Microbric Edison token specification version 1.9 (for firmware 0xe0, token stream 0x60, and legacy firmware 0xac/0x20, tokens 0xf1/0x20)

Brian Danilko, Likeable Software

18 November 2018

Contents

1 Version history

Ver	Date released	Notes
1.9	18/Nov/2018	Documented a new audio output sequence which helps circumvent some audio enhancement software.
1.8	06/Aug/2016	Left/Right shifts by constants can now by byte or word, but the constant is ALWAYS an 8-bit one.
1.7	26/Jun/2016	Left shifts are now all unsigned!
1.6	25/Jun/2016	Added extra status of distance, new temp registers and saving/restoring _index on entering/exiting an event, and cleared unused registers in _devices.
1.5	17/Jun/2016	Fix pop to accumulator. Added, firmware defined, debug output. Updated tone freq., Bill protocol and todos with input from Bill.
1.4	13/Jun/2016	Fix shiftleft/right by constant to only accept 8-bit constants. Fixed token download check digit. Updated event zeroes for no event
1.3	10/Jun/2016	Updates to CPU flags and download preambles. Also updates to shifts.
1.2	23/Nov/2015	Correct endian info – it is really BIG ENDIAN. Also corrected info about download offsets, and made Sample Rate variable.
1.1	19/Nov/2015	Added info on interpreting tune strings.
1.0	16/Nov/2015	Started from 3.9 version of tokens1.html, with changes for Ed.Py.

2 Background

This document started from the 3.9 version of tokens1.html, and that document should still be consulted where this document isn't completed (and to get some background). Where there are differences, this document should be considered correct, as it has changes for version 0x60 (Ed.Py).

3 Still to do

- Beeper/TuneTempo value interpretation needs to be completed.
- Distances for the new motors/encoders are different. Need to update the info here.
- How is an error in a TuneString handled by the firmware? I prefer that an error in the string just terminates the playback at that point. Then the musician would know that the tune was short and could inspect their code to find out why. But we could use an error indication if desired as well.

4 Memory Map

4.1 Endianness

- 8-bit values take exactly 1 byte and are considered un-signed
- 16-bit values take exactly 2 bytes and are signed (2s complement). The most significant byte (MSB) is in the lower numbered address, and the least significant byte (LSB) is in the higher numbered address. This is big endian (BE).
- There are special cases where 8-bit values are signed and 16-bit values are unsigned. These cases are documented in this file.

4.2 8-bit space

- This may have from 0 to 256 8-bit elements. In version 0x20 there are 256 elements. In version 0x60 the number of elements is encoded in the download of each program and can range from 0 to 255 elements.
- Each element contains an 8-bit unsigned value (range from 0 to 255).

4.3 16-bit space

- This may have from 0 to 256 16-bit elements. In version 0x20 there are 13 elements. In version 0x60 the number of elements is encoded in the download of each program and can range from 0 to 255 elements.
- Each element contains a 16-bit signed value (range from -32768 to +32767).

4.4 Stack space

- This is a firmware defined number of 16-bit elements. The firmware balances 8-bit, 16-bit and stack space, so room for the stack will depend on the sizes of the 8 and 16-bit spaces.
- When an 8-bit value (constant or variable) is pushed, then 0 is added in the MSB.
- When popping to an 8-bit variable, the LSB is put into the 8-bit variable.
- The Stack Pointer, SP, which indexes this space is in the CPU module.
- The SP starts at 0 and grows up on pushes, and decreases on pops.
- On a Push, the value is saved to the current SP, and then SP is incremented. If this increment would go outside the stack space a StackBlown error is raised.
- On a Pop, if SP=0, a StackBlown error is raised. Else, the value is read off the stack, and then the SP is decremented.
- All offsets into the stack, refer to the 16-bit elements, not the individual bytes.
- Pushes onto the stack for changes of control flow (branching) push the following stack frame, which consists of the following 3 16-bit values:
  - cpu:ACC
  - cpu:Flags (high byte) and cpu:Counter (low byte)
  - cpu:PC (this may be of the next instruction to execute on return)

5 Tokens

5.1 General

- All tokens are sequences of binary bits, in multiples of 8 (a byte).
- All reads/writes to memory have to be checked if they are in range. If not then either a StackBlown or an OutOfRange error will be raised.

