Frequency Generator & Frequency Counter Modules for Arduino Pro-Micro (or other ATMega32U4 module)

Overview

Many times it is necessary to produce a programmable frequency output or implement a frequency counter for various uses. This project implements either or both of these functional blocks using only a low cost (≈$5-$17) Pro Micro (Sparkfun DEV-12640 or equivalent clone) module with an ATMega32U4 processor chip and software implemented using the Arduino IDE compiler and environment. These functional blocks can be used separately or together in a larger application or as a stand alone device. Since they are independent functions they will be covered separately. As part of this project a common main module is also included that can be used to set up the frequency generator or read the frequency from the frequency counter. This may be done via a serial port or alternatively, with 4 push button switches and an LCD display to use these functions as a stand alone device.

Unlike many of the projects listed on this site, instead of implementing some final device, this project focuses on the functional blocks for a frequency generator and a frequency counter. It is assumed that the user will then write a main module that deploys these modules for whatever purpose is desired, and may include an LCD display, a keyboard or other external interface, or may be part of some larger project the user needs. Finally, because both the frequency generator and frequency counter depend on the Pro Micro's normally uncalibrated internal crystal oscillator, details of how to modify the module to improve the accuracy of the oscillator is detailed.

It should be noted that other Arduino modules with the ATMega32U4 processor could also be used, including the Arduino Leonardo and Arduino Micro and possibly others.

Frequency Generator

This library implements a variable frequency generator using Timer 4 on an Arduino Pro Micro module (using an ATMega32U4). This signal may be used to output an adjustable frequency square wave signal to any user defined device or circuit. The frequency generator will output a square wave signal with a 50% duty cycle from 1 Hz to approximately 12MHz, using hardware contained within the Pro Micro module itself. It can be used while also using all other functions of the module. Timer 4 on this processor/module is unique in that it can be connected to the processors main clock or can be connected to the PLL contained in the ATMega32U4 providing higher initial frequencies and some additional scaling values including 1.5x clock, providing more combinations of count values to produce an output signal closer to any desired frequency. Several other ATMega/Tiny processors also have this type of timer, but they are not too common

The firmware for setting the frequency contains a unique algorithm that sets the output frequency to the closest value to the requested frequency as is possible by selecting from one of three different input clock frequencies (via the on-chip PLL), a binary prescaler value and a count value. With all these variables, which settings for each to obtain a given frequency could be challenging if they had to be

determined externally. With the algorithm contained in the firmware, the firmware itself determines the optimal value for each of these three values so that the output frequency is as close to the requested value as is possible. Once these values are calculated and set, the Pro Micro module will output the requested frequency with no further firmware/software involvement since it is using the timer/counter in an auto reload mode with no interrupts.

Hardware

The Pro Micro module will output the selected frequency clock signal on Arduino Digital pin 5 (processor pin PC6) as a 0-5V 50% duty cycle square wave signal with no further modifications to the controller. Alternatively, the output can also be configured to output the signal on Arduino Digital pin 10 (processor pin PB6), if desired. The figure below shows the pin(s) for the frequency generator output on the Pro Micro module.

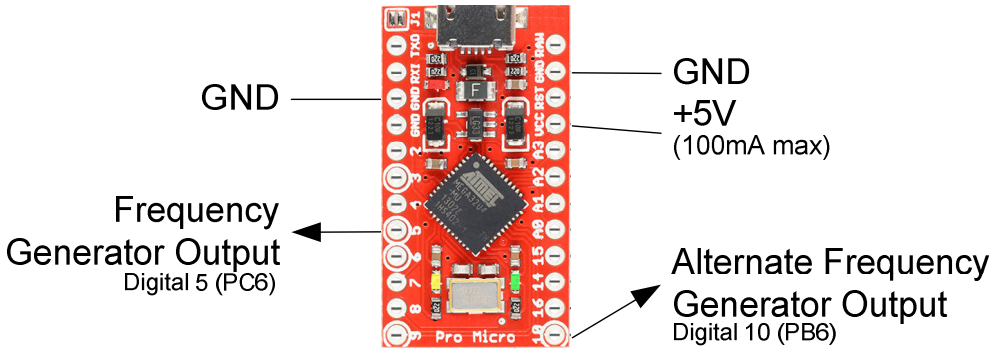

Figure 1 Frequency Generator Output Pins

Depending on your application, no additional circuitry may be required, but if different levels or more power than the modules output pin can deliver is required, an external amplifier, level shifter or resistive divider may be needed. A 5V power source is also shown that can be used to power this external circuitry if needed and its power requirements are less than 100mA.

The frequency accuracy of this generator is directly related to the crystal oscillator on the Pro Micro module and is usually within 0.1-0.2%, but is typically closer than this to the nominal frequency. See the section below that details modifications to the Pro Micros crystal oscillator if greater accuracy is needed. Frequency accuracy is also related to the values that are set in the various registers of the processor chip. While the algorithm tries to determine the best values for the PLL, prescaler and counter, there are frequencies that are unobtainable with the hardware, without additional software intervention, which is beyond the scope of this project. Almost all lower frequencies (below about 5KHz) are perfect (or nearly perfect), meaning the module can output the exact frequency, but as the requested frequency goes higher, there will be ‘gaps’ where the exact frequency is not achievable. In these cases the closest frequency obtainable will be set and is usually close enough for most applications. The duty cycle of the output is fixed at 50%.

Software

Interface Functions

The library is implemented in a single class named “FrequencyGenerator”. This class has two functions: “set” and “read” to set the frequency generator and read the current frequency that the generator is set to.

