Music Synthesizing Device using 8-bit Microcontroller

Shona Curtis-Walcott

Word Count: 3074

***Abstract*—This report details a Microcontroller-based Synthesizer with a full-functioning real-time play mode and optional recording and playback modes. The system made use of timers and interrupts to provide digitally-controlled streams of binary numbers to a signal generator at steady sample rates. The signal generator comprised of a DAC/ Op-Amp configuration and converted the binary numbers to an analog form which was then output through a loudspeaker. The system was built and tested: musical notes were correctly produced bearing sawtooth waveforms and frequencies from the fourth octave ranging from $293.4Hz$ to $492.2Hz$. The temporal uncertainty induced in the calling of each command was calculated to be $2.05ms$. It was also found that in playback mode, the delay between each note slightly deviated from the predicted uncertainty. Possible extensions and improvements to the system are discussed.**

I. INTRODUCTION

Since their rise in popularity in the 1960s and early 1970s, synthesizers have featured as a dominant instrument in numerous genres and are responsible for many of the worlds most famous songs and movie soundtracks today [1]. Synthesizers are electrical musical instruments which produce audio signals, whether melodic or percussive, through the generation of wave-forms with variable frequency and amplitude [2]. A synthesizer contains many parts; most notably a tone generator and a sequencer. The role of a music sequencer is to deal with the note and timing information of a piece of music, normally using Musical Instrument Digital Interface (MIDI) protocol. The tone generator is responsible for generating waveforms from the conversion of a digital signal to analogue form. The output waveforms are then played through a loudspeaker.

Discussed in this report is a music synthesizer based on the PIC18F87K22 Microprocessor from Microchip [3], built using a signal generator with underlying routines written in Assembler. For simplicity, this project obeyed a MIDI-like protocol: each sound called on the controller was represented as a digital command which was sent to the sequencer and then on the tone generator. To produce the sound, a signal generator was built using a Digital to Analogue Converter (DAC) and an Op-Amp connected to the in-built speaker of the PROTO-Board, aided with numerous control buttons. The project as a whole encompassed an extensive number of algorithms with minimal hardware components. This approach increases the systems reliability and potential for further extension.

In this project musical notes within the fourth octave pitch class were generated. Sawtooth waveforms were produced at the appropriate frequencies using a combination of timers and interrupts. The sawtooth form was settled on due to its relatively accurate representation of sound waves created by ensemble instruments, stemming from the force a string places on the bridge of the bowed instruments [4].

II. HIGH LEVEL DESIGN

The Music Synthesizer consisted of three main modes: Real Time play mode, Recording mode, and Playback mode. The system was placed in Real Time mode by default: upon the pressing of each note button the corresponding sound wave was produced through the loudspeaker. Real Time mode provided the basis for the sound generation of all other modes. Storage in RAM was reserved where required, the timers and DAC were set up, and PORTJ and PORTE were set as inputs. The Top-Down diagram of the system is shown in Fig.1.

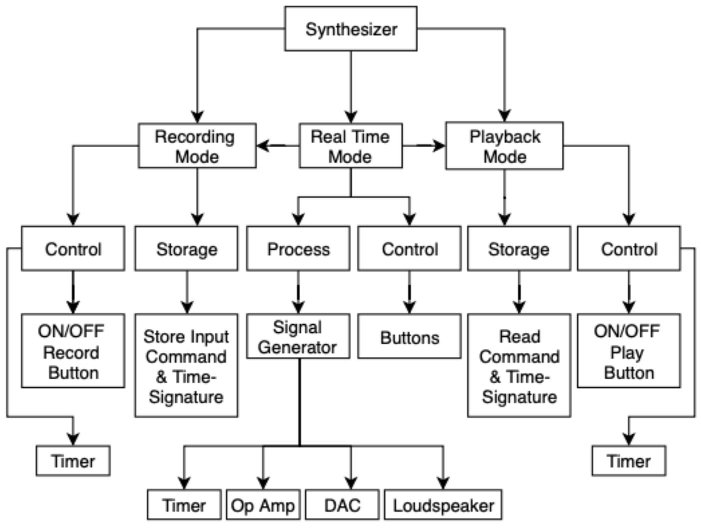

Fig. 1: Top-Down Diagram of the Microcontroller-based Synthesizer. The 3 main system modes -Recording, Real Time, and Playback- are displayed, along with the prominent hardware and software components contained within them.