5.2 Special tokens (prefix 00)

- Bit clear/set in the mod/regs (used for triggering actions and clearing status). The mod/reg and be either 8 or 16-bit, as only the lower 8-bits can be modified anyway.
  - **00 | 00 | 0bbb || mod/reg(8)**
    Clear bit 'bbb' in mod/reg.
  - **00 | 00 | 1bbb || mod/reg(8)**
    Set bit 'bbb' in mod/reg.
- Copy data into memory.
  - **00 | 01 | nnnn || first-8-bit-var(8) || data(nnnn*8)**
    Copy 'nnnn' elements of 8-bit data to 8-bit space starting at first-8-bit-var.
  - **00 | 10 | nnnn || first-16-bit-var(8) || data(nnnn*16)**
    Copy 'nnnn' elements of 16-bit data to 16-bit space starting at first-16-bit-var.
- Store value in CPU ACC to memory.
  - **00 | 11 | 0000 || 8-bit-var(8)**
    Store value in ACC to 8-bit space at 8-bit-var.
    No flags are changed. If S bit is set (so 16-bit value in ACC) then move just the low byte to 8-bit-var.
  - **00 | 11 | 0001 || 16-bit-var(8)**
    Store value in ACC to 16-bit space at 16-bit-var.
    No flags are changed. Check S bit and if not 1, then raise error SizeMismatch.
  - **00 | 11 | 0010 || 8-bit-mod(4)reg(4)**
    Store value in ACC to 8-bit module register, mod/reg.
    No flags are changed. If S bit is set (so 16-bit value in ACC) then move just the low byte to mod/reg.
  - **00 | 11 | 0011 || 16-bit-mod(4)reg(4)**
    Store value in ACC to 16-bit module register, mod/reg.
    No flags are changed. Check S bit and if not 1, then raise error SizeMismatch.
- Move 16-bit variable into 8-bit module register (as Ed.Py just supports 16-bit variables, this reduces numbers of tokens needed for simple operations).
  - **00 | 11 | 0100 || 16-bit-var(8) || 8-bit-mod(4)reg(4)**
    Move low byte of 16-bit-var to 8-bit mod(4)reg(4). No flags are changed.
- Unused tokens
  - 0011|0101 - 0011|1111 so eleven tokens are free

5.3 Move Data Tokens (prefix 01)

- The token first byte is made up of 2 bits for prefix, 4 bits for source and 2 bits for destination.
- Sources
  - **01 | 0nns | dd**
    The source is a small constant 'nn' (so 0-3). If 's' = 0, then it's interpreted as 8-bit, else 16-bit.
  - **01 | 1000 | dd || 8-bit-mod(4)reg(4)**
    The source is the 8-bit value in the 8-bit mod/reg.

- **01 | 1001 | dd || 16-bit-mod(4)reg(4)**
  The source is the 16-bit value in the 16-bit mod/reg.
- **01 | 1010 | dd || 8-bit-const(8)**
  The source is the 8-bit constant.
- **01 | 1011 | dd || 16-bit-const(16)**
  The source is the 16-bit constant.
- **01 | 1100 | dd || 8-bit-var(8)**
  The source is the 8-bit value in the 8-bit variable.
- **01 | 1101 | dd || 16-bit-var(8)**
  The source is the 16-bit value in the 16-bit variable.
- **01 | 1110 | dd || 16-bit-var(8)**
  The source is the system time (see Timers module, register e) with the value of the 16-bit variable added to it. The destination must be an 8-bit-variable.
- **01 | 1111 | dd || 16-bit-const(16)**
  The source is the system time (see Timers module, register e) with the 16-bit constant added to it. The destination must be an 8-bit-variable.

- Destinations
  - **01 | ssss | 00 || as per the source || mod(4)reg(4)**
    Destination is the mod(4)reg(4). The size of the destination mod/reg will be taken from the source size.
  - **01 | ssss | 01 || as per the source || 8-bit-var(8)**
    Destination is the 8-bit-var. If the source is not 8-bit then a SizeMismatch error should be raised.
  - **01 | ssss | 10 || as per the source || 16-bit-var(8)**
    Destination is the 16-bit-var. If the source is not 16-bit then a SizeMismatch error should be raised.
  - **01 | ssss | 11 || as per the source**
    Destination is the CPU accumulator (ACC). Size bit, S will be set reflecting the size of the source.
- Notes
  1. The E flag (in CPU module) is updated on EVERY move. If the source of the move is 0, then E = 1, else E = 0.
  2. Small constants are encoded in the token so that writing 0 or 1 to a module only takes a token of 2 bytes - the same as now.
  3. The smallest token is 2 bytes, the largest is 4.
  4. Moves can also be done through the INDEX module which uses an incrementing cursor for the reads or writes 8-bit and 16-bit variables.
  5. The System time add and moves seem complicated but are not. And they allow for easy timed operations so that a Microbric instruction of motor forward for 10 seconds becomes 4 tokens: 1) motor on, 2) move time data +10 seconds, 3) compare times (see below), 4) loop back 1 if less then. The point of all of this is so that no (or very few) tokens take long to execute. Then it is an easy for the interpreter to execute a token, then check for events, then repeat. Polling can work for most things then.

