**Please read,**

**or take a peek**

**Buzz Wire Game Description**

**Scan here to read later:**

A qr code with black squares

Description automatically generated

Buzz Wire Game

# Quick Overview

You get a timed lap of the Snetterton Racetrack with this game! The game will speak your player number and lap time, and you may get on the leaderboard or make a new lap record.

When the game first powers up it checks the battery capacity and tells you to place the car at the left or right start point, if it’s not already there.

The game then says it’s initialised and suggests you press the yellow Help Button, which speaks some instructions. To start racing, you press the green Race Button, wait for the green light, lift the car clear of the start point and place it at the other end of the track.

The race is bidirectional, so leave the car at the end and this becomes the start point for the next player. During the game you’ll be told if you’re going too fast or too slow based on your crashes and time. At the end of the game your lap time is spoken and you may get some advice if you crashed too often or went too slow! If you get on the leaderboard of top 20 places, you’ll get some praise and if you get the lap record you’ll get a fanfare too.

If you start the race too soon, you’ll get a 10 second time penalty. You’ll also get a 3 second time penalty if you skid off course by touching the buzz wire. Correct your course quickly because the crash penalty is added every 0.5 seconds. “When” you crash you’ll hear a buzz tone and some car crashing sounds too. If your crash penalties add up to more than your driving time you’ll also get an extra 1 minute penalty for “stock car racing”.

If you’re not crashing too often during a race, you’ll hear some background music or sound effects played. During idle times different sounds are played, mostly with a car theme, which are interspersed with adverts every 90 seconds to encourage game play.

If you want a practice, just lift the handle and return it to an endpoint when finished. Unless you press the Race Button you won’t get a player number or lap time though.

Pressing the Leaderboard Button will speak out the top 4 player numbers and their lap times. The leaderboard records were cleared for today and the results are not lost when the game is switched off. The fastest lap of the day will receive the Piston Cup Trophy.

The battery lasts about 10 hours when the volume is high, or about a day if the volume is low.

# Technical Summary

The game uses an MCU (Microcontroller Unit), which is a low power computer chip as used in something like a washing machine. It has 4 Kbytes of program memory, 56 bytes of RAM for general purpose variables and 256 bytes of permanent storage where the leaderboard is placed. The game almost filled of all these memory locations. The program is about 1,000 lines of code and is written in BASIC because it’s simple, fun and I’m old. The MCU is a PICAXE 20X2.

An audio module stores sound on an SD card, consisting of songs, effects and downloaded AI generated words or phrases. These sounds are strung together and controlled by a communication link to the microcontroller. The microcontroller has a timer and the electrical input and output pins for the buttons, buzz wire connections, buzz sound, LEDs (Light Emitting Diodes). The inputs use transient voltage suppression to protect the electronics against static electricity, up to 30 kV.

# Photos

|  |  |
| --- | --- |
| **Working Prototype** | **The Workbench** |
|  |  |
| **Handle V1 (Final was brazed)** | **1st Semblance** |
|  |  |
| **Some 30 Year Old Varnish** | **The Underneath** |
|  |  |

|  |  |
| --- | --- |
| **Button Panel** | **Assembled Components** |
|  |  |
| **Testing Australian Style** | **The Piston Cup Trophy** |
|  |  |
| **A Look From The Top** | **A Look Underneath** |
|  |  |
|  |  |

|  |
| --- |
| **It’s Finished!**  **(Until another 22 tracks were added!)** |
|  |

# The Circuit Diagram

The Electronics Schematic was produced in KiCad and the connectivity of all components was exported as a “netlist”, so that the circuit board design could be produced in VeroRoute.

A computer diagram with text and symbols

Description automatically generated with medium confidence

# Circuit Board

The game uses a type of circuit board, which is referred to as “Stripboard” these days. It has holes 0.1” apart, which early electronic chips used and this size spacing is still common today. The board has copper strips, which are easy to visualise and connect components together. Components are passed through the holes and soldered to the copper strips. Where you want to isolate components a hole is cut in the copper strip.

Some of the components I used are over 40 years old, from when I first played with electronics. These days it’s increasingly hard to use this type of layout because components are miniaturised to tiny sizes and then surface mounted on printed circuit boards.

A diagram of a computer

Description automatically generated

# Detailed Features

The game was going to be used in a charity event with a prize attached. So, for the first time ever, I created a list of detailed features for testing. This helped to perform a final code review in addition to identifying test cases. It’s challenging to test all the program states of the microcontroller, so both steps proved worthwhile. As a bonus the list helps to describe the game’s features too:

1. Initialisation, Battery Checks and Program Updates:
   1. The game tells you when it’s powered on or restarted.
   2. The battery voltage is checked at restart and between idle tracks.
   3. If either of the 2 battery cells are nearly exhausted (below 3,000mV) a warning will be spoken and the game will shut down into a low power state to prevent over discharge. If either cell is low (below 3,300mV) then a warning will be spoken between sound tracks when the game is idle, or directly after a game has finished, then the game continues as normal.
   4. The Reset Button only works during a restart and it clears the games stats.
   5. Program updates do not clear the game stats.
   6. After a restart the previous number of games is spoken and if games have been played the lap record is spoken.
   7. After a restart the handle must be placed at a starting position, if not it will speak a warning every 3 seconds before proceeding.
2. When the game is waiting to be played:
   1. Background music with a driving or fun theme is always played. There are several soundtracks played in a loop.
   2. Every 90 seconds an advert is played to encourage game play. These adverts play in a loop:
      1. “Roll-up, roll-up! It's only 50p a game. Have some fun and a chance to win the coveted Piston Cup Trophy!”
      2. “Most racing car legends start their careers with BUZZ WIRE RACING!  It's not too late to start now, so give it a try!”
      3. “Here is an important announcement: It's championship race day here at Snetterton. Get the fastest lap of the day and get that darn tooting trophy!”
      4. “Please, please, PLEASE have a game. I'm processing 4 million lines of code a second, so I'm kind of getting bored over here! “
      5. “Hey you! Yes you! Do you want some fun? It's a mere 50p for a chance to become a racing car legend!”
      6. “Come on, have a game! I promise you won't EVEN kill yourself if you crash.”
      7. “This game has a special 2 for 1 offer! You have some FUN! And your money will go to restoring the Village Sign TOO!”
      8. “Is there no one else! Is there no one else!”, a Brad Pitt extract from Troy.
3. Starting a practise:
   1. If the handle is raised from either start point without pressing the Race Button first, then a practise session is assumed. “Having a practise hey! There's no lap time for you.” is spoken, but other sound effects of a real game are still played.
   2. When the handle is returned to either endpoint, “Practise over and ready to race! Press the green button next time!” is played.
4. Starting a race:
   1. Pressing the Race Button only functions when the game is in idle mode, or while the help information is being spoken.
   2. The game is bidirectional and can be started at either start point, even if there’s been a practise and the previous race endpoint is changed.
   3. After pressing the Race Button a coin drop effect is played, then the player number is incremented and spoken, then starting beeps and the LED sequence (Red 1, Red 1 and 2, Red 1, 2 and 3, and then Green only) are synced to run in parallel.
   4. To allow for an accidental wiggle of the handle, the handle can lose contact with the start point for up to 100mS before a premature start is determined.
   5. If the player starts before the green light a 10 second penalty is created and a message “Time penalty! Started too soon!” is played.
5. During a race:
   1. Race themed background music starts with the game. There are several soundtracks in a loop and the next track is played if a game is started, or if a track finishes during a game.
   2. The player gets some feedback:
      1. The audio clip “Taking it slow huh. I can respect that” is played after 10 seconds if crash count is zero, or after 30 seconds if crash count is less than 2.
      2. The audio clip “Slow down!” is played after 10 seconds if crash count is more than 2, or after 30 seconds if crash count is more than 5.
      3. “You've been playing a while now, if there's someone waiting let them have a go to” is spoken after 90 seconds.
   3. Whenever the handle touches the buzz wire:
      1. There’s a buzz tone.
      2. The background sound is interrupted with a crash sound effect, the sound effect changes each time and ultimately loops.
      3. There’s a 3 second penalty, which is repeated in 0.5 seconds unless the handle is free from the buzz wire.
6. Cancelling the race:
   1. If the handle is returned to the start point during a race, then the race is cancelled and the next race will use the same player number. To debounce any handle wobbles during the race start phase, this event won’t be acknowledged until at least 2 seconds after the normal race start, or a premature race start.
   2. Then “Going back huh. I would have done the same” and “Press the Race Button to start again!” are spoken.
7. Finishing the race:
   1. If the handle touches the finish point:
      1. The green race light goes out and the log of player numbers is incremented and stored permanently.
      2. “You’ve finished!” is spoken.
      3. If the total crash penalty time is higher than the driving time an audio clip “Slow down stop the insanity!” is played and a cartoon voice states “You spent more time crashing than racing!” and a normal voice states “A 1-minute penalty is added for stock car racing!”.
      4. If the player’s adjusted lap time is not quick enough for a leaderboard entry then “Aarh you just missed out on the leaderboard! Have another try!” is spoken.
      5. If the player’s adjusted lap time achieves a leaderboard ranking their player number and lap time is inserted in the permanent memory of the leaderboard. All lower positions are shuffled downwards and eventually out of the leaderboard, which is limited to 20 places.
      6. If the player is ranked 1st the audio clip “A new record!” is played. If this isn’t the 1st player, then 1 of 4 random victory audio clips is played.
      7. If the player achieves a leaderboard ranking that is less than half the number of games played, the announcement “Here's a musical tribute just for you!” is spoken. A short fanfare is then played from a loop of tracks. Then a voice states their position on the leaderboard.
      8. If the crash count is zero then “No crashes. That's amazing for this crowd!” is spoken. If it’s not a lap record the audio clip “Wow all I can say is wow!” is played.
      9. If the crash count is greater than 9 “Unfortunately the car is a right off” is spoken.
      10. If there were any crashes, a voice states the number of crashes and how much time each crash cost.
      11. Then the lap time is spoken in seconds to 1 decimal place.
      12. Then “Waiting for next player!” is spoken and the game returns to idle/waiting mode.
8. Help Button:
   1. The help button only functions when the game is in idle mode, waiting to be played.
   2. This soundtrack is spoken: “This is a souped-up buzz wire game! The idea of this game is to get your race car around a lap of the racetrack as quickly as possible. If the car touches the metal tube you'll hear the car crashing, your parents worrying about the repair bill and most importantly time will be added to your lap of the race! To start a race, press the green button and wait for the green light. All you have to do is lift your car up and place it down at the other end of the track. The game will tell you your lap time and whether you've set a record for the day. Take it in turns to play. Have fun and Good luck!”.
   3. As an admin function, if the Help Button is held down and the Leaderboard Button pressed then the battery cell voltages will be spoken.
9. Speaking the Leaderboard:
   1. The Leaderboard Button only functions when the game is in idle mode, or while the help information is being spoken.
   2. If any games have been played an announcement starts with “The top N positions are”, where N is the number of games played up to the maximum of 4.
   3. Then from 1st to 4th place: the place, player number and lap time in tenths of a second are spoken.
   4. If the number of games is more than 4 the number of games played is spoken.
   5. Then “Press the green race button to beat them!” is played.

# The Program

Written in PICAXE BASIC, the program occupies 4,072 bytes of the 4,096 available on the 20X2 microcontroller. As more functionality, like the Leaderboard, was added some memory had to be saved by reducing the debug logging.

The program is organized into the following regions to improve quality:

1. Program Notes
2. Compiler Directives
3. Resources
   1. Pins
   2. Variables
   3. Constants
   4. Utilities
4. Initialisation
5. Main:
6. Interrupt:
7. Subroutines

Four software modules were written for the game, these are intended to be re-useable utilities for future projects:

1. "Debug\_Terminal.basinc" helps to easily display status information to a computer screen while the program is running.
2. "Timing.basinc" sets the processor clock rate and provides timing functions regardless of the clock rate.
3. "DF\_Player\_Mini.basinc" provides functions to easily control sound output from a DF Player Mini audio module mounted on the circuit board.
4. "Voltage.basinc" calculates the chip supply voltage and the input voltage levels of the battery cells.

# Program Listing – “Buzz\_Wire\_Race.bas”

#Region "Program Notes"

#REM

File: "Buzz\_Wire\_Race.bas"

License: MIT (See end of file)

Change History: 2024/10/27 by Alan Hunt - 1st issue.

Microcontroller Pin Usage

==========================

Pin Usage PICAXE 20X2 Pinout & Functions Pin Usage

-------------------------------------------------------------------

+5V Supply |1 Vdd Gnd 20| 0V Supply

Programming Input |2 Serial In Serial out 19| Debugging Output

BATTERY (ADC3 VoltageBatt) |3 C.7(ADC3/Out/In) B.0(In/Out/ADC1/hint1) 18| AUDIO\_TX (to DF Player @T9600)

HelpButton |4 C.6(In) B.1(In/Out/ADC2/hint1/SRQ) 17| CELL (ADC2 for Cell 1 voltage)

LeaderboardButton |5 C.5(hpwmA/pvmC.5/Out/In) B.2(In/Out/ADC4/Comp2+) 16| AUDIO\_STATUS (DF Player active low)

RaceButton |6 C.4(hwpmB/SRNQ/Out/In) B.3(In/Out/ADC5/Comp2-) 15| LED\_RED1 (1st "Get Ready")

InputPosL (Left Start Point) |7 C.3(hwpmC/ADC7/Out/In) B.4(In/Out/ADC6/hpwmD/Comp1-) 14| PWM\_BUZZ (Tone for Buzz Wire)

InputPosR (Right Start Point) |8 C.2(kbclk/ADC8/Out/In) B.5(In/Out/ADC10/hi2csda/hspisdi) 13| LED\_RED2 (2nd "Get Ready")

InputBuzz (Buzz Wire) |9 C.1(hspisdo/kbdata/ADC9/Out/In) B.6(In/Out/ADC11/hserin) 12| LED\_RED3 (3rd "Get Ready")

RESET\_BUTTON |10 C.0(hserout/Out/In) B.7(In/Out/hi2cscl/hspisck) 11| LED\_Green(Race "Go" light)

-------------------------------------------------------------------

PICAXE 20X2 Notes

=================

\* Chip hardware is the Microchip PIC18F14K22 MCU:

- Power supply 1.8V to 5.5V. Default clock 8MHz.

- Pullup and pulldown input currents measured respectively as 1.5nA and 300pA on B.1 when set as a digital input.

