
Fax: 02-2719-5951

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-6250-3583

Japan
NEC Semiconductor Technical Hotline
Fax: +81- 44-435-9608

I would like to report the following error/make the following suggestion:

Document title: ______________________

Document number: ______________________ Page number: ____________

____

____

____

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	☐	☐	☐	☐
Technical Accuracy	☐	☐	☐	☐
Organization	☐	☐	☐	☐

CS 99.1

[MEMO]