Upon pressing buttons on the development board, musical notes play in real time through the in-built speaker of the PROTO-board. The option to record and play-back a musical sequence was added, with a further in-built button dedicated to each of these modes. The bar length of the output music was flexible, set simply as the length of time Recording mode was switched on for (of maximum value 134s).

It was decided that to avoid the complications of enabling polyphonous tunes to be played whilst developing the project's basic structure, the system would be designed purely for monophonous musical sequences. This was considered a viable extension to the project however, and is discussed in section V. of this report.

III. Software and Hardware Design

A. Hardware Design

The `EasyPIC PRO v7` development board containing the PIC18F87K22 Microprocessor was connected to the signal generator circuit on the `PROTO-BOARD PB-503` through PORTD. The input ports of the development board were set as PORTJ and PORTE: the RE7 button on PORTE turned ON/OFF Recording mode, RJ0 on PORTJ turned ON/OFF Playback mode, and the remaining PORTJ keys (RJ1-RJ7) corresponded to the various musical notes available.

The signal generator circuit comprised of the `TLC7524CN` DAC and the `741CN` Op-Amp, connected as shown in Fig.2.

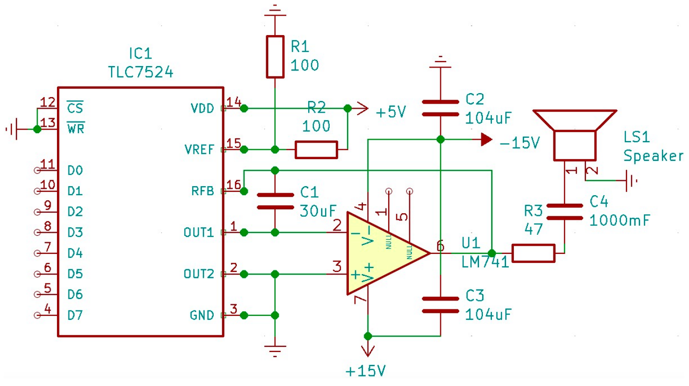

Fig. 2: A schematic of the Signal Generator circuit. Main features include the `TLC7524CN` DAC (labelled `IC1`), the `741CN` Op-Amp (in yellow), the PROTO-board in-built Loudspeaker (labelled `LS1`) and various capacitors and resistors to smoothen any internal oscillations produced in the Op-Amp.

The circuit was designed for the generation of sawtooth waveforms. The `IC1` takes an 8 bit binary counter with a rollover as input and converts it to an analogue equivalent in the form of a current. This current is then fed into the Op-Amp for amplification and inversion. The Op-Amp was set up in closed-loop mode, via capacitor `C1`, to reduce the gain of the amplifier at high frequencies, helping maintain amplifier stability. During the test run it was discovered that the Op-Amp suffered from parasitic and feedback internal oscillations which were affecting the form of the output signal. To combat this, the capacitors `C2`, `C3`, and `C4` were added to smoothen the output signal by moderating the feedback oscillations. Care was taken when choosing their values to ensure the resultant reduction in gain did not diminish the strength of the loudspeaker output.

This project made use of two timers: Timer2 in 8-bit mode and Timer0 in 16-bit mode, where the mode of the timers was chosen to reflect the required resolution and precision of their tasks. Timer2 was used for the signal generation, whereas Timer0 was used in the Recording and Playback routines. Timers are pieces of hardware built into the Microprocessor which run based on the internal clock and can be programmed using special registers. The value of the Period register `PR2` is defined as the number up to which the Timer2 register `TM02` increments before rolling over to zero. In this project `PR2` was set with different values depending on the musical note to be played. Timer2 was initialised with prescaler and postscaler values of 1:1.

It was decided that for Timer0, even if the Period register was set at its maximum possible value, the timer would still run extremely fast and therefore limit the possible bar length of output music sequences. To tackle this, a counter `tmr0_cnt` was implemented to lift the restrictions on the bar length. Additionally, the prescaler of Timer0 was set so that there were 16 internal clock cycles per timer increment to extend the time delay.

