

YAPP - Yet Another PIC Program

for Motor and two Actuators

Martin Newell

June 2005

YAPP is a program for driving motor and two actuators from a PIC chip. It is a little different from previously published programs. The main difference is that the output is up to 50 times higher frequency than most PICs currently used, but there are other differences too.

Summary

- Written in C
- For PIC12F629/675 (4 MHz) and PIC12F635/683 (8 MHz), 12F635 recommended, about 80 mg
- Adapts automatically to both positive- and negative-going input
- Adapts automatically to different transmitter channel assignments
- High frequency Pulse Frequency Modulation, PFM, output in the range 5.26 KHz to 52.6 KHz (26.3 KHz for 4 MHz parts)
- Resolution 20 positions for each of Left, Right, Up, Down and Throttle (10 for 4 MHz parts)
- Supports modes that are programmable from the transmitter at start up

- Aileron-Elevator or Elevon mode
- Linear or negative-exponential mode
- Automatic calibration to transmitter's center sticks
- Modes remembered in EEPROM and restored on next power up
- Goes through progressive standby if signal lost, until found again
- Control surfaces to 1/5 left rudder, neutral elevator
- Throttle gradually reduced to zero
- Decodes 4 channels, easy to make 4th channel chip
- Very stable output - no jitter if input pulse stream is stable
- And most important - Plays a tune on startup

The program is available in source form, as a formatted, syntax colored PDF version for easier reading, and as hex files for the PIC12F629, PIC12F635, PIC12F675 and PIC12F683 versions.

Overview

The purpose of this program is to read and decode the incoming binary pulse string from the RF front end, and to drive the aileron, elevator and motor outputs with separate binary pulse strings. The input pulse string encodes the control channels by varying the time between consecutive pulses. The output pulse strings encode values by the ratio of on to off. The main challenge is to arrange to examine

the input string often enough to be able to time the transitions sufficiently accurately, while at the same time outputting transitions to the output pins on a consistent basis to keep the required on-to-off ratios, and to do all the necessary conversion computing in between. The basic structure of the program is similar to that developed by Andy Birkett, from whose code I learned a lot, but is a complete rewrite with most details being different.

Output Coding

The output signals can be encoded in at least two ways - Pulse Width Modulation, PWM, or Pulse Frequency Modulation, PFM. There is code in here for both, but PFM is preferred due to the higher frequencies attainable. It also allows greater resolution in the available memory. There is a body of opinion that higher frequencies are better for both efficiency reasons and for longevity of the motor, particularly coreless ones like the pager motors in common use today. I have convinced myself through a number of experiments that the efficiency claim for pager motors is bogus. But there may be something to the longevity claim, and since we typically drive these motors way beyond their design parameters, anything that will aid longevity is a good thing.

As for performance, this implementation generates PFM output between 5.26 KHz at the ends of the range up to 52.6 KHz in the middle (half that for 4 MHz parts), compared with the constant roughly 1.25 KHz PWM output that seems to be standard with Andy's code and many commercially available receivers. Since we drive the motors at the middle of the range a lot of the time, the higher frequencies may be relevant. Another issue is that the output on some receivers can be a little jittery. While this seems to matter little in flight, I wanted to clean that up.

Loss of Signal

A further characteristic of the above-mentioned receivers is that some of them don't behave well on loss of signal. Try switching off your transmitter and see what happens to the receiver. I believe the code here goes a long way to overcoming these problems, though it is still not perfect.

Additional Functions

The program implements additional functions under Startup Configuration and Progressive Standby, see below. It detects different transmitter configurations. It expects aileron, (which is typically used to control the rudder), elevator and rudder to be on increasing channels, with throttle on any of the four channels. The code normalizes any of these configurations to Aileron, Elevator, Throttle, Rudder in that order. Channel 4 can be used to make a 4th channel chip, see `#define RUDDER_ELEVATOR` in the code.

Some versions of PIC chips, including those used here, can support interrupts which can be triggered by an internal clock to time the output, and by input pin transitions. This would seem ideal for consistently serving the timing needs, and indeed, can be used for that purpose. However, I have been unable to achieve the same output frequency using interrupts as I can with a polling system. This is due to the high overhead of getting into and out of the interrupt routine. Therefore this program does not use interrupts, but relies on polling and careful coding in the two areas where timing is critical. The output streams are very stable, though for some frequencies they may appear not to be, This is because the PFM output is on a

cycle that repeats every 190 uS, but the pulses within that cycle are not absolutely uniform, by design. It is therefore necessary to synchronize an oscilloscope to trigger on multiples of 190 uS to see the stable pattern.

