LABORATORY ASSIGNMENT NUMBER 3 FOR CMPE 118
Due by 5:00pm on Sunday, May 8, 2005
Pre-Lab Due by 5:00pm on Tuesday, May 3, 2005

Purpose:
This lab is intended to acquaint you with:
- Controlling a DC motor.
- Controlling a stepper motor.
- Finding information in data sheets.
- Pulse width modulation.
- Phenomena encountered with a DC motor.
- Limitations of techniques implemented purely in software.

Minimum Parts Required:
1 each: DC motor and stepper motor (provided at each lab station), 1N4744A 15 Volt zener diode, 50K potentiometer. See a TA for any parts you need that are not in your parts kit.

Pre-Lab:
Complete the following exercises AFTER you have read through the lab assignment and BEFORE coming in to complete the lab.

0.1) Decide which port and bit(s) you will use to control the motors (all parts of lab).

0.2) Decide what, if any, initialization is required for that port.

0.3) Describe the mode in which the A/D converter will be operating.

0.4) Determine which pins on which cables/connectors you need to hook up.

In the report:
Include a description of which of the port lines on the 9S12C32 you used to implement the control as well as a justification for why you chose to use that (those) particular line(s). Include the answer to question 0.3.

Part 1 Interfacing to a DC Motor and Potentiometer

Reading:
AD_LIBS reference documentation, the CMPE118 HC12 IO Board documentation, the CMPE118 DS3658 Module board documentation, the Metrowerks CodeWarrior Quickstart manual.

Assignment:
You are to design the software necessary for the 9S12C32 to drive the supplied DC motor. Motor drive should use Pulse Width Modulation with a minimum resolution of 1 part in 256 (i.e., 8 bits). This control is to be implemented completely in software without the use of the PWM libraries. The speed at which you run the motor should be determined by the setting of an external potentiometer, which your software should read using the A/D converter on the 9S12C32. Your program should continuously read the voltage set by the potentiometer and set the motor speed directly proportional to the voltage (full scale A/D is full speed). You should be able to exit the program by pressing any key. When you have completed this task, you will be prepared to continue with the laboratory exercises in Part 2. At this point find a coach or TA and get a sign-off sheet after demonstrating Part 1.

Set-Up:
1.1) For this, and the next, part you will be using the CMPE118 DS3658 Module board to drive the DC motor. Since the DS3658 has open collector outputs, you will need to supply the power to the motor separately. To bring the power to your proto-board for connection to the
motor, use your banana plug jumpers to bring the power from the positive adjustable outputs to your proto-board.

1.2) Adjust the positive output to approximately +7.5V using the control dial on the power supply. Now connect the supply (at the proto-board) to one lead of the DC motor using the connector from the motor assembly. Also, you will need to supply power to the CMPE118 DS3658 Module board, so connect the +7.5V supply to J4, pin 1 on the DS3658 module, and connect J4, pin 2 to the GND terminal on the power supply.

1.3) Now connect the DS3658’s clamp diode for Channels A & B to the motor supply at the proto-board, and the output for Channel A to the remaining lead of the DC motor. The motor should not be rotating at this point, right? If the motor is rotating, this is due to the fact that you haven’t hooked up anything to the input of the DS3658, so the state of the input is not guaranteed – it’s floating!

1.4) Hook up the 9S12C32 output(s) you selected in the pre-lab to the input of the channel you are using on the DS3658.

1.5) Hook the two outer leads from the 50K potentiometer to the power supply’s +5V & GND terminals. Be very careful NOT to use the power going to the motor, since that is +7.5V and will damage the 9S12C32 once you connect it. After doing this, check to see that the voltage at the wiper (the remaining terminal on the potentiometer) swings between 0 and +5V.

1.6) Using one of the jumpers that goes from a wire to a push-on connector, make the connection from the wiper to the pin labeled E0 on the HC12 IO Board (it’s on J5). We could have used a unity gain buffer to bring the signal to the board, much like you did in the last lab. Why would you not want to do this? What are the limitations that this would impose?

1.7) It's finally time to try out your code. In CW12, build your project into an executable file. Then, with the uBug12 window active, turn on (or reset) the 9S12C32 board and type “CON 1” <ENTER>. You should get the connection message. If not ensure SW1 is in the boot position. At this point you are ready to download your code. Downloading and testing your code should proceed exactly as it did in Lab 0. Before you can download your code, you need to clear the FLASH, so you must type “f bulk” and press <ENTER>. Now, everything is ready for the new code to be downloaded. You need to tell the Terminal window which file to download, so type “fload ;b” and select your file, which should be named something like RCHDEMO.ABS.S19 (the *.S19 file is compiled code in the proper format for downloading). Once your code is finished downloading, the You are now ready to execute your code. Type “go” or restart the 9S12C32 with SW1 in the “Run” position and connect using HyperTerminal to use serial IO. The proper settings are at 115.2Kbaud, no flow control, and be sure to init the SCI using TERMIO_Init().

In the report: You must include the wiring diagrams of the circuits you used, noting which bits were used and what connections were made on the HC12 IO Board and the CMPE118 DS3658 Module board, the sign-off sheet, the answers to the questions in part 1.6, as well as a listing of the software used to control the motor. All wiring diagrams and schematics should be completed using the PSPICE Capture program provided on each workstation.

Part 2 Exploring DC Motors

Reading: Lecture Notes on DC motors.

Assignment: Complete the following exercises:

Basic Waveforms 2.1) Set the 'scope up for dual trace and examine the waveforms at the input and output of the DS3658. Draw a neat and readable representation of the two wave-forms. Include at least 2 full cycles. Label the parts of the waveform to show the active (driven) portion of the waveform, the decay period as the motor field collapses, the peak of the inductive kickback, and the back-EMF generated by the motor. Do this at 2 duty cycle levels (i.e. 20% and 80%). Make a separate scope face drawing for each. Note the different back-EMF values at the different motor speeds. To best see the back-EMF, you might want to draw a composite
of two waveforms, one with the motor stalled (hold the shaft to keep it from turning) and one with the motor turning.

2.2) Disconnect the clamp diode from the circuit and replace it with the 15 volt zener in a configuration like the one shown below. **Note:** the band on the zener diode body corresponds to the horizontal bar in the zener symbol. Repeat the labeled waveform drawing from Part 2.1. Contrast the decay times for the 2 techniques; be specific about durations. To do this you will probably need to use a different time scale on the 'scope. This portion will be most clear if you examine the alternatives at a very low duty cycle (i.e. active for 1 part in 256). This way the effects of back EMF do not obscure the decay time. From this exercise you can see how a zener protects the output transistor by limiting the peak voltage to the zener voltage, and at the same time minimizes the decay time of the collapsing field. It is important to minimize this decay time in order to make the duty cycle response of the motor as linear as possible. At very high duty cycles it would be possible for a long delay time to completely overlap the un-driven portion of the waveform.