Set Function

The frequency is set with a frequency set command, which sets the frequency to the closest obtainable value. The function ‘FrequencyGenerator::set’ accepts a long integer that is the frequency to output in Hz. This may range from a value less than 0 (return current frequency) to 0 (generator off) to a value some over 1/2 the clock frequency of the micro (8MHz). With the stock Pro Micro modules available, it has been observed to work to about 12MHz. The ‘FrequencyGenerator::set’ function returns the set frequency if the frequency requested was set or -1 if unable to set to the desired frequency. Since the timer is set up to automatically reload, no interrupts or other software overhead is required — Just call the function and then the hardware will produce the output frequency with no additional intervention.

To turn off the frequency generator, simply call the ‘FrequencyGenerator::set’ function with a frequency of zero. This will disable the frequency generator and put its output pin in a high impedance state. The current frequency can also be requested by calling the ‘FrequencyGenerator::set’ function with -1 (or any negative number).

Read Function

A function ‘FrequencyGenerator::read’ is also provided to request the current frequency of the frequency generator module. The value returned will be the actual frequency that the controller is outputting (and the closest value the hardware is capable of outputting) and may be slightly different than the last set frequency. This is returned as a long integer.

Code Details and Options

The frequency generator functionality is contained in the file “FrequencyGenerator.cpp”. The header file that contains the class prototype is in the file “FrequencyGenerator.h”. This module was specifically written for Timer 4 of the ATMega32U4 on a Pro Micro device (or other microcontroller devices with similar functional blocks) assuming the controller is running as a USB device with the PLL set at 96MHz and a crystal of 16MHz (This is the default for the Pro Micro device). It could possibly be reworked for other timers, but the resolution would be less.

The code in the “FrequencyGenerator.cpp” module contains the algorithm to set the various registers in the processor to the best values for the input frequency desired. It does so by successively doing the calculations for each PLL setting and then determining which PLL setting provides the closest output frequency to the desired value. The prescaler and count value are determined by calculating the log2 of the desired frequency and then converting these to a prescaler and count values for the registers. Once the best combination is determined the physical registers are set to these values. Comments in the code detail this algorithm in more detail.

Within the FrequencyGenerator.cpp module there are 2 conditional compilation defines that can either be defined or not. One of them is ‘FRQGENUSEPB6’ which, if defined to non-zero, will use Arduino Digital pin 10 (processor pin PB6) for output of the generator, instead of the default Arduino Digital pin 5 (processor pin PC6). The other define is ‘FRQGENDEBUG’, which, if defined to non-zero, will output some of the variables values to the serial port when setting the frequency for debug purposes.

Frequency Counter

This library implements a frequency counter using hardware timer/counter resources that are part of the Pro Micro module. It uses Timer 0 and another timer as a "gate" timer on the Pro Micro module. It works either as a traditional frequency counter by counting the number of pulses that occur on an input pin for a fixed amount of time (the gate time) or alternatively as a period counter where the time of a single or multiple cycles of the input waveform are measured. The traditional frequency counting mode is best when measuring faster signals while the period measuring mode provides more sub Hertz resolution when measuring lower frequency signals. The interface to this module is via 2 to 3 main functions. The first function provides the frequency read to the user, while a second function sets the frequency counter to either the traditional counting mode or the period measure mode and also sets either the gate time or the number of input pulses to average. Another function is available, if desired, to determine if a new "fresh" reading is available. Since this module uses hardware timers and interrupt routines to implement the frequency counter, it can be used while also using all other functions of the Pro Micro module and the Arduino environment in addition to whatever user written code is desired.

When the counter is configured in the traditional counting mode, Timer 0 is configured to count input pulses and a second timer (Timer 1) is used as the gating source. When counting, every time Timer 0 overflows, an interrupt routine adds one to a saved count variable. The gating timer (Timer 1) is used to periodically move this accumulated count to another variable that will then be returned to the user when the frequency is requested, and then resets the count in the Timer 0 counter (and it's saved count variable) for the next cycle. The gating timer can be configured to one of 10 mS, 100 mS, 1 Second, 10 Seconds or 100 Seconds. The gating timer is also interrupt driven, so that the counter module can accurately time the gating window. There is also an external gating mode where a level on an external pin can be used to "gate" the frequency counter.

When the counter is in the period measuring mode, Timer 0 is used to count either a single transition or multiple transitions on the input pin. The system microseconds count (that is part of the Arduino environment) is saved upon the receipt of the first transition, and then when the second transition occurs, the microseconds value is again read and the saved microseconds value is subtracted from it to determine the period of the input signal. Then when the user requests the frequency, this value is converted to a frequency and returned to the user. In the period measuring mode, one of three averaging modes is available. Either no averaging or an average of 10 or 100 input pulses is selectable. This averaging is done by setting the Timer 0 counter to generate an interrupt on either 1, 10 or 100 transitions, effectively averaging the input signal.

Hardware

Input to the frequency counter is on Arduino Digital pin 6 (processor pin PD7) for Pro Micro (ATMega32U4) and is assumed to be a low going train of pulses to count. The figure below shows the counter input and also the external gate input pin (detailed later). The external gate input, if used, is on Arduino Digital pin 9 (processor pin PB5), but can be moved to another pin if desired.