5.4 Math Tokens (prefix 10)

- Unary operations on the ACC
  - **10 | 00 | s | 0 | 00**
    Not the CPU ACC. If the size 's' does not equal the S flag then a SizeMismatch error should be raised.
  - **10 | 00 | s | 0 | 01**
    Increment the CPU ACC by 1. If the size 's' does not equal the S flag then a SizeMismatch error should be raised.
  - **10 | 00 | s | 0 | 10**
    Decrement the CPU ACC by 1. If the size 's' does not equal the S flag then a SizeMismatch error should be raised.
  - **10 | 00 | s | 0 | 11**
    Convert the CPU ACC to the other size. If the S flag is 0, then convert the ACC to 16-bit, else convert it to 8-bit. When converting from 16-bit to 8-bit, if s=0, use the LSB, else use the MSB.

- Unary operations on memory variables. Result goes back into the memory variable, and the ACC.
  - **10 | 00 | s | 1 | 00 || var(8)**
    Not the value of variable var . If s=0, then it's an 8-bit variable, else it's 16-bit.
  - **10 | 00 | s | 1 | 01 || var(8)**
    Increment the value of variable var by 1. If s=0, then it's an 8-bit variable, else it's 16-bit.
  - **10 | 00 | s | 1 | 10 || var(8)**
    Decrement the value of variable var by 1. If s=0, then it's an 8-bit variable, else it's 16-bit.
- System time compares
  - **10 | 00 | 0 | 1 | 11|| first-8-bit-var(8)**
    Compare the system time (see module Timers, register e) against the 4 bytes starting at first-8-bit-var, which will update E, G & L flags. The ACC and 4 bytes in 8-bit space are unchanged by this operation.
- Basic arithmetic. All operations are on the ACC and a constant or the value of variable var (size determined by s). Result goes into the ACC, the var is not changed.
  - **10 | 01 | s | 0 | 00 || var(8)**
    Add the value of variable var into the ACC. If s=0, then it's an 8-bit variable, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 01 | s | 0 | 01 || var(8)**
    Subtract the value of variable var from the ACC. If s=0, then it's an 8-bit variable, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 01 | s | 0 | 10 || var(8)**
    Multiply the ACC with the value of variable var. If s=0, then it's an 8-bit variable, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 01 | s | 0 | 11 || var(8)**
    Compare the value of variable var with the ACC. If s=0, then it's an 8-bit variable, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error. Flags E, G & L are updated.
  - **10 | 01 | s | 1 | 00 || const(8/16)**
    Add the constant into the ACC. If s=0, then it's an 8-bit constant, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 01 | s | 1 | 01 || const(8/16)**
    Subtract the constant from the ACC. If s=0, then it's an 8-bit constant, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 01 | s | 1 | 10 || const(8/16)**
    Multiply the ACC with the constant. If s=0, then it's an 8-bit constant, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 01 | s | 1 | 11 || const(8/16)**
    Compare the value of constant with the ACC. If s=0, then it's an 8-bit constant, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error. Flags E, G & L are updated.
- Shifts and divides. All operations are on the ACC and a constant or the value of variable var (size determined by s). The result goes back into ACC and the variable is not changed.
  - **10 | 10 | s | 0 | 00 || var(8)**
    Shift left the ACC by the value in var. If s=0, then it's an 8-bit variable, else it's 16-bit. If the value of the variable is greater then 7 for 8-bit ACC (flag S=0), or less then 0 or greater then 15 for 16-bit ACC (flag S=1), then raise an OutOfRange error.
  - **10 | 10 | s | 0 | 01 || var(8)**
    Shift right the ACC by the value in var. If s=0, then it's an 8-bit variable, else it's 16-bit. Shifts are **all** unsigned, regardless of the flag S. If the value of the variable is greater then 7 for 8-bit ACC (flag S=0), or less then 0 or greater then 15 for 16-bit ACC (flag S=1), then raise an OutOfRange error.
  - **10 | 10 | s | 0 | 10 || var(8)**
    Divide the ACC with the value of variable var. If s=0, then it's an 8-bit variable, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 10 | s | 0 | 11 || var(8)**
    Modulus the ACC with the value of variable var. If s=0, then it's an 8-bit variable, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.

  - **10 | 10 | s | 1 | 00 || const(8)**
    Shift left the ACC by the 8-bit constant. Note that this constant is always 8-bit for both 8-bit (flag S=0) and 16-bit (flag S=1) in the accumulator. If the flag S and 's' do not match then raise a SizeMismatch error. If the value of the constant is greater then 7 for 8-bit ACC (flag S=0), or less then 0 or greater then 15 for 16-bit ACC (flag S=1), then raise an OutOfRange error.
  - **10 | 10 | s | 1 | 01 || const(8)**
    Shift right the ACC by the 8-bit constant. Note that this constant is always 8-bit for both 8-bit (flag S=0) and 16-bit (flag S=1) in the accumulator. If the flag S and 's' do not match then raise a SizeMismatch error. Shifts are **all** unsigned, regardless of the flag S. If the value of the constant is greater then 7 for 8-bit ACC (flag S=0), or less then 0 or greater then 15 for 16-bit ACC (flag S=1), then raise an OutOfRange error.
  - **10 | 10 | s | 1 | 10 || const(8/16)**
    Divide the ACC with the constant. If s=0, then it's an 8-bit constant, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 10 | s | 1 | 11 || const(8/16)**
    Modulus the ACC with the contant. If s=0, then it's an 8-bit constant, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.