![Zener Diode Diagram](image)

2.3) Using the oscilloscope and/or the counter-timer, determine the frequency of the duty cycle waveform that you are generating. What is the frequency?

2.4) Can you find a way to increase the operating frequency? (Look at how to shorten your cycle time to the minimum.)

2.5) What happens to the upper frequency limit if we only have 7 bits of resolution? How about 10 bits? We are making a trade-off between resolution and frequency of operation.

2.6) Given a resolution, what is the key parameter in determining the total period of the PWM signal?

2.7) Can your solution achieve a full range of operation? (i.e. What happens if you try to get 0% or 100% duty cycle). If it can't, don't try and fix it, just explain why it won't work.

2.8) For a given duty cycle, what are the sources of error in the approach you have chosen? (i.e. if you ask for 50% duty cycle do you get it? ...theoretically, that is, you probably will not be able to measure the error.)

**In the report:** Several scope face drawings are requested, please make your drawings legible. Using different colors for different waveforms is helpful. Include the discussion requested in Part 2.2. Be sure also to include in the lab report your answers to the all the questions in Parts 2.3-2.8. To ease the grading task please quote the question you are answering before your answer. This way the grader can read the question you think you are answering followed by your answer.

---

**Part 3 Interfacing to a Stepper Motor and Potentiometer**

**Reading:** Stepper Motor Handbook (in the Course Reader), the CMPE118 MC3479 Stepper Motor Driver
Assignment: You are to design the necessary software to run on the 9S12C32 to drive the supplied stepper motor. The speed at which you run the motor should be determined by the setting of an external potentiometer, which your software reads using the A/D converter on the 9S12C32. Your program should continuously read the voltage set by the potentiometer and set the motor speed directly proportional to the voltage (full scale A/D is full speed). You should be able to exit the program by pressing any key. When you have completed this task, get a coach to sign off on its function. For this, and subsequent sections of the lab, you may use the supplied libraries, providing pulse generation and PWM output.

Set-Up:

3.1) For this part, you should use the CMPE118 MC3479 Stepper Motor Driver Module board.

3.2) You will need to connect power and ground to J5 of the CMPE118 MC3479 Module, which will in turn supply current to the stepper motor. Note that you must supply it with a voltage between +7.2V and +10V.

3.3) You will need to supply pulse and direction inputs to the chip. For this test, you may simply hook the Full/Half input to ground (J6).

3.4) You will need to experiment with the motor coil polarities to get the motor to rotate. If at first you don't succeed, try, try again (there are only a few possibilities).

In the report: Include the wiring diagram of the circuit you used, a copy of the code that drives the stepper motor and the sign-off sheet from the coach. Be sure to indicate on the diagram which bits of which ports of the 9S12C32 you used.

Part 4 Interfacing to a DC Motor and H-Bridge

Reading: CMPE118 SN754410 Dual 1A H-Bridge Module documentation.

Assignment: You are to duplicate the functionality of Part 1, this time using the H-Bridge to drive the motor. You will also need to add a little complexity. You should include a provision to change the direction of rotation of the motor. When you have completed this task, get a coach to sign off on its function. For this section of the lab, you should use the supplied libraries providing PWM output.

Set-Up:

4.1) You will need to add a power connector to supply power to the H-bridge on the SN754410 Module. Connect power and ground to J3 of the module.

4.2) You will need to supply enable and direction inputs to the chip.

4.3) Mount the motors that were loaned to you in Lab 2 and the SN754410 module to your platform and demonstrate it to obtain a check-off.

In the report: Include the wiring diagram of the circuit you used, a copy of the code that drives the DC motor and the sign-off sheet from the coach. Be sure to indicate on the diagram which bits of which ports of the 9S12C32 you used.

Hints on working this assignment

When you go into the lab, follow a well thought-out and systematic approach to testing both the hardware and the software. It is a good idea to get into the habit of testing your software, as much as possible, separately from testing your hardware.

If you really did your preparation properly, you should come into the lab ready to build and test the hardware in Part 1. If you have spent more than one hour on any single task, after coming in prepared as described above, something is wrong! STOP and ask a TA or your neighbor to take a look at what you are doing. Often a new look will spot simple
problems that you've missed.
Lab #3

Time Summary

Be sure to turn this in with your lab report

This information is being gathered solely to produce statistical information to help improve the lab assignments.

<table>
<thead>
<tr>
<th>Category</th>
<th>Preparing Outside of the lab</th>
<th>In the lab working this part</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Lab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Part 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Part 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Part 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Part 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Report</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The CMPE118 HC12 IO Board

Background:
The 9S12C32 Module is a single-board computer produced by Technological Arts, which integrates a Motorola 9S12C32 microcontroller, 2K of RAM scratchpad memory, 32K of EEPROM program memory, RS232 level-shifters, voltage regulation, and CPU voltage monitoring, all into a tidy little PC board.

CMPE118 uses the 9S12C32 as the single board computer platform throughout all of the lab exercises and final project. Since the 9S12C32 wasn’t designed with just CMPE118 in mind, it’s not exactly what we want, so we’ve designed and manufactured the CMPE118 HC12 IO Board to get us closer to the ideal.

The HC12 IO Board has 4 major functions:

1. Bring all connections from the 9S12C32’s main header out so that they are easy to access and clearly labeled
2. Provide voltage regulation and power supply protection circuitry
3. Increase the number of port pins available for use by providing 8 additional input pins and 8 additional output pins
4. Provide over-voltage and over-current protection circuitry on all port pins.

Using the CMPE118 HC12 IO Board:
Using the CMPE118 HC12 IO Board should be straightforward. All the connections that will typically be made to the board are located around the edge of the board, and are clearly labeled and logically grouped.
**Digital Input/Output and A/D Converter Inputs:**

Following is a description of the I/O available on the HC12 IO Board:

**Port A (J7 and J8):**

“Port A” on the IO Board refers to Port A on the older 68HC11 MCU. The Port A interface maps to Port T on the newer 9S12C32 and act as general purpose input and output ports and can be configured for a wide variety of uses. In CMPE118, we use it mostly to provide PWM and Pulse signals with our PWM and PLS software libraries. PWM outputs are limited to PT0-PT4

**Port E (J5 and J6):**

“Port E” on the IO Board refers to Port E on the older 68HC11 MCU. The Port E interface maps to Port AD on the newer 9S12C32 and acts as general purpose I/O pins and also analog inputs for the analog to digital converter.