- When an ADC channel is enabled it draws approximately 280 uA and the recommended source impedance is 10 kOhm.

- Input over/under voltage protection has a maximum clamp current of 20mA.

Program Notes

=============

\* The game has 7 digital inputs monitored on Port C. Five of the inputs are monitored by software interrupts with "setint" (Only c.1 to c.5 permitted for 20M2 and 20X2). The symbol "getInputs" (pinsC) is used to read PortC simultaneously and place it in the symbol "GameInputs" (variable b3). Then bits of b3 are used to check specific inputs, like InputBuzz(bit25).

\* Clock rate was set to 16MHz but it's not necessary. It probably helps to speed up inputs, particularly as I left debugging on for show.

\* Timer3 word value is set to increment every 131 mS (Prescale8:1 \* 65536 \* 4 / Clock), so very roughly divide 8 to get seconds.

#EndREM

#EndRegion

#Region "Main Compiler Directives"

'Compiler Directives

#picaxe 20X2

#no\_data

#no\_table

#no\_debug

#define Debugger\_On 'Comment out this line to disable Debugging.

#Endregion

#Region "Resources"

'Pins

symbol AUDIO\_TX = B.0 '@DF\_Player\_Mini.basinc: Serial Tx to audio module via 1K resistor.

symbol CELL1 = pinB.1 '@Voltage.basinc: Cell 1 voltage monitoring is ADC2(B.1).

symbol AUDIO\_STATUS = pinB.2 '@DF\_Player\_Mini.basinc: Busy is active low after time lag (AUDIO\_IDLE and AUDIO\_BUSY).

symbol LED\_RED1 = B.3 '1st track light.

symbol PWM\_BUZZ = B.4 'Tone for Buzz Wire.

symbol LED\_RED2 = B.5 '2nd track light.

symbol LED\_RED3 = B.6 '3rd track light.

symbol LED\_GREEN = B.7 '4th "go" and running track light.

symbol RESET\_BUTTON = pinC.0 'Circuit board button to clear game stats if pressed during initialisation.

symbol HELP\_BUTTON = pinC.6 'Button to explain game.

symbol BATTERY = pinC.7 '@Voltage.basinc: Battery voltage monitoring is ADC3(C.7).

'Variables

symbol GameTime = timer 'Alias for the system "timer", measures elapsed game/idle time time in tenths of a second.

symbol UpTime = timer3 'Alias for "timer3", used for random number generation and uptime in 8th's of a second.

symbol \_Debugger = b0 '"Debugger.basinc" module is limited to using b0 for Byte to ASCII bitstream conversion.

symbol InterruptSetting = b1 'Interrupt setting for input pins.

symbol IntSetBuzz = bit9

symbol IntSetPosR = bit10

symbol IntSetPosL = bit11

symbol IntSetRace = bit12

symbol IntSetLeaderboard = bit13

symbol InterruptMask = b2 'Interrupt mask for input pins.

symbol IntMaskBuzz = bit17

symbol IntMaskPosR = bit18

symbol IntMaskPosL = bit19

symbol IntMaskRace = bit20

symbol IntMaskLeaderBoard = bit21

symbol GameInputs = b3 'Game inputs on PortC with quick byte getter "getInputs" and bit checkers in b3.

symbol getInputs = pinsC

symbol InputBuzz = bit25

symbol InputPosR = bit26

symbol InputPosL = bit27

symbol RaceButton = bit28

symbol LeaderboardButton = bit29

symbol HelpButton = bit30

symbol GameStatus = b4 'Game Status has enumerated state constants

symbol GameDirection = b5

symbol SoundLoopIdle = b6 'Pointer for a list of tracks played when no game is running.

symbol SoundLoopBackground = b7 'Pointer for a list of tracks played as background music during the game.

symbol SoundLoopCrash = b8 'Pointer for a list of crash sounds that interupt the background music.

symbol SoundLoopFanfare = b9 'Pointer for a list of tracks played at the end of a game.

symbol SoundLoopAdvert = b10 'Pointer for a list of adverts played when the game is idle for too long.

symbol CrashCount = b11

symbol LapTime = w6 'Time in tenths of a second.

symbol PenaltyTime = w7 'Time added, in tenths, for a premature race start.

symbol LapRecord = w8

symbol PlayerNumber = w9

symbol PlayerRanking = b20

symbol GameTimerLog = b21

symbol GameTimer1 = w11

symbol GameTimer2 = w12

symbol GameTimer3 = w13

symbol VoltsCell2 = w14

symbol VoltsCell1 = w15

symbol BatteryCapacity = b32

'Internal variables for gosubs (Try and keep these unique due to ease Interrupt handling)

symbol \_LeaderBoard\_ResultPTR = b33 'For Gosub PlayerRank only.

symbol \_LeaderBoard\_TimePTR = b34 'For Gosubs PlayerRank and SayLeaderboard.

symbol \_LeaderBoard\_PlayerPTR = b35 'For Gosubs PlayerRank and SayLeaderboard.

symbol \_GameEndW = w18 'For Gosub GameEnd only.

symbol \_LeaderBoardW = w19 'For Gosubs PlayerRank and SayLeaderboard.

symbol \_LeaderBoardB = b40 'For Gosubs PlayerRank and SayLeaderboard.

'@DF\_Player\_Mini.basinc Variables

symbol \_Timing = w21 'Assign symbol to any spare Word.

'@DF\_Player\_Mini.basinc Variables

symbol \_Audio\_Digit = b41 'Assign symbol to any spare Byte.

symbol \_Audio\_Integer = w22 'Assign symbol to any spare Word.

symbol \_Audio\_Fraction = w23 'Assign symbol to any spare Word.

symbol \_Audio\_Track = b48 'Assign symbol to any spare Byte.

symbol \_Audio\_Byte = b49 'Assign symbol to any spare Byte.

'@Voltage.basinc Variables

symbol Voltage\_Vdd = w25 'Assign symbol to any spare Word. Stores PICAXE supply voltage as mV.

symbol \_VoltageADCval = w26 'Assign symbol to any spare Word.

symbol \_VoltageModulus = w27 'Assign symbol to any spare Word.

'Constants

'For interrupts

symbol VAL\_DEFAULT = %00000000 'Interrupt value default

symbol MASK\_DEFAULT = %00111110 'Interrupt mask default

symbol NOT\_MASK\_DEFAULT = %11000001

symbol CONTACT = 1 'Logic level for contact with buzz wire or end points.

symbol NO\_CONTACT = 0

symbol PRESSED = 1 'Logic level for button press.

symbol NOT\_PRESSED = 0

'For GameStatus variable

symbol GAME\_NOT\_SET = 0 'Not at a valid start point.

symbol GAME\_WAITING = 1 'At a valid start point and waiting for Race or practise.

symbol GAME\_BEGINNING = 2 'Race button pressed.

symbol GAME\_NOT\_STARTED = 3 'Race Go signal was given but not moved off yet.

symbol GAME\_PLAYING = 4

symbol GAME\_ENDING = 5 'Reached end point and stating performance.

symbol GAME\_PRACTISE = 6 'Moved off without pressing Race button.

'For GameDirection variable

symbol R\_TO\_L = 1

symbol L\_TO\_R = 2

'For Penalty time (tenths of a second)

symbol PENALTY\_PREMATURE\_START = 100

symbol PENALTY\_CRASH = 30

'Initialise Variables

symbol SOUND\_IDLE\_START = 177 'MP3 177 to 215 play backgound music when no game is running.

symbol SOUND\_IDLE\_END = 215

symbol SOUND\_BACKGROUND\_START = 218 'MP3 218 to 238 play background music during the game.

symbol SOUND\_BACKGROUND\_END = 238

symbol SOUND\_FANFARE\_START = 250 'MP3 250 to 254 play different game endings for leaderboard entries.

symbol SOUND\_FANFARE\_END = 254

symbol SOUND\_CRASH\_START = 1 'ADVERT 1 to 8 play crash sounds that interupt the background music.

symbol SOUND\_CRASH\_END = 8

symbol SOUND\_ADVERT\_START = 15 'ADVERT 15 to 22 interuppt idle music to encourage game playing.

symbol SOUND\_ADVERT\_END = 22

'Timer Values (tenths of a second)

symbol GAME\_TIME1 = 100 'Some encouragement after 10S, or quit a failed race start.

symbol GAME\_TIME2 = 300 'Some encouragement after 30S.

symbol GAME\_TIME3 = 900 'Suggest you let someone else play after 90S.

symbol IDLE\_TIMER = 900 'Play an advert to encourage playing every 90S.

'For EEPROM storage locations

symbol aGAME\_COUNT = 2

symbol LEADERBOARD\_SIZE = 20 'Limited to 20 to make speaking nth's easier.

symbol aLEADERBOARD\_PLAYER = 30 'The Leaderboard player numbers occupy EEPROM addresses from here.

symbol aLEADERBOARD\_PLAYER\_END = 68 'Player numbers are stored as words, the last byte used is this plus 1.

symbol aLEADERBOARD\_TIME = 70 'Laptimes are stored as words from here.

symbol aLEADERBOARD\_TIME\_END = 108 'The last byte is this +1.

symbol LEADERBOARD\_SPOKEN\_SIZE = 4 'Number of entries spoken after LEADERBOARD\_BUTTON pressed (limited to 4 by track 120).

'For Battery Monitoring (Voltage in mV)

symbol VOLTS\_CELL\_GOOD = 3400 'A li-ion Samsung INR18650-30Q with constant 200mA load was tested to be about 18% at 3400 mV.

symbol VOLTS\_CELL\_EXHAUSTED = 3100 'The battery spec is 2.5V cut-off but tested down to 2.8V, this gave about 6% at 3100 mV.

'Caution: Figures change drastically for 3.0V cut-off cells, or high current loads.

'@Timing.basinc Constants

'CLOCK\_SET\_KHZ options: 31,250,500,1000,2000,4000,8000,16000,32000,or 64000 (Dependenton chip). No equals sign and nothing after number.

#define CLOCK\_SET\_KHZ 16000

#define TIMER\_IN\_HUNDRETHS 10

'@DF\_Player\_Mini.basinc Constants

symbol AUDIO\_VOLUME = 31 '0 to 31(Maximum).

symbol AUDIO\_BAUD = T9600\_16 'Tx is idle high (T) and 9600 bps, the last characters you adjust to your clock speed.

symbol AUDIO\_ACTIVE = 0 'The AUDIO\_STATUS pin is active low, indicating audio playing.

symbol AUDIO\_IDLE = 1 'AUDIO\_STATUS high indicates the module is idle, or seeking to play a track on the disk.

'@Voltage\_X2.basinc Constant

symbol VoltageDividend = 10595 'Adjust value for FVR inaccuracy, see "Voltage\_X2.basinc" Calibration.

'Modules

#include "Debugger.basinc"

#include "Timing.basinc"

#include "DF\_Player\_Mini.basinc"

#include "Voltage\_X2.basinc"

#Endregion

#Region "Programme Initialisation"

Init:

if PenaltyTime = PENALTY\_PREMATURE\_START then goto Main 'A premature start causes a Run 0

Pause1S 'Allows PICAXE Editor time to open the terminal window.

DebugLine(ppp\_filename) 'Displays the programming filename in PICAXE Editor Terminal if "Debugger\_On" is defined.

DebugLine(ppp\_datetime) 'Displays the programming date and time.

'Setup Pins

dirsB = %11111001 'Set port B7 to B0 pins (1=output).

dirsC = %00000000 'Ensure all C pins are inputs.

adcsetup = %0000000000000110 'Use ADC3(Pin 3, C.7) and ADC2 (Pin 17, B.1), this disables digital interface circuitry.

hpwm PWMDIV16, 0, 0, %1000, 255, 511 'Use PWM to produce lowest tone possible for a potential buzz wire contact sound.

'Setup Timers

tmr3setup %10110001 'Enables timer3 to generate randomness and for approx timing (8 per Sec, see notes).

'Initialise Variables

SoundLoopIdle = SOUND\_IDLE\_START 'MP3 150 to 172 play background music when no game is running.

SoundLoopBackground = SOUND\_BACKGROUND\_START 'MP3 200 to 212 play background music during the game.

SoundLoopCrash = SOUND\_CRASH\_START 'ADVERT 1 to 9 play crash sounds that interupt the background music.

SoundLoopFanfare = SOUND\_FANFARE\_START 'MP3 250 to 254 play different game endings.

SoundLoopAdvert = SOUND\_ADVERT\_START 'ADVERT 15 to 21 to encourage play when idle too long.

GameTimer1 = GAME\_TIME1 'Some encouragement after 10S, or quit a failed race start.

GameTimer2 = GAME\_TIME2 'Some encouragement after 30S.

GameTimer3 = GAME\_TIME3 'Suggest you let someone else play after 90S.

'Initialise Audio Module and check battery

gosub AudioInitialise

PlayMP3(063,"Restarted") 'Play "The system has restarted"

gosub CheckBattery

'Reset game records during startup if necessary

if RESET\_BUTTON = PRESSED then

DebugLine("@Reset Stats")

PlayerNumber = 0

write aGAME\_COUNT, WORD PlayerNumber

LapRecord = 0

for PlayerRanking = aLEADERBOARD\_Player to aLEADERBOARD\_TIME\_END step 2

write PlayerRanking, WORD PlayerNumber

next PlayerRanking

PlayMP3(0145,"game stats reset")

gosub AudioWaitForIdle

endif

'Speak the game stats

gosub SayGamesPlayed

gosub SayLapRecord

'Ensure the car is at one end of the track and prepare interrupt settings

gosub GameGetReady

InterruptSetting = VAL\_DEFAULT

InterruptMask = MASK\_DEFAULT

'Confirm Initialisation

PlayMP3(0151," Hello the game is ready")

PlayMP3(0133," Press Help or start now")

DebugLine("@Initialised")

debug 'Display values in PICAXE Editor unless "no\_debug" is defined.