Naturally, the program running on a microprocessor runs Assembly instructions sequentially. A software interrupt is an instruction asynchronous with the current software execution. When an interrupt is raised to the CPU, the CPU stores the location of the current instruction and stops running. The interrupt service routine attends to the interrupt instructions then returns to the main program using the location stored by the CPU, and the program continues running from where it left off [5]. There exists a hierarchy of interrupts: high-priority interrupts can interrupt low-priority interrupts, but not vice versa. Interrupts were implemented in this software to aid the production of the sawtooth waveforms. A high priority interrupt was called every time Timer0 rolled over, or Timer2's `TMR0` register reached the value of the PR2 register. Special care was taken to ensure that no variables were overwritten when executing interrupts, and that the `TMROIF` bit in register `INTCON` was cleared during each interrupt to prevent the interrupt from being continuously triggered.

B. Software Design

A flowchart of the main program loop is displayed in Fig.3. The program ran through a main loop which continuously checked the mode and state of the system. Only if a new 'action'- be it a change of mode or change of button state- was detected would the algorithm proceed to further subroutines.

1) Checking for Change of System State : Upon setting up the system, the routine `state_init` was called. This function accessed the current binary state of PORTJ -a binary number with a bit corresponding to each note button, where 1 indicates the button is ON and 0 indicates the button is OFF- and moved it into the special file register `state`. This routine was called during each subroutine to ensure the number stored in `state` was continuously updated.

In order to quickly determine whether a change had been made to system the routine, the high-level comparison routine `state_check` was called from the main loop of the program. `state_check` operated using a bitwise comparison of the current state of the input PORTJ, saved in register `PORTJ`, with the most recently saved state in `state`.

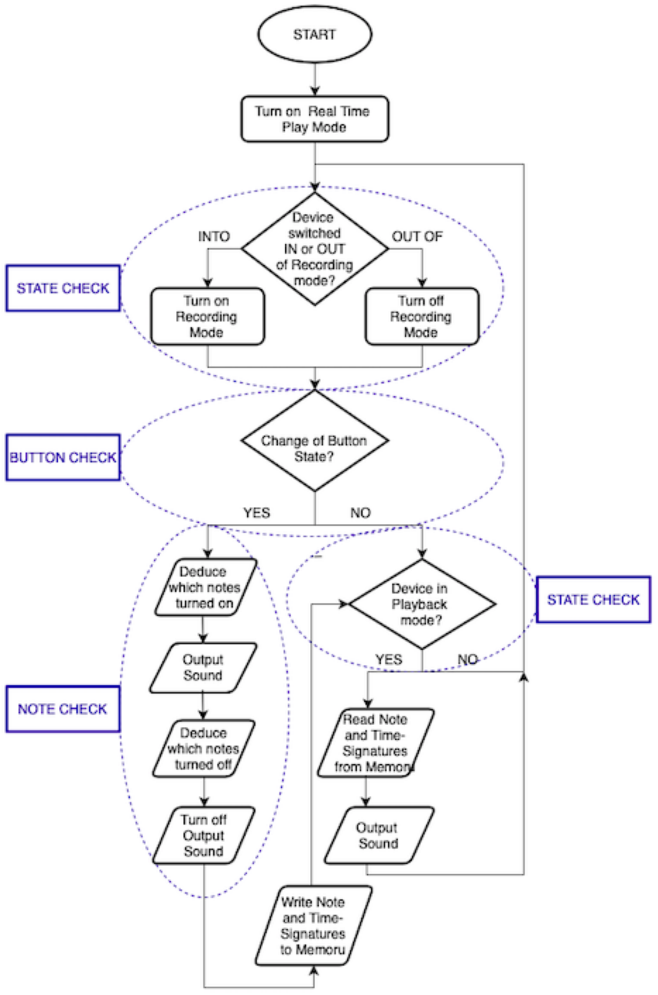
START
Turn on Real Time Play Mode
Device switched IN or OUT of Recording mode?
INTO
OUT OF
STATE CHECK
Turn on Recording Mode
Turn off Recording Mode
BUTTON CHECK
Change of Button State?
YES
NO
Deduce which notes turned on
Output Sound
Deduce which notes turned off
Turn off Output Sound
NOTE CHECK
Device in Playback mode?
STATE CHECK
YES
NO
Read Note and Time-Signatures from Memory
Output Sound
Write Note and Time-Signatures to Memory

Fig. 3: Flowchart summarising the basic logic behind the main loop of the Assembly program. The flowchart is split up into the main checks of the system: State, Button, and Note check.