**Port F (J9 and J10):**

Port F is a digital input-only port, and has been created on the CMPE118 HC12 IO Board to compensate for the fact that the 9S12C32 has very few such input pins available if the user wishes to use Port A for PWM or Pulse outputs.

Port F has been created by using an 8-bit parallel-to-digital shift register (a 74HC165) connected to the 9S12C32’s SPI (the Serial Peripheral Interface, the on-chip synchronous serial module). A discussion of how to access these input pins follows in the section on software, below.

**Port G (J3 and J4):**

Port G is a digital output-only port, and has been included on the CMPE118 HC12 IO Board for the same reasons that Port F was created – we wanted more I/O!

Each time the 9S12C32 is powered up or reset, Port G is automatically cleared so that all 8 pins are logic-level low’s (0’s).

Port G was created by using an 8-bit serial-to-parallel shift register, and is also connected to the 9S12C32’s SPI module. A discussion of how to control Port G’s output pins follows in the section on software, below.

All port pins available for the user on the CMPE118 HC12 IO Board are protected from over-voltage and over-current conditions by the use of series resistors (to limit current) and clamping diodes (to limit voltage). To see how this is implemented, please refer to the attached schematic.

**Power Regulation and Protection Circuitry:**

The CMPE118 HC12 IO Board must be powered by connecting J1 (labeled “Vin”) to a voltage supply where:

\[ +6VDC < \text{Vin (J1)} < +18VDC \]

This input voltage is regulated down to +5VDC for the 9S12C32 and the protection circuitry by an LM2940CT-5.0 low drop-out voltage regulator. Reverse-polarity protection is provided by a low-forward-voltage-drop diode (D1) on the positive Vin terminal. ESD protection and protection against supplying the Interface PCB with extremely high input voltage is provided by a 27V-trip transient voltage suppressor (TVS), at D3. In addition, the current flow for the entire board is limited to 500mA by a fuse.

Provisions are made for connecting other circuitry to the ground of the CMPE118 HC12 IO Board: a hook for connecting scope probes or other test devices, and a banana jack for hooking up any number of other circuit elements or proto-boards.
Software Considerations for the CMPE118 HC12 IO Board:

Just as the 9S12C32 is the single-board computer platform for all the microcontroller exercises in CMPE118, CW12 is the C cross-compiler.

Throughout CMPE118, we use the 9S12C32 and CW12 in a completely stock, off-the-shelf way – no modifications have been made to the hardware or the software setup. The main reason for leaving everything unmodified is so that students can get experience dealing with this hardware/software environment, and immediately go forth into the world and put their knowledge to use outside of CMPE118’s somewhat sheltered environs.

Since we have added Port F and Port G to increase the amount of I/O available to the user, we wrote a software module to control them and added it to CW12’s standard library. Keep in mind that these ports are unique to the CMPE118 CMPE118 HC12 IO Board – they are not available on the stock 9S12C32, and the software library is also not part of the stock CW12 compiler setup.

Reading and writing to PORTF and PORTG is a little different than reading or writing to PORTA or PORTE. Here are the details:

Public Header File for PORTF and PORTG software library:

- You will need to include the following statement at the beginning of your program to use the functions listed below:

  ```
  #include <shiftreg.h>
  ```

PORTF (input only):

- Reading PORTF will return the state of the input pins (on J9 and J10). To read from PORTF, you will need to use the following function:

  ```
  unsigned char ReadPORTF(void);
  ```

  Usage example (“data” is a variable of type unsigned char):

  ```
  data = ReadPORTF();
  ```

- There is no way to write to PORTF (it’s input only).

PORTG (output only):

- Writing to PORTG will cause the port pins (on J3 and J4) to reflect the data you have written – each bit may take on a logic-level 0 or logic-level 1. To write to PORTG, you will need to use the following function:

  ```
  void WritePORTG(unsigned char data);
  ```

  Usage example:

  ```
  WritePORTG(0xD4);  /* writes the data 0xD4 to PORTG pins */
  ```

- Even though PORTG is output only, you may still read it – reading from PORTG will return the data that was last written to PORTG. To read from PORTG, you will need to use the following function:

  ```
  unsigned char ReadPORTG(void);
  ```

  Usage example (“data” is a variable of type unsigned char):

  ```
  data = ReadPORTG();  /* reads the data last written to PORTG pins */
  ```

For complete details about the software library, please refer to the documentation for the module `shiftreg.c`. 
PURPOSE OF MODULE

This module provides read and write access to PORTF and PORTG on the ME118 MicroCore-11 Interface PCB. Since these ports are created by adding a parallel-to-serial input shift register (PORTF) and a serial-to-parallel output shift register (PORTG) to the SPI on the MicroCore-11's 68HC11, special functions are required to read and write data.

INTERFACE

PORTF
This port is input-only. This is created by connecting a 74HC165 to the SPI's data/clock/SS* pins and shifting data in on request. Reads of this port return the current state of the 8 parallel input pins. There is no way to write to this port.

PORTG
This port is output-only. This is created by connecting a 74HC594 to the SPI's data/clock/SS* pins and shifting data out on request. Writes to this port force the 8 parallel output pins of the '594 to take on the values specified. Reads from this port return the value that was last sent to the pins.

Defined Constants
There are no defined constants necessary for the operation of this module.

MODULE FUNCTIONS

ReadPORTF
Prototype: unsigned char ReadPORTF(void)
Parameters: none
Returns: the current state of the PORTF input pins
Description: Reads and returns the current state of the PORTF input pins.

ReadPORTG
Prototype: unsigned char ReadPORTG(void)
Parameters: none
Returns: the current state of the PORTG output pins
Description: Returns the current state of the PORTG output pins.

Since PORTG is created by interfacing a serial-to-parallel output latch to the 68HC11's SPI, the output pins cannot be read directly. This module keeps track of what was most recently written to PORTG and returns that number when requested here.

WritePORTG
Prototype: void WritePORTG(unsigned char data)
Parameters: unsigned char data the data to write to the PORTG output pins
Returns: none
Description: Changes the state of the PORTG output pins to the states specified in the parameter data.

USAGE EXAMPLE

```c
#include <stdio.h>
#include "shiftreg.h"

int main() {
    int i;

    /* Jumper all pins of PORTG to PORTF. This program will then test to ensure that everything written to PORTG is correctly read by PORTF */

    printf("\n 1st read of PORTF = %x", ReadPORTF());
    printf("\n 1st read of PORTG = %x", ReadPORTG());
    printf("\n 2nd read of PORTF = %x", ReadPORTF());
    printf("\n 2nd read of PORTG = %x", ReadPORTG());
    for(i=0;i<30000;i++);
    while(1) {
        WritePORTG(0xAA);  PORTF = \x", ReadPORTF(), ReadPORTF());
        for(i=0;i<30000;i++);
        WritePORTG(0x55);  PORTF = \x", ReadPORTF(), ReadPORTF());
        for(i=0;i<30000;i++);
    }
}
```
The Timer Module

Purpose:
This module provides for 8 independent timer channels and a function to return the value on a free running timer. The timers may be used to either time external events or to schedule timed events in other code. These timers can all be independently be started, stopped and tested for completion. All timers share a common time base provided by the Real Time Interrupt (RTI) system on the 9S12 processors. That time base is selectable from the available RTI rates during the initialization of the timer system.