#Endregion

#Region "Main"

Main:

'Contiunally loop around Main to see if music has ended or GameTimers have expired.

setint NOT InterruptSetting, InterruptMask 'Set interupts for low state on C1 to C5 (BuzzWire, PosR, PosL, Race and Leaderboard)

'Check if the audio has finished playing.

if AUDIO\_STATUS = AUDIO\_IDLE then

DebugLine("@Main Without Audio")

gosub DebugGameStatus

select case GameStatus

case GAME\_NOT\_SET

PlayNow(0134," Place car at 1 end of the track")

Pause10S

case GAME\_WAITING

gosub CheckBattery

gosub AudioWaitForIdle

DebugOut("Play Idle ",#SoundLoopIdle,cr,lf)

PlayNow(SoundLoopIdle," IdleTrack") 'Play the next rotating idle track

inc SoundLoopIdle

if SoundLoopIdle > SOUND\_IDLE\_END then

SoundLoopIdle = SOUND\_IDLE\_START

endif

case GAME\_NOT\_STARTED, GAME\_PLAYING, GAME\_PRACTISE

DebugOut("Play Background ",#SoundLoopBackground,cr,lf)

PlayNow(SoundLoopBackground," GameTrack") 'Play the next rotating game track

inc SoundLoopBackground

if SoundLoopBackground > SOUND\_BACKGROUND\_END then

SoundLoopBackground = SOUND\_BACKGROUND\_START

end if

else

'Other values GAME\_BEGINNING and GAME\_ENDING are temporary and should not occur.

endselect

endif

'Check if timers have expired.

select case GameStatus

case GAME\_WAITING

if GameTime > IDLE\_TIMER then

PlayAdvert(SoundLoopAdvert,"Roll-up etc")

Gametime = 0

inc SoundLoopAdvert

if SoundLoopAdvert > SOUND\_ADVERT\_END then

SoundLoopAdvert = SOUND\_ADVERT\_START

endif

endif

if HELP\_BUTTON = PRESSED then

DebugLine("Help Pressed")

GameTime = 0 'Reset the timer for encouragement adverts

PlayNow(0132," GameIntro")

gosub AudioWaitForIdle

endif

case GAME\_BEGINNING

gosub GameBegin

case GAME\_NOT\_STARTED

'DebugOut("GameTime: ",#GameTime,cr,lf)

'DebugOut(", GameTimer1: ",#GameTimer1,cr,lf)

if GameTime > GameTimer1 then

DebugLine("Not started after 10S")

gosub GameQuit

PlayNow(0153, "Oh no it's nap time")

PlayMP3(0152, "Press the race button to start again")

gosub AudioWaitForIdle

endif

case GAME\_PLAYING

select case GameTime

case > GameTimer1

if CrashCount = 0 then

PlayAdvert(11,"Too slow")

elseif CrashCount > 2 then

PlayAdvert(14, "Slow down")

endif

GameTimer1 = 65535

case > GameTimer2

if CrashCount < 2 then

PlayAdvert(11,"Too slow")

elseif CrashCount > 5 then

PlayAdvert(14, "Slow down")

endif

GameTimer2 = 65535

case > GameTimer3

PlayNow(0135, "You've been playing a while, let someone else too")

GameTimer3 = 65535

endselect

endselect

Pause10mS

goto Main

#Endregion

#Region "Interrupts"

Interrupt:

GameInputs = getInputs

DebugLine("@Interupt")

gosub DebugGameStatus

gosub DebugGameInputs

select case GameStatus

case GAME\_NOT\_SET

if InputPosL = CONTACT OR InputPosR = CONTACT then 'Car now placed correctly at one end of the track.

GameStatus = GAME\_WAITING

if InputPosL = CONTACT then

GameDirection = L\_TO\_R

else

GameDirection = R\_TO\_L

endif

PlayNow(062," Thank you")

endif

case GAME\_WAITING

if InputPosL = NO\_CONTACT AND InputPosR = NO\_CONTACT then 'Car moving without Race button, so it's a practise.

GameStatus = GAME\_PRACTISE

SoundLoopCrash = SOUND\_CRASH\_START

CrashCount = 0

PlayNow(0147,"having a practise hey. There's no laptime for you")

endif

if LeaderboardButton = PRESSED then

if HelpButton = PRESSED then

gosub AudioClear

gosub CheckBattery 'If both pressed then state battery cell voltages

PlayNow(0070, "The cell voltages are")

AudioSpeakNumber(VoltsCell1)

PlayMP3(0130, "millivolts")

PlayMP3(0060, "and")

AudioSpeakNumber(VoltsCell2)

PlayMP3(0130, "millivolts")

else

gosub SayLeaderboard

endif

GameTime = 0 'Reset the timer for encouragement adverts

endif

if RaceButton = PRESSED then

GameStatus = GAME\_BEGINNING

endif

case GAME\_BEGINNING

if InputPosL = NO\_CONTACT AND InputPosR = NO\_CONTACT then 'Car moved too early while game still begining (before green light).

Pause100ms 'Pause a while to debounce a player wiggling the handle.

GameInputs = getInputs

if InputPosL = NO\_CONTACT AND InputPosR = NO\_CONTACT then 'Car is really moving too early, so start the game and log the penalty.

GameStatus = GAME\_PLAYING

'DebugLine("Started too Soon")

PlayNow(0148,"Time penalty. Started too soon.")

low LED\_RED1, LED\_RED2, LED\_RED3

high LED\_GREEN

GameStatus = GAME\_PLAYING

GameTimer1 = GAME\_TIME1

GameTimer2 = GAME\_TIME2

GameTimer3 = GAME\_TIME3

SoundLoopCrash = SOUND\_CRASH\_START

CrashCount = 0

PenaltyTime = PENALTY\_PREMATURE\_START

GameTime = 0

IntSetPosL = InputPosL

IntSetPosR = InputPosR

IntSetBuzz = InputBuzz

run 0

endif

endif

case GAME\_NOT\_STARTED

if InputPosL = NO\_CONTACT AND InputPosR = NO\_CONTACT then 'Car moving correctly after given the green light.

DebugLine("Started OK")

GameStatus = GAME\_PLAYING

endif

case GAME\_PLAYING

if InputPosL = CONTACT then 'Car touching left end of the track.

if GameDirection = R\_TO\_L then

gosub GameEnd

else

if GameTime > 20 then 'debounce a return to start

PlayNow(0144," Going back huh")

PlayMP3(0152," Press the race button to start again")

gosub AudioWaitForIdle

gosub GameQuit

endif

endif

endif

if InputPosR = CONTACT then 'Car touching right end of the track.

if GameDirection = L\_TO\_R then

gosub GameEnd

else

if GameTime > 20 then 'debounce a return to start

PlayNow(0144," Going back huh")

PlayMP3(0152," Press the race button to start again")

gosub AudioWaitForIdle

gosub GameQuit

endif

endif

endif

if InputBuzz = CONTACT then 'Buzz wire contact

gosub GameCrash

endif

case GAME\_PRACTISE

if InputPosL = CONTACT or InputPosR = CONTACT then 'Car finished practising on the track.

GameStatus = GAME\_WAITING

if InputPosL = CONTACT then

GameDirection = L\_TO\_R

else

GameDirection = R\_TO\_L

endif

PlayNow(0150," Practise over, press green button")

endif

if InputBuzz = CONTACT then 'Buzz wire contact

gosub GameCrash

endif

endselect

IntSetPosL = InputPosL

IntSetPosR = InputPosR

gosub DebugGameStatus

DebugLine("@Interupt End")

setint NOT InterruptSetting, InterruptMask

return

#Endregion

#Region "Subroutine GameGetReady"

GameGetReady:

DebugLine("@GameGetReady")

GameInputs = getInputs

gosub DebugGameInputs

if InputPosL = NO\_CONTACT and InputPosR = NO\_CONTACT then

do until InputPosL = CONTACT or InputPosR = CONTACT 'Ensure car is at 1 end of the track

PlayMP3(0134,"Place car at 1 end of the track")

Pause3S

GameInputs = getInputs

loop

PlayNow(062," Thank you")

endif

if InputPosL = CONTACT then

GameDirection = L\_TO\_R

else

GameDirection = R\_TO\_L

end if

IntSetPosL = InputPosL

IntSetPosR = InputPosR

GameStatus = GAME\_WAITING

return

#Endregion

#Region "Subroutine GameBegin"

GameBegin:

DebugLine("@GameBegin")

GameStatus = GAME\_BEGINNING

read aGAME\_COUNT, WORD PlayerNumber

inc PlayerNumber 'Don't flash the PlayerNumber increase until a successful end.

PlayNow(173," Arcade token")

PlayMP3(157," Player")

AudioSpeakNumber(PlayerNumber)

PlayMP3(174," Wait for the green light")

PlayMP3(175," Mario Kart beeps")

high LED\_RED1

Pause1S

high LED\_RED2

Pause1S

high LED\_RED3

Pause1S

low LED\_RED1, LED\_RED2, LED\_RED3

high LED\_GREEN

GameStatus = GAME\_NOT\_STARTED

GameTimer1 = GAME\_TIME1

GameTimer2 = GAME\_TIME2

GameTimer3 = GAME\_TIME3

SoundLoopCrash = SOUND\_CRASH\_START

CrashCount = 0

PenaltyTime = 0

GameTime = 0

return

#Endregion

#Region "Subroutine GameCrash"

GameCrash:

DebugLine("@GameCrash")

PlayAdvert(SoundLoopCrash, "Crash") 'There is a timed delay in PlayAdvert, which helps as a crash rate limiter.

if CrashCount < 255 then

inc CrashCount

inc SoundLoopCrash

if SoundLoopCrash > SOUND\_CRASH\_END then

SoundLoopCrash = SOUND\_CRASH\_START

endif

endif

DebugOut("CrashCount=",#CrashCount,cr,lf)

return

#Endregion

#Region "Subroutine GameQuit"

GameQuit:

DebugLine("@GameQuit")

low LED\_RED1, LED\_RED2, LED\_RED3, LED\_GREEN

gosub GameGetReady

GameTime = 0

return

#Endregion

#Region "Subroutine GameEnd"

GameEnd:

DebugLine("@GameEnd")

'Record Game and notify of GameEnd

GameStatus = GAME\_ENDING

LapTime = GameTime

low LED\_GREEN

gosub AudioClear

write aGAME\_COUNT, WORD PlayerNumber

PlayNow(0136," You finished!")

'Add 1 minute penalty if crash time is greater than laptime.

\_GameEndW = PENALTY\_CRASH \* CrashCount

if \_GameEndW > Laptime then

PenaltyTime = PenaltyTime + 600

PlayMP3(246," Slow down stop the insanity!")

PlayMP3(155," 1 minute penalty because crashtime > laptime")

endif

'Calculate laptime and penalty time (measured in tenths of a second)

DebugOut("GameTime=", #Laptime)

DebugOut(", CrashTime=", #\_GameEndW)

PenaltyTime = \_GameEndW + PenaltyTime

LapTime = LapTime + PenaltyTime

DebugOut(", Adjusted=",#LapTime,cr,lf)

'Leaderboard ranking and associated messages.

gosub PlayerRank

select case PlayerRanking

case 0

PlayMP3(0170," Ah you just missed out on the leaderboard, have another try!")

case 1

LapRecord = LapTime

PlayMP3(0149," New lap record")

\_GameEndW = UpTime % 4

\_GameEndW = \_GameEndW + 240

PlayMP3(\_GameEndW," Winners track")

gosub PlayFanfare

else

gosub PlayFanfare

endselect

'Speak about crashes

if CrashCount = 0 then

PlayMP3(0142," No crashes!")

if LapTime <> LapRecord then

PlayMP3(0247," Wow all I can say is wow!")

endif

PlayMP3(0137," Your time was")

else

if CrashCount > 9 then

PlayMP3(0143," The car is a right off.")

endif

PlayMP3(0140," You crashed")

AudioSpeakNumber(CrashCount)

PlayMP3(0141," times & each cost")'

AudioSpeakNumber(PENALTY\_CRASH/10)

PlayMP3(0083," seconds")

PlayMP3(154," Your adjusted time was")

endif

'Speak the laptime seconds

AudioSpeakTenths(LapTime)

PlayMP3(0083," seconds")

If PlayerRanking <> 1 then gosub SayLapRecord

gosub GameQuit

PlayMP3(0156," Waiting for next player!")

return

#Endregion

#Region "Subroutine SayGamesPlayed"

SayGamesPlayed:

read aGAME\_COUNT, WORD PlayerNumber

DebugOut("@SayGamesPlayed: ",#PlayerNumber,cr,lf)

PlayMP3(0138," There have been")

AudioSpeakNumber(PlayerNumber)

PlayMP3(0139," games")

return

#Endregion

#Region "Subroutine SayLeaderboard"

SayLeaderBoard:

read aGAME\_COUNT, WORD PlayerNumber

DebugLine("@SayLeaderBoard")

if PlayerNumber = 0 then

'@No games yet

gosub AudioClear

gosub sayGamesPlayed

else

'@Speak Leaderboard

PlayNow(0158, "The top")

If PlayerNumber > LEADERBOARD\_SPOKEN\_SIZE then

PlayMP3(LEADERBOARD\_SPOKEN\_SIZE, "{n}")

else

PlayMP3(PlayerNumber, "{n}")

endif

PlayMP3(0159," leaderboard positions are:")

\_LeaderBoard\_PlayerPTR = aLEADERBOARD\_PLAYER

\_LeaderBoard\_TimePTR = aLEADERBOARD\_TIME

\_LeaderBoard\_ResultPTR = 0

do

read \_LeaderBoard\_PlayerPTR, WORD \_LeaderBoardW

if \_LeaderBoardW = 0 then exit 'Quit leaderboard if the entry isn't filled yet.

\_LeaderBoardB = 160 + \_LeaderBoard\_ResultPTR 'Track 0160 to 0163 is "1st place" to "4th place".

PlayMP3(\_LeaderBoardB," {nth} place is")

PlayMP3(0157," Player")

AudioSpeakNumber(\_LeaderBoardW)

read \_LeaderBoard\_TimePTR, WORD \_LeaderBoardW

PlayMP3(0164," with laptime")

AudioSpeakTenths(\_LeaderBoardW)

PlayMP3(0083," seconds")

inc \_LeaderBoard\_ResultPTR

\_LeaderBoard\_PlayerPTR = \_LeaderBoard\_PlayerPTR + 2

\_LeaderBoard\_TimePTR = \_LeaderBoard\_TimePTR + 2

loop until \_LeaderBoard\_ResultPTR = LEADERBOARD\_SPOKEN\_SIZE 'Only speak the first n entries.

if PlayerNumber > LEADERBOARD\_SPOKEN\_SIZE then gosub sayGamesPlayed 'Speak game count if more records.

