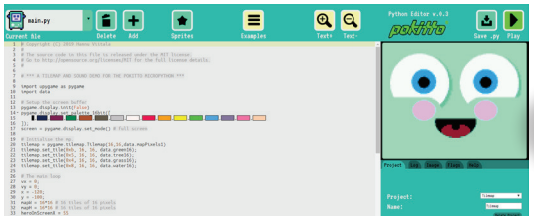
YOUR ONE-STOP SOURCE OF INFORMATION ABOUT YOUR FAVOURITE DIY CONSOLE

pokitto®

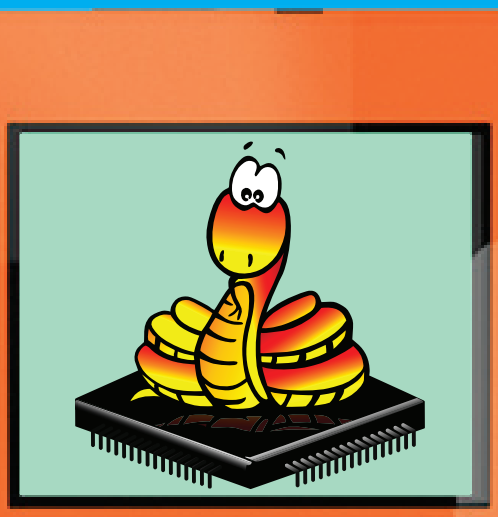
MAGAZINE

ISSUE 001, JULY 2019

Getting Started with Python Editor



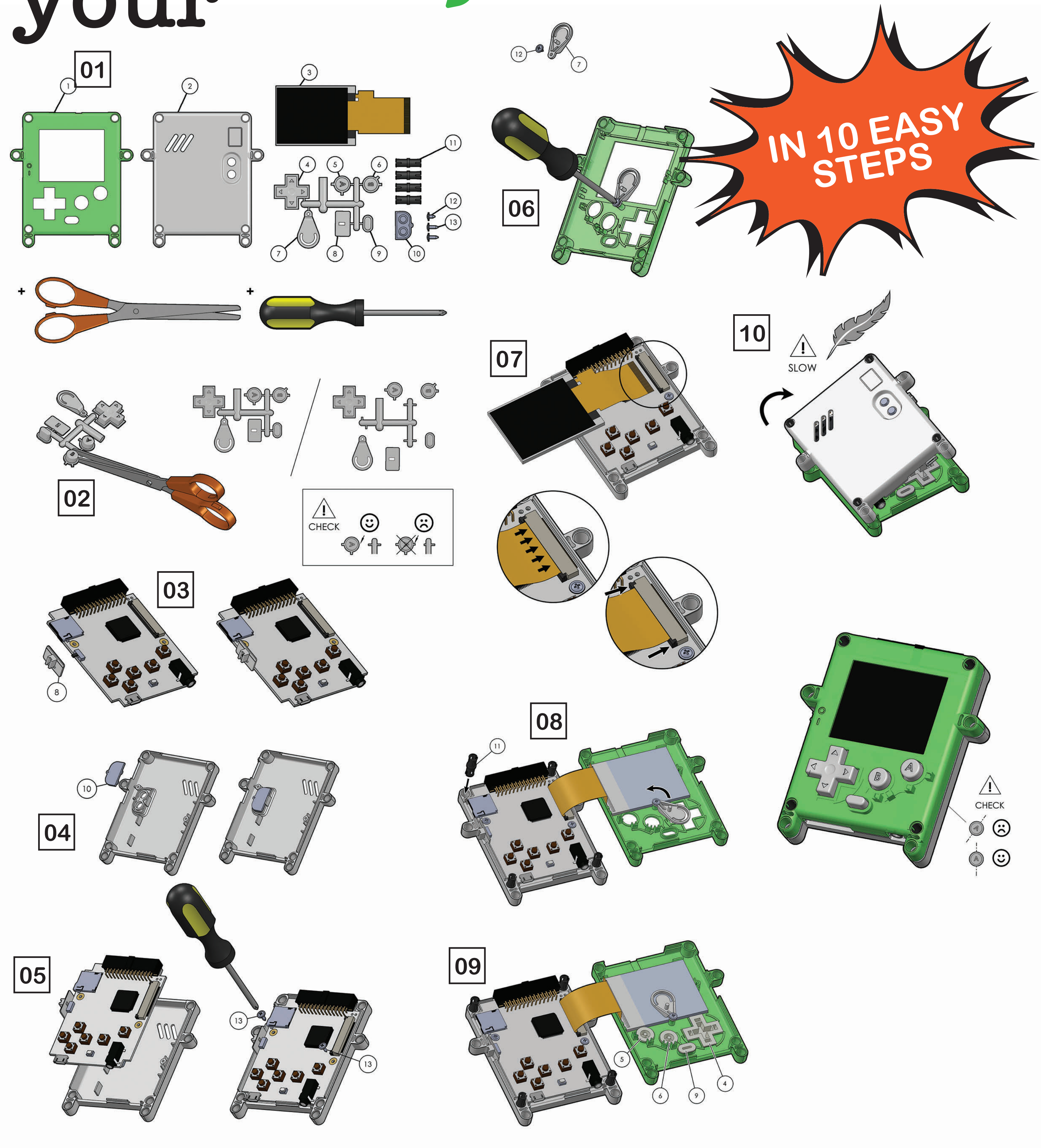
Abbaye Des Morts POST MORTEM



AND MUCH MORE!

ALSO FEATURING INTERVIEWS, PEX EXPERIMENTS, POKITTO SCHEMATICS

How to assemble your pokitto



ARRAY Of Contents

Welcome to the first Pokitto Magazine!

This issue comes with an exciting lineup of articles straight from the community itself. We will learn how to use the amazing online Python editor with **Hanski**. Get an inside look at Noggin along with some tips and tricks from the developer **Spinal**. We will have a look at an awesome hardware project by **jpfli** adding a speedometer to a bicycle using Pokitto. A postmortem from one of the developers behind porting Abbaye Des Morts to Pokitto, **HomineLudens**, shows the in depth challenges of bringing a title from another platform. An interview with the winner of the Python Pokitto game challenge shows us what it is like to develop for the platform as **dir3kt** tells all. **FManga** joins us with an article on taking photos with Pokitto, a fantastic starter project for all sorts of amazing camera related projects. **Jonne** is finally giving us the Pokitto schematics and give us some nice details on how to overclock a Pokitto!

With the diverse and creative culture of the Pokitto community, have a look through the Pokitto photo booth gallery to see how the community shows their own style and creativity. Maybe even get some ideas of your own!

Have something you'd like to share or contribute for future issues of the magazine? Feel free to check out the Pokitto Forums to find out more.

-Tor
community member.

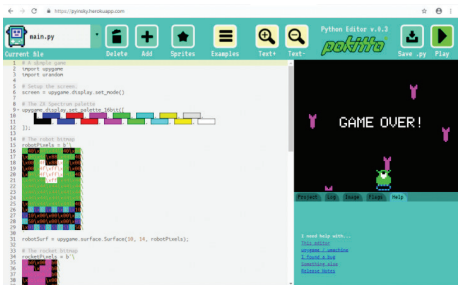
The More You Know

With Hanski

Getting started with Online Python Editor

Python Editor is a web-based development environment for creating Python games and other applications for Pokitto. It does not require any installing. The editor contains an integrated Pokitto emulator, so you do not even necessarily need Pokitto HW. In fact, some very good entries in the past Pokitto Python GameJam were created using only the emulator.

If you have a Pokitto, you can first develop the game in the Python Editor, and later test it on HW. The editor even creates the flash image automatically for you :-)



Pic. 1. Python Editor UI

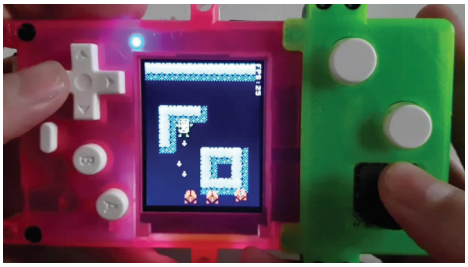
Overview of features

Python Editor has many advanced features that make creating games a blast! In addition to an integrated emulator, there is also an inline bitmap editor, a color palette selector, a drag-and-drop code, graphics, and audio file import tool, an automatic project management system, and an easy to use FPS counter, FPS limiter and font selector. The code editor itself uses syntax highlighting for the Python language, and supports code completion. There are also direct links to the documentation and to the external Pyxel tool.

The Python language itself is a Pokitto port of MicroPython, a great scripting language specially tailored for low resource MCU devices. Don't be fooled by the low specs of the MCU of Pokitto, because with MicroPython it is possible to get over 80 FPS speed in graphics drawing. A lot of power without the complexity of normal CPU based devices, like Raspberry Pi.

Pokitto also has a gaming library, uPyGame, which has all the functions you need to make great games, e.g. key event handling, bitmap and primitives drawing, music and sound effect playing.

Not to mention the ability to take control of the GPIO pin interface of Pokitto for use with your own HW projects or with ready made “hats”. There is really no limits for your imagination!



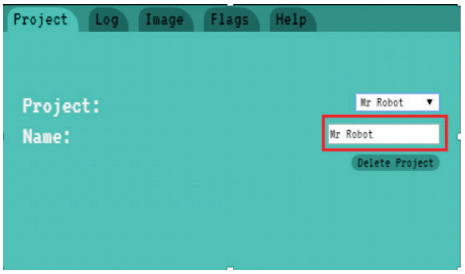
Pic. 2. The PokiTron demo and the Pokitto Joy Rumble hat

Start making a game with Python Editor

Let's make a simple game with Python Editor. The target of the game is to move the poor robot on the bottom of the screen so that it will avoid hitting into the rockets. You are expected to have a basic knowledge of how to code in Python, or if not, you can find the reference here: https://docs.python.org/3/

To start Python Editor just open this web page: https://pyinsky.herokuapp.com/.

First thing to do when making a new project is to decide which example will be used as a base. As this a very simple game that we are going to create from scratch, open the "Hello World" example from the "Examples" drop down list. Then rename the "Hello World" project to "Mr Robot". You can do that, in the "Project" tab in the bottom right window. Renaming creates a new project under the given name. In the "Project" tab you can select, rename or delete a project.



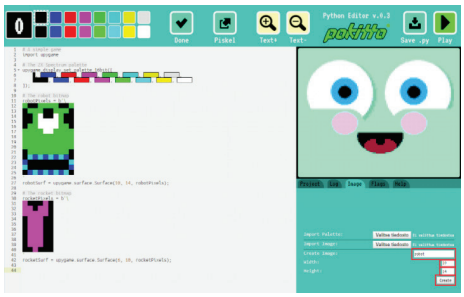
Pic. 3. The Project tab.

Drawing pixels has never been this much fun!

Now we are ready to start making our "Mr Robot" game! Let's start by making graphics, as that is one of the best parts of game making in my mind. Before that we need to have one important python library imported, "uPyGame". All the graphics, audio, and key handling are there because they are not a part of the Python language or the standard libraries. Clear everything in the code editor window, and add "import uPyGame" as the first line (#1). That makes all the uPyGame functions and classes available in our game. The uPyGame reference is here: https://talk.pokitto.com/t/wiki-pokitto-gaming-api-for-python-reference-upygame-and-umachine/1615.

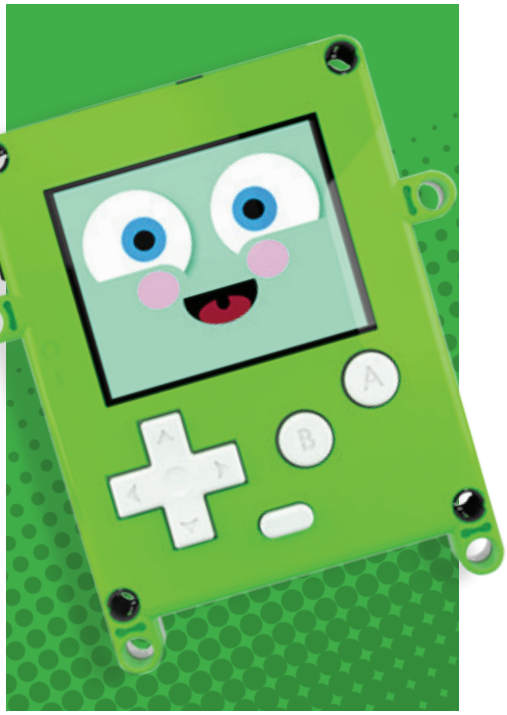
Now we can start drawing graphics. In the bottom right window, select "Image" tab (pic 4). Here you can import ready made palettes and graphics (as e.g. png files), or a create new graphics. Set the image name as "robot", size as 10x14, and press "Create". Now you can see an empty image in the code editor. One of the very nice features is that you can edit the images in the code editor! That is very easy and fun to do, and you can also see the connection of image data and image in a concrete way. Press "Sprite" to switch to the image editing mode. Now you can draw the image using the predefined palette. You can also zoom the image using "Zoom-" and "Zoom +" buttons. Just select any color from the palette and start editing.

Before editing the bitmap, we need some different colors, so copy another palette in the code window. The good old Sinclair ZX Spectrum color palette suites well (#2). Remember to switch off the "Sprite" mode before editing the code. Now recreate the picture of the robot image by using the image editor. If you do not feel like drawing, you can just copy the data below (#3). Create the "rocket" image (6x10 pixels) and draw the picture the same way as the "robot" image. Now it might be a good time to take a local backup by pressing "Save *.py" ;-)



Pic. 4. The image tab and the inline-editing mode ("Sprite").