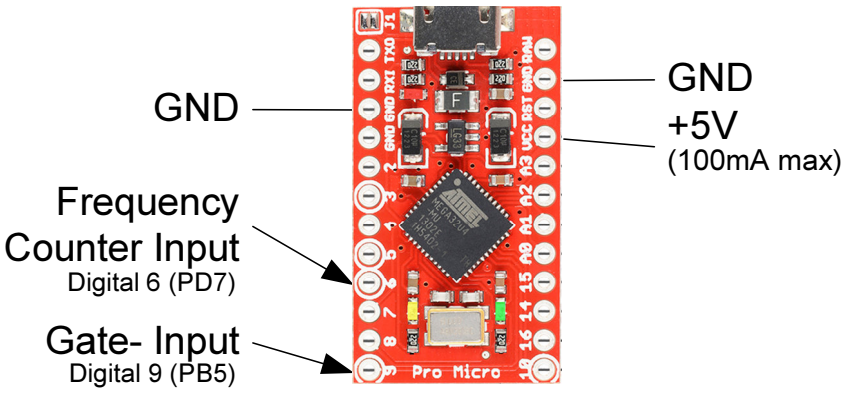

Figure 2 Frequency Counter Pins

Obviously the input signal must be an appropriate TTL level to be counted. Input amplifiers to amplify a lower level signal to this TTL level (if needed) are beyond the scope of this work. A 5V power source is also shown that can be used to power any external circuitry if needed and its power requirements are less than 100mA.

The frequency accuracy of this counter is directly related to the crystal oscillator on the Pro Micro module and is usually within 0.1-0.2%, but is typically closer than this to the actual frequency. See the section below that details modifications to the Pro Micros crystal oscillator if greater accuracy is needed.

The accuracy is also directly related to the accuracy of the gate timer for the traditional frequency counter mode and the milliseconds timer for the period measuring mode. Both of these timers are interrupt driven in this project, meaning that they occur autonomously without any software intervention and occur as close to immediately as is possible in the Arduino environment. While there is some overhead for the interrupt routine, this is usually negligible and the frequencies read are typically very close to or normally exactly the correct frequency.

Testing on different modules that were available during development of this project revealed that in the traditional frequency counter mode, the Pro Micro module was able to reliably count frequencies up to about 8MHz (1/2 of the processor clock frequency). In the period measure mode, which is intended for measuring lower frequencies with more resolution, frequencies up to 20 KHz were possible and reliable when the averaging setting is 10 and 100 and 10 KHz when the averaging was turned off (average of 1). In the period measure mode, if the frequency is above these values, the module will either report a value of ‘999999’ or in some rare cases an erroneous value.

If the application requires measuring frequencies higher than about 8 MHz, then an external divider chain (prescaler) can be added to the input signal prior to connecting it to this counter module and then adjusting the values returned by this module accordingly. A 74196 or a collection of 74F74’s can be used for frequencies up to about 60-100MHz and other prescaler IC’s can be used for even higher frequencies. If a prescaler is used then a define named ‘FCPRESCALER’ in the “FrequencyCounter.cpp” module can be adjusted to the divisor value so that the output frequency returned is correct with the prescaler being used.

Software

Interface Functions

The library is implemented in a single class named “FrequencyCounter. This class has two main functions and a couple of ancillary functions that may be needed, depending on the exact implementation chosen.

Mode Function

The first function, ‘FrequencyCounter::mode’, is used to set or read the counting mode, which includes the mode and either the gate time or the number of averages. This is called with a single parameter that specifies the mode to use, or requests the current mode. If the input parameter is -1 (or any value less than 0), then the function returns the current mode value. If the input parameter is in the range of 0 to 9 the following mode is set:

Value	Function
0	Frequency counter off (All frequency counter interrupts disabled, I/O pins set to input)
1	Counter mode. Gate time of 1 Second.
2	Counter mode. Gate time of 10 mS.
3	Counter mode. Gate time of 100 mS.
4	Counter mode. Gate time of 10 Seconds.
5	Counter mode. Gate time of 100 Seconds.
6	Counter mode. Gate time is from external input on Arduino Digital pin 9 [PB5]. *
7	Period measure mode. No averaging. **
8	Period measure mode. Average 10 input transitions. **
9	Period measure mode. Average 100 input transitions. **

* Only if 'FCEXTERN' is defined. If 'FCEXTERN' not defined then mode 7..9 become 6..8.

** Only if ‘FCPERIOD’ is defined

The function always returns the current mode selected or -1 if the input parameter is outside the range of the values mentioned above. Modes 6 thru 9 may or may not exist, depending on how the module is compiled. If the ‘FrequencyCounter::mode’ function is called without a parameter, it will return the current mode.

Read Function

The second function, ‘FrequencyCounter::read’, is used to read the frequency counter and return a string that is a numeric floating point value of the frequency read. This function is called with a pointer to a string buffer that will be filled in by this function and a parameter called ‘Wait’, that if non-zero will cause the function to wait until a new “fresh” reading of the frequency counter is available. The function returns a pointer to a string that contains this frequency value and is of the form “12345678” or “12345.67890” (or similar). The string buffer location passed to this function must be large enough to hold the string that this function will create – 15 or more characters is suggested. The parameter ‘Wait’ should be zero to get the last frequency read or non-zero to wait for a new “fresh” frequency count. It should be noted that if the function is called with the ‘Wait’ parameter as a non-zero value, the function may take up to 100 seconds to return, depending on the mode the frequency counter module is in. If this is unacceptable, a polling function is also available to see if a new frequency reading is available. This function returns a string that can be a floating point value instead of an actual binary floating point value type so that the compiler’s floating point functions (that consume a lot of code space) are not required.

Available Function

As mentioned in the last paragraph, a function to determine if a new “fresh” frequency reading is available is also included. The function ‘‘FrequencyCounter::available’’ requires no parameters and returns a 1 if a new value is ready and 0 if a new value is not ready.

Most users will only need the two or three functions mentioned above. In special circumstances, a couple of other functions are available.