PlayMP3(0165, "Press green button to beat them")

endif

GameTime = 0

return

#Endregion

#Region "Subroutine SayLapRecord"

SayLapRecord:

read aLEADERBOARD\_TIME, WORD LapRecord

DebugOut("@SayLapRecord: ",#LapRecord,cr,lf)

if LapRecord > 0 then

PlayMP3(0146," The lap record is")

AudioSpeakTenths(LapRecord)

PlayMP3(0083," seconds")

endif

return

#Endregion

#Region "Subroutine PlayFanfare"

PlayFanfare:

PlayMP3(0166, "Congratulations you are")

\_GameEndW = 32 + PlayerRanking

PlayMP3(\_GameEndW, "nth")

PlayMP3(0167, "on the leaderboard")

\_GameEndW = PlayerNumber / 2

if PlayerRanking <= \_GameEndW then

PlayMP3(168," Musical tribute just for you.")

PlayMP3(SoundLoopFanfare," Fanfare")

inc SoundLoopFanfare

if SoundLoopFanfare > SOUND\_FANFARE\_END then

SoundLoopFanfare = SOUND\_FANFARE\_START

endif

endif

return

#Endregion

#Region "Subroutine PlayerRank"

PlayerRank:

'DebugLine("@PlayerRank")

'@Work down the leaderboard to determine ranking from the 1st beaten or blank time entry.

\_LeaderBoard\_ResultPTR = aLEADERBOARD\_TIME

do

read \_LeaderBoard\_ResultPTR, WORD \_LeaderBoardW

if \_LeaderBoardW = 0 OR LapTime < \_LeaderBoardW then exit

inc \_LeaderBoard\_ResultPTR

inc \_LeaderBoard\_ResultPTR

loop until \_LeaderBoard\_ResultPTR > aLEADERBOARD\_TIME\_END

'@Chose action based on ranking.

select case \_LeaderBoard\_ResultPTR

case = aLEADERBOARD\_TIME\_END

'@Last entry on the leaderboard.

PlayerRanking = aLEADERBOARD\_TIME\_END - aLEADERBOARD\_TIME / 2 + 1

write aLEADERBOARD\_PLAYER\_END, WORD PlayerNumber

write aLEADERBOARD\_TIME\_END, WORD LapTime

'DebugOut("RankLast=",#PlayerRanking,cr,lf)

case > aLEADERBOARD\_TIME\_END

'@Not on the leaderboard.

PlayerRanking = 0

'DebugLine("Rank=None")

return

else

'@Other placing on the leaderboard.

PlayerRanking = \_LeaderBoard\_ResultPTR - aLEADERBOARD\_TIME / 2 + 1

'DebugOut("RankElse=",#PlayerRanking,cr,lf)

'@Work up the leaderboard to shuffle slower entries down.

\_LeaderBoard\_PlayerPTR = aLEADERBOARD\_PLAYER\_END - 2

\_LeaderBoard\_TimePTR = aLEADERBOARD\_TIME\_END - 2

do

'Move time entry down

read \_LeaderBoard\_TimePTR, WORD \_LeaderBoardW

\_LeaderBoard\_TimePTR = \_LeaderBoard\_TimePTR + 2

write \_LeaderBoard\_TimePTR, WORD \_LeaderBoardW

\_LeaderBoard\_TimePTR = \_LeaderBoard\_TimePTR - 4

'Move player entry down

read \_LeaderBoard\_PlayerPTR, WORD \_LeaderBoardW

\_LeaderBoard\_PlayerPTR = \_LeaderBoard\_PlayerPTR + 2

write \_LeaderBoard\_PlayerPTR, WORD \_LeaderBoardW

\_LeaderBoard\_PlayerPTR = \_LeaderBoard\_PlayerPTR - 4

loop until \_LeaderBoard\_TimePTR < \_LeaderBoard\_ResultPTR

'@Insert new leaderboard entry

\_LeaderBoard\_PlayerPTR = \_LeaderBoard\_PlayerPTR + 2

write \_LeaderBoard\_PlayerPTR, WORD PlayerNumber

\_LeaderBoard\_TimePTR = \_LeaderBoard\_TimePTR + 2

write \_LeaderBoard\_TimePTR, WORD LapTime

endselect

'DebugOut("Rank=",#PlayerRanking,cr,lf)

#rem'For Debug output only:

for \_LeaderBoard\_PlayerPTR = aLEADERBOARD\_PLAYER\_END to aLEADERBOARD\_PLAYER STEP -2

'DebugOut("PlayerRankPTR=",#\_LeaderBoard\_PlayerPTR)

read \_LeaderBoard\_PlayerPTR, WORD \_LeaderBoardW

DebugOut("Player=",#\_LeaderBoardW)

\_LeaderBoard\_TimePTR = \_LeaderBoard\_PlayerPTR + aLEADERBOARD\_TIME - aLEADERBOARD\_PLAYER

read \_LeaderBoard\_TimePTR, WORD \_LeaderBoardW

DebugOut(", Time=",#\_LeaderBoardW,cr,lf)

next \_LeaderBoard\_PlayerPTR

#endrem

return

#EndRegion

#Region "Subroutine CheckBattery"

CheckBattery:

GetVdd

VoltsByVdd(VoltsCell2,3) 'Get half battery voltage on ADC3 and store in VoltsCell2 as mV.

VoltsByVdd(VoltsCell1,2) 'Get cell voltage on ADC2 and store in VoltsCell1 as mV.

VoltsCell2 = VoltsCell2 \* 2 - VoltsCell1 'Convert ADC3 input to Cell2 voltage.

'DebugOut("Vdd: ",#Voltage\_Vdd,"mV",cr,lf)

'DebugOut("Cell 2: ",#VoltsCell2,"mV",cr,lf)

'DebugOut("Cell 1: ",#VoltsCell1,"mV",cr,lf)

if VoltsCell2 > VOLTS\_CELL\_GOOD AND VoltsCell1 > VOLTS\_CELL\_GOOD then

'DebugLine("Battery cells good")

else

if VoltsCell2 < VOLTS\_CELL\_EXHAUSTED OR VoltsCell1 < VOLTS\_CELL\_EXHAUSTED then

'DebugLine("Battery cells exhausted")

PlayMP3(0069,"System shutting down due to low battery")

sleep 0 'Enter low power state, essentially off.

else

'DebugLine("Battery cells low")

PlayMP3(0068,"Warning, battery is below 20%")

endif

endif

return

#Endregion

#Region "Subroutine DebugGameStatus"

#ifndef Debugger\_On

DebugGameStatus:

Return

#else

DebugGameStatus:

DebugOut("GameStatus: ")

select case GameStatus

case GAME\_NOT\_SET

DebugOut("NotSet")

case GAME\_WAITING

DebugOut("Waiting")

case GAME\_BEGINNING

DebugOut("Beginning")

case GAME\_NOT\_STARTED

DebugOut("NotStarted")

case GAME\_PLAYING

DebugOut("Playing")

case GAME\_ENDING

DebugOut("Ending")

case GAME\_PRACTISE

DebugOut("Practising")

endselect

DebugOut(", Direction: ")

select case GameDirection

case L\_TO\_R

DebugLine("Left to Right")

case R\_TO\_L

DebugLine("Right to Left")

else

DebugLine("Not Set")

endselect

return

#endif

#Endregion

#Region "Subroutine DebugGameInputs"

#ifndef Debugger\_On

DebugGameInputs:

Return

#else

DebugGameInputs:

DebugOut("GameInputs: ", #GameInputs, " ")

if GameInputs = 128 then

DebugLine("Nothing")

else

if InputBuzz = CONTACT then

DebugOut("BuzzWire ")

endif

if InputPosR = CONTACT then

DebugOut("Right ")

endif

if InputPosL = CONTACT then

DebugOut("Left ")

endif

if RaceButton = PRESSED then

DebugOut("Race ")

endif

if HelpButton = PRESSED then

DebugOut("Help")

endif

DebugOut(cr,lf)

endif

return

#endif

#Endregion

#Region "License"

#Rem

MIT License

Copyright (c) 2024 Alan Hunt

Permission is hereby granted, free of charge, to any person obtaining a copy

of this software and associated documentation files (the "Software"), to deal

in the Software without restriction, including without limitation the rights

to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

SOFTWARE.

#EndRem

#EndRegion

# Program Listing – “Debugger.basinc”

#Region "Read Me"

#REM

File: "Debugger.basinc"

License: MIT (See end of file)

Change History: 2024/10/1 by Alan Hunt - Improved help text.

2024/9/30 by Alan Hunt - 1st issue.

Description

===========

This simple module provides:

\* A debugging feature that can easily be turned on or off using a single compiler directive.

\* Improved program readability in addition to debugging.

\* An efficient and visually easy way to represent a byte (or port) as individual bit values.

\* The option of having 2 debug levels.

It can be useful to easily turn off debugging because this avoids serial transmissions delays that could affect program performance. These transmissions also temporarily disable software interrupts too.

The statements in this module start with "Debug", which I find it easier on the eye than sertxd. In fact the sertxd command is being used, so any information about program execution points or particular program state can be sent. Hopefully, you will find a key benefit of using brief text is that the debug messages help static reviews of the program too, such as ``DebugLine("Not started after 10S")``.

Taking this further, it's nice to see macros with text explanations highlighted by the PICAXE Editor, rather than over use of comments in your program. For example, it would be awful to remember or lookup the contents of audio track 112, but this line makes it very clear ``PlayNow(0112, "The cell voltages are")``. In the PlayNow macro you can use the text with debugging statements and comment out these lines when things are working, or turn all the debugging off in one go. Whatever you choose the PICAXE Editor keeps on displaying the bright red text as part of your program.

The DebugBinary(val) statement can be used when a byte is holding important information in individual bits. For readability the byte is written as two nibbles separated by a space. For example, to view logic level inputs on port C it's just ``DebugBinary(pinsC)``, which would show "0010 0001" if C.0 and C.5 were high.

The Debugger's sertxd output is the usual way to perform PICAXE debugging because it sends text or data to the PICAXE Editor Terminal window with the same connection as the programming interface. Most of the functionality in this module is achieved by creating a unique alias for sertxd. When debugging is turned off this alias becomes comment lines instead off sertxd. Any sertxd commands in your program remain intact, so these can provide always-on basic information in addition to turning on or off detailed debugging. However, in normal situations, just using the Debugger statements is recommended.

In comparison to the Debugger/sertxd output, the PICAXE debug command only performs a large snapshot of variable values. For an X2 chip with default clock rate the debugger/sertxd output will send 10 bytes in 15mS, compared to the debug output which takes 180mS, without any indication of the execution points. For M2 chips at 4800bps the times will double. The PICAXE Editor will also try to disable its Terminal when debug is active, rendering sertxd useless. Debugging output would then need to be directed to another pin using serout/hserout and to a different com port and Terminal on the PC.

Programming Guide

==================

The following statements are defined in this module:

DebugOut This is an alias for sertxd, so use in the same way, e.g. ``DebugOut("Player ",#player," Wins",cr,lf)``.

DebugBinary(val) This is a 2 line macro that displays a byte value as two nibbles, e.g. ``DebugBinary(18)`` displays "0001 0010".

NB: It's very useful to display the status of 8 bit input/output ports, e.g. ``DebugBinary(pinsC)``.

DebugLine(msg) This is a 1 line macro that takes text and adds a CR and LF to it, e.g. ``DebugLine("@Initialising")``.

This module uses just a few lines of pre-processor directives, with no program memory space unless the debug statements are used and debugging is turned on.

The debugging information can be descriptive text, such as status or program position, and/or variables and constants. For example, ``Debugger("@Interrupt")`` would indicate the passing of a significant program execution point. To output a value as text use the "#" sign, such as ``Debugger("Battery = ",#BattVoltage," mV",cr,lf)``. The cr and lf provide a carriage return and line feed to move the cursor down to the start of the next line.

The Debugger statements are turned off in your main program with the compiler directive ``#define no\_debugger`` and turned on by just commenting out that line. The compiler directive must be defined before the include statement for this file.

Variable b0 is used by this module to provide the byte to binary string conversion. To remind yourself, it's recommended that you add the line ``symbol \_Debugger = b0`` in your program.

The PICAXE Editor will take some time to open its Terminal window for the debug output, so a two second pause is recommended in your program initialisation before its first use.

The problem with keeping both sertxd commands and debug statements in your program is that debugging can't be fully turned off. The recommended approach is to rely only on the Debugger statements, just create them and then comment out one's that were used for detailed debugging.

Debugging quickly consumes program memory space, so text should be kept as short as possible and debugging that is no longer necessary should be commented out.

Below is a sample configuration, assuming the following program structure:

1) Program Notes

2) Compiler Directives

3) Resources

3a) Pins

3b) Variables

3c) Constants

3d) Modules

4) Initialisation

5) Main:

6) Interrupt:

7) Subroutines

Sample Configuration

====================

'Compiler Directives

#picaxe 20X2

#no\_data

#no\_table

#no\_debug

#terminal 9600

'#define no\_debugger 'Comment out this line to enable Debugging.

'Variables

symbol \_Debugger = b0 '"Debugger.basinc" module is limited to using b0 for Byte to ASCII bitstream conversion.

'Modules

#include "Debugger.basinc"

'Initialisation

pause 2000 'Wait for the PICAXE Editor Terminal window to come online.

#EndRem

#EndRegion

#Region "Program"

'If ``#define no\_debugger`` is present the pre-processor comments out the Debugger statements.

#ifndef no\_debugger

#define DebugOut sertxd

#define DebugBinary(val) b0=val : sertxd(#bit7,#bit6,#bit5,#bit4," ",#bit3,#bit2,#bit1,#bit0)

#define DebugLine(msg) sertxd(msg,cr,lf)

#else

#define DebugOut 'Replaced with comment.

#define DebugBinary(val) 'Replaced with comment.

#define DebugLine(msg) 'Replaced with comment.

#endif

#Endregion

#Region "License"

#Rem

License

=======

MIT License

Copyright (c) 2024 Alan Hunt

Permission is hereby granted, free of charge, to any person obtaining a copy

of this software and associated documentation files (the "Software"), to deal

in the Software without restriction, including without limitation the rights

to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

SOFTWARE.

#EndRem

#EndRegion

# Program Listing – “Timing.basinc”

#Region "Read Me"

#REM

File: "Timing.basinc"

License: MIT (See end of file)

Change History: 2024/10/4 by Alan Hunt - Added the error handling of #terminal if it is already defined in the main program.

2024/10/4 by Alan Hunt - 1st issue.