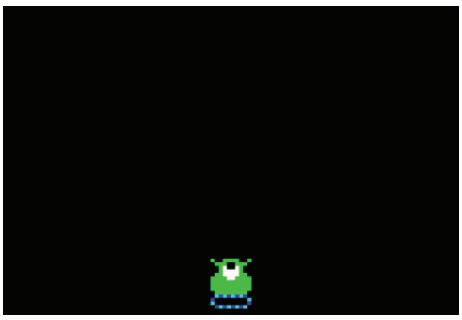
Good! All the graphics are there. Let's draw the robot on the bottom of the screen next. Before drawing anything, the screen object must be created first (#4). Now we can draw the robot using the blit() function (#5).



```
# A simple game
import upygame #(1)

# Setup the screen. #(4)
screen = upygame.display.set_mode()

# The ZX Spectrum palette #(2)
upygame.display.set_palette_16bit([
    0, 0x001a, 0xd000, 0xd01a, 0x06a0, 0x06ba, 0xd6a0, 0xd6ba,
    0x0000, 0x001f, 0xf800, 0xf81f, 0x07e0, 0x07ff, 0xffe0, 0xffff
]);

# The robot bitmap #(3)
robotPixels = b'\
\x40\x04\x44\x40\x04\
\x04\x44\x88\x44\x40\
\x00\x4f\x88\xf4\x00\
\x00\x4f\xff\xf4\x00\
\x04\x4f\xff\xf4\x40\
\x44\x44\xff\x44\x44\
\x44\x44\x44\x44\x44\
\x44\x44\x44\x44\x44\
\x44\x44\x44\x44\x44\
\x04\x44\x44\x44\x40\
\x05\x15\x15\x15\x10\
\x10\x00\x00\x00\x05\
\x50\x00\x00\x00\x01\
\x01\x51\x51\x51\x50\
'
robotSurf = upygame.surface.Surface(10, 14, robotPixels);

# The rocket bitmap
rocketPixels = b'\
\xb0\x00\xb0\
\xbb\x0b\xb0\
\xbb\xbb\xb0\
\x0b\xbb\x00\
\x0b\xbb\x00\
\x0b\xbb\x00\
\x0b\xbb\x00\
\x0b\xbb\x00\
\x0b\xbb\x00\
\x00\xb0\x00\
'
rocketSurf = upygame.surface.Surface(6, 10, rocketPixels);

# End of graphics

# Draw the robot #(5)
screen.blit(robotSurf, 50, 70)

# Update the display
upygame.display.flip()
upygame.display.flip() # The second flip() is needed if there are no main loop.
```

Listing 1.

Trying the game in the emulator



We are now ready to start our game-in-progress. Do you have a Pokitto device? Don't worry if you don't yet have it. You can make the game from start to finish using the integrated emulator in Python Editor. Start the emulator by pressing the "Play" button on the right end of the menu bar. Note that the first time the emulator starts it might take a long time, because the editor must rebuild all the Pokitto source files. The next time the build results are cached in the editor, so startup is much faster.

Did you get an error in the "Logs" tab? The bottom-most line number in the error stack tells the file and the line where the problem is.

When the program has been successfully built and the emulator starts, press the "A" button to start the game. Now you can see the robot, right? If not double check the changes or copy-paste all text from the listing 1 and start the emulator again.

The integrated emulator emulates almost all of Pokitto HW, but there are a few exceptions. The integrated emulator does not support audio, SD card functions or GPIO pins at all. Also the speed of the emulation is much less than the speed of the HW.

Now it is a good time to get familiar with the "Flags" tab on the bottom right window. There are many useful options which you can select for the game:

•**"Maximum FPS"** - This is the maximum frames-per-second you want to have in the game. If you do not limit the FPS value and the timing in your game is based on frame count, you have to change internal timing of the objects every time when total FPS changes. Unfortunately, the value is not working as expected, e.g. giving the 100 FPS limit, limits the FPS to about 50 FPS (!).

•**"Print FPS"** - Prints the FPS value to the top of the screen. It is quite accurate, and with that you can follow the performance of your game as you add features.

•**"Font"** - Again, a very useful feature. Select the font used in your game.

•**"High Res"** - The higher resolution mode, 220 x 176 as opposed to normal 110 x 88 pixels. Note that the graphics will be slower as there is more pixels to draw. It reduces to the available memory for Python too.

Want to try it on the Pokitto device already? That can be done any time. Look at the flashing instructions near the end of this article.

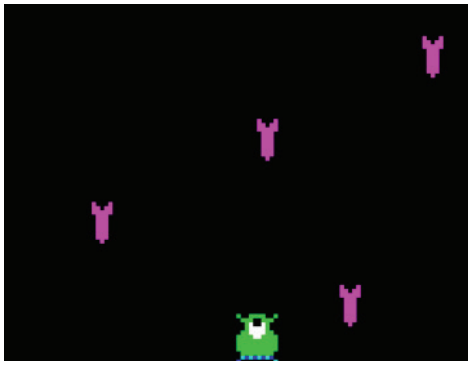
The game loop

To be able to move the robot we can use another uPyGame feature, the key event handling. That gets the up and down events for all the keys of Pokitto: "A", "B", "C", and the directional pad. See is the uPyGame reference for all the event ids. For this game we only use left and right movement. We need to also add something that is common in all games: every game needs: a game loop where there is a screen update in the end. The game loop is used to read the user input at regular intervals as well as move and draw the objects (#6).
... Do not repeat the beginning of the file here as it has no changes.

```
# End of graphics
# The main loop.
robotX = 55
robotY = 74
speedX = 0
exit = False
while not exit: #(6)

    # Read a key event.
    eventtype = upygame.event.poll()
    if eventtype != upygame.NOEVENT:
        if eventtype.type== upygame.KEYDOWN:
            if eventtype.key == upygame.K_RIGHT: speedX = 1
            if eventtype.key == upygame.K_LEFT:  speedX = -1
        if eventtype.type== upygame.KEYUP: speedX = 0

    # Move the robot
    robotX += speedX
    if(robotX < 0 ): robotX = 0
    if(robotX > 100 ): robotX = 100

    # Draw the robot
    screen.blit(robotSurf, robotX, robotY)

    # Update the display
    upygame.display.flip()
```

Listing 2.

Now we are ready to try the emulator again. Try to move the robot to the left and to the right. Nice!



Enemies make the game

Something is missing. What is that? Lets see...the..the enemies! No game is exciting without fierce enemies, which try to bring you down. The eternal battle between good and evil. This time the rockets can play the bad boy role. As there are many of them we create a list of coordinates. For each main loop iteration, we go through the rocket coordinates list and do these operations for each rocket:

1) Move the rocket lower (#7)
2) Check if the rocket has collided with the robot. If collided, this ends the game. (#8)
3) Check if the rocket has gone off the screen. If this is true, we have to raffle the new x-coordinate and move the rocket to the top. (#9)
4) Draw the rocket (#10)

Pro tip: You can use the standalone emulator for SD card support and faster emulation. The standalone emulator can be found here: https://github.com/felipemanga/PokittoEmu

Even more advanced tip: For testing both audio and SD card with Python you can use Pokitto Simulator. That is the same environment that is used for C++ development with the simulator, so you need full Pokitto SDK and tools. In the "Code::Blocks" development environment you should select the "MicroPython" project. The python project files are in the"PokittoLib\Pokitto\POKITTO_LIBS\MicroPython\src_py\" folder. Replace the content with your own project files and rebuild "MicroPython" project. Now you can hear the audio also :-)

Look at the listing 3 on next page for the final game. Congratulations! You have just finalized your first little game on Pokitto! Now you can start the emulator to see it all in action.

```
# A simple game
import upygame
import urandom

# Setup the screen.
screen = upygame.display.set_mode()

# The ZX Spectrum palette
upygame.display.set_palette_16bit([
    0, 0x001a, 0xd000, 0xd01a, 0x06a0, 0x06ba, 0xd6a0, 0xd6ba,
    0x0000, 0x001f, 0xf800, 0xf81f, 0x07e0, 0x07ff, 0xffe0, 0xffff
]);

# The robot bitmap
robotPixels = b'\
\x40\x04\x44\x40\x04\
\x04\x44\x88\x44\x40\
\x00\x4f\x88\xf4\x00\
\x00\x4f\xff\xf4\x00\
\x04\x4f\xff\xf4\x40\
\x44\x44\xff\x44\x44\
\x44\x44\x44\x44\x44\
\x44\x44\x44\x44\x44\
\x44\x44\x44\x44\x44\
\x04\x44\x44\x44\x40\
\x05\x15\x15\x15\x10\
\x10\x00\x00\x00\x05\
\x50\x00\x00\x00\x01\
\x01\x51\x51\x51\x50\
'
robotSurf = upygame.surface.Surface(10, 14, robotPixels);

# The rocket bitmap
rocketPixels = b'\
\xb0\x00\xb0\
\xbb\x0b\xb0\
\xbb\xbb\xb0\
\x0b\xbb\x00\
\x0b\xbb\x00\
\x0b\xbb\x00\
\x0b\xbb\x00\
\x0b\xbb\x00\
\x0b\xbb\x00\
\x00\xb0\x00\
'
rocketSurf = upygame.surface.Surface(6, 10, rocketPixels);

# The list of rocket x and y coordinates
rocketList=[[20,-40],[40,0],[60,-60],[80,-20],[100,-80]]

# The main loop.
robotX = 55
robotY = 74
speedX = 0
exit = False
while not exit:

    # Read a key event.
    eventtype = upygame.event.poll()
    if eventtype != upygame.NOEVENT:
        if eventtype.type== upygame.KEYDOWN:
            if eventtype.key == upygame.K_RIGHT: speedX = 1
            if eventtype.key == upygame.K_LEFT:  speedX = -1
        if eventtype.type== upygame.KEYUP: speedX = 0

    # Move and draw the rockets
    for r in rocketList:

        # Move the rocket #(7)
        r[1] += 1

        # Check for collision to the robot #(8)
        if( abs((r[0]+3) - (robotX+5))<6 and (r[1] + 9 >= robotY) ):
            upygame.draw.text(30,44-6,"GAME OVER!",15)
            exit = True

        # Move the rocket #(9)
        if( r[1] > 88 ):
            r[1] = 0
            r[0] = 5 + urandom.getrandbits(7) * 100 // 128

        # Draw the rocket #(10)
        screen.blit(rocketSurf, r[0], r[1])

    # Move the robot
    robotX += speedX
    if(robotX < 0 ): robotX = 0
    if(robotX > 100 ): robotX = 100

    # Draw the robot
    screen.blit(robotSurf, robotX, robotY)

    # Update the display
    upygame.display.flip()
```

Flashing the game on HW

At last now it is time to try the game on the real Pokitto HW. The "Play" button automatically created the flash image for you. Click the “Download bin” link in the "Logs" tab on the bottom right window to download the flash image. You can then drag-and-drop the flash image file to the Pokitto flash window like any other flash image. I suggest to use PokittoPOP Guided Programmer for the job. It is very handy to use.

Final thoughts

The game is very simple and it can easily be extended in many ways, like adding animation, sound effects, and background music. There are also many features in uPyGame that are not covered here, like the graphics primitives (dot, line, circle), the scrolling tilemap class and the GPIO pin interface.

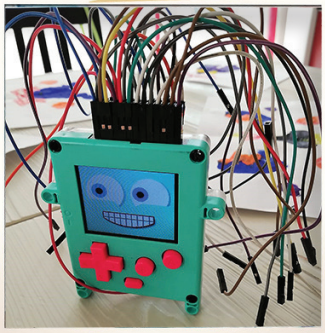
I believe you now have a good idea how to use Python Editor, and hopefully have a lot of fun with Python Editor and Pokitto!

PHOTO BOOTH!

Ready for fun

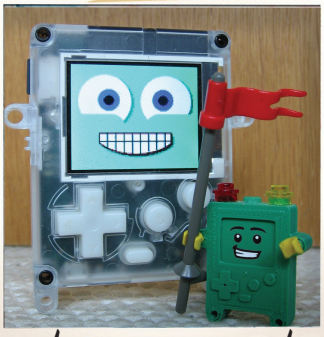
No more coffee for you

Are critters that cute?



Rasta Pokitto

Planet Escape!



Mini Pokitto and Me

POST MORTEM

With Homineludens

A walk in porting a Pc Game to a nice micro controller Console.

Let's compare a modern PC with 4GB of Ram with Pokitto, an embedded console with just 36Kb of Ram. If a Kb of Ram would be a sheet of paper in A4 format, and dropping one sheet over the other, you'll end with a tower about 11 meters tall and more than 500Kg in weight. Against that single sheet of paper. But isn't just a single sheet needed, to make a paper plane fly? (Considering one sheet 0.1mm and 5 gram, 4000000Kb/36Kb=111111*0.1=11mt).

Getting back on track, finally I've collected some time to write the Postmortem of: "**ABBAYE DES MORTS**" porting to Pokitto.

Contact!

When I first heard about this game I didn't know of any Locomalito's productions. It was this post by @ExeDmgDeb that lit a sparkle of interest in this game. The thread exposes, in a rising of emotional sentence, how the game would fit well with Pokitto resolution, looks nice and seems easy to port.
This statement by @jonne stayed printed in my mind:
Game port question - L'Abbaye des Morts
Yes, its very nice source code. Extremely well organized and easy to read. Let's see who gets to put their name in the "Pokitto version by" line.

Discussion then stalled and nothing really happened. At the time I had no free time (what a surprise!) to look further. It was a shame because it really fitted perfectly for Pokitto, and a finished platformer was missing from the available titles at the time.
I just gave a look at Locomalito site and kept mental note to my ever longer todo/wish list.

Abbaye what?