Header:
The header file TIMERS12.h should be included in any module wishing to use the functions provided by this module.

Initialization

Function:
void TMRS12_Init(unsigned char Rate)

Parameters:
unsigned char Rate
one of the TMRS12_RATE_XX defines to choose the RTI rate:
- TMRS12_RATE_1MS  1.024mS
- TMRS12_RATE_2MS  2.048mS
- TMRS12_RATE_4MS  4.096mS
- TMRS12_RATE_8MS  8.192mS
- TMRS12_RATE_16MS 16.384mS
- TMRS12_RATE_32MS 32.768mS

Returns:
Nothing

Description:
Initializes the timer module by programming the RTI interrupt and setting up the necessary data structures to maintain the timers. Must be called before using any of the other TMR routines. The Rate parameter sets the 'tick rate' of the RTI clock which is the basic unit of timing for this timer system.

Notes:
Does not perform any sort of sanity check on the Rate parameter. Should only be used with the defined constants listed above.

Usage:
TMR12_Init(TMRS12_RATE_4MS);
would initialize the Timer subsystem to run with a basic timing unit of 4.096mS per tick.

Program and Start a Timer

Function:
signed char TMRS12_InitTimer(unsigned char Num, unsigned int NewTime)
Parameters:

unsigned char Num
the number (0-7) of the timer to start

unsigned int NewTime
the number of ticks of the RTI clock to be counted.

Returns:

Signed char
TMRS12_ERR if the requested timer does not exist, TMRS12_OK otherwise.

Description:
Sets the NewTime into the chosen timer, clears any previous event flag for that timer and sets the timer active to begin counting the interval.

Notes:
Because of the inherent uncertainty of interacting with an asynchronous clock, the actual interval that will be timed will be between ((RTITickRate * NewTime)-1) and ((RTITickRate * NewTime)+1).

Usage:
TMRS12_InitTimer(0, TIME_OUT_DELAY);
Would set up the channel 0 timer and start counting TIME_OUT_DELAY number of ticks of the RTI clock.

Function:
signed char TMRS12_SetTimer(unsigned char Num, unsigned int NewTime)

Parameters:

unsigned char Num
the number (0-7) of the timer to start

unsigned int NewTime
the number of ticks of the RTI clock to be counted.

Returns:

Signed char
TMRS12_ERR if the requested timer does not exist, TMRS12_OK otherwise.

Description:
Sets the NewTime into the chosen timer, clears any previous event flag for that timer. Does not make the timer active to begin counting.

Notes:
Because of the inherent uncertainty of interacting with an asynchronous clock, the actual interval that will be timed will be between ((RTITickRate * NewTime)-1) and ((RTITickRate * NewTime)+1).

Usage:
TMRS12_SetTimer(0, TIME_OUT_DELAY);
CMPE 118 Timer Library for the MC9S12C32

Would set up the channel 0 timer to count TIME_OUT_DELAY number of ticks of the RTI clock.

Start a Timer

Function:

```c
signed char TMRS12_StartTimer(unsigned char Num)
```

Parameters:

- `unsigned char Num`:
  - the number (0-7) of the timer to start

Returns:

- `Signed char`:
  - TMRS12_ERR if the requested timer does not exist, TMRS12_OK otherwise.

Description:

Starts a timer counting down a count that was previously programmed into the timer. This function can be used to start a timer that was programmed with a previous call to TMRS12_SetTimer(). It may also be used to re-start a timer that was stopped with a call to TMRS12_StopTimer() before it expired.

Usage:

```c
TMRS12_StartTimer(0);
```

Would start the channel 0 timer counting ticks of the RTI clock.

Stop a Timer

Function:

```c
signed char TMRS12_StopTimer(unsigned char Num)
```

Parameters:

- `unsigned char Num`:
  - the number (0-7) of the timer to stop

Returns:

- `Signed char`:
  - TMRS12_ERR if the requested timer does not exist, TMRS12_OK otherwise.

Description:

Stops a timer counting. After a timer is stopped, there may be a count remaining that may be resumed with a call to TMRS12_StartTimer(). After a timer has been stopped, it is no longer considered 'Active'.

Usage:

```c
TMRS12_StopTimer(0);
```

Would stop the channel 0 timer from counting ticks of the RTI clock.

Check If a Timer is Active (counting)

Function:
signed char TMRS12_IsTimerActive (unsigned char Num)

Parameters:
  unsigned char Num
  the number (0-7) of the timer to stop

Returns:
  Signed char
  TMRS12_ERR if the requested timer does not exist, TMRS12_NOT_ACTIVE if timer is not active, TMRS12_ACTIVE if it is active

Description:
  Check to see if a timer is 'Active', that is, actively counting RTI ticks.

Usage:
  If(TMRS12_IsTimerActive(0) == TMRS12_ACTIVE)
  Would test the channel 0 timer to see if it was actively counting.

Check If a Timer has Expired

Function:
  signed char TMRS12_IsTimerExpired(unsigned char Num)

Parameters:
  unsigned char Num
  the number (0-7) of the timer to stop

Returns:
  Signed char
  TMRS12_ERR if the requested timer does not exist, TMRS12_NOT_EXPIRED if timer has not expired, TMRS12_EXPIRED if it has expired.

Description:
  Check to see if a timer has expired, that is, the previously programmed number of RTI ticks has elapsed.

Usage:
  If(TMRS12_IsTimerExpired(0) == TMRS12_EXPIRED)
  Would test the channel 0 timer to see if the previously programmed interval has expired.

Clear a Timer's Expired Flag

Function:
  signed char TMRS12_ClearTimerExpired(unsigned char Num)
Parameters:

unsigned char Num
the number (0-7) of the timer whose expired flag should be cleared

Returns:

Signed char
TMRS12_ERR if the requested timer does not exist, TMRS12_OK otherwise.

Description:
Clears the internal flag that shows that a timer has expired. Once a timer has been started and its interval expired, repeated calls to TMRS12_IsTimerExpired will continue to show that the timer has expired. Use TMRS12_ClearTimerExpired so that subsequent calls to TMRS12_IsTimerExpired will not return TMRS12_EXPIRED until the timer has been re-programmed and expired after the re-programming.