Description

===========

This software was originally used for a game where the clock rate hadn't been finalised and the audio interface required consistent timings to be near minimum levels. Modifying more than a few pause commands by hand is unwieldy and error prone, particularly because arithmetic like "Pause 1 \* ClockMultiplier" is not allowed with the pause commands. Ultimately, a short list of logarithmic timings was easy to remember and they provided all the accuracy needed on most projects. It became a bit of fun to delve deeper into PICAXE timing and record some timing on an oscilloscope and this module emerged. This module's list of pause statements doesn't consume any extra program memory, and the module has these features:

\* A list of pause statements that provide timing delays regardless of the PICAXE clock rate: Pause1mS, Pause3mS, Pause10mS, Pause30mS, Pause100mS, Pause300mS, Pause1S, Pause3S and Pause10S.

\* The macro ``Pause\_mSec(t)`` allows you to specify specific pause times regardless PICAXE clock rate.

\* The PICAXE clock rate is set using a kHz value: 31, 250, 500, 1000, 2000, 4000, 8000, 16000, 32000, or 64000 (Dependent on chip).

\* The debug terminal baud rate is automatically set to the appropriate value for the PICAXE clock rate.

\* For X2 chips, the "timer" variable is set to increment at a configured rate if ``TIMER\_IN\_HUNDRETHS`` is defined in your program and given a value. This proved useful to measure lap time in a game.

\* A reference list of actual pause timings is shown in the Programming Guide notes below, these are valid if you use this module, or just use regular pause commands and do integer math based on the ratio of actual clock rate to default clock. From this some subjective recommendations are made for minimum clock rates with certain durations. Just don't expect a 1mS delay to be added when you use ``pause 1`` :)

Limitations

===========

Processing Delays

-----------------

There are processing delays around a pause statement, just to process each line of a program. For example, an output toggling test with ``high b.1`` immediatley followed by ``low b.1`` created a 267uS pulse width with a 20X2 or 419uS with a 20M2 when using their default clock rates. Please note, if you just wanted an accurate or short time pulse output then you would use the ``pulsout pinName, timeWord``, it supports 10us increments based on either an M2 or X2 running at 4MHz.

Continuing with the testing and processing delays, the processing time is approximately doubled when inserting a ``pause 1 `` between the high and low sequence. The test pulse then became 1.50mS with a 20X2 and 2.1mS with an 20M2, instead of the idealistic 1.00mS. The 2 lines of processing delay grow significantly at low clock rates. For an 20X2 with a 31kHz clock the ClockDivider is 256, so ``pause 1`` and ``pause 2`` should cause a 256mS and 512mS pulse, but the results were 371mS and 633mSm respectively. They were roughly 256mS apart but with an added 115mS. Even with a minimum ``pauseus 1``, this added 54mS to the output toggling test. Changing to an X2 running at 64MHz the ClockMultiplier is 8, so 125us and 250us expected, but 185uS and 316us totals were seen, so the gap is still about right and the processing delays have come down to 60us.

So it's worth keeping in mind that: incrementing pause values gives quite accurate results; the pause command introduces a procesing delay in addition to its timed delay; the processing delays are a large factor for small pause times or slow clock rates; and high clock rates will make pauses more accurate, albeit at the cost of power and heat.

It would be possible to compensate for processing delays in most situations, but it should not be necessary. A pause statement may be only one of many statements each causing their own processing delay. If more accurate delays are needed, you can take a measurement and make a delta change if the clock rate is fixed. Just remember to check timings again if you modify any of your program between the events you're trying to pause.

Available Clock Rates

---------------------

Parameters for CLOCK\_SET\_KHZ have chipset limitations as defined in PICAXE documentation, see Manual 2, setfreq. Only the common internal clock rates are included in this module: K31, K250, K500, M1, M2, M4, M8, M16, M32 and M64, where K=kHz and M=MHz.

Other Factors

-------------

When using clock rates below the chips default, the accuracy of integer division needs to be considered. This module compensates for most of this by scaling up to use ``pauseus``. Short delays are realy not be possible anyway due to the processing time mentioned earlier. Another key consideration is the granularity of the Pause\_mS(t) macro at lowered clock rates. Its simplistic implementation means the macro will only affect changes if the value of t increases by the modulus of the clock divisor amount, so if an X2 is clocked at 1MHz instead of the default 8MHz, the actual pause value increments at 8, 16, 24 etc.

Unfortunately, when using a clock rate of 31kHz it's impossible to use the sertxd command with the PE (PICAXE Editor) Terminal. The data is still being sent at 37.5bps, so it takes a very long time, but the minimum baud rate for the PE Terminal is 110 bps.

Programming Guide & Timing Tests

================================

Below is a list of statements defined in this module, these can be used just like any other PICAXE command if this module is included. The statements are equivalent to using ``pause t \* ClockRatio`` when using integer maths and converting some statements to ``pauseus t \* ClockRatio \* 100`` to help with scaling of low clock rates. The timings shown are based on the test described in the Processing Delays section above. The recommended minimum clock rates are subjective, based on needs:

Pause1mS Minimum recommended clock rate is 8MHz, the X2 default:

- 20X2@8MHz(Default) ``Pause 1`` and ``Pause1mS``=1.50mS, @64MHz=1.07mS.

- 20M2@4MHz(Default)=2.11mS, @8MHz=1.55mS, @32MHz=1.15mS

Pause3mS Minimum recommended clock rate is 4MHz, the M2 default:

- 20X2@4MHz=3.98mS, @8MHz=3.56mS, @64MHz=3.08mS.

- 20M2@4MHz=4.23mS, @8MHz=3.66mS, @32MHz=3.15mS

Pause10mS Minimum recommended clock rate is 1MHz:

- 20X2@1MHz=14.1mS, @8MHz=10.6mS, @64MHz=10.1mS.

- 20M2@1MHz=14.4mS, @4MHz=11.4mS, @8MHz=10.6mS, @32MHz=10.2mS

Pause30mS Minimum recommended clock rate is 500kHz.

- 20X2@500KHz=38.3mS, @8MHz=30.6mS.

- 20M2@500kHz=40.1mS

Pause100mS Minimum recommended clock rate is 250kHz.

- 20X2@250kHz=118.6mS, @8MHz=100.7mS.

- 20M2@250kHz=120.1mS

Pause300mS Use any clock rate.

- 20X2@31kHz=418.4mS, @250kHz=318.7mS.

- 20M2@31kHz=477.8mS, @250kHz=320.2mS.

Pause1S Use any clock rate.

Pause3S Use any clock rate.

Pause10S Maximum clock rate is 32MHz due to word variable overflow at 64MHz.

This macro is available for use from this module:

Pause\_mSec(t) This macro takes a variable or constant "t" and conveniently creates the pause according to the clock rate.

A good use of this macro is where you need to increment a pause fairly accurately, for example ``Pause\_mSec(1)``=2.15ms, whereas values above 1 add precisely 1.024mS per step. The macro also provides a quick and dirty solution when playing with clock timing, but other than that this macro should be avoided. The issue is that it consumes program memory and speed/accuracy in addition to requiring "\_Timing" to be defined to a spare word variable, for example "#define \_Timing w21". For fixed pause times that are not already included in this file, you can create your own based on the definitions of the standard delays shown in this file. They are calculated in pre-compilation, which avoids memory usage and calculation delays.

This module requires the type of PICAXE chip to be declared in your program, which is normal practise, e.g. #picaxe 20X2.

If you have ``#terminal`` defined in your main program you should delete or comment out this line. The #terminal directive is defined within this module with the optimal baud rate for your clock rate. If the terminal is not open it should open after a program download, but the PE will not change the Terminal baud rate if the Terminal is already open. So if you download a new clock rate, either close the Terminal beforehand or change the baud rate manually.

As shown in the sample configuration below, your program needs to "#define" pre-processor directives before the inclusion of this file:

Sample Configuration

====================

#Region "Resources"

'Variables

'Timing.basinc Variables

#define \_Timing w21 'Assign symbol to any spare Word.

'Constants

'@Timing.basinc Constants

'CLOCK\_SET\_KHZ options: 31,250,500,1000,2000,4000,8000,16000,32000,or 64000 (Dependent on chip). No equals sign and nothing after number.

#define CLOCK\_SET\_KHZ 16000

'The defintion below is optional, if present settimer is used to set the major tick rate. The example below makes the Timer variable increment every tenth of a second.

#define TIMER\_IN\_HUNDRETHS 10

'Modules

#include "Timing.basinc"

#Endregion

#EndREM

#EndRegion

#Region "Program"

'Determine the Default Clock Rate in kHz

'=======================================

'Your program should specify the chip that it's aligned to, for example ``#picaxe 20X2`` will define "\_20X2".

#ifdef \_20X2

#define CLOCK\_DEFAULT\_KHZ 8000

#elseifdef \_28X2

#define CLOCK\_DEFAULT\_KHZ 8000

#elseifdef \_40X2

#define CLOCK\_DEFAULT\_KHZ 8000

#else

'The chip is assumed to have a 4MHz default clock if the preceding if commands are false.

#define CLOCK\_DEFAULT\_KHZ 4000

#endif

'Determine and Set key parameters

'================================

'NB: TerminalRate is the bit rate for the Serial Out pin, as used for debugging with the sertxd command.

#ifndef CLOCK\_SET\_KHZ

#define CLOCK\_SET\_KHZ CLOCK\_DEFAULT\_KHZ

#endif

#if CLOCK\_DEFAULT\_KHZ = 8000

#if CLOCK\_SET\_KHZ = 31

#define ClockDivider 256

setfreq K31

'NB: PE Terminal will not operate at 37.5 bps but PICAXE still sends.

#elseif CLOCK\_SET\_KHZ = 250

#define ClockDivider 32

setfreq K250

#terminal 300

#elseif CLOCK\_SET\_KHZ = 500

#define ClockDivider 16

setfreq K500

#terminal 600

#elseif CLOCK\_SET\_KHZ = 1000

#define ClockDivider 8

setfreq M1

#terminal 1200

#elseif CLOCK\_SET\_KHZ = 2000

#define ClockDivider 4

setfreq M2

#terminal 2400

#elseif CLOCK\_SET\_KHZ = 4000

#define ClockDivider 2

setfreq M4

#terminal 4800

#elseif CLOCK\_SET\_KHZ = 8000

#define ClockMultiplier 1

setfreq M8

#terminal 9600

#elseif CLOCK\_SET\_KHZ = 16000

#define ClockMultiplier 2

setfreq M16

#terminal 19200

#elseif CLOCK\_SET\_KHZ = 32000

#define ClockMultiplier 4

setfreq M32

#terminal 38400

#elseif CLOCK\_SET\_KHZ = 64000

#define ClockMultiplier 8

setfreq M64

#terminal 76800

#endif

#else

#if CLOCK\_SET\_KHZ = 31

#define ClockDivider 128

setfreq K31

'NB: PE Terminal will not operate at 37.5 bps but PICAXE still sends.

#elseif CLOCK\_SET\_KHZ = 250

#define ClockDivider 16

setfreq K250

#terminal 300

#elseif CLOCK\_SET\_KHZ = 500

#define ClockDivider 8

setfreq K500

#terminal 600

#elseif CLOCK\_SET\_KHZ = 1000

#define ClockDivider 4

setfreq M1

#terminal 1200

#elseif CLOCK\_SET\_KHZ = 2000

#define ClockDivider 2

setfreq M2

#terminal 2400

#elseif CLOCK\_SET\_KHZ = 4000

#define ClockMultiplier 1

setfreq M4

#terminal 4800

#elseif CLOCK\_SET\_KHZ = 8000

#define ClockMultiplier 2

setfreq M8

#terminal 9600

#elseif CLOCK\_SET\_KHZ = 16000

#define ClockMultiplier 4

setfreq M16

#terminal 19200

#elseif CLOCK\_SET\_KHZ = 32000

#define ClockMultiplier 8

setfreq M32

#terminal 38400

#elseif CLOCK\_SET\_KHZ = 64000

#error "An M2 chip cannot clock at 64MHz"

#endif

#endif

'Determine ClockRatio and Tick Rate and define Pause\_mSec(t) macro

'=================================================================

'NB: ClockRatio is a pre-processor symbol that is used as a multiplier or divisor in timing symbols to generate pause statements.

'NB: For X2 chips, TickRate is the incrementing rate for the system "timer" variable.

#ifdef ClockMultiplier

symbol MinorTickuS = 32 / ClockMultiplier 'Minor ticks = 256/Clock (Default=32uS, x2=16uS, x4=8uS, x8=4uS).

#define ClockRatio \* ClockMultiplier

#ifdef \_Timing then

#define Pause\_mSec(t) \_Timing = t \* ClockMultiplier: pause \_Timing

#endif

#else

symbol minorTickuS = 32 \* ClockDivider 'Minor ticks = 256/Clock (/2=64uS, /4=128uS, /8=256uS, /16=512uS, /32=1024uS, /256=8096uS).

#define ClockRatio / ClockDivider

#ifdef \_Timing then

#define Pause\_mSec(t) \_Timing = t / ClockDivider: pause \_Timing

#endif

#endif

'If an X2 chip then set the timer rate

'=====================================

#if CLOCK\_DEFAULT\_KHZ = 8000

#ifdef TIMER\_IN\_HUNDRETHS then

symbol MinorTicksIn\_10mS = 10000 / MinorTickuS

symbol TicksRequired = TIMER\_IN\_HUNDRETHS \* MinorTicksIn\_10mS

symbol TickToSet = 65536 - TicksRequired

settimer TickToSet 'Sets incrementing rate for the system "timer" word variable.

'For example, at 16MHz the "timer" word value increments every 0.1S with a setting of 59,286 (65,536-(100,000/minorTickuS).

#endif

#endif

'Pre-Processor definitions for standard pause statements

'=======================================================

'NB: These pause deifintions work regardless of clockrate, just enter a statement like "Pause10mS" in your program:

'A "pauseus" has 10uS units at default clock rate. It's used here to provide better accuracy and low range.

symbol ustime1mS = 100 ClockRatio

#define Pause1mS pauseus ustime1mS

symbol ustime3mS = 300 ClockRatio

#define Pause3mS pauseus ustime3mS

symbol ustime10mS = 1000 ClockRatio

#define Pause10mS pauseus ustime10mS

symbol ustime30mS = 3000 ClockRatio

#define Pause30mS pauseus ustime30mS

'This prevents a word overflow when the chip is clocked at 8 times its default.

#ifdef ClockMultiplier

symbol time100mS = 100 ClockRatio

#define Pause100mS pause time100mS

#else

symbol ustime100mS = 10000 ClockRatio

#define Pause100mS pauseus ustime100mS

#endif

'This extends the use of pauseus for low clock rates.