If the binary values were equal it indicated no actions had been taken since the last loop and immediately returned to the main loop. If the values were not equal however, it was directed into the note-checking branch `neq` to investigate which buttons had been pressed.

2) Checking for Change of Button State : The `neq` branch was split in two: firstly, it was determined which notes had switched on in a rising edge transition (i.e. the corresponding bit value had changed $0 \rightarrow 1$), and then which had switched off in a falling edge transition (i.e. the bit value had changed from $1 \rightarrow 0$). To achieve this, logic operations were employed. To deduce which bits turned on, the following command was implemented: `NOT(state) AND PORTJ` $= RE$. The output of this command, RE, contained the bitwise mapping of notes being turned on.

A similar process was then carried out to identify which notes turned off: `NOT(PORTJ) AND state` $= FE$. These bitwise mappings then directed the algorithm to the appropriate subroutine for each note: either to turn on the DAC and output the signal through the loudspeaker, or to turn off the DAC.

3) Creation of Sawtooth Waves : Once the program detected a specific note had been switched on, it proceeded to set up the DAC for note-generation. To create periodically-repeating sawtooth waves, an 8-bit counter triggered by the roll-over of Timer2 was supplied to the DAC via the data bus. The counter was ensured to have the correct period of oscillation for the specific note called by altering the choice of the `PR2` register. Once the note was called, Timer2 was initialised and switched on. The value of the Timer2 register `TMR2` incremented on each clock cycle and was compared to that of `PR2`. Once the values matched, an interrupt was generated and the `TMR2` value rolled over to zero. Each interrupt called caused the 8-bit value on PORTD to be incremented by one and therefore 256 timer cycles were required for a one complete oscillation of the output-wave. A more detailed description of the Real Time play mode code structure is given in Fig.4.

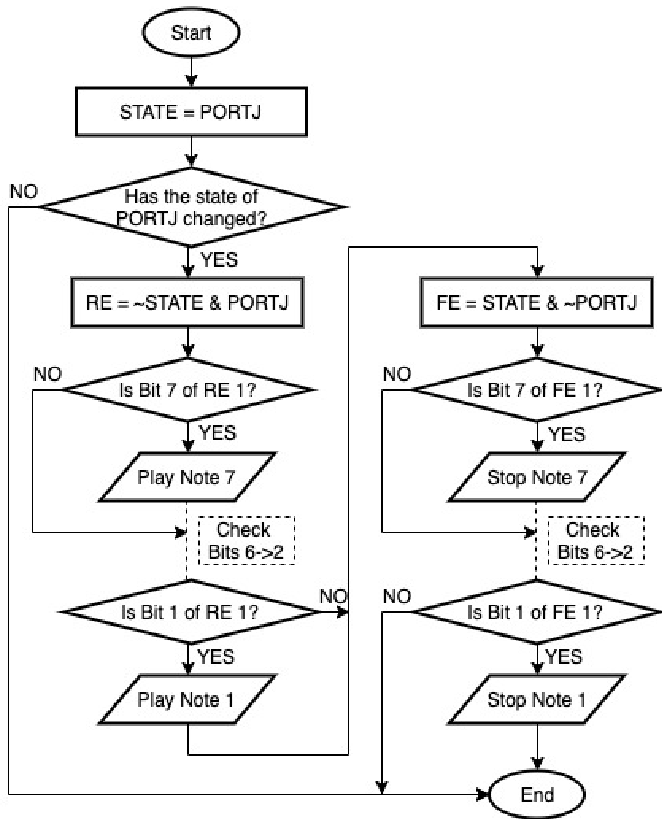

Fig. 4: Flowchart detailing the general decision structure of the Real Time play mode algorithm, where RE and FE are the outputs of the binary logic commands discussed in section (2) of the software design section.

4) Note and Time-Signature Storage in Recording mode : It was decided that per action, three bytes of information containing the note, command, counter value and time signature would be written to a data array. The array `seq_array` was created with 120 bytes of memory reserved for a single music sequence. This amounted to a maximum number of 20 notes played per sequence.