Usage:

TMRS12_ClearTimerExpired(0);
Would clear the expired flag associated with channel 0.

---

**Read the Free Running Timer**

**Function:**

unsigned int TMRS12_GetTime(void)

**Parameters:**

None

**Returns:**

Unsigned int
a number between 0 and 65535 reflecting the current number of ticks on the timer.

**Description:**
Simply returns the value on a free-running counter kept by the module. The count on the timer ticks once for each interval set by the RATE parameter in the call to TMRS12_Init.

**Usage:**

Foo = TMRS12_GetTime();
Would get the current time count and assign it to Foo.

---

**Example Usage**

```c
#include <timers12.h>

/* TIME_OUT_DELAY = 10 S w/ 8mS interval */
#define TIME_OUT_DELAY 1220

void main(void)
{
    unsigned int i;

    puts("Starting\n");
    TMRS12_Init(TMRS12_RATE_8MS);
```
if (TMRS12_InitTimer(0, TIME_OUT_DELAY) != TMRS12_OK)
    puts("Init of Timer 0 Failed\n");
if (TMRS12_InitTimer(1, TIME_OUT_DELAY) != TMRS12_OK)
    puts("Init of Timer 1 Failed\n");
if (TMRS12_InitTimer(2, TIME_OUT_DELAY) != TMRS12_OK)
    puts("Init of Timer 2 Failed\n");
if (TMRS12_InitTimer(3, TIME_OUT_DELAY) != TMRS12_OK)
    puts("Init of Timer 3 Failed\n");
if (TMRS12_InitTimer(4, TIME_OUT_DELAY) != TMRS12_OK)
    puts("Init of Timer 4 Failed\n");
if (TMRS12_InitTimer(5, TIME_OUT_DELAY) != TMRS12_OK)
    puts("Init of Timer 5 Failed\n");
if (TMRS12_InitTimer(6, TIME_OUT_DELAY) != TMRS12_OK)
    puts("Init of Timer 6 Failed\n");
if (TMRS12_InitTimer(7, TIME_OUT_DELAY) != TMRS12_OK)
    puts("Init of Timer 7 Failed\n");
while(TMRS12_IsTimerExpired(0) != TMRS12_EXPIRED)
{
    puts("Timed Out\n");
    TMRS12_InitTimer(7, TIME_OUT_DELAY);
    for (i=0;i<10,000 ;i++ )
        {/* kill some time */
    }
    TMRS12_StopTimer(7);
    if (TMRS12_IsTimerActive(7) != TMRS12_NOT_ACTIVE)
        puts("Timer Stop Failed\n");
    else
        puts("Timer Stop Succeeded\n");
    TMRS12_StartTimer(7);
    while(TMRS12_IsTimerExpired(7) != TMRS12_EXPIRED)
{
    }
    puts("Timed Out Again\n");
}
Pulse Generation (PLS) Module

Purpose:
This module provides a set of functions to give 2 channels capable of creating streams of pulses using the timer output compare features of the '9S12C32. It is started with a call to PLSS12_Init(). PLSS12_StartPulse() is called to generate a stream of a specified number of pulses at a specified period. The PLSS12_Done() function should be used to determine when the pulse stream is finished. When the program finishes, PLSS12_End() is used to shut down the PLS System.

Header:
The header file PLSS12.h should be included in any module wishing to use the functions provided by this module.

Initialization

Function:
void PLSS12_Init(void)

Parameters:
None

Returns:
Nothing

Description:
This is the initialization routine for the PLS functions. It will
1) Ensure that the timer system is enabled
2) Set the pre-scaler on the timer to give a 1.5MHz clock
3) Give control of Port T pins 3 & 4 to the Timer for use by the PLS library
4) Set Port T pins 3 & 4 to be outputs and initially low
5) Program no initial action on a compare
6) Enable interrupts for timer channels 3 & 4.

Notes:
Assumes a 24MHz bus clock.

Usage:
PLSS12_Init();
would initialize the PLS library and prepare for the generation of pulse stream on Port T bits 3 & 4.

Start a pulse sequence

Function:
PLSS12ReturnTyp PLSS12_PulseStart( unsigned int numPulses, unsigned int newPeriod,
PLSS12ChanTyp channel)
Parameters:

- unsigned int numPulses
  the number of pulses to be generated. A pulse is a high period followed by a low period.
- unsigned int newPeriod
  the period of the output waveform specified in 1/10mS increments. The period is in tenth milli-second increments between 1 and 87.3 ms. Output period = 0.1mS*newPeriod.

- PLSS12ChanTyp channel
  Which channel to start pulsing. Choices are PLSS12_Chan0 or PLSS12_Chan1.

Returns:

- PLSS12ReturnTyp
  a value indicating the result of the request:
  - PLSS12_ErrNotInstalled = The PLS system was not initialized prior to the call.
  - PLSS12_ErrBadChannel = The channel requested was not a legal value
  - PLSS12_ErrBadPeriod = The vales of the period did not fall between 10 & 873.

Description:

Starts an output channel producing a number of pulses.

Notes:

The output waveform always has a 50% duty cycle. The high time will be one-half of the total period specified.

Usage:

```c
PLSS12_PulseStart( 10, 873, PLSS12_Chan1);
```

would begin a stream of 10 pulses at the maximum period of 87.3mS.

Test for the end of a pulse sequence

Function:

```c
PLSS12ReturnTyp PLSS12_PulseFinished(PLSS12ChanTyp channel)
```

Parameters:

- PLSS12ChanTyp channel
  Which channel to test. Choices are PLSS12_Chan0 or PLSS12_Chan1.

Returns:

- PLSS12ReturnTyp
  a value indicating the result of the request:
  - PLSS12_ErrNotInstalled = The PLS system was not initialized prior to the call.
  - PLSS12_ErrBadChannel = The channel requested was not a legal value
  - PLSS12_Done = The pulse sequence on that channel has completed.
  - PLSS12_NotDone = The pulse sequence on that channel has not completed.

Description:

Test to see if a previously requested series of pulses has completed.

Usage:

```c
if(PLSS12_PulseFinished(PLSS12_Chan1) == PLSS12_Done)
```
End the PLS library control of timers & pins

Function:

    void PLSS12_End(void)

Parameters:

    None

Returns:

    Nothing

Description:

    Disables the PLS system, releasing control of Port T pins 3 & 4 and disabling the interrupts on timer channels 3 & 4.

Usage:

    PLSS12_End();

    would terminate the PLS system.