- Bitwise arithmetic. All operations are on the ACC and a constant or the value of a variable (size determined by s). The result goes back into ACC and the variable is not changed.
  - **10 | 11 | s | 0 | 00 || var(8)**
    OR the ACC by the value in var. If s=0, then it's an 8-bit variable, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 11 | s | 0 | 01 || var(8)**
    AND the ACC by the value in var. If s=0, then it's an 8-bit variable, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 11 | s | 0 | 10 || var(8)**
    XOR the ACC by the value in var. If s=0, then it's an 8-bit variable, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 11 | s | 1 | 00 || const(8/16)**
    OR the ACC by the constant. If s=0, then it's an 8-bit constant, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 11 | s | 1 | 01 || const(8/16)**
    AND the ACC by the constant. If s=0, then it's an 8-bit constant, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.
  - **10 | 11 | s | 1 | 10 || const(8/16)**
    XOR the ACC by the constant. If s=0, then it's an 8-bit constant, else it's 16-bit. If the flag S and 's' do not match then raise a SizeMismatch error.

- Control event processing. These tokens will enable or disable event processing. This is similar to enabling or disabling interrupts in a normal uProc.
  - **10 | 11 | 0 | 0 | 11**
    Disable event processing.
  - **10 | 11 | 0 | 1 | 11**
    Enable event processing. This is the default on starting every run.

- Notes
  - All 8-bit variables and constants use unsigned arithmetic, 16-bit variables and constants use signed arithmetic.
  - The E flag (in CPU module) is affected by all operations. When the operation is not compare then it is set when the result is zero, cleared otherwise. On compare D is set if the values are the same, cleared otherwise. On compare G & L are updated as well.

- Unused tokens
  - 1011|1011, 1011|1111 so two tokens are free

5.5 Control Flow Tokens (prefix 11)

- Branching. The 'f' bit is the Far bit - if 0 then the offset is signed 8-bits, else it is signed 16-bits. The 'p' bit is the Push bit - if 0 then don't push a stack frame, else push one. The stack frame layout is the first 3 16-bit words of the CPU module (6 bytes), comprising ACC, Flags, Counter and PC.
  - **11 | 0 | f | p | 000 || SignedOffset(8/16)**
    Branch to the new location (PC = PC + SignedOffset).
  - **11 | 0 | f | p | 001 || SignedOffset(8/16)**
    Branch to the new location (PC = PC + SignedOffset) if E=1 (equal, or zero).
  - **11 | 0 | f | p | 010 || SignedOffset(8/16)**
    Branch to the new location (PC = PC + SignedOffset) if E=0 (not equal, or not zero).
  - **11 | 0 | f | p | 011 || SignedOffset(8/16)**
    Branch to the new location (PC = PC + SignedOffset) if G=1 (greater).
  - **11 | 0 | f | p | 100 || SignedOffset(8/16)**
    Branch to the new location (PC = PC + SignedOffset) if G=1 or E=1 (greater or equal).
  - **11 | 0 | f | p | 101 || SignedOffset(8/16)**
    Branch to the new location (PC = PC + SignedOffset) if L=1 (less).
  - **11 | 0 | f | p | 110 || SignedOffset(8/16)**
    Branch to the new location (PC = PC + SignedOffset) if L=1 or E=1 (less or equal).
  - **11 | 0 | f | p | 111 || SignedOffset(8/16)**
    Decrement the CPU register counter (and update the E flag) then branch to the new location (PC = PC + SignedOffset) if E = 0 (not zero).

- Push and pops. 's' = 0 for 8-bit, 's' = 1 for 16-bit.
  - **11 | 1 | s | 0000 || const(8/16)**
    Push the constant on the stack. If s=0, then it's an 8-bit constant, else it's 16-bit. Check for stack overflow and raise StackBlown if needed.
  - **11 | 1 | s | 0001 || var(8)**
    Push the variable on the stack. If s=0, then it's an 8-bit variable, else it's 16-bit. Check for stack overflow and raise StackBlown if needed.
  - **11 | 1 | s | 0010 || mod(4)reg(4)**
    Push the module register on the stack. If s=0, then it's an 8-bit register, else it's 16-bit. Check for stack overflow and raise StackBlown if needed.
  - **11 | 1 | 0 | 0011**
    Push the ACC on the stack. Check for stack overflow and raise StackBlown if needed.
  - **11 | 1 | s | 0100**
    Pop the top of the stack into the ACC. If s=0, then assume the value on the stack is 8-bit, else it's 16-bit. Check for stack underflow and raise StackBlown if needed.
  - **11 | 1 | s | 0101 || var(8)**
    Pop the top of the stack into the variable. If s=0, then it's an 8-bit variable, else it's 16-bit. Check for stack overflow and raise StackBlown if needed.
  - **11 | 1 | s | 0110 || mod(4)reg(4)**
    Pop the top of the stack into the module register. If s=0, then it's an 8-bit register, else it's 16-bit. Check for stack overflow and raise StackBlown if needed.
  - **11 | 1 | 0 | 1000**
    Return from subroutine by popping the return frame off the stack. Check for stack underflow and raise StackBlown if needed.