Read Binary Function

When the function ‘‘FrequencyCounter::read’’ is called with only a “Wait” parameter (and no pointer to a string parameter) the read function returns a long integer that is the uncorrected count or period of the frequency read. The user would then have to scale / convert this to the proper value for the needs of the application. Like the string version of ‘‘FrequencyCounter::read’’ function this function also accepts a ‘Wait’ parameter that works the same as it does for that function. This function is normally not used, but is available in cases where other code (perhaps in the host application) needs the uncorrected binary value read for whatever reason. See the code for the string version of this function to see how this value is scaled and converted to a floating point string to see how this uncorrected value becomes “corrected”.

FreqCtrGateISR Function

Finally a function called ‘FreqCtrGateISR’ can be called by a user routine, if the default linkage to the system timer and its 10mS interrupt is not desired. This function is not part of the class, accepts no parameters and returns nothing. (This function is not needed if using the default library files as published.)

Code Details and Options

The frequency counter functionality is contained in the file “FrequencyCounter.cpp”. The header file that contains the class prototype is in the file “FrequencyCounter.h”. This module contains the code for the functions mentioned above. The modules “SysTimer.cpp” and “PCInterrupt.cpp” may also be needed, depending on the configuration desired by the user of this project.

In the traditional frequency counter mode this module requires a “gating” source that is typically a different timer that occurs at a fixed time rate. If not using the supplied “SysTimer.cpp” module then the function ‘FreqCtrGateISR’ needs to be called at the proper periodic rate (10mS). The accuracy of this module is directly affected by the accuracy of this gate timing source!

System Timer and Processor Resources

This module is designed for the Arduino Pro Micro module that uses an Atmel ATMega32U4, and uses Timer 0 for the counting input. It would be ideal to use another timer for this function, but unfortunately the ext clock pin for Timer 1 (ATMega32U4 pin 26) is not routed to the modules connector and only Timer 0 and Timer 1 have an external clock input, so no alternative exists other than to use Timer 0. As is known, Timer 0 is normally used for the Arduino system timing functions (delay, millisec, microsec, etc.), so the only alternative is to move these functions to some other timer. This is done by reworking the stock wiring.c module that is part of the Arduino system to use a different timer for this purpose. Then this “system” timer is expanded to also use it for the gate timer for the frequency counter function.

By simply including “SysTimer.cpp” in the build, the functions normally implemented by Timer 0 (delay, millisec, microsec, etc.) are moved to use this alternate timer (configurable to Timer 1 or Timer 3), and a hook to the timer's interrupt service routine (ISR) is accessible, allowing this timer to be used for the frequency counter gating function. It might be best to use Timer 1 for the gate, because it has the highest interrupt priority (even above Timer 0), but either Timer 1 or Timer 3 is able to be used. If the

user's application does not include the SysTimer module, then it must call the function 'FreqCtrGateISR' using other code that must be written. The SysTimer module can also be used for purposes other than the frequency counter, if desired. It provides a convenient way to call a user created function each millisecond and moves the default system timer functions (delay, millisec, microsec, etc.) to another timer.

Obviously, when using an alternate timer for the "system" timer function and frequency counter gating function, the timer will not be available for PWM functions or any other Arduino "built in" library functions normally depend on Timer 0 or the timer selected for the "system" timer in the Arduino software.

It should be noted that if using USB to communicate with the Pro Micro module, then the count returned may not be as precise as when using other or no external communications methods. This is because the USB interrupts have a higher priority than the Timer 1 / Timer 3 timer that is used for the gate timer. A possible alternative would be to set up another timer for the gate time and output it on a pin and then use a pin change or external interrupt as the gate timer source (Pin change and external interrupts have a higher priority than the USB interrupts). Also note that when measuring frequencies above about 2MHz the count returned may, in rare cases, be short a count or two. This is because the code in this module has to clear the counter, turn it off and then back on again, which takes a couple of CPU cycles.

External Gating

This frequency counter module can also be set up to use external gating. To do so, define 'FCEXTERN' as non-zero and include the module "PCInterrupt.cpp" in this sketch. Then by setting the 'Mode' function to 6, the Arduino pin defined by 'FCEXTGATEMSK' will be used as the gate input. By default, this signal is input on Arduino Digital pin 9 (processor pin PB5), but can be moved to another pin if desired. See "PCInterrupt.h" for a list of 'PCINTMASKxx' pins – Any of these pins can be used. When activated, a low on this pin turns on the gate, and when high the gate is turned off. Including the external gate function is optional, but included as defined in the distribution files. Adding the external gating function adds about 250 bytes to the code size of the module. If not included, then the period counting functions will have an index of one less than mentioned above in the function ''FrequencyCounter::mode''.

Using the external gate function and another timer set up to work autonomously and then output its signal on some other pin, and then connecting this pin to the 'Ext Gate' input is a possible way to get around the USB problem mentioned above that might affect the count, because the external gate inputs are all higher in priority than the USB interrupt are.

Period Measuring Modes

When using the frequency counter module in the period measure mode, Timer 0 is used to detect transitions on the input pin and then measures the number of microseconds between these transitions using the Arduino system microseconds counter. When in this mode, the input is still on the same Timer 0 input pin, but instead of counting the number of pulses that occur in a given time, Timer 0 is set to generate an interrupt on the first, tenth or hundredth transition on the Timer 0 input and the number of microseconds that have passed between two of these transitions is measured and then converted to a frequency. This allows a more accurate measurement of lower frequency signals in a minimal amount of time. When Timer 0 is set up to count 10 transitions, this effectively averages 10 periods of the input signal, and when set to 100, it averages 100 periods of the input signal. Including the period measuring modes is optional and only included if the define 'FCPERIOD' is defined as a non-zero value (which it is for the distribution files). Including the period measuring modes adds about 1000 bytes to the code size of the module.