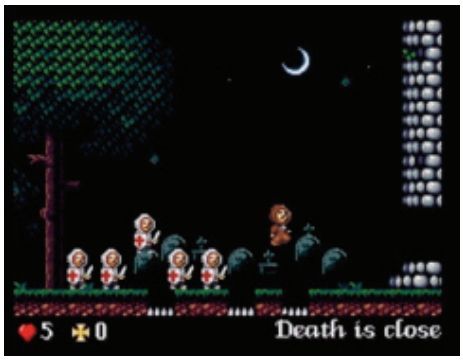
Now just a brief flash-forward to all of what I discovered after the port was finished. The game is dated 2010, original written in GameMaker in a couple of weeks, and later ported, with my surprise, to an incredible high amount of other platforms:

THE MANY PORTS OF ABBAYE DES MORTS

Commodore 64

ZX Spectrum

Mega Drive



OUYA

PC (Win/Linux/Mac) using SDL

HTML5

With my even bigger astonishment, I've discovered there are also some physical versions (box, cassette), that sell at commercial price.

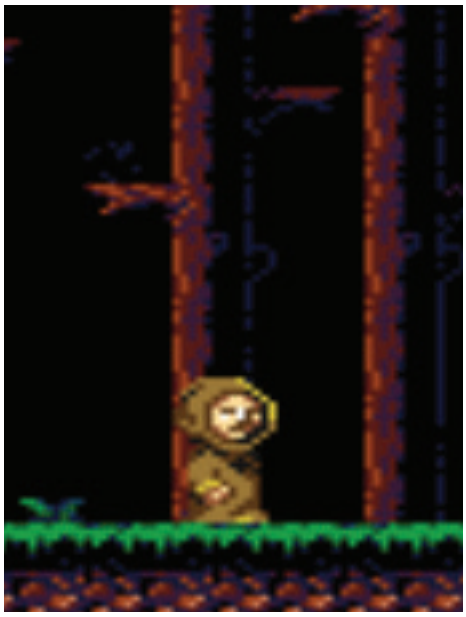
Maybe we made it all wrong guys? Seems like a quite a famous game indeed. All ports look visually similar. They all try to transpose the look and feel of old good times, with that retro feeling and operation nostalgia so abused lately, but that here is fruit of real passion.

There's something else that really connect all of them. Game-play is hard (**VERY HARD**) something that all the various version respect faithfully. It's a platform but with lot of different puzzle, often quite unusual and cryptic. People love it or hate it. Both at the extreme.

Back to the future

Fast forward long time after (Oct 2017 Nov 2018 !!)
I finally found myself with some free time! Newborn in family allow me to take some holidays from work to look after the babies.
So sneaky, I had also some spare time to relax, and spent some coding in a little project.
What could be best of that little game that seems so easy to catch and luckily nobody have already taken over?
I started looking at the PC port, using this github repository .The code is not by Locomalito, but by Nevat.
It's a porting for Linux,Pandora, CGW0, Wii and PSP. All that device in a single codebase? Promising!
The master branch seems the most updated, there's also and older version (v1) that "esta escrito en español". So not an option for me .

At the first look everything seems fine, assets are all there and really high level. All the music and effects are in ogg format, and play awesome. The tiles are all grouped in a big image, with low color and high color mode layered one above the other. (see next page)

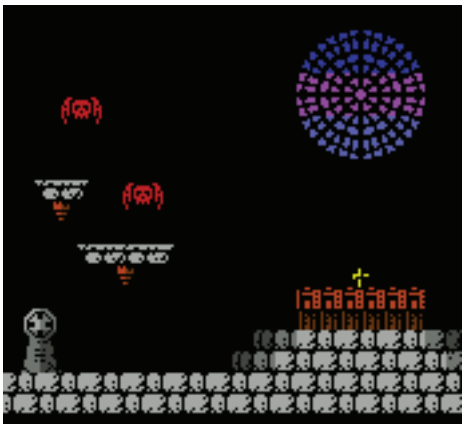


ABBAYE DES MORTS TILESET

This raised the first question: how the game manages the tiles? Warning: technical details ahead! From now on I'll go deeper into code aspects. You could skip some parts, but hey, it's part of the joy of being a programmer, crashing against developing problems.

Jumping around in the code, I quickly found that the game uses SDL mainly to copy some portion of this big image to the screen. It basically setup a rectangle source coordinate and destination, then just copy to the screen. Also there's an in game option to enable a second color set, this it simply adds an offset of 120 pixel to the y-coordinate of the source rectangle, quite odd but direct way, isn't it?

There are just a couple of SDL functions (SDL_RenderCopy and SDL_RenderCopy_Ex) that do all the work.



Nice! I thought, I probably have just to override some method and take place of the SDL ones. So I start to cut everything SDL related in the code, excepted the copy image functions.

I even just didn't pose the problem of the resolution, that by the way you can easily find in the main.c where it set up a screen logic size of 256x192 pixel SDL_RenderSetLogicalSize (renderer, 256, 192);

Pokitto is 220x176 at highest resolution, missing some piece of screen was not my major concern at the time. I created some dummy functions for SDL sound and music, something to finish later. Empty body and a big "TODO:implement sound". Done!

Taking confidence with the code reveal it was all C: no classes, no operator overloading, no exceptions, no enum, no virtual function, etc. Quite straightforward. (What!? Even no Enum !?)

Magic numbers! OMG

Some doubt started to rise in my simple mind. Is it me or are all the calls to the drawing function are hard-coded?
I mean everything: animation, GUI elements, shots, everything graphic is copy-pasted from screen using coded coordinate. Magic numbers, Magic numbers everywhere! The Pandora's box was open and I was still not completely aware of it.

I thought, well just fix this function, just try to show something on screen… Where's the problem with this constants? If it works in PC why shouldn't it work on Pokitto? It's C! It's portable!

It was a slow spiral that attracted me to code more, understanding something, change some aspect to work with low constraints, have some working code, optimize, fix bugs **repeats**.

But I had a plan! One step after the other, just little steps if you feel burdened.

First step: having something to see on simulator

It was really not so hard adjusting the code and having something that could be compiled. Really its just C so at least for the simulator the raw thing can work. Obviously compiled does not mean correct…

To be honest, there was a lot of ugliness here and there, but every time I said to myself: "just fix that small thing and everything will run flawlessly, no need to worry for all

the other mess". I put a lot of dust under the carpet, but Jean (the unlucky protagonist) came to life on the screen.

Step two: see something running on Pokitto

Half of the assets (one type of colorization) can fit on Pokitto flash so that was good enough.
I was aware of memory constraints and the original game was using a big 3d array [room][column][row] of int to store the map. We're speaking of [25]x[22]x[32]x(4 byte)=70Kb of Ram. Nothing on a common PC but simply too much on Pokitto.

To run the test I moved everything to flash, avoiding also to load the map from text file (like the original did). Only the actual room was loaded in RAM allowing to the hard-coded game logic (O_o) to run inside the room. On exiting and entering the room, everything was lost. But some basic game logic run, enemies move, lever acts…

Here again I get a boost on seeing how nice the game was on Pokitto screen. Carrot and stick.

Some defects came out. For example the original game checked for collision outside the array size boundaries (but only in rooms where going out of the bigger array doesn't hurt too much) and there was some strange behavior to fix (you could collect hearts without removing from game) , but the main problem was now speed .

Not so fast, not so furious (yes I'm already tired with the "step" plan)

Everything being hard-coded also affected the overall speed the code ran at. PC version should be something between 60FPS to 65FPS (don't want to search now). Inside the SDL while loop some counters were increased and zeroed on overflow, to regulate animations and speed movements. Again all hard coded, no time elapsed calculation no separation between update and draw.

I try hard to optimize the drawing routine that copy the data from flash directly to the screen buffer. Memcopy, raw copy, changing screen mode to been faster.
After some testing, the best result I could achieve was obtained using Pokitto MODE 13: High memory usage, low resolution, fast copy.

The down side was reducing the view port (110x88) making the game even harder. Anyway this forced me to implement a scroll camera system, and it comes handy later. But no way I could reach 60FPS, simply too much.

So I split some of the code to being called on request, I couldn't simply have some update and draw function, because the logic was mixed in lot of different functions.

At the end I set the game at 20FPS, calling 3 times the game logic but enabling the draw bit only at first cycle. A monstrosity.

It was also the time I made some play on the PC version of Abbaye. I found it too hard and complain about my choice. Why did I embark on this adventure?

I was ready to leave everything, lot of roller-coaster emotion on the porting till now, but with so much done and so little to finish… (fool me)

911? There's an emergency. Time to call for help

It was 10 Nov when I expose the game state to Jonne.
I sent a PM asking the best way to speed up drawing and we quickly fix some of the problems. Jonne even add some music ! Awesome. Even more, Jonne said he would like to make a special promotion for the "black Friday" (23 Nov) featuring Abbayes on Pokitto! So cool. Super happy.

It was time to close the problem with the room serialization. I was only trying to play the first couple of rooms, and was terrorized by the idea that the logic was broken somewhere else. I managed to read the map from file in the simulator, but nothing works on Pokitto with SDF (and advanced SD Filesystem library with extended function to manipulate files and directory on sd card). Something broke on my setup. Cross testing with Jonne gave strange results too.

I propose we explain the problem to Felipe and quickly involve two of the greatest coder in Pokitto community: Felipe @FManga and Miloslav @drummyfish .

That raised my mood, and made me hope again for a happy ending.

Pokitto special force to the rescue

Keep close, at this point story speeds up! Few messages to share the problem: the team warm up fixing one issue after the other. The problem with the file system was quickly knocked out.

Great piece of code rain to my base rising FPS, loading and saving map correctly, allowing images to be loaded from disk and drastically reducing ram usage.

I know I'm over simplifying but everything flowed the right way. That's probably the luck of teaming up with the right people.

Some of these improvements are now part of PokittoLib and everyone can benefit. FOSS power! Resolution went back to 220x176 pixel at 16 color keeping the scrolling camera streaming at 30FPS.

Meanwhile Jonne kept pressing us, showing beautiful posters of black Friday events. No pressure, no pressure at all!

Great leadership!

We used to share piece of code in a private thread on Pokitto Forum. That was quickly getting tedious and risky. We developed in different OS and toolchain and we needed a common way to contribute the project. So I embarked on setting up a private GitLab repository for the first time. If you work on group it's a must have. It's the best way to work together and share the code. You can clearly see what's changed and keep working on your code without hurting the others. Due to different time zone, the code kept flowing without stops. Glorious.

Time for official disclosure came, we knew some minor issues (i.e. slow down in dragon room) was still there but final result seemed good enough.

Too hard to try (Lurking bugs)

There were bugs even on the ending sequence, that we left un-played for a long time, with a good reason. It's not easy to test a game where you need to be a PRO to finish. Miloslav played the game something like 80 times in a row, trying to finish without cheating. And never did it. That's forced us to bypass some game logic and add God mode. Even that way... it takes some times to face a problem at the end of the game. Abbaye is a work of passion, handcrafted to tell a story, and the final room is part of this. And was flawed. The very last night we discovered the game was not ending the same way some long-play video show us. It was just showing odd images and crashes. In a rush of furious debug and rewrite the fix was ready and so the final binary.

Probably that bug came from the needs of optimization, like moving data from float to int, or saving the room to disk. Some other defects were present also in the original source code (crouch same speed as walking, array overflow). Some were hard to find or reproduce, and time was always ticketing away. Towards the end, I had the feeling that the code was cryptic by choice. Why avoid using enum for game state? Why avoiding using struct to organize player data on a easier way? There is still an array of int in "Jean" data, marked as flags[ ] that I don't know what they exactly do. Some are time related and other action related for story cutscenes.

Things can be made in many ways, some are good some just work, but for sure without Nevat's porting I'd never start porting Abbaye to Pokitto.

It's ship time!

Releasing was both exciting and frightening. Fixed all major bugs fine tuning some arts: converting image to Pokitto format (or the new handy "*binary SD card format* " developed those days in the spare time), test various sound effects and soundtrack for the game, menu navigation and secret hard mode (Ups! that's should be secret, and was added after the release also… so better not speaking anymore).

Binary sd card format: the idea here was to just load all the byte in the framebuffer directly from sd card, without passing from Pokitto RAM. This way was extremely inexpensive to load full screen images like the one on the start screen. It is not really fast but works great to save precious RAM. It has been Felipe, at time still living in Brazil, that pop out with the idea, probably when he was waiting for us to wake up here in Europe.

The end

The game is fully playable (if you're good enough) and reflects the original work. It comes in two flavours, low color and high color mode with music sound track and sound effects.

Source code available at https://gitlab.com/HomineLudens/abbaye-des-morts-pokitto

POKITTO PORT COMES IN 2 FLAVOURS

LOW COLOR MODE

HIGH COLOR MODE

Thanks. Many thanks!

I want to thanks all teams members that allowed me, at least once in my life, to be part of a real Game DevelopmentTeam:

Jonne, to be always available to give, without something in return

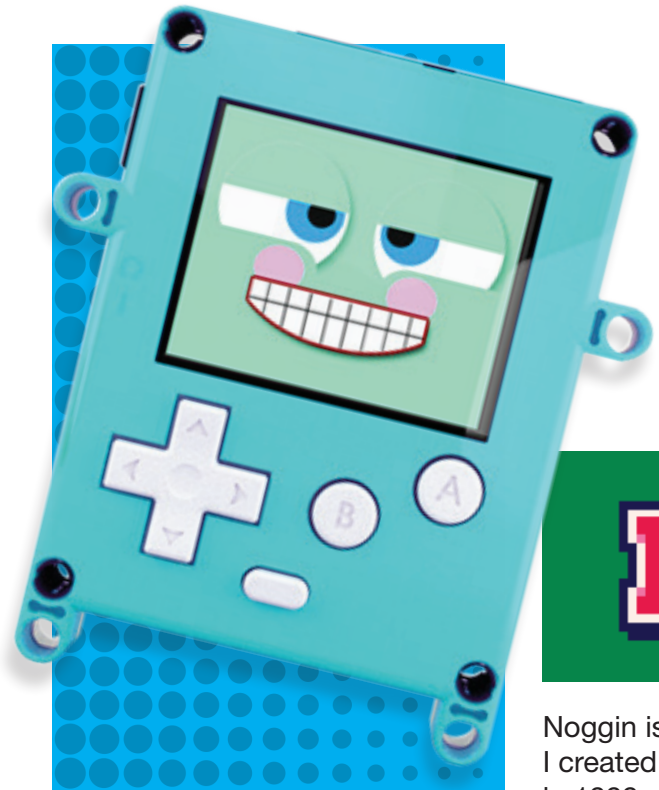
Felipe, to share freely his incredible coding experience

Miloslav, to show clearly the right way to make things across different development aspects

Tor @torbuntu to help me redact this article and not let me procrastinate again the submission of this article.