Startup Configuration

There are two features in addition to the fundamental operation of the program - Startup Configuration and Progressive Standby. Startup Configuration provides for setting parameters in the program to change the behavior of the receiver. The parameters are Elevon mode, so-called Negative Exponential mode, and Center Calibration. Startup Configuration is entered by starting the receiver with the transmitter on and the throttle stick above zero, usually about $\frac{3}{4}$ of max. The motor will not come on, but the control surfaces will operate normally. This also provides a measure of safety in case the throttle was accidentally left above minimum position. While in this state, parameters can be set by moving the control sticks to extremes of their ranges and holding them there for about half a second. These modes are set in the procedure StartupConfig(). Modes take effect immediately. The following modes can be set:

- Elevon mode turned On - Right-Top (Full right rudder, full down elevator). The left elevon actuator should be connected to the aileron output, with left aileron corresponding to up elevon. The right elevon actuator should be connected to the elevator output, with up elevator corresponding to up elevon. Full elevon deflection is achieved for full deflection of either aileron or elevator individually.
- Elevon mode turned Off, i.e. Aileron-Elevator mode - Left-Top (Full left aileron, full down elevator).

- Exponential mode turned On - Right-Bottom (Full right aileron, full up elevator). By default the system is in linear mode - equal movements of the aileron and elevator control sticks generate equal changes in the effective voltage output to the actuators. In Exponential mode, the control stick has to be moved further to get one step in the output when the stick is near the center, but less far to get one step when the stick is near full throw. Exponential mode applies only to the aileron, elevator and rudder channels, not to the motor.
- Exponential mode turned Off, i.e. Linear mode On - Left-Bottom (Full left aileron, full up elevator).
- Center Calibration mode - Move Throttle stick to max with other controls centered. This calibrates the Rx to neutral at the current Aileron, Elevator and Rudder stick positions. This is not strictly a mode since it takes effect immediately and is not repeated.
- Normal running mode - Move throttle stick to minimum. All modes are locked in their current state, and the motor is enabled. After this, moving the control sticks to their extremes will have no effect on modes.

Elevon and Exponential modes are remembered in EEPROM in the chip and are re-instated to their previous values next time the receiver is started. Normal starting with the throttle at minimum will simply re-instate the old values. Starting with the throttle at maximum will

immediately calibrate to center stick positions, after which moving the throttle to minimum will make the system ready for use.

The reasoning behind Center Calibration is a belief that on small actuator-controlled planes you shouldn't use trim controls to get the plane trimmed out (except for initial trimming), for two reasons - it consumes power, and it is hard to reproduce the correct trims with a non-computer-based transmitter. Trimming should be done by adjusting the neutral positions of the control surfaces, or with trim tabs, or by moving the CG. Due to slight timing differences between chips and the transmitter, it is typically necessary to use the transmitter trims to zero out the control surfaces. This is a nuisance when starting up with a different receiver, but is at least possible with low frequency PIC chips since you can hear the output and can adjust trims until nothing is heard. With the high frequency output used here you cannot hear when the control surfaces are at neutral since the frequencies are above human (or at least, my) hearing. The Center Calibration does it for you. There is a single `#define` that can remove this feature from the code.

Progressive Standby

Progressive Standby is activated when synch is lost. This can happen due to loss of signal or too much interference. While attempting to re-synchronize, the system progressively shuts down the controls.

1. After 1.5 seconds - Aileron 1/5 left, Elevator neutral
2. After another 1.5 seconds - Throttle reduced to 75% of its last value

3. Repeat step 2 until Throttle zero

This behavior is controlled by the StandBy() procedure. If synch is re-established at any time during, or after, this standby process, normal operation is resumed with the same modes in effect.

I have tested the Progressive Standby by turning off the transmitter while flying Bipe4 with a homemade Leichty receiver. The plane landed safely, and one of the people present commented that it was better than the landings I usually make (thanks Brian).

Overview of the Code

The heart of the system is the output procedure OC() (renamed from OutputCycle()). The main purpose of this procedure is to set the GPIO outputs to the appropriate values each time it is called. It relies on being called no less frequently than every 38 instruction cycles, which allows up to 8 cycles between calls. OC() begins with a piece of code that synchronizes it so that the body of the procedure will always run exactly 38 instruction cycles after the previous time, provided the limitation of no more than 8 cycles outside the procedure is maintained. This provides a solid time base for the output pulses. It also provides a clock with a period of 38 cycles, called a tick, which is used by the code that times the input pulses. It does this by simply incrementing a global variable, OutputTimer, once for every entry.