Timeout for period measuring modes

When using the traditional frequency counting mode, the function 'FrequencyCounter::read' is guaranteed to return after a fixed amount of time; However in the period measure mode, if there is no input signal, a new "fresh" frequency count will never occur and the read function will not return. This could be problematic in some cases. To guarantee that this function will always return after a given time, the frequency counter module contains a time out counter that will time out after a given amount of time when using the period measure mode. This time out timer uses the same timer interrupt as the gate counter uses and is part of the function 'FreqCtrGateISR'. If no input is found, then after this timer times out, a new "fresh" frequency of zero Hertz is reported (and the frequency read function will return). This time out value defaults to 5 seconds but can be changed, if desired, by changing the value of the define 'PERIODTIMOUT'. If this define is 0, then no time out functionality will be included with period measure functions.

Main Module

As mentioned in the introduction, this project also includes a "main" module (FreqGenCtrApp.ino) that is used to test the frequency generator and frequency counter module. It provides a simple ASCII command interface with mostly 1 or 2 character commands to control the modules that are part of this project or alternatively a stand alone device can be created with the addition of 4 pushbutton switches and an LCD display. It can be used with either the frequency generator module or the frequency counter module or both. The use of this module is optional and is included merely to provide a convenient way to test and verify the operation of the two functional blocks of this project and show the user how each function in this project is called. Normally the user of the modules in this project will include just the frequency generator and/or the frequency counter modules in their own work.

Serial Interface

The "FreqGenCtrApp.ino" module contains code and defines that allow it to use the USB virtual com port as it's command interface or it can optionally use the Pro Micro's UART serial interface for the same interface. Defining the "COMM" #define to "Serial" will use the USB interface, while defining it as "Serial1" will use the UART interface. The module also contains a simple printf interface that allows the use of the compiler's printf functions.

Serial Commands

The commands that the main module interprets are mostly 1 or 2 character commands that are terminated with a <CR> (Enter) to invoke them. For setting type commands, the command letter can optionally be followed by an equal sign, the value and a <CR>. For query commands that return a value, just the command letter(s) followed by a <CR> is entered. They then call functions in the generator or counter module and return the results of these functions. The commands are:

Read	Set	Description
G	G=<freq>	Get or set the frequency of the frequency generator. <freq> is a number from <0 (return current frequency) to 0 (generator off) to 20000000 (20 MHz) and is an integer value. The command returns the frequency that the generator is set to.

<table>
<tr><td>T</td><td>T[0..9]</td><td>Set the frequency counter mode and gate time / averaging.
[0..9] is a one character number from zero to nine where
<table><tr><th>Value</th><th>Function</th></tr><tr><td>0</td><td>Frequency counter off (All frequency counter interrupts disabled, I/O pins set to input mode).</td></tr><tr><td>1</td><td>Counter mode. Gate time of 1 Second.</td></tr><tr><td>2</td><td>Counter mode. Gate time of 10 mS.</td></tr><tr><td>3</td><td>Counter mode. Gate time of 100 mS.</td></tr><tr><td>4</td><td>Counter mode. Gate time of 10 Seconds.</td></tr><tr><td>5</td><td>Counter mode. Gate time of 100 Seconds.</td></tr><tr><td>6</td><td>Counter mode. Gate time is from external input on Arduino Digital pin 9.</td></tr><tr><td>7</td><td>Period measure mode. No averaging.</td></tr><tr><td>8</td><td>Period measure mode. Average 10 input transitions.</td></tr><tr><td>9</td><td>Period measure mode. Average 100 input transitions.</td></tr></table>The command returns the current new mode value or -1 if the command fails.</td></tr>
<tr><td>F</td><td></td><td>Read the frequency counter. – Do not wait for a new reading.
The command returns the last frequency read.</td></tr>
<tr><td>F1</td><td></td><td>Read the frequency counter. – Wait for a new reading.
The command returns the frequency read after a new value is recorded.</td></tr>
<tr><td>FS</td><td></td><td>See if a new frequency counter value is ready.
The command returns a 1 if a new value is available or 0 if not.</td></tr>
<tr><td>R</td><td></td><td>Turn on/off automatic reporting of the frequency read by the frequency counter module. Each time ‘R’ is entered this state is toggled. When on, each time a new frequency count is obtained it will be sent to the serial port.</td></tr>
<tr><td>?</td><td></td><td>The command returns a help screen showing the commands.</td></tr>
</table>

Stand Alone Device

Because it is believed that a number of users will want a simple self contained frequency generator and/or a frequency counter without writing it themselves, the main module also includes a simple controller that uses 4 push button switches and a 2x16 LCD display to implement both of these functions. Two of these pushbuttons control the frequency generator while the other two buttons are used to set the frequency counter gate mode. This self contained stand alone device can be used instead of or in addition to the serial interface that was detailed earlier.

The generator setting control will change the output frequency to one of 18 different settings from 10Hz to 4MHz in a 1, 2, 5 sequence for all the decades. The frequency counter selector will select any of the 9 modes and gate times that the frequency counter is capable of being set to. This includes gate times of 10mS, 100mS, 1, 10 and 100 seconds and an external gate mode, in addition to the 3 period measurement modes with averaging of 1, 10 or 100 averages. All that is needed to implement this stand alone functionality is an LCD display and four push button switches. The LCD display uses the standard Arduino “LiquidCrystal” library. The pushbutton interface is implemented as simple debounced digital inputs. The LCD display shows the frequency counter count on the first line and

either the frequency counter mode or the generator frequency on the second line, depending on which was changed last.