Complete example

```c
void main(void) {
    /* start by checking to be sure that the calls that should, recognize that
        the module has not been initialized */
    if ( PLSS12_PulseStart( 10,10, PLSS12_Chan0) == PLSS12_ErrNotInstalled )
        puts("PulseStart Recognized that things were not initialized\r\n");
    else
        puts("PulseStart Failed to Recognized that things were not initialized\r\n");

    if ( PLSS12_PulseFinished( PLSS12_Chan0) == PLSS12_ErrNotInstalled )
        puts("PulseFinished Recognized that things were not initialized\r\n");
    else
        puts("PulseFinished Failed to Recognized that things were not initialized\r\n");

    PLSS12_Init();
    TMRS12_Init(TMRS12_RATE_8MS);
    PLSS12_PulseStart( 10, 10, PLSS12_Chan0);
    while( PLSS12_PulseFinished(PLSS12_Chan0) != PLSS12_Done)  
        ; /* wait for pulses to finish */

    /* set up for button test  on TechArts carrier board */
    ATDCTL23 = 0x0;
    ATDCTL45 = 0x0;
    ATDDIEN = 0xFF;

    while ( (PTAD & BIT7HI) == BIT7HI)
        ; /* wait for SW3 button */

    while(1) { // produce sequence of 10 pulses at 87.3mS period
        PLSS12_PulseStart( 10, 873, PLSS12_Chan1);
        while( PLSS12_PulseFinished(PLSS12_Chan1) != PLSS12_Done)
            ; /* wait for pulses to finish */
```

```
TMRS12_InitTimer(1, 122); /* delay approx 1 sec */
while(TMRS12_IsTimerExpired(1) != TMRS12_EXPIRED)
{
}
}
9S12C32 Pulse Width Modulator Interface Module

Purpose:
This module provides a set of interface functions to perform a simplified initialization of the Pulse Width Modulation system on the Freescale MC9S12C32 microcontroller. The initialization function should be called before any use is made of the other functions. After the initialization is complete bits 0, 1 & 2 of Port T are dedicated to the PWM output functions for PWM channels 0, 1 & 2.

Header:
The header file PWMS12.h should be included in any module wishing to use the functions provided by this module.

---

Initialization

Function:
void PWMS12_Init(void)

Parameters:
None

Returns:
Nothing

Description:
Initializes the PWM subsystem for all three channels. Gives the PWM subsystem control of Port T bits 0, 1 & 2. Sets the initial PWM period to 2mS. Sets the Duty Cycle resolution to 1% and output duty cycle to 0%.

Notes:
The duty cycle resolution is fixed at 1% for this library.

Usage:
PWMS12_Init();
would initialize the PWM subsystem.

---

Set Duty Cycle

Function:
signed char PWMS12_SetDuty(unsigned char dutyCycle, unsigned char channel)

Parameters:
unsigned char dutyCycle
a number between 0 & 100, the new duty cycle
unsigned char channel
PWMS12_CHAN0, PWMS12_CHAN1 or PWMS12_CHAN2 to specify which channel's duty cycle to set.

Returns:
Signed char
PWMS12_ERR if requested channel does not exist or the duty cycle is over 100%.
PWMS12_OK otherwise
Description:
Sets the output duty cycle on the specified channel. Setting the duty cycle to 0% (full off) or 100% (full on) allows you to control the PWM channel as if it were a digital output.

Usage:
PWMS12_SetDuty(50, PWMS12_CHAN0)
Would set channel 0 (Port T bit 0) to output a 50% duty cycle signal at the previously programmed period.

---

**Set PWM Period**

**Function:**

`signed char PWMS12_SetPeriod(unsigned short newPeriod, unsigned char group)`

**Parameters:**

- `unsigned char newPeriod`
  a defined constant to specify the new period. Do not enter a number in mS or any other time units. This constant encodes the values needed to program the 9S12C32 timer system. Use only the pre-defined constants documented here or in PWMS12.h.

- `unsigned char group`
  PWMS12_GRP0 to specify group 0, which applies to channels 0 & 1 or PWMS12_GRP1 to specify group 1, which applies to channel 2.

**Returns:**

- Signed char
  PWMS12_ERR if requested group is invalid.
  PWMS12_OK otherwise

**Description:**
Sets the output period on the specified channel group.

**Notes:**
Takes newPeriod directly. Makes no sanity check on the period being programmed.

**Usage:**
PWMS12_SetPeriod(PWMS12_PERIOD_4MS, PWMS12_GRP0)
Would set channel group 0 (channels 0 & 1) to output a duty cycle with a 4mS period.

---

**Disable PWM Subsystem**

**Function:**

`void PWMS12_End(void)`

**Parameters:**
None

**Returns:**
Nothing
Description:
Disables the PWM subsystem for all three channels. Gives the control of Port T bits 0, 1 & 2 back to the DDR and port registers.

Usage:
PWMS12_End();
would disable the PWM subsystem.

Defined Constants For Period

#define PWMS12_4000US 20510
#define PWMS12_3300US 8390
#define PWMS12_2500US 8342
#define PWMS12_2000US 16414
#define PWMS12_1950US 4330
#define PWMS12_1650US 4294
#define PWMS12_1250US 4246
#define PWMS12_1000US 12318
#define PWMS12_825US 198
#define PWMS12_650US 4174
#define PWMS12_500US 8222
Purpose:
This module provides a set of interface functions to perform a simplified initialization of the A/D converter system on the Freescale MC9S12C32 microcontroller. The initialization function should be called before any use is made of Port AD. After the initialization is complete, reads of the analog input pins should be made using the ADS12_ReadADPin() function. Reads of digital input pins on the AD port should be made by reading from PT1AD. Writes to outputs defined on port AD should be performed by making assignments to PTAD.

Header:
The header file ADS12.h should be included in any module wishing to use the functions provided by this module.

Initialization

Function:
ADS12ReturnTyp ADS12_Init(char * modeString)

Parameters:
char [9] modeString
A null terminated string of 8 ASCII characters to describe the mode of each of the pins on Port AD. The legal values are: I for digital input, O for digital output, A for analog input. The string positions, reading left to right, correspond to the pins MSB to LSB (modeString[0]=MSB, modeString[7]=LSB)

Returns:
ADS12_Err if the input string is malformed
ADS12_OK if the mode string was OK

Description:
Initializes Port AD data direction register & ATDDIEN for digital I/O and the A/D converter to multi-channel, continuous conversion.

Notes:
Assumes a 24MHz bus clock, but simply sets the default values for now. Enforces a single block of A/D channels even though it is possible to make a contiguous sequence that occupies a non-contiguous block. For example, channels 6,7,0,& 1 are sequence contiguous but occupy 2 blocks so would be rejected by this code.

Usage:
if (ADS12_Init("OOIIAAAA") == ADS12_OK)
would initialize the A/D converter and port AD with bits 6 & 7 as outputs, 4 & 5 as inputs and bits 0-3 as analog inputs and test to see that the initialization was successful.

Analog Pin Read

Function:
short ADS12_ReadADPin( unsigned char pinNumber)

Parameters:
unsigned char PinNumber
the Port AD pin number to read the analog value from