- Variables on the stack
  - **11 | 1 | s | 1001 || StackOffset(8)**
    Copy value from the stack at SP - StackOffset to ACC. If 's' = 0, then it's 8-bit, else it's 16-bit. Check for stack underflow and raise StackBlown error if so. Update S flag and E flag.
  - **11 | 1 | 0 | 1010 || StackOffset(8)**
    Add StackOffset to SP (this can be used to create space for local variables). Check for stack overflow and raise StackBlown error if so. No flags change.

  - **11 | 1 | 0 | 1011 || StackOffset(8)**
    Subtract StackOffset from SP (this can be used to remove local variable space). Check for stack underflow and raise StackBlown error if so. No flags change.
  - **11 | 1 | s | 1100 || StackOffset(8)**
    Copy value from ACC to the stack at SP - StackOffset. Raise SizeMismatch if 's' doesn't match flag S. Check for stack underflow and raise StackBlown error if so. No flags change.

- Miscellaneous tokens
  - **11 | 1 | s | 1110 || var(8)**
    Debug output of variable, if firmware supports it.
    If there is no support then this token is ignored. No flags are updated. If s=0, then the variable is in 8-bit space else it's in 16-bit space.
    The idea is that the firmware could output the value in some way. Possible by blinking the LEDs. The rate may be too fast for the human eye, but could be captured with an oscilloscope. This would allow for more efficient debugging using any unit and not requiring a speciallised debugging platform.
    Output, if any, is firmware defined.
  - **11 | 1 | 0 | 1111 || error(8)**
    Tells the interpreter to raise this error.
  - **11 | 1 | 1 | 1111**
    Tells the interpreter that it has reached the end of code.

- Notes
  1. The stack grows up and starts at virtual offset 0.

- Unused tokens
  - 11110011, 11111000, 11111010, 11111011
  - 111s0111, 111s1101
  - so 4+(2*2) = 8 are free

6 Device Module Registers

6.1 Edison configuration

The following module numbers are assumed by the firmware, and are used as the module number in all tokens where mod/reg is needed.

0. Line Tracker
1. LED (left)
3. Motor (left)
5. IR Rx
6. Beeper
7. IR Tx
8. Motor (right)
11. LED (right)

6.2 Line Tracker (module 0)

6.2.1 Layout

Register	Description
0. Status bits (r/w-8)	Format: ??????cl. C is set if there is a change, L is 1 if the tracker is over a line, 0 otherwise
1. Power on (r/w-8)	Writing a 0 turns off power to the line tracker, 1 turns it on. Reading gives last value written
2. Lightlevel (r-16)	Format: ??????nnnnnnnnnn (10-bit). N is the input light level detected.

6.3 LED (module 1 and 11)

6.3.1 Layout

Register	Description
0. Status bits (r/w-8)	Format: ????????. No status bits used.
1. Output driver level (r/w-8)	Write 0 for LED off, 1 for LED on. Reading returns the last written value.
2. Lightlevel (r-16)	Format: ??????nnnnnnnnnn (10-bit). N is the input light level detected.

6.4 Motor (module 3 and 8)

6.4.1 Layout

Register	Description
0. Motor status (r/w-8)	Format: ??????DS. D is set to one when zero is written into the distance register. S is set to one when the strain on the motor goes over a safe value.
1. Motor control (r/w-8)	Format: cccrssss. CCC are control bytes (00?-coast, 010-forward continuous using speed in SSSS, 011-forward using distance register, 100-backward continuous using speed in SSSS, 101-backward using distance register, 11?-brake). Speeds in SSSS can range from 0 to 10. If R is 1, then the motor will remain running after the distance is reached (but D will still be set when 0 is reached).
2. Distance (r/w-16)	When this register is non-zero, and the control register has forward or backword distance set, then the motor will do this many half-motor rotations which equates to the value in the register X 7.5 degrees, or the value X 2.5 mm (.1 inch) As the unit is moving, this register will be updated (counting down).

6.5 IR Rx (module 5)

6.5.1 Layout

Register	Description
0. Status bits (r/w-8)	Format: ?dlcrvmb. D is 1 if an obstacle was detected (if the IR transmitter is configured to do obstacle detection). If D is set, then the actual obstacle detection is on the left (L), centre (C) or right (R). V means that the last check was valid, M means that an IR transmission matched in the IR table, and B means that a Bill protocol character was received.
1. Action bits (w-8)	Format: ???????c. C means to check the check index to see if it has valid bits captured.
2. Check index (r/w-8)	Index of the IR table to check for valid bits using the C action bit.
3. Match index (r-8)	The last IR transmission matched the bits at this index in the IR table.
4. Received character (r-8)	The last received IR character in Bill protocol.