The main Arduino module of this project (FreqGenCtrApp.ino) has a number of defines that control how the module is compiled and hence controls the functionality that the program will perform. In addition to the defines that control the inclusion of the frequency generator and frequency counter modules, there are two defines that enable the stand-alone mode code, the serial interface code and whether there is an LCD display or not. The define 'FREEIF' controls whether the stand alone button interface is included. The define 'COMIF' controls whether the serial interface is available. Finally, the define 'HASLCD' should be defined to include the LCD display functionality. The LCD display can be used with the serial interface even if the stand alone interface is not included.

The schematic shown below shows the connection of the LCD display and the four pushbutton switches required to use the stand alone mode or to add an LCD display to the design.

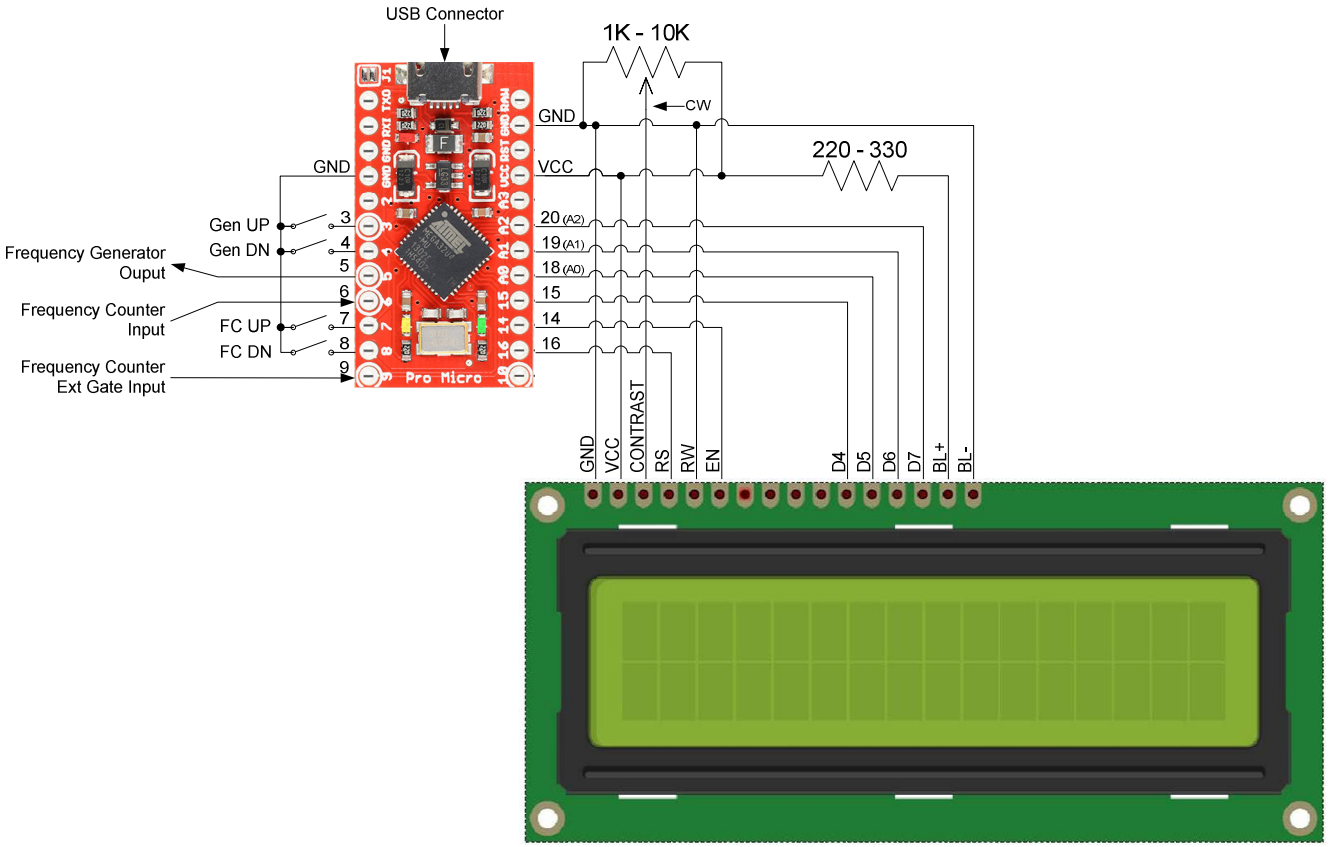

Figure 5 Stand Alone Version Schematic

Most of the parts shown are available at Digikey or Mouser. The Pro-Micro module is available from Sparkfun (part DEV-12640), or an equivalent clone can be obtained for significantly less on eBay. The LCD display shown is a Hantronix HDM16216H-5-S00S (no backlight) or HDM16216L-5-L30S (LED backlight), but other displays are also usable as long as they work with the Arduino LCDDisplay library. The potentiometer and the resistor can be any common parts desired/available. The four switches should be any momentary contact pushbutton switches that are suitable for the mechanical design of the project.

Using this additional interface allows the frequency generator and frequency counter modules to be used as a stand-alone piece of test equipment.

System Notes

Powering the Pro Micro Module

Throughout this document, no mention is made of how to power the Pro-Micro Module and possibly the LCD display. It is assumed it will be powered by connecting a USB cable to the USB connection on the Pro Micro module. The USB connection provides 5V to the module and any external circuitry required as long as the power required by any external circuitry is less than about 100mA. The other end of the USB cable can be connected to either a computer for power and access to the virtual com port and programming, or a simple 5V wall supply if using the system as a stand alone device. The Pro Micro module can also be powered via the "RAW" input with a 7-16V DC supply or via the "VCC" pin with a regulated 5V DC supply, only if the USB connection is not used.