#ifdef ClockDivider

symbol ustime300mS = 30000 ClockRatio

#define Pause300mS pauseus ustime300mS

#else

symbol time300mS = 300 ClockRatio

#define Pause300mS pause time300mS

#endif

'A "pause" has 1mS units at default clock rate. Using pause here instead of pauseus gives top range.

symbol time1S = 1000 ClockRatio

#define Pause1S pause time1S

symbol time3S = 3000 ClockRatio

#define Pause3S pause time3S

symbol time10S = 10000 ClockRatio

#define Pause10S pause time10S

#Endregion

#Region "License"

#Rem

MIT License

Copyright (c) 2024 Alan Hunt

Permission is hereby granted, free of charge, to any person obtaining a copy

of this software and associated documentation files (the "Software"), to deal

in the Software without restriction, including without limitation the rights

to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

SOFTWARE.

#EndRem

#EndRegion

# Program Listing – “DF\_Player\_Mini.basinc”

#Region "Read Me"

#REM

File: "DF\_Player\_Mini.basinc"

License: MIT (See end of file)

Change History: 2024/10/16 by Alan Hunt - 1st issue.

Description

===========

Designed for the DF Player Mini audio board, this module provides macros and subroutines to speak numbers and play audio tracks to emulate speech and play music and sound effects.

The DF Player module does not behave entirely as expected, so this module eases its use. The busy pin takes time to become active and effectively reports audio output without file seek time delays. Testing also found the minimum pause times between different instructions, so this module takes care of that too and shares the info with inline comments. Using the module should mean that individual words can be sequenced to be as quick and reliable as practical.

The list of macros include:

PlayMP3(Track, Title) Plays the Track after the previous track is finished. Title is for debugging/information.

PlayNow(Track, Title) Stops anything currently played or queued and plays the selected Track.

PlayAdvert(Track, Title) Plays the Advert and temporarily suspends anything currently playing.

AudioSpeakNumber(Val) Speaks the number Val, which is a Word value from 0 to 65,535.

AudioSpeakTenths(Val) Speaks the number Val divided by 10

AudioSpeakHundreths(Val) Speaks the number Val divided by 100

AudioSpeakThousandths(Val) Speaks the number Val divided by 1000

The list of subroutines include:

AudioInitialise To be included in program initialisation.

AudioWaitForIdle Wait for sound to stop.

AudioClear Stop sound.

\_AudioPlayMP3 Internal routine to play the track defined by \_Audio\_MP3Track.

\_AudioPlayAdvert Internal routine to play the advert defined by \_Audio\_AdvertTrack.

\_AudioSpeakInteger Internal routine to speak the number defined by \_Audio\_Integer.

\_AudioSpeakDecimalPlaces Internal routine to speak a number as individual digits, as spoken after a decimal place.

Limitations

-----------

The DF Player Mini comms is 9,600bps, so the minimum clock speed for a PICAXE chip is 8MHz, this is the X2 default whereas an M2 needs setting to run faster than default.

The DF Player Mini can support several folders and thousands of files but this module is limited to using the default MP3 and ADVERT folders, so a maximum of 256 files in each.

Even with trimming silences, chaining tracks with individual words is a little staccato. For personal use it's quick and very clear but not smooth, or so impressive for others. To get the best results use joined phrases, such as "The total solar power is" {n} "kilo watt hours over the last week". Otherwise, you need solutions to replace the DF Player, such as online interfaces or a far more powerful speech synthesis device.

Dependencies

------------

Certain pause statements and symbols must be defined for timing, pin usage and variable usage, see the Programming Guide section.

About the DF Player Mini

========================

Sourcing the hardware

---------------------

The DF Player Mini can be obtained from many sources and in many variants. Some of these are unreliable or appeared faulty. Here's a summary of types I compared:

1) Good: The PICAXE SPE035, labelled "DF Player Mini", with processor "YX5200-24SS, OE64 1607" provided good results and this utility is based on its timings. Like other units, completion of a track is reported twice from its serial output and the track number appears to be a random number, which is perhaps the file position in the SD card FAT.

2) Good: The "DF Player Mini, VO5.1, HW-247A" with processor "TD5580A". Initially I thought only 1 of 3 modules worked but this was due to timing differences from (1). The differences I noted were that: Boot-up delays were more than 100mS, so the subroutine AudioIniailise now pauses for 1 second; the Busy signal went active earlier but stayed on later, typically 160mS after the modules track completion message; and there's a 10ms gap in the double reporting of track completion.

3) Bad: A batch labelled "MP-TF-16P-V3.0" with processor "MH2024K-24SS, 240113" were not usable. They were very slow, only reported track completion occasionally and did not reliably accept track requests.

4) Based on the findings above I'd buy from an electronics store unless I had time on my hands to check cheap packs from eBay, Amazon, AliExpress etc.

Some Reference Notes

--------------------

The DF Player Mini is loaded with an SD card containing .mp3 or .wav tracks in the root MP3 folder, these are then easily selected for playback. The filename simply must begin with a number, I use 0000 to 0255 followed by a space and a short description. Likewise, numbered Advert tracks can also be played and these temporarily suspend the main playing track. The Adverts have uses such as playing a crash sound during a game and they are placed in the root ADVERTS folder. Adverts are a really cool feature!

\* The "DF Player Mini" supply is 3.2V to 5.0V. TTL is 3.3v level so a 1kOhm series must be placed between its Rx pin and 5V signals.

\* At startup the module may need up to 4 seconds to boot and load a very long file list.

\* The Busy signal only indicates audio output. After requesting a track to be played there's a delay before it goes active.

\* Various commands take time to respond or need short wait periods before another command is supplied. This module takes care of it and includes comments on them.

\* The serial comms is True(Idle high), 9600 bps. This module needs the baud rate parameter adjusting when PICAXE clock speed changes from 8MHz.

\* The command packet format is: $7E(Start), $FF(Version), $06(Length), $xx(Command), $00(No feedback), $xx(Param1), $xx(Param2), {Optional 2 byte checksum},$EF(End)

- Normally the optional checksums are not used, the packet length stays as $06 regardless.

- If required, the checksum is easily calculated as zero minus each of the byte values in the message.

\* Manuals are widely available on the Internet, including https://picaxe.com/docs/spe033.pdf.

\* A simple starter kit is shown here: https://picaxe.com/docs/spe035.pdf.

\* A nice little weatherproof speaker is the Eagle B185.

\* The speaker volume is software controllable, and the maximum is very loud indoors. Outdoors it won't carry far in a noisy environment.

\* Playing an Advert when an advert is already running seems to either clear the underlying track or cause longer delays on the Busy signal (I haven't investigated this).

Programming Guide

==================

The main 3 macros PlayMP3, PlayNow and PlayAdvert include a Title parameter, which allows short descriptive text to be passed. The Title can greatly improve readability of your program, and it consumes no program memory if debugging remains commented out, e.g. if track 29 is configured to say "thousand" the program statement to use it is PlayMP3(29,"thousand"). The Title output can be helpful when debugging is enabled, but program memory will diminish very quickly, so the text should be short.

This module requires the following pause statements to be defined Pause3mS, Pause30mS, Pause100mS, Pause300mS and Pause1S. These can be defined in your main program with pre-processor directives like '#define Pause3mS pause 3'; or instead, you can '#include "Timing.basinc"' before this file, which will provide these statements and give consistent timing across different clock rates.

When changing PICAXE clock rates be sure to change the AUDIO\_BAUD constant too, valid settings are T9600\_8, T9600\_16, T9600\_32 and, if using X2 chips, T9600\_64.

Using phrases instead words for most tracks will greatly reduce program memory space. It will also make speech smoother and the code more readable with fewer lines.

When reading voltage levels it's tempting to use millivolts but this adds extra words like "thousand", "and" and "hundred". If you have a variable representing millivolts, try using AudioSpeakThousandths(Val) because "12325" would be spoken as separate digits like twelve-point-three-two-five, which is quicker and seems natural with a little staccato.

Your program can and probably should work asynchronously with the DF Player. In the sample config below, you can see the DF Player Busy Pin is connected to the PICAXE B.2 input, which is symbolised by AUDIO\_STATUS; and the Busy Pin idle state of 1 is symbolised by AUDIO\_IDLE. So for example, in a main program loop you can test to see if a music track is finished and play the next track with "if AUDIO\_STATUS = AUDIO\_IDLE then {.....} endif". Taking this a stage further you could probably trigger interrupts on the pin represented by AUDIO\_STATUS but take care of time delays in this going active and inactive.

You should be careful that your program isn't blocked unintentionally by the PlayMP3(Track, Title) macro. This macro waits for any existing track to complete before it sends its track request to the DF Player. PlayNow is an alternative macro, but this simply stops anything being played and plays what it wants.

My preferred program structure is:

1) Program Notes

2) Compiler Directives

3) Resources

3a) Pins

3b) Variables

3c) Constants

3d) Modules

4) Initialisation

5) Main:

6) Interrupt:

7) Subroutines

Based on the program structure above, here is an example configuration for this module:

#Region "Resources"

'Pins

symbol AUDIO\_TX = B.0 '@DF\_Player\_Mini.basinc: PICAXE Serial Tx to DF Player via 1K resistor.

symbol AUDIO\_STATUS = pinB.2 '@DF\_Player\_Mini.basinc: Busy indication from DF Player is active low after about 230mS.

'Variables

'@DF\_Player\_Mini.basinc Variables

symbol \_Audio\_Digit = b43 'Assign symbol to any spare Byte.

symbol \_Audio\_Integer = w22 'Assign symbol to any spare Word.

symbol \_Audio\_Fraction = w23 'Assign symbol to any spare Word.

symbol \_Audio\_Track = b48 'Assign symbol to any spare Byte.

symbol \_Audio\_Byte = b49 'Assign symbol to any spare Byte.

'Constants

'@DF\_Player\_Mini.basinc Constants

symbol AUDIO\_VOLUME = 15 '0 to 31(Maximum). You can assign this symbol to a variable if you don't want fixed volume.

symbol AUDIO\_BAUD = T9600\_8 'Tx logic is idle high (T) and 9600 bps, adjust the last character(s) to your clock speed in MHz.

symbol AUDIO\_ACTIVE = 0 'The AUDIO\_STATUS pin is active low, indicating audio playing.

symbol AUDIO\_IDLE = 1 'AUDIO\_STATUS high indicates the module is idle OR seeking a track to play on the disk.

'Modules

#include "Timing.basinc"

#include "DF\_Player\_Mini.basinc"

'Initialisation

gosub AudioInitialise 'Prepares PICAXE output pin, Resets the DF Player, selects SD input and sets volume.

#Endregion

#EndREM

#EndRegion

#Region "Program"

goto DF\_Player\_End 'This module only contains macro and subroutine definitions.

'Debugging

'=========

'Caution, debugging provides some nice info but causes serial transmission delays and consumes program space.

'#define DebugAudio\_On 'Comment out this line to turn debugging off.

#ifdef DebugAudio\_On

#define DebugAudio sertxd

#else

#define DebugAudio 'Do nothing

#endif

'Audio Macros

'============

#macro PlayMP3(Track, Title)

'DebugAudio("@PlayMP3 ",#Track,Title,cr,lf) 'Enabling this line consumes program space extremely quickly with track title info.

\_Audio\_Track = Track

gosub AudioWaitForIdle

gosub \_AudioPlayMP3

#endmacro

#macro PlayNow(Track, Title)

'DebugAudio("@PlayNow ",#Track,Title,cr,lf) 'Enabling this line consumes program space extremely quickly with track title info.

\_Audio\_Track = Track

gosub AudioClear

gosub \_AudioPlayMP3

#endmacro

#macro PlayAdvert(Track, Title)

'DebugAudio("@PlayAdvert ",#Track,Title,cr,lf) 'Enabling this line consumes program space extremely quickly with track title info.

\_Audio\_Track = Track

gosub \_AudioPlayAdvert

#endmacro

#macro AudioSpeakNumber(Val)

\_Audio\_Integer = Val

gosub \_AudioSpeakInteger

#endmacro

#macro AudioSpeakTenths(Val)

\_Audio\_Integer = Val / 10

\_Audio\_Fraction = Val % 10

gosub \_AudioSpeakInteger

PlayMP3(031,"point")

\_Audio\_Byte = 1

gosub \_AudioSpeakDecimalPlaces

#endmacro

#macro AudioSpeakHundreths(Val)

\_Audio\_Integer = Val / 100

\_Audio\_Fraction = Val % 100

gosub \_AudioSpeakInteger

PlayMP3(031,"point")

\_Audio\_Byte = 2

gosub \_AudioSpeakDecimalPlaces

#endmacro

#macro AudioSpeakThousandths(Val)

\_Audio\_Integer = Val / 1000

\_Audio\_Fraction = Val % 1000

gosub \_AudioSpeakInteger

PlayMP3(031,"point")

\_Audio\_Byte = 3

gosub \_AudioSpeakDecimalPlaces

#endmacro

'Audio Subroutines

'=================

AudioInitialise:

DebugAudio("@AudioInitialise",cr,lf)

high AUDIO\_TX 'Output pins are default low, so this prepares communication.

Pause1S 'Some types of DF Player need more than 100mS boot time.

serOut AUDIO\_TX, AUDIO\_BAUD, \_

(**$7E**, **$FF**, **$06**, **$0C**, **$0**, **$0**, **$0**, **$EF**) 'Command $0C (Reset)

Pause1S 'AUDIO\_ACTIVE after 220mS and module messages going to sleep at 900mS.

serOut AUDIO\_TX, AUDIO\_BAUD, \_

(**$7E**, **$FF**, **$06**, **$09**, **$0**, **$0**, **$2**, **$EF**) 'Command $09 (Playback source), Arg $0002 (SD card)

Pause1S 'Pause to read SD card

serOut AUDIO\_TX, AUDIO\_BAUD, \_

(**$7E**, **$FF**, **$06**, **$06**, **$0**, **$0**, AUDIO\_VOLUME, **$EF**) 'Command $06 (Set Audio volume)

Pause100mS 'Pause to ensure volume set

return

AudioWaitForIdle:

DebugAudio("@AudioWaitForIdle",cr,lf)

do while AUDIO\_STATUS = AUDIO\_ACTIVE

Pause3mS

loop

return

AudioClear:

DebugAudio("@AudioClear",cr,lf)

serOut AUDIO\_TX, AUDIO\_BAUD, \_

(**$7E**, **$FF**, **$06**, **$15**, **$0**, **$0**, **$0**, **$EF**) 'Command $15. Stop Advert(Crash), if playing.