Returns:
Description:
Reads and returns A/D conversion result from the appropriate A/D converter register.

Usage:
ADS12_ReadADPin(0)
Would read the analog value on bit 0 of port AD.
The ME118 SN754410 Dual 1A H-Bridge Module
Rev. 1
Smart Product Design Laboratory – Stanford University
Matt Ohline – February 2, 2000

Background:
Texas Instrument’s SN754410 is a quadruple half-H driver with control logic, integrated into a 16-pin DIP IC. Each driver has 1A maximum current drive capability. Two half-H drivers may be combined together to form a full H-bridge, so that it can be used for bi-directional control of motors or other peripherals (this is how the ME118 SN754410 Module uses it). The IC has thermal shutdown protection, which will disable the chip if it is operated beyond its current-drive specifications.

The ME118 SN754410 Dual 1A H-Bridge Module provides a convenient and robust interface to the SN754410. Separate connectors provide access to the logic-level inputs, H-bridge output connections, and the IC’s power supply. The module makes use of a 74HC14, which simplifies the drive logic by reducing it to two inputs per H-bridge: direction and enable. Clamping diodes are included on the module PCB to protect the circuit from inductive kickback.

Using the ME118 SN754410 Dual 1A H-Bridge Module:
In order to make use of the ME118 SN754410 Dual 1A H-Bridge Module, you will need to be familiar with the various connectors and their purposes. Since each connector has a single logical function (inputs, outputs, power supply, etc.) this is straightforward.

Logic-Level Inputs (J1):
Access to the logic-level inputs of the SN754410 is provided through J1. Directional and enable control for each of the IC’s two H-Bridges is specified through these connections.

Logic ground connections must also be made through this connector. This will ensure that the ground of the (off-board) logic command circuitry and the ground of the ME118 SN754410 Module agree.

The pinout of J1 is as follows:

<table>
<thead>
<tr>
<th>J1 Connection</th>
<th>Pin 1</th>
<th>Pin 2</th>
<th>Pin 3</th>
<th>Pin 4</th>
<th>Pin 5</th>
<th>Pin 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic ground</td>
<td>logic ground</td>
<td>Channel A direction</td>
<td>Channel A enable</td>
<td>Channel B direction</td>
<td>Channel B enable</td>
<td>logic ground</td>
</tr>
</tbody>
</table>

Motor/Load Connections (J2):
Connections of up to two independent motors or similar loads should be made through the screw-terminal connector located at J2. The pinout of J2 is as follows:

<table>
<thead>
<tr>
<th>J2 Connection</th>
<th>pin 1 &amp; pin 2</th>
<th>pin 3 &amp; pin 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motor/load A</td>
<td>motor/load A</td>
<td>motor/load B</td>
</tr>
</tbody>
</table>

SN754410 Power Supply and High-Current Ground Connector (J3):

J3, pin 1: The SN754410 requires a power supply for its logic circuitry. Provisions for this are made through J3, pin 1. An LM2931Z-5.0 low drop-out, 100mA voltage regulator is provided on the SN754410
Module PCB so that any voltage between 5.6V (minimum) and 18V (maximum) may be supplied to J3, pin 1. This makes the SN754410 module much easier to use, since it includes its own voltage regulation and does not require an externally regulated +5V supply.

**SN754410 Power Supply Requirements:**  
+5.6V < Vin (J3, pin 1) < +18V

**J3, pin 2:** Since the SN754410 is capable of switching two H-Bridge channels at a maximum of 1A each, care must be taken in the methods employed returning this substantial current to ground. For this reason, a separate high-current ground connection is available at J3, pin 2. A separate connection should be made between J3, pin 2 and the power supply of the motors/loads connected to the IC’s outputs. This will ensure that the logic power supply maintains a clean ground.

**ME118 SN754410 Dual 1A H-Bridge Module Schematic:**
The ME118 MC3479 Stepper Motor Driver Module
Rev. 1
Smart Product Design Laboratory – Stanford University
Matt Ohline – February 2, 2000

Background:
Motorola’s MC3479 provides logic and power drive stages for controlling 2-phase bipolar stepper motors. It has output stages capable of supplying up to 350mA/phase. The maximum current supplied to a motor is internally limited by the MC3479, and this current limit may be adjusted by the selection of a resistor. Logic-level inputs are provided for the step clock, step direction (CW/CCW), and stepping mode (full/half-step). An integral clamp diode protects the IC from inductive kickback resulting from the switching of the stepper motor’s phases.

The ME118 MC3479 Stepper Motor Driver Module provides a convenient and robust interface to the MC3479. Separate connectors provide access to the logic-level inputs, stepper motor phase connections, current-limit-setting bias resistor, and the IC’s power supply. The stepping mode is jumper-selectable: the user may choose to drive the MC3479 in Full-Step Mode only, or to actively control the mode (Half-Step or Full-Step Mode) via external logic. In addition, the state of inactive outputs is jumper-selectable. An external clamp diode is included for extra protection from inductive kickback.

Using the ME118 MC3479 Stepper Motor Driver Module:
In order to make use of the ME118 MC3479 Stepper Motor Driver Module, you will need to be familiar with the various connectors and their purposes. Since each connector has a single logical function (inputs, outputs, power supply, etc.) this is straightforward.

Logic-Level Inputs (J1):
Access to the logic-level inputs of the MC3479 is provided through J1. At a minimum, inputs for the stepping clock and direction must be supplied. The attached data sheet for the MC3479 provides complete specifications for the drive requirements of these inputs. The pinout of J1 is as follows:

<table>
<thead>
<tr>
<th>J1 Connection</th>
<th>Pin 1</th>
<th>Pin 2</th>
<th>Pin 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>stepping clock</td>
<td>stepping direction – CW*/CCW</td>
<td>step mode – FULL*/HALF^1</td>
<td></td>
</tr>
</tbody>
</table>

Note 1: step mode is also jumper-selectable: see the section below describing J6

Stepper Motor Connections (J2):
Connections to the terminals of a 2-phase bipolar stepper motor should be made through the screw-terminal connector located at J2. The pinout of J2 is as follows:

<table>
<thead>
<tr>
<th>J2 Connection</th>
<th>pin 1 &amp; pin 2</th>
<th>pin 3 &amp; pin 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>motor phase A</td>
<td>motor phase B</td>
<td></td>
</tr>
</tbody>
</table>

Output Impedance Control Header – OIC (J3):