The routine `write_action` was written and called to after every button state change. The routine performed an initial check of whether the system was in Recording mode by doing a bit-comparison with the state of the RD0 pin: if it was low the system was not in recording mode and hence the program returned from the routine without any information being stored. If the pin was high it indicated the system was in recording mode and information collected per action was written to `seq_array` through indirect addressing, which used the special function register `FSR0` as a memory pointer to the data memory location. The precise information stored per action included one byte for the note code (of value 1-7 in reference to the RJ1-RJ7 note buttons) and note command (f for note ON, 0 for note OFF). The file register which stored the current value of Timer0 was `TMR0`, and upon every action the highest byte of the time signature `TMR0` was also accessed and stored in the array. The final byte stored per action was the value of the `tmr0_cnt` register at the time the command was called. This method of storing three bytes of information per command ensured that the details of every action made during the whole live sequence had been stored and was therefore available for reading once system was placed in playback mode.

5) Playback Mode : A bit-comparison was carried out with the state of the Play button (RJ0) as part of the system's high-level state check. As soon as the program recognised the system had been placed in playback mode, Timer0 was initialised and indirect addressing was used to step through the stored data array to unpack the `tmr0_count` value and Time Signature of the first recorded note into temporary registers. The algorithm firstly waited until the `tmr0_cnt` value matched that of the read-in counter byte, then waited until the Time Signature matched the value of the highest byte of Timer0. The choice was made to only compare the highest byte of time information so as to introduce a time range in which the values match. This was deemed necessary to account for the processing speed of the routine; the system did not run fast enough to check every single value of `TMR0` between 0 and 65535 (the number of numbers we can store in two bytes) accurately. Once both pieces of stored information matched with the live program, the appropriate note algorithm was called and a signal was generated through the DAC. This whole process was then repeated sequentially within a loop so that the full recorded sequence could be played back.

IV. Performance

Overall, the approach taken to only proceed to the low-level comparison routines once a high-level comparison of state was run greatly increased the efficiency of the central algorithm. An improvement to the code would have been to change `goto` commands to `branch` commands, as `goto` commands require an extra line of code to execute and therefore take slightly longer.

The output sawtooth signals were visualised in the time-domain using an oscilloscope. Fig.5 presents the 440.2Hz sawtooth signal which produced the central-A musical note.

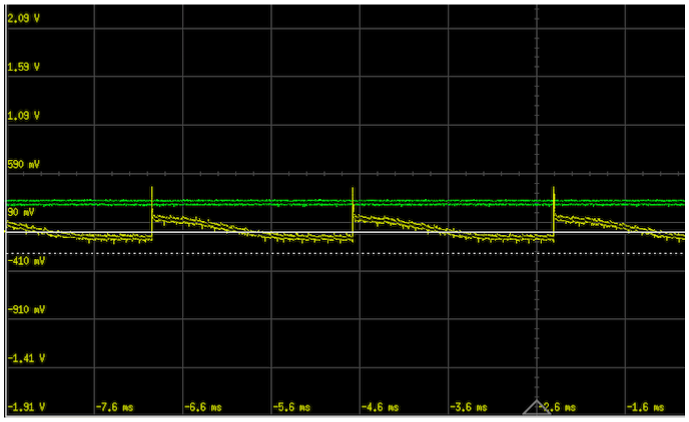

Fig. 5: Output sound wave of frequency 440.2Hz, representing musical note A from the fourth octave

The slight wavering of the signal can be traced back to the remaining affects of the Op-Amp internal oscillations. The overshoots arose due to the implementation of extra resistors which suppressed the signal amplitude.

Table.1 presents the final frequency of each output musical note compared to the theoretical frequency expected.

Note	B	A	G#	F#	F	E	D
Theory(Hz)	493.9	440	415.3	370.0	349.2	329.6	293.7
Output(Hz)	492.2	440.2	413.9	369.8	349.2	330.7	293.4

TABLE I: A comparison of musical note frequencies: Theoretical expectation vs Sequencer output. The sequencer output values were measured using an oscilloscope. All values have units of Hertz.

There was a 0.17% average percentage error in musical note frequency which occurred due to hardware constraints and resolution of the timers. The maximum deviation of output frequency from what was expected was $1.7Hz$ which lies within the frequency resolution range of human hearing, $3.6Hz$ [6]. Therefore it could be assumed that the produced note sounds were indistinguishable from the theoretical ideal, providing confidence in the design of the Signal Generator.