Compiling and file Information

This project and the libraries were developed using Arduino IDE Version 1.8.5. Newer versions, including the latest (Vers. 1.8.15) should also be able to be used without issue. The settings in "Tools" of the IDE should be:

Board: Arduino Leonardo,
Port: COMxx (as it is assigned by your system)
Programmer: ArduinoISP.

The Frequency Generator and Frequency counter modules were initially written and debugged in a single project directory, "FreqGenCtrApp" that contained the following files:

Filename	**Description**	**Part of Library**
FreqGenCtrApp.ino	The main application file	
FrequencyGenerator.h	Header file for the Frequency Generator	FrequencyGenerator
FrequencyGenerator.cpp	Code file for the Frequency Generator	FrequencyGenerator
FrequencyCounter.h	Header file for the Frequency Counter	FrequencyCounter
FrequencyCounter.cpp	The code file for the Frequency Counter	FrequencyCounter
PCInterrupt.h	Header file for external gate input for Freq Counter	FrequencyCounter
PCInterrupt.cpp	Code file for external gate input for Freq Counter	FrequencyCounter
systimer.h	Header file for moving system timer to timer 1 or 3	FrequencyCounter
systimer.cpp	Code file for moving system timer to timer 1 or 3	FrequencyCounter

Note: PCInterrupt is only used if the external gate mode is enabled in the FrequencyCounter.cpp file.
Systimer is used for the frequency counter

As mentioned all the above files can be in a single directory and compiled that way, or they can be part of a frequency generator library or frequency counter library. If using the libraries, there is an example folder in each one that contains a main application file for either just the frequency generator or frequency counter and a folder that contains a combined application.

Filename	**Description**
FreqGenCtrApp.ino	The main application file for both the frequency generator and frequency counter.

FreqGenApp.ino	The main application file for just the frequency generator.
FreqCtrApp.ino	The main application file for just the frequency counter.

(The application files are all basically the same except for commenting out '#defines' for what functionality is not used.)

The FrequencyGenerator library file is on Github at https://github.com/Rick-G1/FrequencyGenerator
The FrequencyCounter library file is on Github at https://github.com/Rick-G1/FrequencyCounter.

Frequency Generator / Counter and Timebase Accuracy

The timebase used for the generator and frequency counter is the microprocessor's clock, which is a crystal oscillator with its inherent accuracy and stability, but since it is not calibrated, it will normally vary from its nominal frequency of 16MHz by a few Hertz. This is typically within about 0.1 to 0.2%, but could be more than that depending on the exact Pro Micro module used. Since the crystal frequency can vary some, the output from the frequency generator and values read by the frequency counter will vary by the same percentage.

A check of several modules that were on hand during the writing of this document revealed that most of them fell within 0.1% and only a couple of them were within the 0.2% accuracy value. If this level of accuracy is sufficient for the intended application then no further effort is needed to use this frequency generator or counter functionality.

If higher accuracy is desired, it is possible to modify the Arduino module to either include a trimmer capacitor to tune the modules crystal frequency to exactly 16MHz or inject an external high accuracy 16MHz clock signal into the module to improve the accuracy of the frequency generated or the frequency measurements the controller makes. Obviously a calibrated frequency counter will be needed to calibrate the Arduino Pro Micro module to the exact frequency.

When measuring the frequency of any crystal oscillator, do not measure the crystal frequency directly on either pin of the crystal itself. Just the added capacitance of the measuring device is enough to change the frequency of the crystal oscillator circuit significantly. Instead, set up the controller to output a known frequency clock signal on one of its output pins and measure that signal instead. Since part of this project contains frequency generator functionality, this output signal is the perfect signal to use to measure and calibrate the accuracy of the crystal timebase of the controller. The frequency generator function can be set to 8MHz, which is perfect to check and tune the crystal oscillator on the module.

Adding a Trimmer Capacitor

One of the methods that can be used to tune the systems timebase is to add a trimmer capacitor to the Arduino module. Then by outputting a known frequency using the frequency generator portion of the project connected to a calibrated frequency counter and adjusting this trimmer capacitor to set the output frequency, the systems timebase can be set to exactly 16MHz and the accuracy of the frequency generators output and the timebase for the counter will be zero, or as accurate as the reference equipment used to tune it.

To install a trimmer capacitor, remove the C2 capacitor from the module and install a trimmer capacitor that has a range of about 5 to 40pF. A suggested part that can be used for this is the Knowles Voltronics JR300 (5.5-30pF) or JR400 (8-40pF) (Digikey 1674-1020-1-ND or 1674-1021-1-ND). This is a small surface mount part that can be glued to the processor chip and then small wires used to connect it to GND and the pad of where C2 was removed. Because trimmer capacitors have some variability based on temperature, the stability of the oscillator is directly related to the stability of this trimmer capacitor.

If more stability is desired, an alternate and more stable method would be to replace C2 with a smaller value capacitor (perhaps 5-15pF) and then use a trimmer capacitor that has a smaller capacitance range (and hence less variability of the total capacitance) (like the JR150 or JR200) such that the total capacitance of the fixed cap and the variable capacitance can be varied to a value near 22pF. It is important to note on this part that one terminal is connected to the rotor (and the metal of the adjustment screw). This terminal should be connected to the Arduino module's GND connection and the other terminal should be connected to the pad of C2 that is not grounded (and goes to pin 17 of the microprocessor on the Pro Micro module). This will provide the most frequency stable circuit. A convenient location to connect the GND connection of the trimmer cap is the minus side of the module's capacitor C19. A picture of the suggested trimmer capacitor is shown next along with a modification detail of the Pro Micro module. The terminal on the top of the trimmer capacitor picture is the 'rotor' terminal. Once the connections are made, is best to glue the trimmer cap to top of the microprocessor chip to keep it from moving, which improves the stability of the crystal's frequency. You may also want to consider stabilizing the wiring to maintain the greatest accuracy by putting hot glue or epoxy around the wires to prevent them from being accidently damaged or moved.