6.6 Beeper module (module 6)

6.6.1 Layout

Register	Description
0. Status bits (r/w-8)	Format: ????ecou: E-tune string error found, C-clap was detected, O-tone is done, U-tune (either coded or string) is done. Write 0 to clear a status, also corresponding status cleared on start of action (i.e. Action S starting clears E and U).
1. Action bits (w-8)	Format: ????sbou: S-play string tune, B-do a beep, O-sound a tone, U-play coded tune.
2. Tone Frequency (r/w-16)	Tone frequency, f (see below)
4. Tone Duration (r/w-16)	Tone duration, d (see below)
6. TuneCode Pointer (r/w-8)	first-8-bit-variable where an encoded tune starts.
7. TuneString Pointer (r/w-8)	first-8-bit-variable where a string tune starts. Firmware decodes this tune string.
8. Tune Tempo (r/w-16)	Tune playback temp. 0-normal, negative-slower, positive-faster. Actual numbers TBD

6.6.2 Notes

- Tone frequency value, f = 32E+6 / DesiredFreq in Hz, where 1000Hz <= DesiredFreq <= 5000Hz.
- Tone duration value, d, is in units of 10ms, where 0 <= d <= 32767 (for max of 327.67 seconds).

- Tune Code data uses 1 8-bit variable for each note and duration, coded in the bit fields: rtttnnnn, where R is a rest marker (and NNNN is ignored), TTT is the not time in units of 50ms each, and NNNN is the note to play, from this table:

  0. A (6th octave)
  1. A sharp
  2. B
  3. C (7th octave)
  4. C sharp
  5. D
  6. D sharp
  7. E
  8. F
  9. F sharp
  10. G
  11. G sharp
  12. A
  13. A sharp
  14. B
  15. C (8th octave)

- Tune String data uses 1 8-bit variables for a note, and 1 8-bit variable for the duration (so 2 8-bit variables for each note played), repeated for the number of notes to be played, terminated by the 'z' character. So a Tune String would look like: "ndndndnd...z".

Note Char	Means	Duration Char	Means
m	A (6th octave)	1	Whole note
M	A sharp	2	Half note
n	B	4	Quarter note
c	C (7th octave)	8	Eighth note
C	C sharp	6	Sixteenth note
d	D		
D	D sharp		
e	E		
f	F		
F	F sharp		
g	G		
G	G sharp		
a	A		
A	A sharp		
b	B		
o	C (8th octave)		
r	Rest		
z	End of tune		

- The tune is considered finished (and the status is updated) when any of these happens:
  1. When fetching the next (or first) note, a 'z' is read, or
  2. When fetching the next (or first) note, an invalid note character is read, or
  3. When fetching the duration for a note, an invalid duration char is read
- TODO – need to decide this. Could treat 2. and 3. from above as errors instead. This would help the programmer realise their mistake quicker.

6.7 IR Tx (module 7)

6.7.1 Layout

Register	Description
0. Action bits (w-8)	Format: ??????dt. D means to do obstacle detection, T means to transmit a character. Note that D and T can not both be set to 1!
1. Transmit character (w-8)	Character to transmit in Bill protocol.

6.7.2 Notes

- Bill protocol:
  IR data is sent as 8bit characters. The 'frame' consists of a start bit followed by the 8 data bits, msb first, followed by the compliment of the data byte, also msb 1st. There is no stop bit Each bit is sent out as a burst of 38kHz, followed by a 600uS space. A start bit is a 2.4mS burst, a one bit is a 600uS burst, and a zero bit is a 1200uS burst.

7 Internal Module Registers

7.1 Index (module 12)

This module provides for indexed access to memory spaces with incrementing after the read or write. Note this module is saved/restored on entry/exit to/from an event. This means that non-event processing can use multiple tokens to set-up indexing operations without disabling event processing.

7.1.1 Layout

Register	Description
0. Action bits (w-8)	Format: ?rwc?gpm. R is read 8-bit1, W is write 8-bit1, C is copy from 8-bit1 to 8-bit2, G is read 16-bit1, P is write 16-bit1, M is copy from 16bit1 to 16bit2. Note that all of these operations update the relevant cursors using the step registers.
1. 8-bit1 cursor (r/w-8)	Cursor 1 into 8-bit space
2. 8-bit1 step (r/w-8)	Amount to add to 8-bit1 cursor after use.
3. 8-bit1 window (r/w-8)	Variable to write to before triggering W or C, or read from after R.
4. 8-bit2 cursor (r/w-8)	Cursor 2 into 8-bit space (only used as a copy destination).
5. 8-bit2 step (r/w-8)	Amount to add to 8-bit2 cursor after use.
6. 16-bit1 cursor (r/w-8)	Cursor 1 into 16-bit space
7. 16-bit1 step (r/w-8)	Amount to add to 16-bit1 cursor after use.
8. 16-bit1 window (r/w-16)	Variable to write to before triggering P or M, or read from after G.
a. 16-bit2 cursor (r/w-8)	Cursor 2 into 16-bit space (only used as a copy destination).
b. 16-bit2 step (r/w-8)	Amount to add to 16-bit2 cursor after use.
c. 16-bit1 temp (r/w-16)	Temporary 1 16-bit value
e. 16-bit2 temp (r/w-16)	Temporary 2 16-bit value