The MC3479 gives the designer the option of configuring inactive outputs as high-impedance outputs or as low-impedance outputs to Vm (the motor supply voltage). By choosing the location of the jumper at J3, one of these 2 configurations will be selected. This option is only relevant when the Half-Step stepping mode is selected. Installing the jumper as described below will determine the state of inactive outputs:

<table>
<thead>
<tr>
<th>J3 Connection</th>
<th>IOC Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>pin 1 + pin 2 jumpered</td>
<td>unused outputs are in high impedance state (referenced to Vm)</td>
</tr>
<tr>
<td>pin 2 + pin 3 jumpered</td>
<td>unused outputs are in low impedance state (referenced to Vm)</td>
</tr>
</tbody>
</table>
Bias Resistor Connector (J4):
The coil-drive stage of the MC3479 has internal current limiting. The current limit is selected through the choice of the resistor installed in J4 (between the MC3479’s BIAS pin and GND). Refer to the attached data sheet for complete specifications relating the value of the Bias resistor to the MC3479’s current limit.

MC3479 Power Supply (J5):
The MC3479 has an internal voltage regulator, so powering the IC is quite simple. Power and ground for the chip should be supplied through J5, according to the following requirements:

<table>
<thead>
<tr>
<th>J5</th>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>pin 1</td>
<td>+7V &lt; +Vin &lt; +16.5V</td>
</tr>
<tr>
<td>pin 2</td>
<td>-Vin = GND</td>
</tr>
</tbody>
</table>

Full Step Mode Header (J6):
The MC3479 is capable of stepping a 2-phase bipolar stepper motor in either Full Step or Half Step mode. Since the MC3479 is often used in Full Step mode, J6 has been provided so that the user can select with a jumper whether he/she wishes to use Full Step only, or allow the stepping mode to be under external control through J1, pin 3. Installing the jumper as described below will configure the Step Mode as follows:

<table>
<thead>
<tr>
<th>J6</th>
<th>Step Mode Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>pin 1 + pin 2 jumpered</td>
<td>user control – input derived form J1, pin 3</td>
</tr>
<tr>
<td>pin 2 + pin 3 jumpered</td>
<td>Full Step mode – input at J1, pin 3 is not connected</td>
</tr>
</tbody>
</table>

ME118 MC3479 Stepper Motor Driver Module Schematic:
The ME118 DS3658 Driver Module
Rev. 1
Smart Product Design Laboratory – Stanford University
Matt Ohline – February 2, 2000

Background:
The DS3658 is a four-channel, high-current peripheral driver, manufactured by National Semiconductor. It integrates logic-level inputs, four open-collector output channels (each capable of sinking a maximum of 600mA), and internal kickback diodes.

The ME118 DS3658 Driver Module PCB provides a convenient and robust interface to the DS3658. Separate connectors provide access to the logic-level inputs, high-current switched outputs, kickback diodes, and the Module’s power supply input.

Using the ME118 DS3658 Driver Module:
In order to make use of the ME118 DS3658 Driver Module, you will need to be familiar with the various connectors and their purposes. Since each connector has a single logical function (inputs, outputs, power supply, etc.) this is straightforward.

Logic-Level Inputs (J1):
Access to the logic-level inputs to each of the four channels of the DS3658 is provided through J1. In order to turn on a channel’s high-current, open-collector output, a logic-level high must be supplied to that channel’s input pin. The attached data sheet for the DS3658 provides complete specifications for the drive requirements of these inputs.

Logic ground connections must also be made through this connector. This will ensure that the ground of the (off-board) logic command circuitry and the ground of the ME118 DS3658 Driver Module agree.

The pinout of J1 is as follows:

<table>
<thead>
<tr>
<th>J1</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>pin 1</td>
<td>logic ground</td>
</tr>
<tr>
<td>pin 2</td>
<td>Channel A input</td>
</tr>
<tr>
<td>pin 3</td>
<td>Channel B input</td>
</tr>
<tr>
<td>pin 4</td>
<td>Channel C input</td>
</tr>
<tr>
<td>pin 5</td>
<td>Channel D input</td>
</tr>
<tr>
<td>pin 6</td>
<td>logic ground</td>
</tr>
</tbody>
</table>

High-Current Switched Outputs (J2):
Access to the open-collector switched outputs is provided through the screw-terminal connector at J2. Each channel is capable of sinking a maximum of 600mA, and multiple channels may be used in parallel to provide additional current capability (see the attached data sheet). The pinout of J2 is as follows:

<table>
<thead>
<tr>
<th>J2</th>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>pin 1</td>
<td>Channel A output</td>
</tr>
<tr>
<td>pin 2</td>
<td>Channel B output</td>
</tr>
<tr>
<td>pin 3</td>
<td>Channel C output</td>
</tr>
<tr>
<td>pin 4</td>
<td>Channel D output</td>
</tr>
</tbody>
</table>

Integrated Kickback Diodes (J3):
Access to the DS3658’s integral kickback diodes is provided through the screw-terminal connector at J3. There are 2 kickback diodes built into the DS3658, each protecting two of the IC’s four channels. If they are to be used to protect the circuitry of the DS3658 from being subjected to excessive voltages at the open-
collector outputs, external connections from the cathode of the internal diodes to the power supply of the connected switched load are required. If this protection is not needed, the diodes may be left unconnected.

J2 has the following pinout:

<table>
<thead>
<tr>
<th>J2</th>
<th>Diode</th>
<th>Protects these outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>pin 1</td>
<td>CLAMP 1</td>
<td>Channel A &amp; Channel B</td>
</tr>
<tr>
<td>pin 2</td>
<td>CLAMP 2</td>
<td>Channel C &amp; Channel D</td>
</tr>
</tbody>
</table>

**DS3658 Power Supply and High-Current Ground Connector (J4):**

**J4, pin 1:** The DS3658 requires a power supply for its logic circuitry. Provisions for this are made through J4, pin 1. An LM2931Z-5.0 low drop-out, 100mA voltage regulator is provided on the DS3658 Driver Module PCB so that any voltage between 5.6V (minimum) and 18V (maximum) may be supplied to J4, pin 1. This makes the DS3658 module much easier to use, since it includes its own voltage regulation and does not require an externally regulated +5V supply.

*DS3658 Power Supply Requirements:* \(+5.6V < Vin (J4, pin 1) < +18V\)

**J4, pin 2:** Since the DS3658 is capable of switching four channels at a maximum of 600mA each, care must be taken in the methods employed returning this substantial current to ground. For this reason, a separate high-current ground connection is available at J4, pin 2. A separate connection should be made between J4, pin 2 and the power supply of the load(s) connected to the IC’s outputs. This will ensure that the logic power supply maintains a clean ground.

**ME118 DS3658 Driver Module Schematic:**

[Diagram of the DS3658 Driver Module Schematic]