It has been epic and a lot of fun. Let's do it again!

I hope you'll appreciate it and have fun.



Game Highlight

With Spinal

NOGGIN

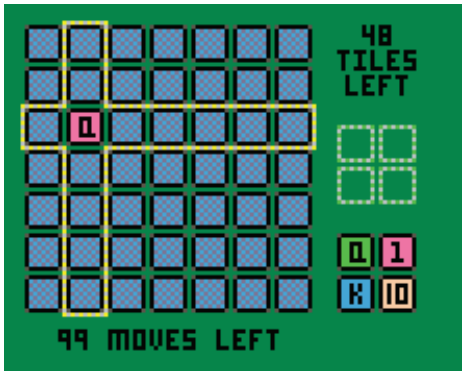
Noggin is a puzzle card game that I created alongside my stepbrother in 1999. The idea came about when he was sat playing pairs with a deck of cards and was trying to develop a system that would allow him to cheat at the game. He started by turning the first card over and would then move that number of spaces away from it to find the pair. For example, if he turned a 6 he would only move 6 spaces away from the card. While this technique didn't work as a way of cheating at pairs it did create an interesting idea for its own game, so we decided to make some rules for it.

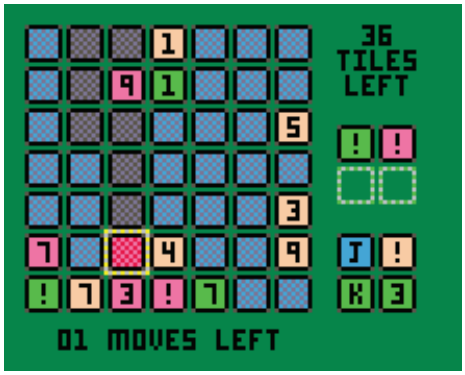
Noggin starts with a 7 x 7 grid of randomly shuffled cards all face down with the exception of one, the aim of the game is simple, turn the rest of the cards face up by moving the number of spaces shown on the most recently turned card. While it sounds simple on the surface, it does take a bit of strategy and a lot of luck to accomplish thanks to the rules we created. These include, only being able to move horizontally or vertically, not being able to cross over a card that is already face-up and special abilities for the picture cards. The Kings allow you to move to any face-down card that is vertical or horizontal to it, even if there is a face-up card in the path, the Queens will turn over all cards on either the vertical or horizontal path and the Jacks take a card from the pile of leftover cards, using that card for the number of spaces it can move.

After watching my stepbrother and other family members having fun in their attempts to turn over all the cards, I decided to develop a Windows version of the game. Unfortunately, due to a couple of coding bugs and the original rules, the game was nearly impossible to win and in the 20 years since being built to my knowledge, only one person has ever won.

When the Pokitto python contest was announced, I decided it was the perfect time to remake Noggin. Using the original rules I created with my stepbrother, I set about building the game, however, once it was complete it was still nearly impossible to win. That was until my wife made the suggestion of adding 4 jokers into the pack with a special 'get out of jail free' ability. This new rule meant that when

the player turned over a joker they could only move one space, however, they would also collect the card and should they become stuck can use it to move to any of the remaining face-down cards. This small change made a big difference, as while the game still takes strategy and luck, it is now a lot easier to win.





7 USEFULL TIPS FOR BEGINNERS

1

Don't flip the cards over randomly, this is the easiest way to block your path and will cause you to get stuck very quickly.

2

Without the aid of a special card, you cannot cross over cards that are already face-up, so the best strategy is to get to the outside edges as quickly as possible.

3

Always try to leave a path to face-down cards, blocking a card will make it more difficult to turn over all the cards.

4

One strategy that works well is to keep going in circles around the outside edge until you make your way to the centre.

5

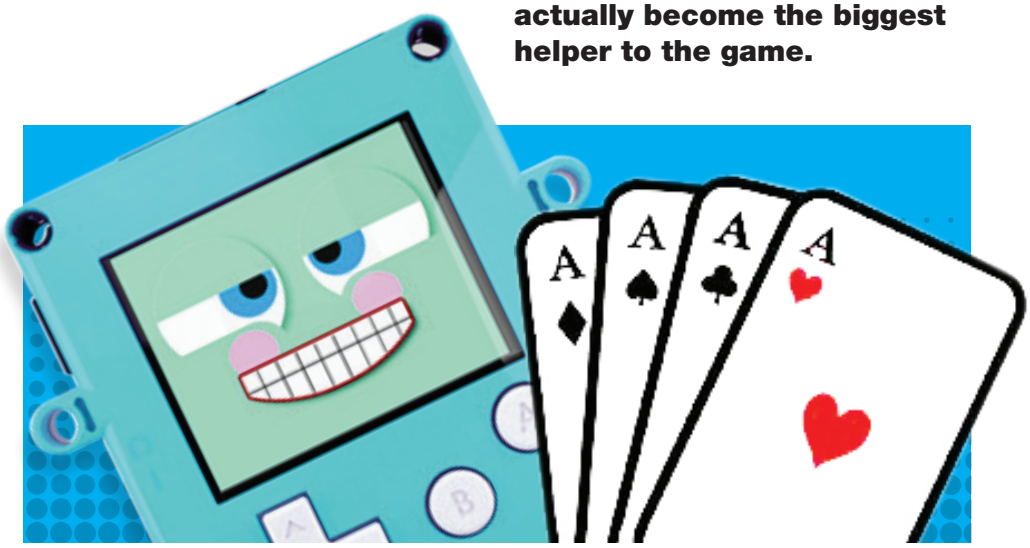
Another strategy is again to start from an outside edge and flip the cards row by row, or column by column from one side to the other.

6

Queens can cause big issues, due to them turning over all the cards in a line, but if you have developed a good gameplay strategy they can actually become the biggest helper to the game.

7

If you have collected one or more Joker cards keep them for as long as possible. These are your lifeline and if you're able to keep them to near the end of the game you will have a better chance of winning.

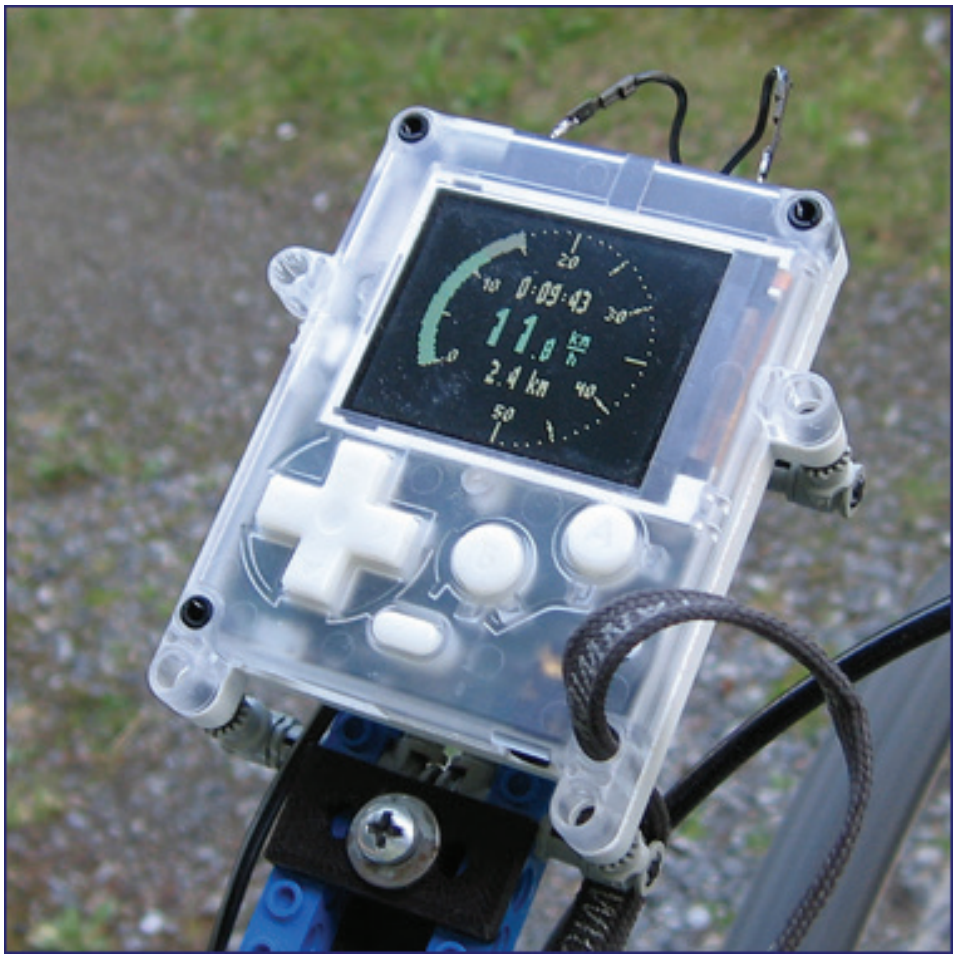


PEXploration LAB Projects

With jpfli

Bicycle speedometer

For my first Pokitto hardware project I decided to make a bicycle speedometer. I cycle quite a bit and thought it would be a good project to start with because the sensor is a simple magnetic switch that is directly connected to the Pokitto without any additional components. This article covers the basics of how to read the sensor using interrupts and calculating the speed.



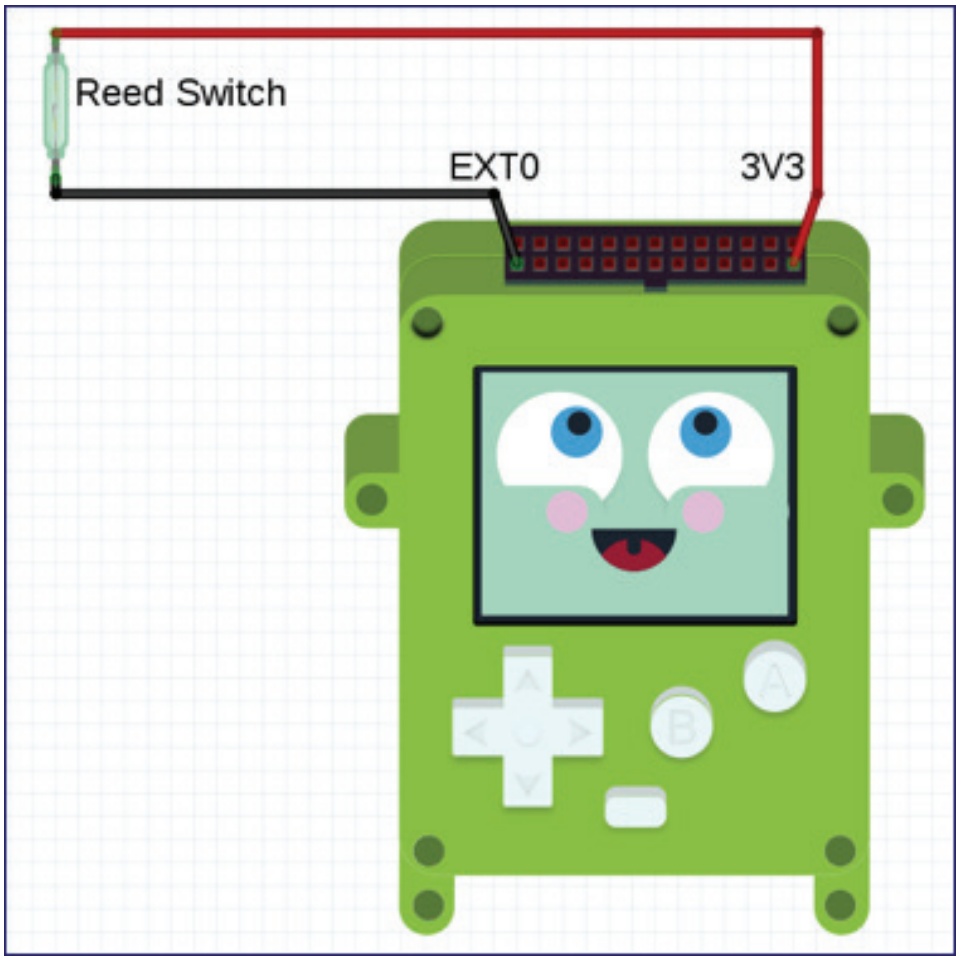
The speedometer works as follows. There is a magnet attached to the front wheel and a reed switch to one of the fork legs. Once every revolution the switch gives a short pulse, which is detected by Pokitto. The speedometer program then calculates the speed based on the time between pulses and the given circumference of the front wheel.

Reading the sensor

The reed switch is connected between pins EXT0 and 3V3 on the Pokitto as shown in the circuit diagram. Normally the switch is open, but each time the magnet comes close to the switch, it closes and sets EXT0 in a high state. An internal pull-down resistor is enabled to set EXT0 in a low state when the switch is open.