7.2 Devices (module 13)

7.2.1 Layout

Register	Description
0. Status bits (r/w-8)	Format:????1234. 1, 2, 3, 4 represent the motherboard buttons being held down (after debounce).
1. Random (r/w-8)	A pseudo-random 8-bit number.

7.3 Timers (module 14)

7.3.1 Layout

Register	Description
0. Status bits (r/w-8)	Format:??????re. R-one shot timer running, E-one shot timer expired.
1. Action bits (w-8)	Format:?????epo: E-enable sleep timer, P-trigger pause timer, O-trigger one-shot timer.
2. Pause timer (r/w-16)	Wait until this time expires before executing next token (though event handling still happens). 10mS units (max 327.67 secs).
4. One shot timer (r/w-16)	Set a timer which when expired, sets the E flag in status bits. Doesn't pause token execution. 10mS units (max 327.67 secs)
6. System Time (r/w-32)	Four byte (32 bit) unsigned time that the system has been powered up in 10ms units (so range is $>$ 5900 hours)

7.4 CPU (module 15)

Virtualisation of the CPU. Note that a stack frame is the first 6 bytes of this structure (ACC, Flags, Counter and PC).

7.4.1 Layout

Register	Description
0. ACC (r/w-16)	Can hold either 8bit or 16bit values (token bits say which one, flag S records
2. Flags (r-8)	it). Flags are: seglhi??. S reflects the size the data in ACC (0-8, 1-16), E for the last compare being equal or the last operation =0, G & L for the last compare being greater (or less respectively), H when executing in a handler, and I means the interrupts (event handling) are enabled (by default I == 1).
3. Counter (r/w-8)	Used to control tight loops (there is a token directly relating to this).
4. PC (r/w-16)	Unsigned 16-bit - address of the token being interpreted.
6. Stack Pointer (r-16)	Index of the current top of the stack (see 4.4 for details).

8 Errors

Whenever an error is detected, the program will stop, and an LED will flash out the error number. This could be supplemented with the buzzer, buzzing out the error number too.

Error #	Name	Description
1	StackBlown	If increasing or decreasing the stack pointer will make it out of range
2	DivideByZero	A divide or modulus operation with 0
3	SizeMismatch	8-bit and 16-bit confusion.
4	OutOfRange	Accessing a variable outside of it's allowed range.
5	VersionMismatch	Download version not compatible with firmware.
6	WriteToReadOnly	Trying to write to a read-only register
7	OpOverflow	Probably will NOT do this!

9 Downloading Protocol for Edison

The Edison uses a

Binary -> Wav_coding -> Play_audio -> LEDs_from_audio -> Line_tracker_photo_transistor -> Binary_in_Edison

input chain to get the binary from the compiler / EdApp into the Edison.

This method is used for both firmware updates and program downloads. A preamble is prepended to the header to indicate a firmware vs. tokens download.

The round button (record) is used on Edison to trigger the firmware in Edison to receive a download. When the firmware starts receiving the data over the LED it interprets the binary as per the following data.

9.1 Firmware Downloads

The total firmware to be downloaded consists of the **Preamble**, followed by the **Header** and then the **Data** (with nothing between them).

Note that preamble values will change in the future to accomodate new firmware. Listed here are two firmware versions – original Edison and Ed.py Edison.

9.1.1 Preamble (original Edison)

	bits	current value
Firmware marker	8	0xac
Version	8	0x20

This firmware only understands token downloads with token marker == 0xf1 (version not checked).

9.1.2 Preamble (Ed.py Edison)

	bits	current value
Firmware marker	8	0xe0
Marker check	8	0x1f

This firmware understands the original Edison tokens (token marker 0xf1, version 0x20) as well as Ed.py Edison (token marker 0x60, marker check 0x9f).

9.1.3 Header

	bits	description
Bytes	16	Number of bytes in the data following this header
CRC	16	CRC (CRC-16-CCITT) of the data following this header

9.1.4 Data

The data is the firmware program.

9.2 Token Program Downloads, Ed.py Edison

The total token program to be downloaded consists of the **Preamble**, followed by the **Header** and then the **Data** (with nothing between them).

9.2.1 Preamble

	bits	current value
Program marker	8	0x60
Marker check	8	0x9f

The Marker check is the complement of the Program marker and should be checked by the firware.

These tokens are understood by firmware type: 0xe0/0x1f.