On the 4 MHz parts an instruction cycle is 1 uS (microsecond), and 0.5 uS on the 8 MHz parts. The time between input pulses varies from about 1100 uS minimum, to about 1500uS in the center, and

to about 1900 uS at maximum. At 4 MHz this corresponds to 28 ticks, 39 ticks and 50 ticks, and at 8 MHz to 56 ticks, 78 ticks and 100 ticks.

OC() outputs values to GPIO that it gets from a "control array" of GPIO words. It simply cycles around this array, outputting the values it finds. To achieve a higher output frequency it actually outputs two consecutive values from the array, separated by 19 instruction cycles, each time it is entered.

In the PWM version, to update to a new pattern of outputs double buffering was used. However, in the PFM version with output resolution of 20 there is not enough memory, so the buffer is overwritten in place with new data while OC() is reading out of it. This is safe to do with PFM since the content of the buffer is still valid even when partially updated.

The front end is responsible for watching and decoding the input stream, then building the control array for OC() to output, all the time being sure to call OC() no more than 8 instruction cycles after the previous call returns. It is necessary to examine the assembler output to ensure that this requirement is always met, though it can be checked by putting a break point in the OC() procedure at the position noted, when running in the simulator.

The input stream can have positive-going or negative-going pulses, depending on the RF section of the receiver. The synchronization with the input stream figures out which is in effect. The front end re-synchronizes every frame before decoding the input pulses. As long as the front end is able to stay in synch with the input stream it

stays in a loop of decoding a frame of input, preparing a new control array and giving that to OC(). If it loses synch, perhaps because of a temporary loss of signal, it starts the Standby process while attempting to re-synchronize. It requires several valid consecutive input frames to be decoded before stopping Standby and resuming control of the outputs as directed by the transmitter.

Other than the need to keep calling OC(), there are two places in the code where timing is critical. These are the places where the leading edges of the input timing pulses are detected. One place is in the procedure SynchUp(), which expects to be called in the inter-frame gap and is looking for the first leading edge of a new frame. It has a code section repeated three times (six times for 8 MHz parts). This allows it to check the input signal for every call to OC(), so that the timing of the input signal will be accurate to within 38 instruction cycles.

Development Environment

The code was developed using the Microchip® MPLab® IDE www.microchip.com, and is best viewed and printed using this system with fixed pitch font and 4 character tabs. WinPic, <http://www.qsl.net/dl4yhf/winpicpr.html>, was used for programming the chips. The program uses 1005 of 1024 available code words, and 63 of 64 available bytes of RAM. You pay for all the memory, so might as well use it all. However, I now find that the 12F683 has double the code words and RAM available, so lots of room for more bells and whistles.

Calibration

It is important that the chip in which this software is loaded is calibrated. Some, like the 12F629 and 12F675, have been factory calibrated and store the calibration value in the last word of program memory. Some, like the 12F635 and 12F683 have no such calibration value. I have found those chips can be significantly out of calibration if their OSCTUNE registers are set to zero, and I have found small errors even in the others. The documentation warns against relying on the accuracy of the internal oscillator since it is not crystal controlled. So, to calibrate the chips I have written a calibration program that can be used either with an oscilloscope or a simple LED or analog voltmeter and stop watch. The program is available in [source](#) form and as hex files for the [12F629](#), [12F635](#), [12F675](#) and [12F683](#) chips. Documentation is in the source file.

Implementations

The code has been flight tested in two Leichty receivers, a Micro and a homemade Mini combo, and in a Dynamics Unlimited RFFS-100 receiver. The Leichty receivers now appear to operate virtually glitch-free in an environment where glitches used to be a problem.

Code is included to program a chip that can replace the standard one in an RFFS-100 receiver from Dynamics Unlimited. I do not recommend doing this as it almost certainly invalidates any warranty, and I disclaim any responsibility for any consequence arising from anyone doing this. The code is included merely for information purposes, and to show what I chose to do. There is one anomaly with this modification that I am aware of. The modified receiver works well and drives a KP-00 motor with a linear response to throttle. However, when driving a much smaller 4.5 ohm 6mm pager motor, the throttle has a flat spot at mid-range. I do not observe this with the Leichty receivers. This is no fault of the RFFS-100 receiver - it is possibly a result of my driving the motor

output at a much higher frequency than that for which it was designed. Other than this the receiver functions very well and also shows good glitch resistance.

Disclaimer

This software is provided "AS IS" and any express or implied warranties, including, but not limited to, the implied warranties of merchantability, fitness for a particular purpose, or non-infringement, are disclaimed. In no event shall the author be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement for the substitute of goods or services, loss of use, data, or profits or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.