In the code below I use interrupts to detect when the state of EXT0 changes. Interrupts are needed because EXT0 is high just a few milliseconds each revolution. Especially if you want to display some graphics at the same time, it would be impossible to reliably detect such short pulses by just polling the state of EXT0. When an interrupt is triggered a different handler function is called depending on whether it is a rising or falling edge. On rising edge the current time is stored in the variable timeLastEdge which is then used in the main update function to calculate time between pulses.

The problem, however, is that with mechanical switches there can be bouncing when the switch opens and closes, causing false readings. Although a simple RC filter could be a better solution, I chose to handle the problem purely in code to avoid using additional components. The way I handled bouncing is that on every rising or falling edge the current time is stored in the variable ***timeLastInterrupt*** and when a new rising edge is detected it is handled only if there has not been any previous interrupts within 30 milliseconds.



Here is the code for the interrupt handlers:

```
// Time (milliseconds) to wait before accepting new rising edge
const uint32_t DEBOUNCE_TIME = 30;

// Time of last interrupt
uint32_t timeLastInterrupt = 0;

// Time of last accepted edge (must be volatile because used in both
// the main code and interrupt handlers)
volatile uint32_t timeLastEdge = 0;

// Init EXT0 as input pin with interrupts and internal pull-down
// resistor enabled
InterruptIn pinExt0(EXT0, PullDown);

// Rising edge interrupt handler
void handleRise() {
  uint32_t timeNow = Pokitto::Core::getTime();
  if(DEBOUNCE_TIME < timeNow-timeLastInterrupt) {
    // Only accept this edge if there has been no previous interrupts
    // within DEBOUNCE_TIME
    timeLastEdge = timeNow;
  }
  timeLastInterrupt = timeNow;
}

// Falling edge interrupt handler
void handleFall() {
  timeLastInterrupt = Pokitto::Core::getTime();
}

init() {
  // Attach interrupt hadlers for rising and falling edge
  pinExt0.rise(&handleRise);
  pinExt0.fall(&handleFall);
}
```

Calculating speed

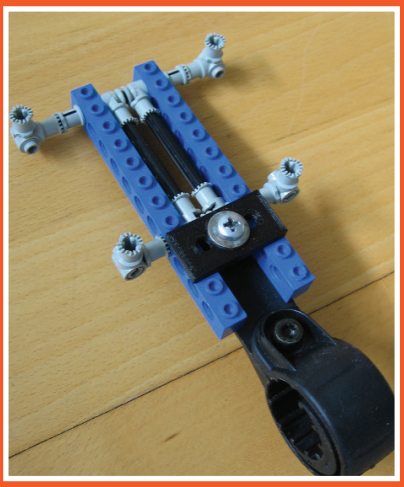
The time between two pulses from the sensor is the wheel revolution time. Dividing wheel circumference by the latest revolution time gives you the current speed. A minor drawback is that the speed value stays constant for a full revolution and then jumps to a new value when a new pulse is detected. The jumps can only be noticed when accelerating or braking rapidly at slow speeds and are not really a problem. If you want the speed to change more like an analog value, you could calculate an average speed using the most recent revolution times within a time window of a few seconds.

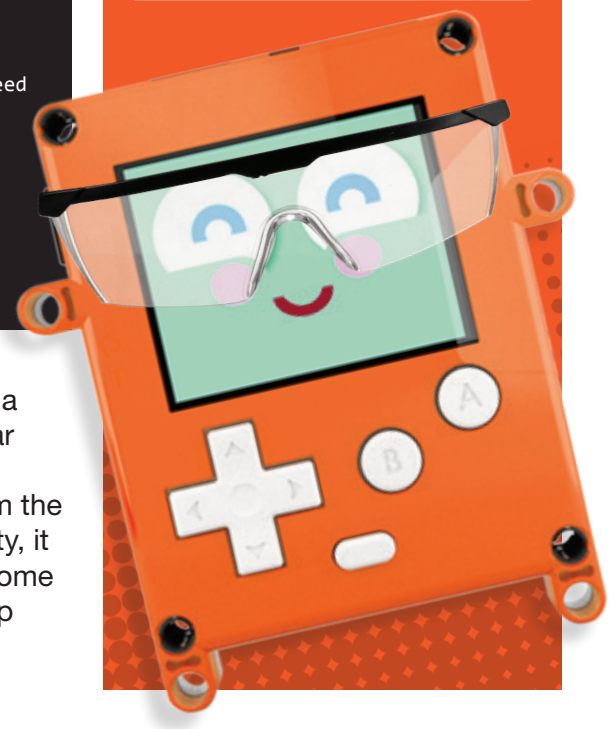
The main update function is shown in the next code listing. First the value of ***timeLastEdge*** is checked. A change in it's value signifies that a rising edge from the sensor was detected in the interrupt handler, and a new revolution time should be calculated. It could also be calculated in the interrupt handler, but later you may want to add more things to be done when a new edge is detected and it's good to keep the number of variables modified in the interrupt handler at a minimum.

Next we get to the speed calculation code. Instead of using only the last revolution time, we take the idle time since the last pulse, and if it's greater than the last revolution time, use that to calculate the speed. This ensures that the speed value starts decreasing if no new pulses are detected. It also helps to smooth out the changes somewhat when speed is decreasing.

Here is the code to calculate the speed:

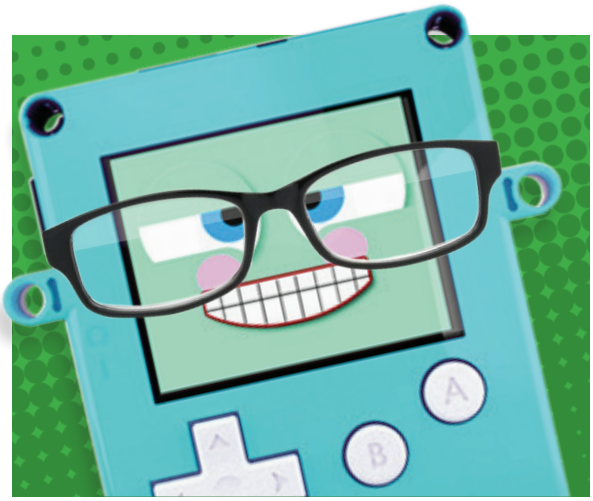
```
// Longest time (milliseconds) between pulses before the bike is
// thought to have stopped
const uint32_t TIMEWIN = 3000;
// Wheel circumference in millimeters
const uint32_t CIRCUMFERENCE = 2120;
// Number of seconds in one hour
const uint32_t SECS_IN_HOUR = 3600;

// Latest revolution time
uint32_t revTime = 0;
// Time of last pulse
uint32_t timeLastPulse = 0;
// Bike speed in meters/h
uint32_t speed = 0;

void update() {
  // Check if a new pulse was detected and calculate new revolution time
  uint32_t timeEdge = timeLastEdge;  // snap a copy of volatile variable
  if(0 < timeEdge-timeLastPulse) {
    revTime = timeEdge-timeLastPulse;
    timeLastPulse = timeEdge;
  }

  // Choose the greater of idleTime and _revTime to use in calculating speed
  uint32_t idleTime = Pokitto::Core::getTime()-timeLastPulse;
  uint32_t dTime = (idleTime>revTime) ? idleTime : revTime;
  if(dTime > TIMEWIN) {
    // No new pulses for a long time, bike has stopped
    speed = 0;
  }
  else {
    speed = (SECS_IN_HOUR*CIRCUMFERENCE)/dTime;  // speed in meters/h
  }
}
```



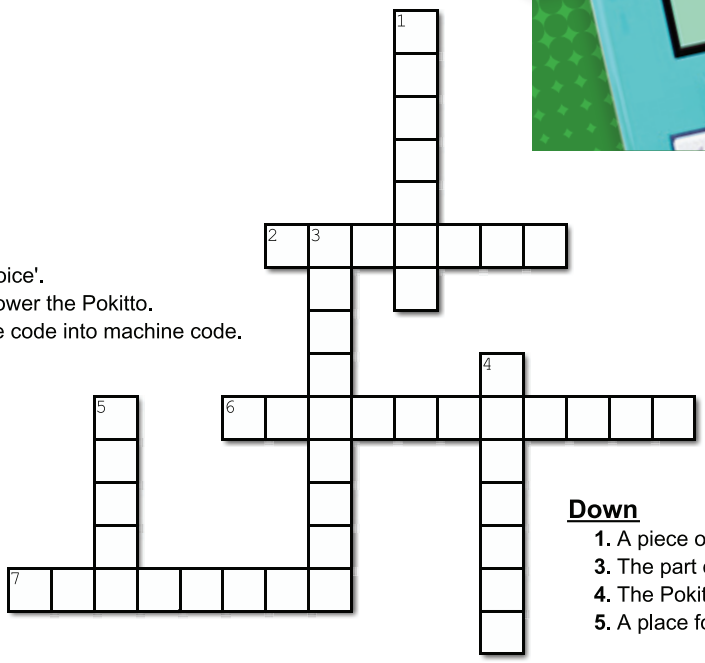


The calculated speed can then be printed on the Pokitto screen as a number, or you can implement some rotating needle or progress bar style speedometer view. Also other information, such as distance, average and maximum speed and cycling time, can be derived from the detected pulses. Once you have implemented the basic functionality, it is easy to add more features to the program. You could even add some game elements, for example earning points the longer you can keep your speed above 30 km/h.

pokitto
CROSSWORDS CORNER



LEVEL 1



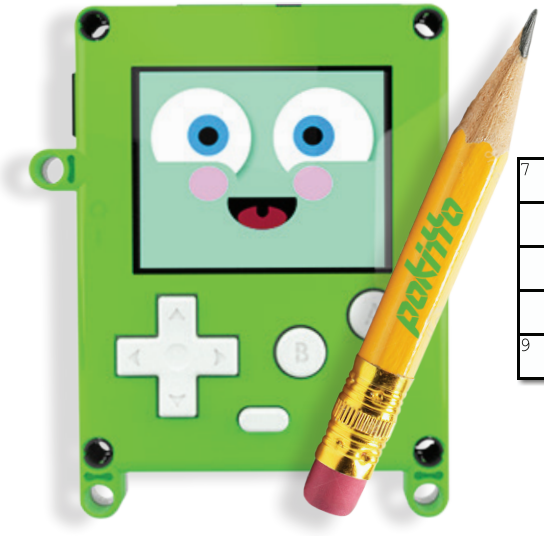
Across

2. The source of the Pokitto's 'voice'.
6. The type of energy used to power the Pokitto.
7. Software that converts source code into machine code.

Down

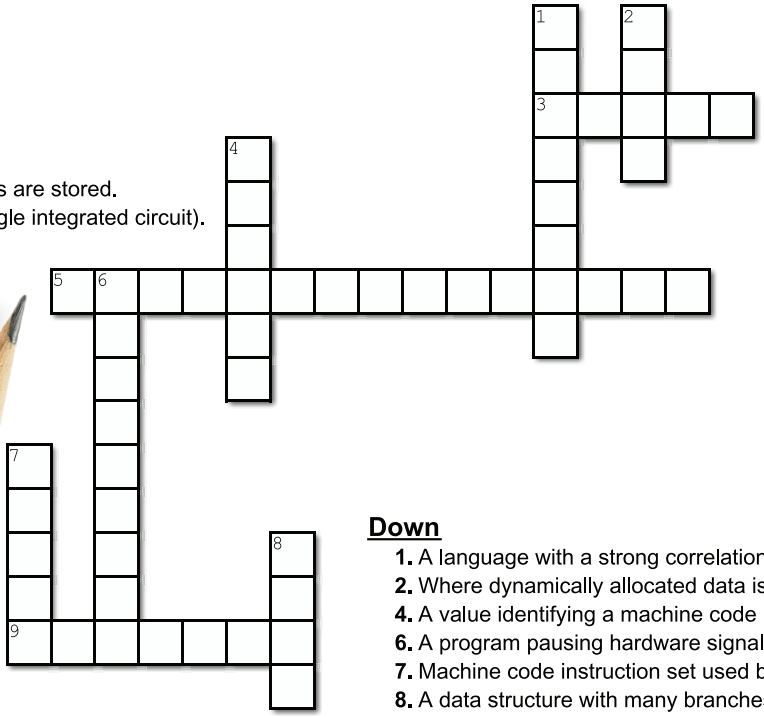
1. A piece of software that runs on a computer.
3. The part of a computer that executes machine code.
4. The Pokitto's homeland.
5. A place for discussion (on the internet).

LEVEL 2



Across

3. Where function return addresses are stored.
5. A very small computer (on a single integrated circuit).
9. Pertaining to bit manipulation.



Down

1. A language with a strong correlation to machine code.
2. Where dynamically allocated data is stored.
4. A value identifying a machine code instruction.
6. A program pausing hardware signal.
7. Machine code instruction set used by Pokitto.
8. A data structure with many branches.

PEXploration LAB Projects

With FManga

Taking Pictures With a Pokitto

That little guy is making a claim to the title of “First Pokitto to Take a Photo.” It’s just a selfie, but he took it mostly on his own. Let’s take a look at how he did it so you can make a photographer out of your Pokitto. While this is partly a hardware project, I think it’s pretty easy even though I’m a software guy.

To do this, you’ll need the following:

- A camera with a remote release connector
- Parts to make a PEX-to-camera cable
- FemtoIDE
- A Pokitto

This is a hardware project, you're not going to get very far in an emulator. Also, the obligatory disclaimer: if something goes up in smoke, you get to use it as a paperweight. I'm a software guy.

Triggering a camera electronically is generally pretty simple, though it varies from one model to another.

To find out how to make the PEX-to-camera cable, check out: https://www.doc-diy.net/photo/remote_pinout/. It has information on many different camera models.