Pause100mS '75mS required for settling time.

'NB: Contrary to manual, command $16 did not stop everything.

serOut AUDIO\_TX, AUDIO\_BAUD, \_

(**$7E**, **$FF**, **$06**, **$16**, **$0**, **$0**, **$0**, **$EF**) 'Command $16. Stop main track, if playing

Pause30mS '14mS required before requesting the new track

return

\_AudioPlayMP3:

DebugAudio("@\_AudioPlayMP3 - ",#\_Audio\_Track,cr,lf)

serOut AUDIO\_TX, AUDIO\_BAUD, \_

(**$7E**, **$FF**, **$06**, **$12**, **$0**, **$0**, \_Audio\_Track, **$EF**) 'Command $12. Play from MP3 folder.

do while AUDIO\_STATUS = AUDIO\_IDLE 'Ensures AUDIO\_STATUS caught up (seen delays up to 290mS)

Pause3mS

loop

return

\_AudioPlayAdvert:

'Known issue:

' - This applies to the PICAXE SPE035 hardware, but not the DF Player Mini, VO5.1, HW-247A.

' - When an advert tries to play over the top of another advert the underlying MP3 track is lost.

' - Finally, a bug I liked because it kept my Buzz Wire Game more active and interesting by shifting to the next track :)

DebugAudio("@\_AudioPlayAdvert - ",#\_Audio\_Track,cr,lf)

serOut AUDIO\_TX, AUDIO\_BAUD, \_

(**$7E**, **$FF**, **$06**, **$13**, **$0**, **$0**, \_Audio\_Track, **$EF**) 'Command $13. Play ADVERT (suspends current track while playing).

Pause300mS '170mS required because AUDIO\_STATUS goes Idle before main track resumed.

return

\_AudioSpeakInteger:

DebugAudio("@\_AudioSpeakInteger - ",#\_Audio\_Integer,cr,lf)

'Thousands

\_Audio\_Byte = \_Audio\_Integer / 1000

if \_Audio\_Byte > 0 then

gosub \_AudioSpeak1to99

PlayMP3(29,"thousand")

endif

'Hundreds

\_Audio\_Digit = \_Audio\_Integer dig 2

if \_Audio\_Digit > 0 then

PlayMP3(\_Audio\_Digit,"aDigit[0to9]")

PlayMP3(28,"hundred")

endif

'Units

\_Audio\_Byte = \_Audio\_Integer % 100

if \_Audio\_Integer > 99 AND \_Audio\_Byte > 0 then

PlayMP3(0030,"and")

endif

gosub \_AudioSpeak1to99

if \_Audio\_Integer = 0 then

PlayMP3(0,"zero")

endif

return

\_AudioSpeakDecimalPlaces:

DebugAudio("@\_AudioSpeakDecimalPlaces - ",#\_Audio\_Byte," places for ",#\_Audio\_Fraction,cr,lf)

dec \_Audio\_Byte

for \_Audio\_Digit = \_Audio\_Byte to 0 step -1

\_Audio\_Track = \_Audio\_Fraction dig \_Audio\_Digit

PlayMP3(\_Audio\_Track,"aDigit[0to9]")

next \_Audio\_Digit

return

\_AudioSpeak1to99:

'Speak numbers 1 to 99 (zero is not sounded here). NB: "dig" operator needs to be the first or only argument.

\_Audio\_Digit = \_Audio\_Byte dig 1

if \_Audio\_Digit >= 2 then

\_Audio\_Track = \_Audio\_Digit + 18

PlayMP3(\_Audio\_Track,"aDecade[20to90]") 'Twenty=20, Thirty=21 etc

\_Audio\_Track = \_Audio\_Byte dig 0

if \_Audio\_Track > 0 then

PlayMP3(\_Audio\_Track,"[0to9]")

endif

else

if \_Audio\_Digit = 1 then

\_Audio\_Track = \_Audio\_Byte dig 0 + 10

PlayMP3(\_Audio\_Track,"[10to19]")

else

\_Audio\_Track = \_Audio\_Byte dig 0

if \_Audio\_Track <> 0 then

PlayMP3(\_Audio\_Track,"[0to9]")

endif

endif

endif

return

DF\_Player\_End:

#Endregion

#Region "License"

#Rem

MIT License

Copyright (c) 2024 Alan Hunt

Permission is hereby granted, free of charge, to any person obtaining a copy

of this software and associated documentation files (the "Software"), to deal

in the Software without restriction, including without limitation the rights

to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

SOFTWARE.

#EndRem

#EndRegion

# Program Listing – “Voltage\_X2.basinc”

#Region "Read Me"

#REM

File: "Voltage\_X2.basinc"

License: MIT (See end of file)

Change History: 2024/10/23 by Alan Hunt - Corrected typos of Denominator to Dividend.

2024/10/22 by Alan Hunt - 1st issue.

Description

===========

This module provides some macros that calculate the power supply voltage and ADC input voltages with the 10bit ADC channels on an PICAXE X2 chip. This is achieved with good accuracy and minimal program space and there's advice for a calibration method included in the notes too.

Hopefully the macros are easy to use and understand, but sorry M2 chips are not supported.

The list of macros included:

GetVdd A word variable symbolised by Voltage\_Vdd is updated with the power supply voltage in mV.

SetVdd(Val\_mV) If the power supply is accurately known and stable, Voltage\_Vdd can be set manually.

VoltsByVdd(Var, Channel) Updates a word variable, Var, with the voltage in mV from the specified Channel number.

Limitations

-----------

M2 chips do not have the X2 operator "\*/" that is used here for simplification. It keeps the code simple and short by obtaining the middle word of a word multiplication, which goes most of the way towards making a division of 1024 while achieving good accuracy on the maths.

In the PICAXE Forum there are several solutions to calculate Vdd and ADC input voltages with some accuracy. There are methods to average ADC readings, or using large dividends and divisors to make the integer math easier, or a neat method of using the DAC to obtain higher value ADC readings for more accuracy etc. Sadly, there's no DAC on my 20X2 and I was extremely short on program space too, even after swapping out a 20M2.

The key problem in calculating Vdd seems to be the accuracy of the PICAXE 1.024V FVR (Fixed Voltage Reference). In my case it was 1.1% high, which is expected as per Microchip application note AN1072 and datasheet PIC18(L)F1XK22. If the power supply is known and stable it's preferable to use a fixed constant for Vdd to calculate ADC input voltages, which is what the SetVdd macro does. If the supply voltage varies, perhaps being connected directly to a battery, another option to improve accuracy is to calibrate, as shown in a section below.

Using the FVR to calculate Vdd presents another issue with the range of voltage uncertainty for each ADC step. For example, each step is 5mV with a 5V supply and there's perhaps 1 bit error making 10mV, then because the calculated Vdd is roughly 5 times larger this value could be 50mV out. A possible approach could be to reduce supply voltage making smaller ADC steps and less multiplication of uncertainty too. In terms of percentage accuracy it's also worth noting that you need lots of steps, so you can't measure small voltages with 1% accuracy and you'd probably want a minimum ADC input of 1V when using a 5V supply.

The good news is that the ADC does seem to be reasonably accurate and linear. Measured with two 20X2 chips, if anything the readings start a little early perhaps by 1mV and finish early too by about 3mV. At these voltage levels it's difficult to be accurate and noise will be a factor, but the readings were taken with an OWON XDM1041 bench multimeter and verified with a handheld digital multimeter. With 5.000V measured on the 20X2 PICAXE supply pins and using SetVdd(5000), the ADC was expected to read 1 between 2.4mV and 7.3mV but this reading started from a 1mV input. Also, an ADC reading of 1023 was expected after 4997mV but the reading started to be seen at 4993mV and was consistent at 4994mV. The table of the results is below:

ADC mV | ADC Val | VoltsByVdd mV Value

--------------------------------------

5000 1023 4995

4994 1023 4995

4993 1022 4990 (occasional 1023, 4995mV)

4000 819 3999

3000 614 2998

2000 410 2001 (some 409, 1997mV)

1000 205 1000

0200 41 200

0100 21 102

0050 10 48

0027 6 29

0021 5 24

0016 4 19

0012 3 14

0007 2 9

0006 1 4 (occasional 2, 9mV)

0002 1 4

0001 0 0 (occasional 1, 4mV)

0000 0 0

Dithering around ADC step changes is a commonly reported issue. I found capacitance on the ADC input didn't help much when there's only 2 obvious ADC values at play. If the readings are fluctuating more than this it probably points to the dynamics of your setup, like varying supply loads and ADC inputs, or noise.

How it Works

------------

The maximum ADC value is 1023 and in basic configuration this approximately represents the PICAXE supply voltage, Vdd. Actually, the specification states this should be a half value, so 1023 represents a voltage over 2047/2048 of Vdd, whereas a value of 1 should represent 1/2048 to 3/2048 of Vdd. In testing the tipping point to 1023, it was found to be just under 1023/1024 of Vdd, which is a good approximation.

To measure Vdd this module simply takes an ADC reading for the internal FVR of 1.024V and scales that up. The math is Vdd=1023\*1.024/Val\_ADC, but the ADC value is quite large divisor for integer maths. Dependent on the chip and its supply voltage range of perhaps 1.8V to 5.2V, the ADC reading will be between 581 (1023\*1.024V/1.8V) and 201 (1023\*1.024V/5.2V). The dividend is therefore scaled up to 10476, which is a rounding of 1023\*1.024\*10. The remainder of this division is then scaled up by 100 for the final part of the division. The integer result is then 1,000 times bigger and accurate to the last digit, meaning the answer for Vdd is given in millivolts.

To calculate an ADC input voltage, the ADC reading is just scaled according to Vdd with ADC\_Volts=Vdd\*ADC\_Val/1024. This formula gives the lowest possible voltage at the input. The math is performed with the X2 operator "\*/", which returns the middle word of a multiplication. This effectively divides the result by 256 and this is then divided by 4 to get the overall division of 1024.

Calibration

-----------

The fixed voltage reference of 1.024V can vary by several percent between chips and it's also dependent on supply voltage and temperature. To halve the supply voltage errors across the possible range, Microchip AN1072 recommends calibrating at 4V. Coincidently, 4V calibration is good when monitoring charge voltages on Li-ion cells too.

To calibrate, set the supply voltage to 4000mV as monitored on the PICAXE supply pins. Uncomment the line "'#define VoltageDebug\_On", which will output information to the PICAXE Editor Terminal, then repeatedly call GetVdd from your program with 1 second pauses. Note the ADC Value and adjust the supply up and down so you find voltages where the ADC value occasionally reads 1 higher and 1 lower. The midpoint between these voltages is used to calculate a constant named VoltageDividend, using the formula VoltageDividend=Midpoint\*ADC\_Value\*10. If your FVR is accurate the reading should be 10476. In my case, two 20X2 chips gave an ADC reading of 265, the edge voltages were 3991mV and 4004mV, the chosen midpoint was 3998mV, so VoltageDividend was 10595. The FVR spec for the 20X2 chip is -8% to +6% and the error seen was +1.1%.

Programming Guide

=================

You should use the ADCSETUP command to disable digital circuitry on the ADC Channels that you plan to use, see Manual 2.

You must use the GetVdd or SetVdd macro before using the VoltsByVdd macro. If you are using a stable regulated power supply, then it's only necessary to do this once. The SetVdd macro is preferred if you have an accurate and stable supply to the PICAXE, the macro can be also be used for testing too. If the PICAXE is connected to a battery where the supply voltage will change over time, then the GetVDD macro should be used immediately before every VoltsByVdd reading.

To debug values to the PICAXE Editor Terminal window just uncomment this line in the module "'#define VoltageDebug\_On".

When monitoring battery voltages be careful not to Phantom Power your circuit through voltage protection diodes on ADC channels. Ideally, batteries should be fully disconnected when your circuit is off to prevent battery drain, otherwise connect the ADC channel via a 1K to 10K resistor to protect the PICAXE chip. Microchip recommends a minimum source impedance of 10K for ADC inputs.

An example configuration is shown below and these statements should appear in your program before the macros are used:

#Region "Resources"

'Variables

'@Voltage\_X2.basinc Variables

symbol Voltage\_Vdd = w25 'Assign this symbol to any spare Word, it stores PICAXE supply voltage as mV.

symbol \_VoltageADCval = w26 'Assign this symbol to any spare Word, it's an internal variable for maths.

symbol \_VoltageModulus = w27 'Assign this symbol to any spare Word, it's an internal variable for maths.

#Constants

'@Voltage\_X2.basinc Constant

symbol VoltageDividend = 10595 'Adjust value for FVR inaccuracy, see "Voltage\_X2.basinc" Calibration.

'Modules

#include "Voltage\_X2.basinc"

#EndRegion

#EndREM

#EndRegion

#Region "Program"

goto VoltageEnd 'There is no code to execute, just macro definitions for the compiler.

'Debugging

'=========

'Caution, debugging provides some nice info but causes serial transmission delays and consumes program space.

'#define VoltageDebug\_On 'Uncomment this line to turn debugging on.

#ifdef VoltageDebug\_On

#define \_VoltageDebug sertxd

#else

#define \_VoltageDebug 'Do nothing

#endif

'Macros

'======

#macro GetVdd

\_VoltageDebug("@GetVdd")

calibadc10 \_VoltageADCval 'Read reference 1.024V (Value = 1023 \* 1.024V / Vdd)

Voltage\_Vdd = VoltageDividend / \_VoltageADCval \* 100 'Obtain first 2 digits, nnxx. Max=5200(5211), Min=1800(1803)

\_VoltageModulus = VoltageDividend // \_VoltageADCval 'Find remainder. Max=580, Min=0

Voltage\_Vdd = 100 \* \_VoltageModulus / \_VoltageADCval + Voltage\_Vdd 'Obtain last 2 digits, xxnn. Max=99(=100\*580/581), Min=0

\_VoltageDebug(" (Ref\_1.024V=",#\_VoltageADCval,", Vdd=",#Voltage\_Vdd," mV)",cr,lf)

#endmacro

#macro SetVdd(VoltsW)

\_VoltageDebug("@SetVdd (Val=",#VoltsW,")",cr,lf)

Voltage\_Vdd = VoltsW

#endmacro

#macro VoltsByVdd(VoltsW, Channel)

'The measured voltage is Vdd(1800 mV to 5200 mV) \* ADC\_Reading(0 to 1023) / 1024(ADC\_Range). NB: ADC range is approximately 1024

\_VoltageDebug("@VoltsByVdd")

readadc10 Channel, \_VoltageADCval 'Read ADC Channel, the range is 0 to 1023 from Gnd to Vdd.