9.2.2 Header

	bits	description
Data Bytes	16	Number of bytes in the data following this header
Data CRC	16	CRC (CRC-16-CCITT) of the data following this header

9.2.3 Data

	bits	description
8-bit vars	8	Number of 8-bit vars used (0-255)
16-bit var	8	Number of 16-bit vars used (0-255)
Program offset	16	Bytes from the start of Header bytes to the start of "Program tokens" (this will be 10 or greater).
Event table data	variable	See table for description of this data. If no events then a single Zeros field (2 zero bytes) is used.
Program tokens	variable	Token stream.

9.3 Token Program Downloads, Original Edison

The total token program to be downloaded consists of the **Preamble**, followed by the **Header** and then the **Data** (with nothing between them).

9.3.1 Preamble

	bits	current value
Program marker	8	0xf1
Version	8	0x20

These tokens are understood by firmware types/versions: 0xac/0x20 and 0xe0/0x1f.

9.3.2 Header

	bits	description
Data Bytes	16	Number of bytes in the data following this header
Data CRC	16	CRC (CRC-16-CCITT) of the data following this header

9.3.3 Data

	bits	description
NOT USED	8	Must be 0
Event table offset	8	Bytes from the start of "Data Bytes" to the start of "Event table data" (if event table not used then this is 0, else it is 8).
Program offset	16	Bytes from the start of "Data Bytes" to the start of "Program tokens" (if event table not used then this is 8, else it's greater then 8).
Event table data	variable	See table for description of this data. Bytes = "Program offset" - 8.
Program tokens	variable	Token stream. Bytes = "Data Bytes" - "Program offset"

9.4 Event table data

	bits	description
Token offset	16	Bytes from start of Header bytes to the first token in the handler
Mod/Reg	8	The register in the module that triggers this event
Mask	8	
Value	8	The trigger condition is when (Mod/Reg & Mask) = Value. When that's true, tokens from "Token offset" will be excuted until a "Stop" token is found.
. . .		(sequence repeated for each event handler)
Zeroes	16	Have value = 0. Marks the end of the table.

9.5 Binary to Audio (wav format) conversion

Binary data to be communicated to Edison is converted to an audio 'wav' file first. The audio file is a 2 channel, 8-bit per channel, file with a frame rate of 44100Hz, though this may change (to allow better playback for some devices). Each byte is encoded with a start bit, 8 data bits (LSB first) and then a stop bit.

All downloads (token programs and firmware) are comprised of the following parts (in this order):

1. Get Ready Sequence
2. Preamble
3. Header
4. Data. When downloading the data, every 1536 bytes, there is a pause of 2 seconds. This pause allows for the flash write to be executed.
5. Postscript Sequence

9.5.1 New Pulse Audio sequences

This new audio sequence helps to circumvent some computer's audio enhancement software.

Each bit uses a sequence where the left and right channels are set to: left=255, right=0 called FAR; left=0, right=255 called NEAR; and left=128, right=128 called MID.

The sequences representing the bits and special cases are:

- Get Ready:
  - Repeated for 20ms (typically about 20 repeats):
  - FAR for 0.5ms
  - NEAR for 0.5ms
- Start bit:
  - FAR for 1 frame
  - NEAR for 1 frame
  - MID for 2ms (minus 2 frames)
- 0 bit:
  - FAR for 1 frame
  - NEAR for 1 frame
  - MID for 1ms (minus 2 frames)
- 1 bit:
  - FAR for 1 frame
  - NEAR for 1 frame
  - MID for 2ms (minus 2 frames)
- Stop bit:
  - FAR for 1 frame
  - NEAR for 1 frame
  - MID for 5ms (minus 2 frames)
- Postscript:
  - Repeated for 20ms (typically about 20 repeats):
  - FAR for 0.5ms

  - NEAR for 0.5ms

- Pause:
  - Repeated for 2 seconds (typically about 2000 repeats):
  - FAR for 0.5ms
  - NEAR for 0.5ms

9.5.2 Old square-wave Audio sequences

Each bit uses a sequence where the left and right channels are set to: left=255, right=0 called FAR; left=0, right=255 called NEAR; and left=128, right=128 called MID.

The sequences representing the bits and special cases are:

- Get Ready:
  - Repeated for 20ms (typically about 20 repeats):
  - FAR for 0.5ms
  - NEAR for 0.5ms
- Start bit:
  - FAR for 0.5ms
  - NEAR for 0.5ms
  - MID for 3ms
- 0 bit:
  - FAR for 0.5ms
  - NEAR for 0.5ms
- 1 bit:
  - FAR for 0.5ms
  - NEAR for 0.5ms
  - MID for 1ms
- Stop bit:
  - FAR for 0.5ms
  - NEAR for 0.5ms
  - MID for 4ms
- Postscript:
  - Repeated for 20ms (typically about 20 repeats):
  - FAR for 0.5ms
  - NEAR for 0.5ms
- Pause:
  - Repeated for 2 seconds (typically about 2000 repeats):
  - FAR for 0.5ms
  - NEAR for 0.5ms