In general, the camera takes a picture when the trigger pin is shorted to ground. In my case, I'll be using a transistor (2N2222, I think… it's what I had in the bin) to do the shorting and the base (middle leg) goes into EXT17. The camera's ground goes into the PEX's ground. That circuit probably needs some resistors somewhere. I'll add them in software. Did I mention I'm a software guy?

For testing, I'll simply plug an LED into EXT17 and ground, without a resistor. If you haven't done so yet, download FemtoIDE from here: https://github.com/felipemanga/FemtoIDE/releases. Extract it somewhere, run IDE and start a new Java project. Most of what we need is ready out-of-the-box. We just need to add one line of code so we can start programming the PEX header:

```
import static femto.hardware.EXT.*;
```

With that we can easily control each pin. To set EXT17 low, then high, we enable output and write to it:

```
PIN17.setOutput();
PIN17.write(false); // low

PIN17.write(true);  // high
```

That will set the pin to 3.3v, without limiting current… which is bad if you're driving a transistor or an LED. To limit the current we need a resistor, as mentioned earlier. Luckily, the Pokitto has some inside and we just need to trick it into using them:

```
PIN17.pullDown();
PIN17.setInput(); // low

PIN17.pullUp();

PIN17.setInput(); // high
```

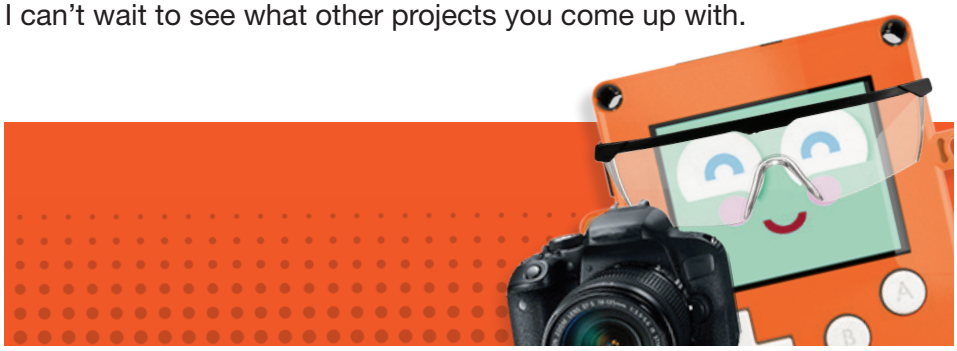
You need to ***call setInput after changing pullDown/pullUp***, but it's still pretty simple. With that, we have all that's necessary to control our camera/LED:

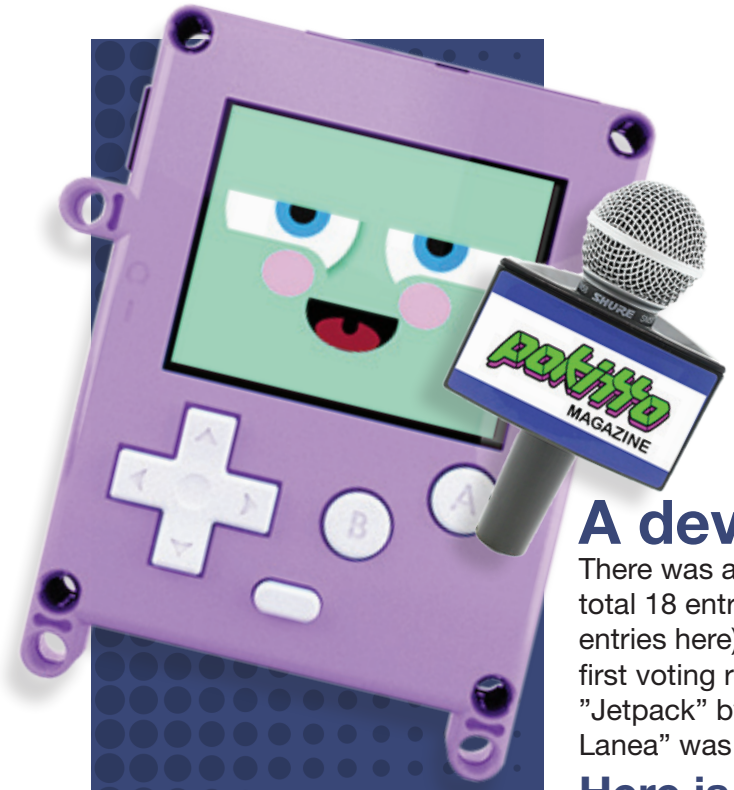
```
import femto.input.Button;
import static femto.hardware.EXT.*;
public class Main {
  public static void main(String[] args){
    PIN17.pullDown();
    PIN17.setInput();                    // initialize with pin LOW
    while(true){                         // loop forever
      if( Button.A.justPressed() ){      // press a button, take a picture
        PIN17.pullUp();
        PIN17.setInput();                // set pin HIGH
        while( Button.A.isPressed() ); // wait until the button is released
        PIN17.pullDown();
        PIN17.setInput();                // set pin LOW
      }
    }
  }
}
```

If we want to make an intervalometer that takes a picture every 5 seconds, we can use ***Thread.sleep***:

```
import java.lang.Thread;
import femto.input.Button;
import static femto.hardware.EXT.*;
public class Main {
  public static void main(String[] args){
    PIN17.pullDown();
    PIN17.setInput();                    // initialize with pin LOW
    while(true){                         // loop forever
      Thread.sleep(5000);                // 5000ms = 5 seconds
      PIN17.pullUp();
      PIN17.setInput();                  // set pin HIGH
      Thread.sleep(100);                 // wait a bit before setting the pin low
      PIN17.pullDown();
      PIN17.setInput();                // set pin LOW
    }
  }
}
```

We've turned our Pokitto into either a remote camera trigger or an intervalometer. Now we just need to make a nice UI, but that's another article. Now that you've seen how to control things with the PEX in Java, I can't wait to see what other projects you come up with.





Community Interview

With dir3kt

A developer interview

There was a Python GameJam for Pokitto on March, 2019. There were total 18 entries in the competition, all were open source (look at the entries here). The community was voting for the overall winner. The first voting round was a tie between ”Legend of Lanea” by dir3kt and ”Jetpack” by bl_ackrain. After the second voting round ”Legend of Lanea” was chosen as a winner!

Here is the interview of the winner, dir3kt.

Tell us something about yourself

My name is Joel, and I’m a dad/husband/coder from Switzerland. My day work is about writing software, and my hobby is… writing software. Ok not only but this is the part of my life that interests you right?

Although I followed a curriculum in computer science, I’m more of the self-taught type. Back 30 years ago, my parents bought the first computer we had at home: An IBM PS/2. As we lived in the countryside and the internet didn’t exist, the little eight-year-old me was alone trying to unlock the mysteries of the magical computer.

One day I eventually discovered something called QBasic which came with two games: An artillery game and a snake game. I quickly started to hack the games by changing bits here and there. But this was not enough. In the end, I decided to create my own game: Star Commander. It was a glorious shoot’em up with err… An ‘x’ character (the aim) that

“Picture of me playing Prehistorik 2 on my good old PS2”

you had to move on a randomly moving ‘*’ character (the enemy). I hope one day I could make Start Commander 2, maybe with more enemies (for example an ‘#’ character for a second enemy).

Thirty years later, I’m here. I am writing games as a hobby, trying to break reality and make a living from it.

How did you get to know Pokitto and the Python GameJam?

Like almost everything nowadays: The Internet. In 2017 I was

participating in the CGAJam for which I made a game called Floppy D!sk (https://zappedcow.itch.io/fl0ppydsk). In the jam's tweet feed I saw a nice looking rogue-like that seemed to work on a strange alien device. This game was Columns&Coffins by Adekto community member (editor note: the game was actually a joint effort by Adekto, vonBednar, wuuff, trelemar and jonne). It was the first time I heard about the Pokitto.

Why did you choose to make a Zelda-style game, Legend of Lanea?

Strangely I have never been a big fan of Zelda. I had some friends playing and praising it back in the NES/SNES era. For some reason, the top-down aesthetics, controls and combat system stuck with me. I always wanted to make a similar game in terms of visuals and mechanics.

As the Python API for Pokitto had support for tilemap, and scrolling seemed fast enough, I thought making a Zelda-style game was a good idea.

Also, the limitation of having only 16 different tiles (4 bit) looked like a fun constraint to play around. I do believe a lot in working with constraints; it's a source of creativity.

How did the development went (time spend, easiest and hardest points, etc.)?

The beginning was super fast and energising, thanks to the Python language, the API and the sprite editor (which I also used as a map editor). Having the first proof of concept was by far the easiest part.

Towards the end of the development, I started to have random crashes when changing areas. I wrote the game in the way I would write any OO program: By creating a new class for each object that exposes a different behaviour. It turned out this was not scaling well with Pokitto's limited memory. Almost one day before the deadline, I had to refactor all my nicely separated entity classes into a single class with lots of if conditions.

How was the development environment (Python Editor, documentation, etc.)?

Great! I could do everything online from a single web page: Write the code, work on the assets and test the game. This makes Pokitto's dev cycle so much faster; it's crazy! The use of Python makes a lot of things faster, and also it's a great language to write data structures, which is something beneficial for game development.

How do you feel about Pokitto Community?

I think the community is rather small, but it is so damn welcoming and friendly. I have rarely seen that. If you have any issue with your Pokitto, or with development, for sure someone will help you real quick.

I want to take the opportunity of this small interview to thank all the member of the community for their kindness and energy!

What other platforms are you coding games for?

Coding games for handled is rather the exception for me. I made a game for the Arduboy and fiddled with Gamebuino a bit. But I mostly write games for desktop, web and sometimes mobile.

Tell us your 3 favorite games for Pokitto

In no specific order:

Noggin!

This game has a simple mechanic and simple aesthetics. But it has been so carefully executed. It definitively proves that less is more.

Pokitto GP

Not only Pokitto GP is technically impressive in terms of graphics rendering and provides smooth racing gameplay. But it also comes with a way to make your tracks. This is typically a game that empowers the spirit of the Pokitto: Making something that the community can use to get creative.

Squiddy

This game has an interesting movement mechanic and makes good use of it. My kid played it a lot. It helped him acquire the hand dexterity and spatial reasoning necessary for playing video games.

How do you think Pokitto is different from the rest of the handhelds?

To me, there is one fundamental difference with Pokitto regarding other handhelds: Jonne. With other (indie or not) handhelds, there is this wall between you and the maker(s). This is because you are regarded as the customer of a product. But with the Pokitto, it feels more like being part of a family. I think Jonne's vision is very different than other creators. He crafted something to share his love for creativity, electronics and hacking. Something that can help people learn while having fun. Of course, I might be all wrong. Maybe Pokitto is the first stepping stone in Jonne's 13 steps plan to conquer the world.

Any advice for the newcomers (coders) in Pokittoland?

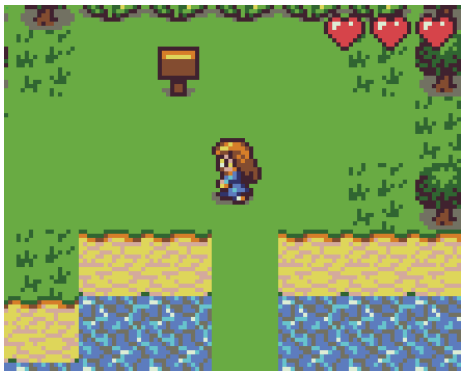
First of all: Welcome to Pokittoland!

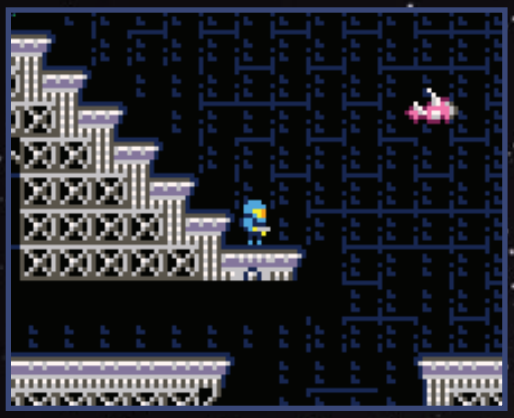
It's always the same: You need to find an edge, a way to enter. Start

with what's the easiest for you. One main advantage of Pokitto over other "8 bit consoles" is that C++ is not a barrier. You can also write software in Python and Java! So pick the language you are most comfortable with and start a project that is within your reach. If you don't have experience in game development, you can also make a tool, an editor, anything. Also, the community, either the forums or the discord server, is there to help.

Are we going to see more great games from you for Pokitto?

Due to another side project that is taking all my time, not now. But Pokitto is here to stay, so I'm sure I will come back to it someday. Hopefully, I could finish Legend of Lanea in the future.





With four massive sectors to explore, will you survive?

Discover and eliminate the unleashed biological specimens.

Use your Omega System to find your way through this tidally-locked hell in orbit.

OMEGA HORIZON MISSION 2

All Omega Soldiers mobilize now! Distress signals have broadcast a security breech at the Exogamma Biological Research Station in orbit around the planet Oltura. Preliminary lifescan reports have identified a massive viral outbreak of an unknown source among the specimens. All attempts by research staff to establish quarantine have been unsuccessful, prompting an eradication protocol to be ordered.

As always, Soldier...Failure is not an option!

Omega Horizon Mission 2 features:

- Massive world to explore
- Map system to help you find your way
- Save system to keep your progress
- Responsive control
- Original in-game music
- And much more...