VoltsW = Voltage\_Vdd \*/ \_VoltageADCval 'Get the middle word of Vdd \* ADC10, so the result is divided by 256.

VoltsW = VoltsW / 4 'Then divide by 4, now we have Vdd \* ADC\_Reading / 1024.

\_VoltageDebug(" (ADC",#Channel,"=",#\_VoltageADCval,", Voltage=",#VoltsW," mV)",cr,lf)

#endmacro

VoltageEnd:

#Endregion

#Region "License"

#Rem

MIT License

Copyright (c) 2024 Alan Hunt

Permission is hereby granted, free of charge, to any person obtaining a copy

of this software and associated documentation files (the "Software"), to deal

in the Software without restriction, including without limitation the rights

to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

copies of the Software, and to permit persons to whom the Software is

furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all

copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR

IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE

AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER

LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,

OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE

SOFTWARE.

#EndRem

#EndRegion

# PICAXE Forum Post: 30/9/2024

## Improved Debugging

In a recent project I made a module called [ICODE]Debugger.basinc[/ICODE]. I found it quite useful so it’s here to share!

It’s very simple and doesn’t consume any extra memory or delays compared to using sertxd for debugging. You can also turn it on and off with a compiler directive.

The main benefit to me was program readability. It was nice to read code without needing extra comments, particularly when text was used in macros. Inside the macro you could debug out the text, using a lot of program space; or disable the debugging either by commenting out a line or turning the whole debugging off. Whatever you choose the PICAXE Editor keeps on displaying the bright red text as part of your program.

I couldn’t find anything similar on the forum, so I hope someone finds it useful.

There’s a lot help information in the file, just rename the extension from .bas to .basinc, or download it from here [URL] https://github.com/alnhnt/PICAXE/blob/main/Library/Debugger.basinc[/URL].

# PICAXE Forum Post: 4/10/2024

## Delving into and supporting Timing with Clock Rates (module/library)

The attached module was originally used for a game where the clock rate hadn't been finalised and the audio interface required consistent timings to be near minimum levels. Modifying more than a few pause commands by hand is unwieldy and error prone, particularly because arithmetic like "Pause 1 \* ClockMultiplier" is not allowed with the pause commands. Ultimately, a short list of logarithmic timings was easy to remember and they provided all the accuracy needed on most projects. It became a bit of fun to delve deeper into PICAXE timing and record some timing on an oscilloscope and this module emerged. This module's list of pause statements doesn't consume any extra program memory, and the module has these features:

[LIST]

[\*] A list of pause statements that provide timing delays regardless of the PICAXE clock rate: Pause1mS, Pause3mS, Pause10mS, Pause30mS, Pause100mS, Pause300mS, Pause1S, Pause3S and Pause10S.

[\*] The macro ``Pause\_mSec(t)`` allows you to specify specific pause times regardless PICAXE clock rate.

[\*] The PICAXE clock rate is set using a kHz value: 31, 250, 500, 1000, 2000, 4000, 8000, 16000, 32000, or 64000 (Dependent on chip).

[\*] The debug terminal baud rate is automatically set to the appropriate value for the PICAXE clock rate.

[\*] For X2 chips, the "timer" variable is set to increment at a configured rate if ``TIMER\_IN\_HUNDRETHS`` is defined in your program and given a value. This proved useful to measure lap time in a game.

[\*] A reference list of actual pause timings is shown in the Programming Guide notes below, these are valid if you use this module, or just use regular pause commands and do integer math based on the ratio of actual clock rate to default clock. From this some subjective recommendations are made for minimum clock rates with certain durations. Just don't expect a 1mS delay to be added when you use ``pause 1`` :)

[/LIST]

The attached file should be renamed to [ICODE]Timing.basinc[/ICODE] with the basinc extension. There’s a lot of help information in the file and you can also download the master file with the basinc extension from GitHub [URL=https://github.com/alnhnt/PICAXE/blob/main/Library/Timing.basinc] here [/URL].

# PICAXE Forum Post: 16/10/2024

## Speech, Effects and Music with DF Player Mini (module/library)

Back in the 80’s the most exciting outputs I could muster usually involved LEDs and 7 segment displays. Returning to electronics I’m finding that generating audio is now easy and can be far more informative and fun. The attached module provides the ability for your PICAXE program to speak numbers and sentences by stringing together words or phrases, it can also play music, sound effects and adverts. The adverts I found useful in interrupting idle music to encourage game play of my Buzz Wire Race Game at the village fete, also they provided crash sound effects during background music when the game was played.

With websites like www.ttsmp3.com it’s easy to create and download MP3 tracks for words or entire paragraphs. I’ve put together a foundation set of tracks for some projects in addition to the one’s used in my game. Together with this “DF Player\_Mini.basinc” module, the audio tracks make it easy to:

[LIST]

[\*] Speak numbers up to 65,535 using the macro AudioSpeakNumber(val).

[\*] Speak decimals assuming a word value represents 10’s, 100’s, or 1000’s, such as AudioSpeakThousandths(4023), which speaks four-point-zero-two-three.

[\*] Speak numbers with ordinal suffixes like 18th, which has its own audio track; or 28th which would be formed by joining the tracks for twenty and eighth.

[\*] Speak modifiable sentences by stringing together a sequence of words, phrases and numbers.

[\*] Recite speech, or play any sound, effects or music, by just pre-recording MP3s that a PICAXE either starts immediately using PlayNow(136, “You finished!”); or starts when the preceding track is finished using PlayMP3(137, “Your time was”).

[/LIST]

With the foundation tracks you make spoken reports like the demo program below:

[CODE]

'Compiler Directives

#picaxe 20X2

#no\_data

#no\_table

'@Timing.basinc directive

#define CLOCK\_SET\_KHZ **8000**

'Pins

symbol AUDIO\_TX = B.0 '@DF\_Player\_Mini.basinc: PICAXE Serial Tx to DF Player via 1K resistor.

symbol AUDIO\_STATUS = pinB.2 '@DF\_Player\_Mini.basinc: Busy indication from DF Player is active low after about 230mS.

'Variables

'@DF\_Player\_Mini.basinc Variables

symbol \_Audio\_Digit = b43 'Assign symbol to any spare Byte.

symbol \_Audio\_Integer = w22 'Assign symbol to any spare Word.

symbol \_Audio\_Fraction = w23 'Assign symbol to any spare Word.

symbol \_Audio\_Track = b48 'Assign symbol to any spare Byte.

symbol \_Audio\_Byte = b49 'Assign symbol to any spare Byte.

'Constants

'@DF\_Player\_Mini.basinc Constants

symbol AUDIO\_VOLUME = **15** '0 to 31(Maximum). You can assign this symbol to a variable if you don't want fixed volume.

symbol AUDIO\_BAUD = T9600\_8 'Tx logic is idle high (T) and 9600 bps, the last characters you adjust to your clock speed.

symbol AUDIO\_ACTIVE = **0** 'The AUDIO\_STATUS pin is active low, indicating audio playing.

symbol AUDIO\_IDLE = **1** 'AUDIO\_STATUS high indicates the module is idle or seeking a track to play on the disk.

'Modules

#include "Timing.basinc"

#include "DF\_Player\_Mini.basinc"

'Initialisation

gosub AudioInitialise 'Prepares PICAXE output pin, Resets the DF Player, selects SD input and sets volume.

Main:

PlayNow(0063, "The system has restarted.")

PlayMP3(0070, "The cell voltage levels are")

AudioSpeakThousandths(3423) 'This speaks "three-point-four-two-three".

PlayMP3(0112, "volts")

PlayMP3(0030, "and")

AudioSpeakNumber(3375) 'This speaks "three-thousand-three-hundred-and-seventy-five".

PlayMP3(0111, "millivolts.")

PlayMP3(0068, "Warning, the battery is below 20%.")

PlayMP3(0103, "Internal")

PlayMP3(0121, "temperature")

PlayMP3(0105, "is")

AudioSpeakTenths(354) 'This speaks "thirty-five-point-four".

PlayMP3(0122, "degrees C.")

PlayMP3(0104, "External")

PlayMP3(0121, "temperature")

PlayMP3(0105, "is")

AudioSpeakTenths(176)

PlayMP3(0122, "degrees C.")

PlayMP3(0106, "Total")

PlayMP3(0101, "solar")

PlayMP3(0116, "power")

PlayMP3(0105, "is")

AudioSpeakHundreths(1263) 'This speaks "Twelve-point-six-three".

PlayMP3(0093, "kilo")

PlayMP3(0117, "watt")

PlayMP3(0087, "hours")

PlayMP3(0108, "over the last")

AudioSpeakNumber(7)

PlayMP3(0089, "days.")

Pause3S

goto Main

[/CODE]

The attached zip file contains a recording that lets you hear what the demo was like. The staccato of individual words is fine and clear for personal use but to save program space and greatly improve presentation it’s best to join phrases rather than individual words. For my game I also edited tracks to speed up the pace a little and occasionally added a cartoon effect and changed pitch.

The DF Player Mini is referred to as a PICAXE SPE033, which isn’t in the shop; or as a SPE035 Project Board which includes a tiny yet powerful speaker, just note the speaker’s pins are too chunky for breadboard. The DF Player Mini is also widely and cheaply available, just make sure you’re obtaining from a quality source. The PICAXE only needs to monitor the DF Player Busy Signal and send serial data to play or stop MP3 or WAV tracks.

The attached “DF\_PLAYER\_MINI” file contains lots of useful information and it should be renamed with a “.BASINC” extension, the master file is [URL=https://github.com/alnhnt/PICAXE/blob/main/Library/DF\_Player\_Mini.basinc] here [/URL].

Please note the audio tracks are not for commercial use, although the majority are from ttsmp3.com using the AI “Alloy” voice and their website claims they don’t have any restrictions. NCH WavePad was used to touch-up the tracks, mainly this was normalising based on peak loudness (RMS) and trimming the start and end to remove small amounts of quiet time.

The audio files and a list of all the tracks can be found [URL=https://github.com/alnhnt/PICAXE/tree/main/Buzzz\_Wire\_Race\_Game/Audio] here [/URL].

My future projects are likely to bias more towards speech feedback in future, particularly with external projects because waterproofing and visibility becomes less of an issue. I’m glad to help if you have any issues. Please also let me know if there’s any potential improvements.

# Forum Post: 22/10/2024

## Voltage monitoring and calibration for X2 chips (module/library)

This module is fairly simple, but maybe of use to someone. It’s my penultimate write-up of a project.

The X2 operator “\*/” is used to gain an accurate ADC to voltage conversion with a small amount of program code and memory. There are some test results in the notes showing that the ADC is reasonably linear and accurate but the internal FVR (Fixed Voltage Reference) is not so great as a basis to calculate the PICAXE supply voltage. To compensate for this there’s an FVR calibration method included in the notes too.

There are just 3 macros:

[LIST]

[\*] GetVdd updates a word variable with the power supply voltage in mV.

[\*] SetVdd(Val\_mV) can be used instead of GetVdd for testing purposes, or if the power supply is accurately known and stable.

[\*] VoltsByVdd(Var, Channel) updates a word variable, Var, with the voltage in mV from the specified ADC Channel number.

[/LIST]

The attached “Voltage\_X2” file contains its own documentation and it should be renamed with a “.BASINC” extension when used, the master file is [URL=https://github.com/alnhnt/PICAXE/blob/main/Library/Voltage\_X2.basinc] here [/URL].

I’m glad to help if you have any issues. Please also let me know if there’s any potential improvements.

# PICAXE Forum Post: 27/10/2024

## Buzz Wire Race Game

A Buzz Wire game started as a fund-raising idea for our village fete. I offered to make something based on a racetrack with timed laps, and the committee jumped at that. To make it more fun I added music, sound effects and speech with a DF Player Mini, as per a previous post. The game was engaging and fun with the audio never stopping and some of the speech speeded up for results etc.

There were advert tracks to encourage game play and to give different crash sound effects during the game. At the race start a beep-beep-beep-beeeeeep was timed to work in parallel to a led sequence. There were timers for feedback during the race. At the end of a race there was more feedback and possible fanfares along with a spoken lap time in tenths of a second. Player times were ranked against a leaderboard and the top 4 positions were spoken if required. There were independent volume controls for audio from the DF Player and for the buzz tone that was created by PWM. The volume was fairly unbearable at maximum.

Soon I ran out of program space on a 20M2, mainly because I liked to see and show the debug output. I swapped out to a 20X2 and kept having to trim back the debug as the program grew. Eventually it was 459 bytes of the 4077 used.

Unfortunately, under twelves performed far worse than expected and a couple were dejected by their lap time, which was sad to see. The adults were so competitive though! The game is a little addictive and frequently people thought they could get the fastest lap by crashing 1 less time. I think the least number of crashes all day was 2, but you must be fairly quick too.

Probably the worst decision was boldly stating I would base the layout on our local racetrack, Snetterton. I already felt committed, then I checked the look of the layout and probably expressed a few expletives based around “oh dear”! 😊 I’ve promised a different track for next year.

I’ve always hand drawn circuits and stripboard layouts in the past, knocked up programs quickly and then wondered what it all meant if I return to it later. This year I wanted to go better and probably took on a bit too many firsts for me:

[LIST]

[\*] KiCad for electronics schematic and exports of a netlist and bill of materials.

[\*] VeroRoute for stripboard layout after importing the netlist.

[\*] InkScape for vector graphics of the laminated sheet.

[\*] Wavepad for audio editing.

[\*] DecalProFx for silk screening panels, which has always been a bug bare. Sadly, they’re now out of business. The decal didn’t peel away properly on the bare metal panel but worked perfectly for the plastic mounted volume controls.

[\*] Brazing to connect the handle and wire, although a nut and bolt seemed almost as good.

[\*] Getting to know GitHub, Markdown Language (with StackEdit), and Licensing for uploads.

[\*] Choosing and learning how to use a table saw after cutting my last wonky bit of wood!

[\*] Trying to make a full documentation set.

[/LIST]

It was fun to do and some audio tracks were amusing too. I’ve placed a few in the ZIP file, including the one about a PICAXE being upgraded and taking over the world!

Excluding any copyright music, I’ve made a full upload of remaining files to GitHub including maker details and a presentation folder [URL= https://github.com/alnhnt/PICAXE/tree/main/Buzz\_Wire\_Race#readme] here [/URL].