A further source of error prominent during sequence playback arose from the decision to only compare the highest byte of time information in the playback routine. This induced an uncertainty in the start/stop times of each note. The theoretical time error per command could be calculated:

$$Err = \frac{255}{f_{T0}} \qquad (1)$$

Where Err is the error per command in units of seconds, f_{T0} is the roll-over frequency of Timer0 in units of Hz, and 255 is the number of the number of increments missed out due to only having considered the highest byte of the time signature. To find a value for f_{T0}, recall that clocks derive their timings from the main oscillator within the Microprocessor, which in the case of the PIC18 has value $f_{osc} = 8MHz$. The instruction clock in this device is a fourth of the internal oscillator frequency [3] so to calculate the frequency of Timer0, f_{osc} was divided by 4 and then further divided by 16 to account for the $1:16$ prescaler. Finally this value was divided by 65535 (the number of Timer0 increments between roll-overs) in order to calculate a frequency corresponding to the number of roll-overs made by Timer0 per second. The frequency was found to be $f_{T0} = 125,000Hz$. Substituting our result into Eq.1 leads to a final time error of $Err = \pm 2.05ms$ per command.

It was expected this error would be prevalent in playback mode of the synthesizer. To investigate, a recording was made of a live sequence as well as two of the play-backs. The true time duration of each note and the gaps between them were compared with their expected durations. The maximum deviation between these values was 5ms. With consideration of both the error of the time signature, $\pm 2.05ms$, and the error due to measurement, $\pm 1ms$, it can be deduced that a further source of error must be prevalent. Despite this, the value of maximum time deviation lay below that of the temporal resolution of the human ear, 10ms [7]. Hence the playback system can still be considered to have performed very well.

V. Modifications and Improvements

The implementation of an interface to a keypad would have been a viable next step for this project. This could be done using serial to parallel data transfer and would allow a greater range of musical notes to be played as well as further control switches to be implemented. The implementation of serial input would also tackle an issue currently present due to the Play mode control button, RJ0, being on the same Port as the note buttons: the `state_check` routine checks all PORTJ inputs for a change of note-state, meaning a change in state of the Play button also falsely leads the algorithm to a full note state-check when it is not necessary.

A prominent limitation of the music sequencer was the requirement that only monophonous music sequences were to be played. Already implemented is the ability to detect multiple button state-changes within one cycle. To advance from this and add polyphony to the device, a further subroutine could be written to carry out logic operations that sum the amplitudes of the waves being produced synchronously before sending the output signal to the loudspeaker.

By not having used exact MIDI protocol in this project, the synthesizer was able to communicate to very few external pieces of musical hardware and software. Through the modification of adding a MIDI port to the circuit, the versatility of the system would significantly increase.

VI. Conclusions

The final product was a fully-functioning music synthesizer consisting of 7 in-tune musical notes with the option to record and playback a music sequence. All three main modes -Real Time, Recording, and Playback- performed very well and with a high level of precision. In the case of sequence play-back, the signals were generated with extreme precision in timing.

VII. Product Specifications

The full Assembly code used for this Microcontroller-based Synthesizer project can be found on the following github web address:

https://github.com/ImperialCollegeLondon/mplab/tree/mc_proc

References

1. Electronics, H. *A Brief History Of Synthesizers* Available at https://www.hi5electronics.co.uk/a-brief-history-of-synthesizers/ [Accessed: 17/11/2019].
2. Pykett, C. *How Sinthesisers Work* Available at http://www.pykett.org.uk/synths.htm [Accessed: 17/11/2019].
3. Microchip. *PIC18F87K22* Available at https://www.microchip.com/wwwproducts/en/PIC18F87K22 [Accessed: 21/11/2019].
4. Society, A. P. *Fiddle Physics* Available at https://www.physicscentral.com/explore/action/fiddle.cfm [Accessed: 17/11/2019].
5. Sindhwani, M. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*.
6. F., H. *Chapter 2: Music, Physics and Engineering. (1967* (Dover Publications. pp. 248–251. ISBN 978-0-486-21769-7).
7. Muchnik C, H. M. *Minimal time interval in auditory temporal resolution.* (1985 Oct;25(4):239-46.).