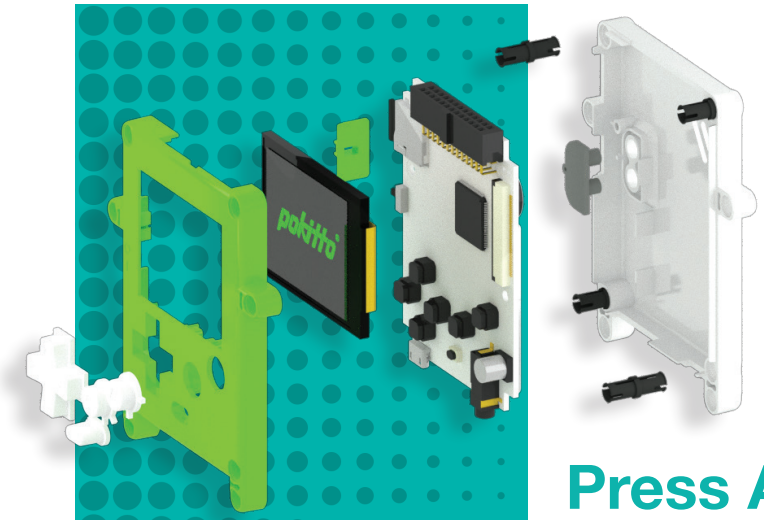
RELEASE
Q4 2019

From the creators of

THE ROGUE Rayne PART II

NEO RETRO

neoretro.games

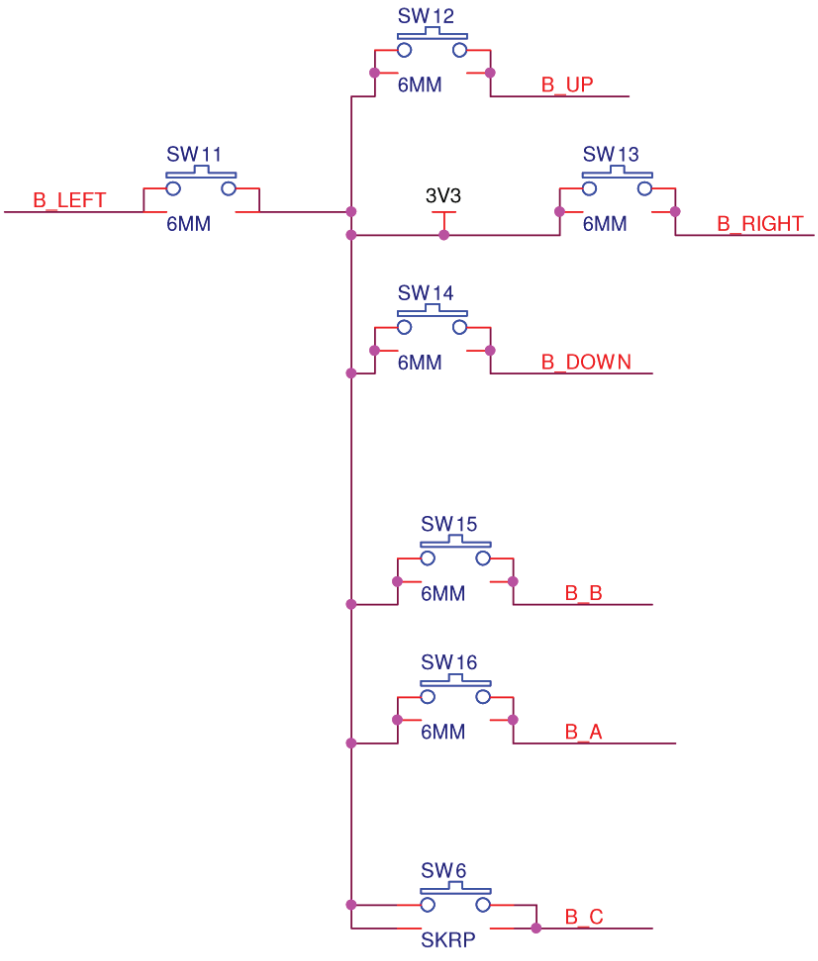


Hardware In Details

With Jonne

Press Any Button!

Let's start with the easiest part of Pokitto hardware schematics, so that you learn how to follow schematics. The Pokitto front buttons (directional pad and A,B,C) are all connected to the 3V3 rail (3V3 signifies the 3.3 volt operating voltage of Pokitto). A button switch is normally open. When a button is pressed down, the pin on the the microcontroller receives the "high" signal (the 3.3 volts) and knows that a button is now pressed. The button-reading pins are marked B_A, B_B, B_C, B_LEFT, B_RIGHT, B_UP and B_DOWN. Next, we will look where they go on the microcontroller.



MCU port 1

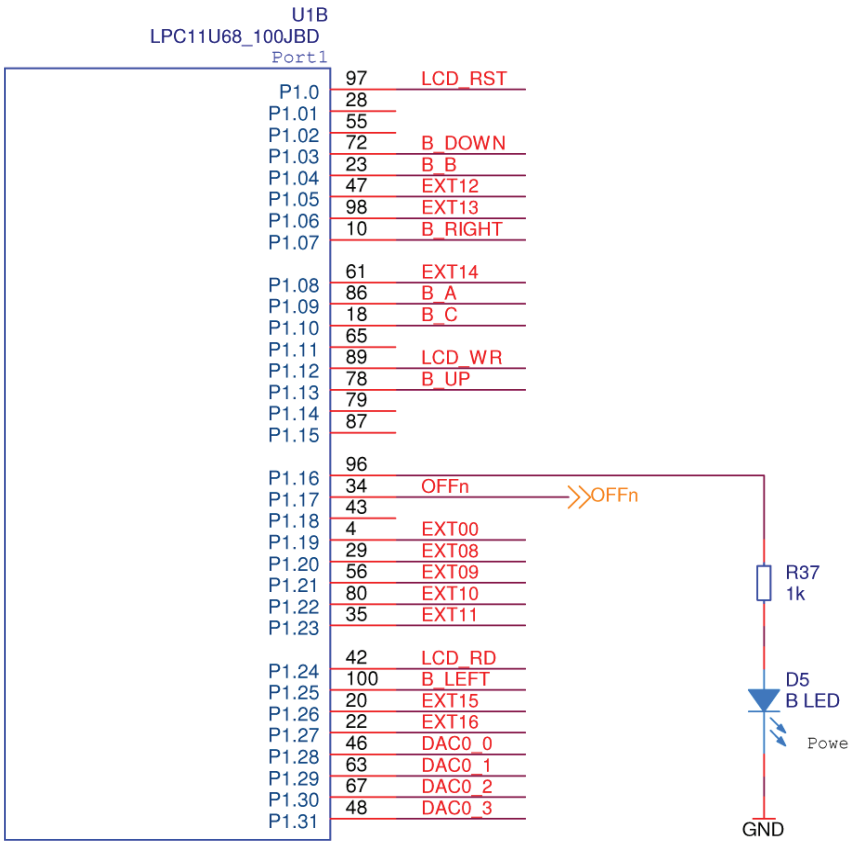
The Microcontroller (MCU in short) is the “brain” that controls all other systems. The MCU in Pokitto is NXP LPC11U64.

The MCU has “ports” and each port has “pins”. The pins are - literally - the legs of the chip.

Pokitto MCU has 3 ports: port 0, port 1 and port 2. Port 1 (shown here) is mostly responsible for reading buttons and working with the PEX expansion connector.

From the top you see P1.0 is connected to LCD_RST. That means Pin 0 of Port 1 is the pin that resets the LCD. Next one down the list is P1.03 that is connected to B_DOWN. Yep, Pin 3 of Port 1 reads button presses from button B. The number 72 means physical leg 72 of the chip.

See those pins marked EXT12, EXT13, EXT14? They are pins that connected to the PEX.

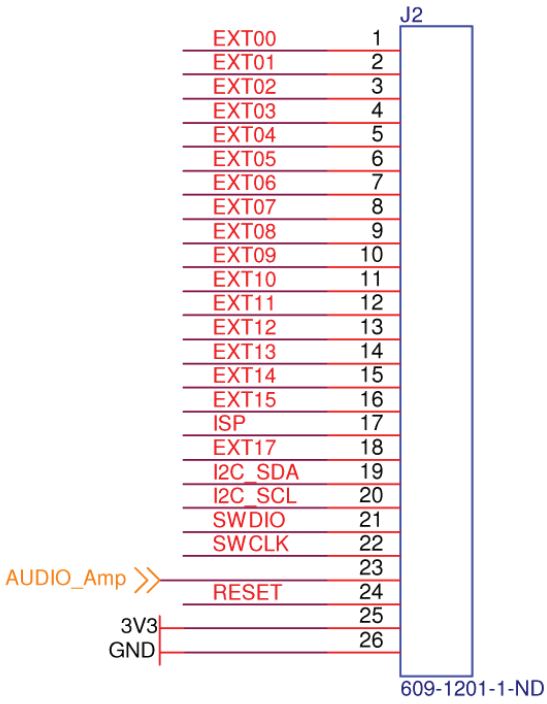


PEX

PEX is the Pokitto Expansion connector on top of Pokitto’s head. EXT00 to EXT15 are IO (input-output) pins that can be configured to blink LED lights, control motors etc.

ISP is a special pin that can be used to put Pokitto into a serial firmware programming mode. SWDIO and SWCLK are hardware debugging pins for hardcore programmers. Audio amp is the audio signal, RESET resets Pokitto. 3V3 supplies 3.3 volt current to external electronics, GND is ground. I2C pins are for I2C communication.

Read more info on PEX at pokitto.com/learn.



MCU port 2

Port 2 of the MCU is dedicated to outputting the graphics to the LCD.

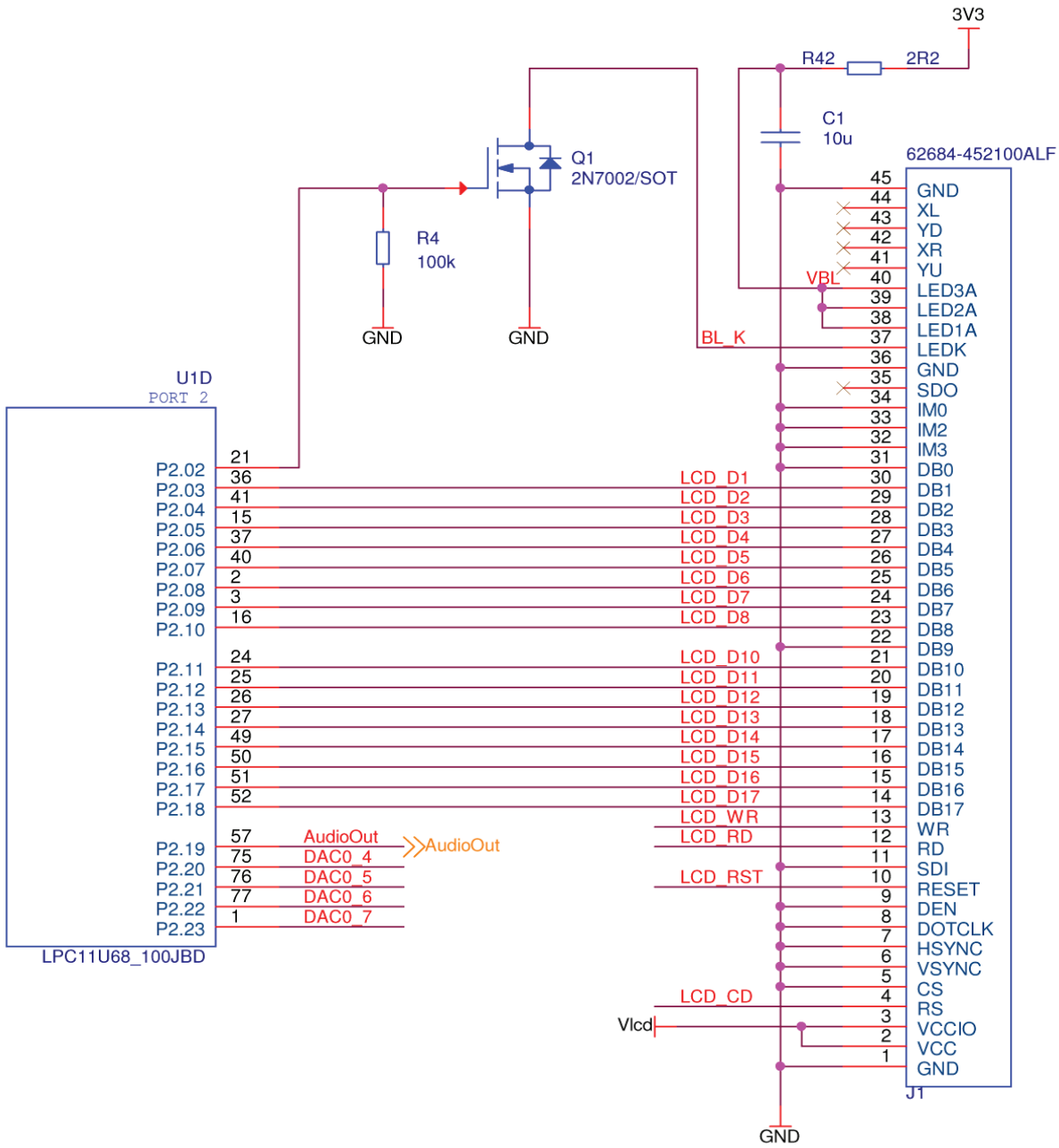
What you see here is one of Pokitto's greatest strengths: a 16-bit wide, very fast parallel bus that allows us to do graphical effects at higher speeds than other similar consoles.

The block on the left side is Port 2 of the Pokitto MCU. The block on the right is the 45 pin flat flex cable connector that is connected to the LCD. By dedicating Port 2 to mostly sending the graphics to the LCD we can push a lot of data at once.

LCD_WR, LCD_RD and LCD_CD tell the LCD controller to either wait for data or commands.

LCD_D1 to LCD_D17 carry the actual data, which can be colour values of pixels or commands to the LCD controller.