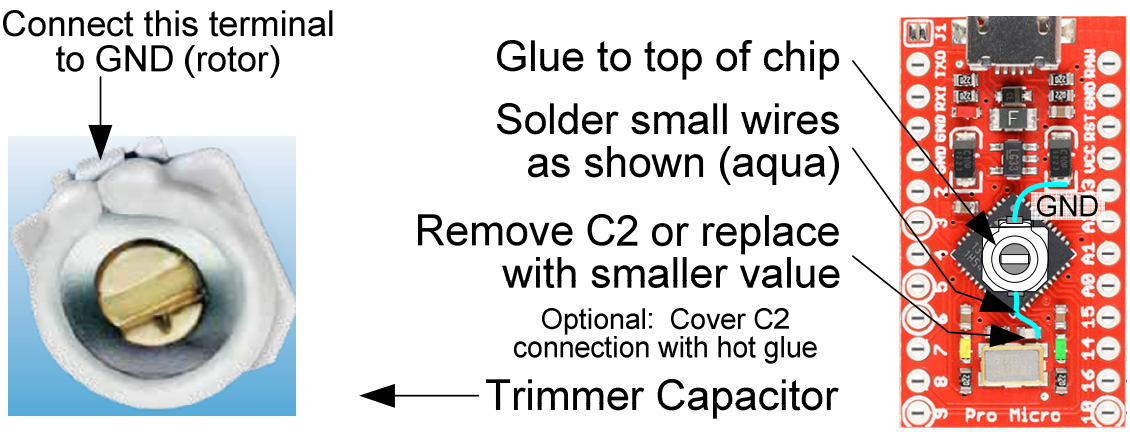

Figure 3 Trimmer Capacitor Modification

Once the modification to the module is made then tuning can be done. For the most accurate adjustment allow the module to remain powered up for at least 15 minutes (several hours is better) at the temperature that the module is expected to be used at before making the adjustment. Then enable the frequency output of the controller and set it to 8MHz. Once the output signal is set, use a calibrated frequency counter and measure this output signal (on digital pin 5)(or digital pin 10). Adjust the trimmer capacitor to exactly 8MHz as shown on the external frequency counter.

Injecting an External Clock

For even higher accuracy an external clock can be used to supply a clock (and hence a timebase) to the controller chip. Using this method the accuracy of the frequency generator and counter is tied directly to the accuracy of the external clock circuit. Various external oscillators with extremely accurate and stable outputs are available, from low cost IC's to very expensive, highly stable TCXO modules. The only requirement of these modules is that they must output 16MHz at 3.3V or 5V signal levels. The output of this external clock can then be injected into a modified Pro Micro module as shown. In this example the external timebase is injected into the "A3" pin using a resistor to prevent overpowering the

module if the external clock is powered but the module is not. A 10-47Ω 1/8W resistor can be used for this purpose. The A3 pin was selected because it is normally programmed as an input and forcing a clock signal on it will not affect the signal or cause excessive current unless the pin is mistakenly programmed as an output.

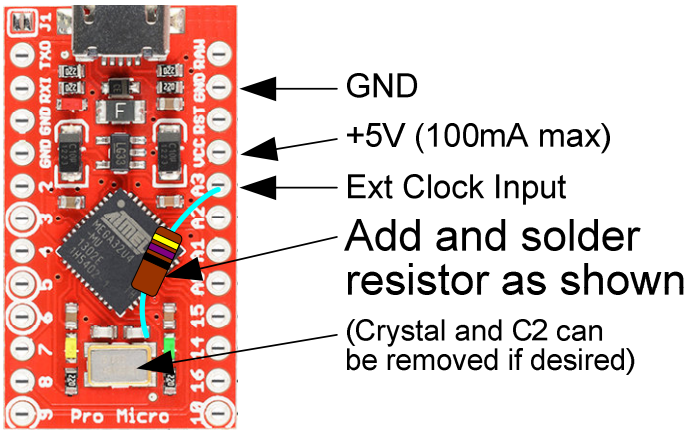

Figure 4 External Clock Modification

Obviously when using this pin the normal analog input function of the A3 pin is lost. Also shown in the detail are the locations of GND (oscillator common) and a source of 5 volts that is usable by the external oscillator to power it, if the external oscillator draws less than 100mA. The 5V supply signal is derived from the USB input power and may vary from 5V by 10-15% (sometimes it is only 4.5V or so).

When using an external clock source, the fuses on the processor can be reprogrammed to the external clock mode, but this should not be necessary because the pin that the clock is injected into is the input from the crystal, so the processor will 'see' the 16MHz from the external source and use it. You could also remove the crystal on the Pro Micro module, but this is also not required to use the external clock.

Conclusion

This project has detailed the implementation of a frequency generator and frequency counter modules using just a low cost Pro-Micro module. The frequency generator can output a square wave signal from 1Hz to about 12MHz while the frequency counter can count pulses input from 1Hz to about 8MHz in the traditional frequency counter mode and from well less than 1Hz to about 20KHz in the period measuring mode. While the main focus of this project is the modules themselves, a simple main program to test these functions with a serial port or 4 pushbutton switches and an LCD display is also provided. The modules are written so that they are self contained functional blocks and can be added to the users code with minimal effort. It is hoped that having these modules will assist the user by simply being able to include the modules, instead of having to write them from scratch.