The LCD controller is a Sitronix ST7775R. For cost reasons, the LCD component (display and cable) is custom made for Pokitto. You can buy LCD's separately from Pokitto web shop.



MCU port 0

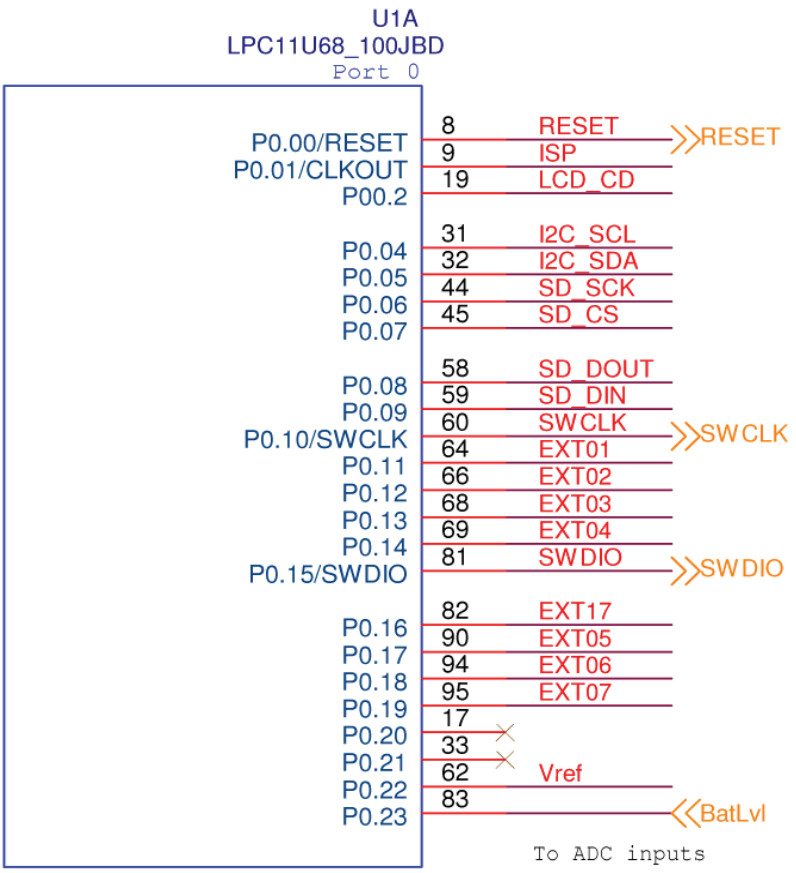
Finally, Port 0 of the MCU, which has some of the “master” functions needed to control the chip.

RESET resets the chip (same as cycling power on/off), ISP puts Pokitto into USB flashing mode for updating the software.

I2C_ pins are for the I2C communication protocol. I2C_SCL is the clock pin, that drives the communication in set intervals. I2C_SDA is where the data moves, one bit at a time.

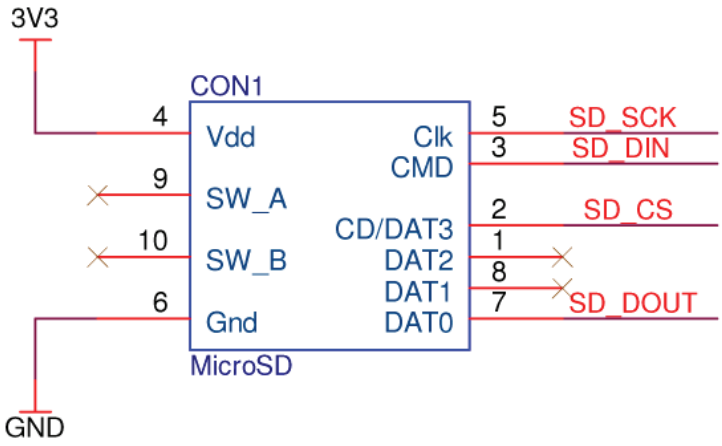
SD_ pins are connected to the MicroSD card socket.

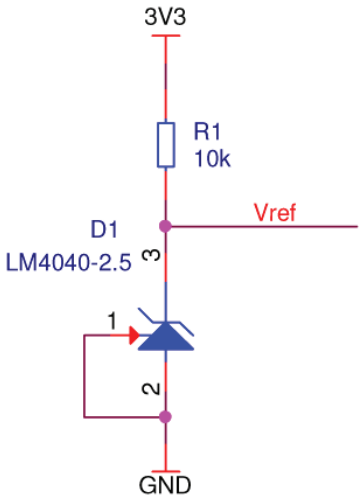
Vref and BattLvl are for battery charge monitoring.



MicroSD

Pokitto uses the SPI protocol to communicate with the MicroSD card. The MCU is the ‘SPI Master’ and the SD card is the ‘SPI Slave’. This Master controls how data is being exchanged. A SPI bus can have several Slave devices. SD_CS is pulled low (0V) to tell the device that it is the active Slave. Only one Slave is active at once.





Battery charge monitoring

Vref is connected to LM4040 shunt voltage reference, that outputs a constant 2.5V - as long as there is over 2.5V in the battery. By comparing the input value of BattLvl pin (P0.23) to the constant value of Vref pin (P0.22) the actual charge level of the battery can be calculated.

MCU Power & Clock, RTC and USB

Before we are 'done' with the MCU, we need to talk about the other connections that are needed to make the Pokitto microcontroller run.

3V3 (3.3 volts) from the battery, through the voltage regulator, is fed into the VDD supply pins. Ground (0V) is connected to the grounding pins (VSS). Together, these pins are what powers the chip. You will often see VDD / VSS marked in other chips as well. Now you know what they are!

XTAL pins are connected to Y1, a ceramic resonator. This resonator provides the chip with a highly accurate 12MHz 'MCU Clock' that is used to keep everything in pace. With chips, if something does not work, its usually either power or clock missing - they're important!

The Real Time Clock (RTC) is something completely different from the MCU clock. RTCXin and RTCxout are wired to Y2, a 32.768kHz oscillator.

The Real Time Clock is an optional feature that was added to Pokitto for the benefit of users. It allows Pokitto to keep the correct time and date - even when Pokitto is turned off! This allows us to create time-dependent applications like alarm clocks or virtual pets.

In order for the RTC to function when Pokitto is powered off, the RTC gets a tiny bit of power from the battery through the MCP1700 Low Quiescent Current LDO (long words that simply mean it wastes as little energy as possible when Pokitto is in 'power off' state).

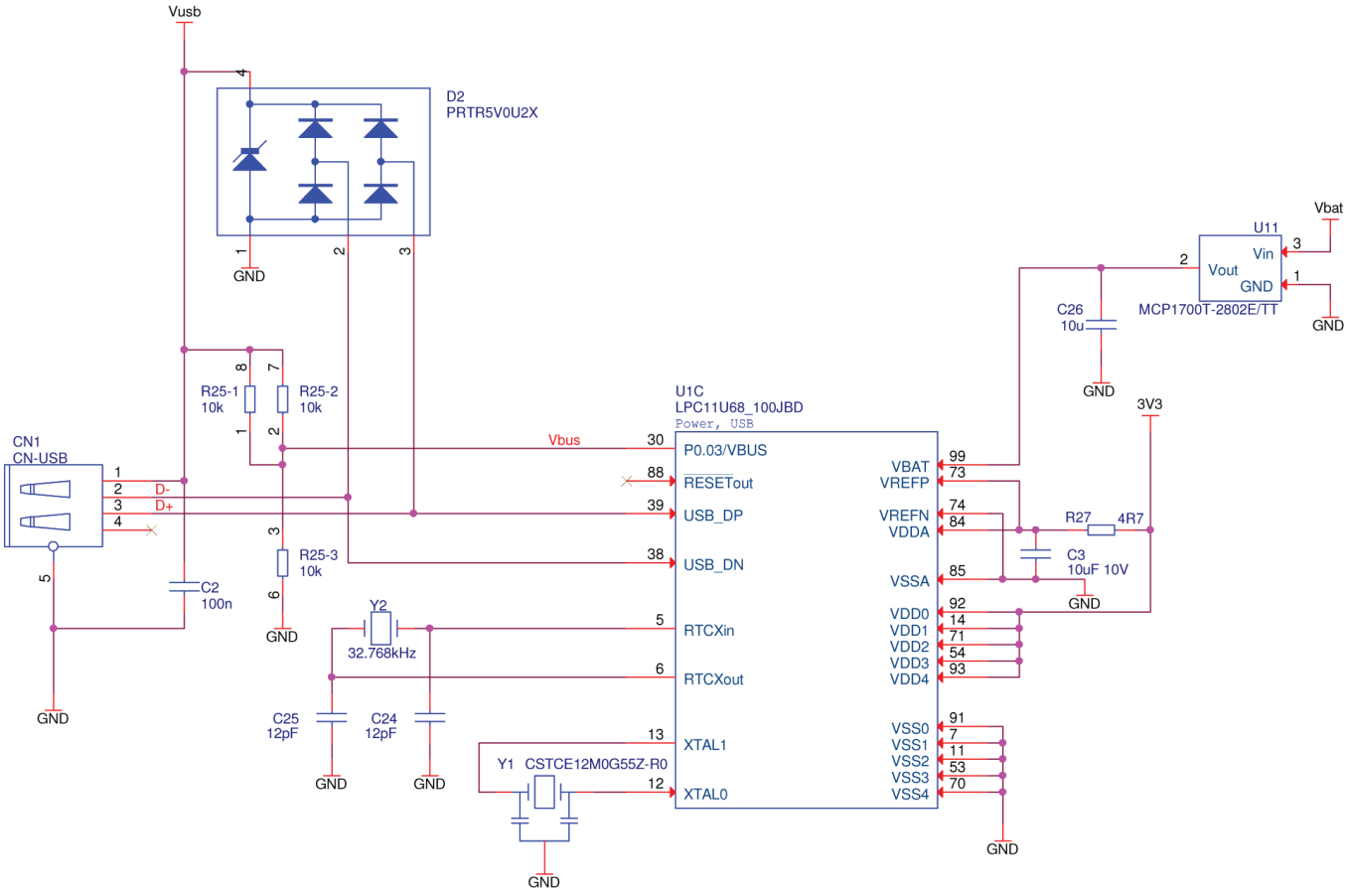
...continued in Pokitto Magazine #2

Whew. I didn't realize explaining the Pokitto hardware in detail would take up so many pages.

We still need to go through the audio circuit, and the main power circuit, and those are easily several pages.

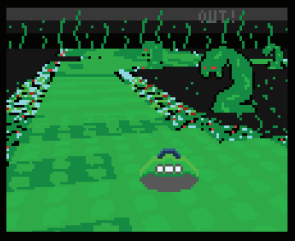
I hope you've enjoyed this tour into the internals of Pokitto

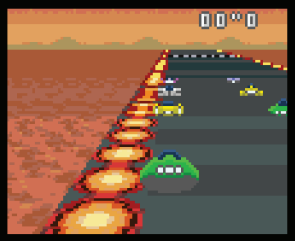
Tune in for more in the next issue of Pokitto Magazine!

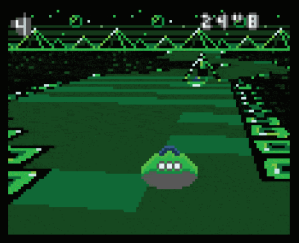


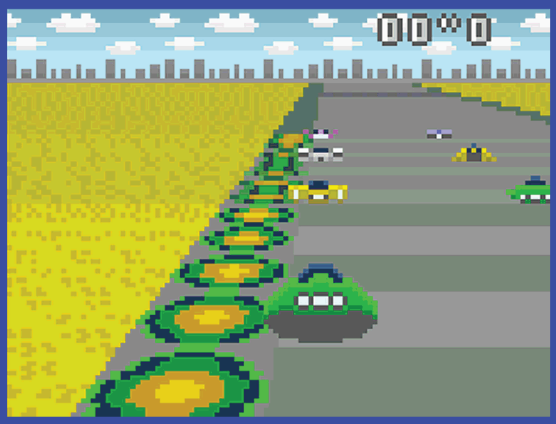
Can you Take The Challenge?

Pokitto Grand prix

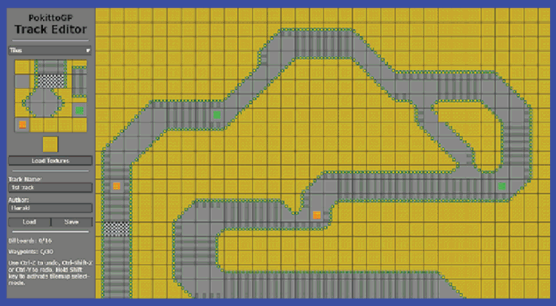






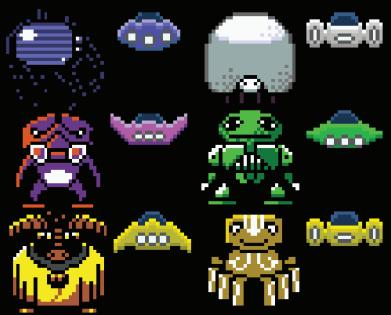


Graphics so real you will forget it's only a game!*



The Online Track Editor!

Intergalactic Opponents!



Game by @Hanski

* Supposing you still live in the 80's.
Note: The licenses for all the graphic assets are listed in the game itself.

Hardware Secrets

With Jonne

Discover your Pokitto's TURBO MODE!

As explained in the Hardware in Details article, the Pokitto MCU is clocked by a 12MHz external resonator. This 12MHz source frequency is multiplied by a device called PLL (Phase-Locked Loop) inside the MCU. The PLL generates multiples of the source frequency. For example, the default System Clock of LPC11u68 MCU is 48MHz and comes from 4x12MHz provided by the PLL

The setup of the PLL multiplier is explained in section 4.3.3. of the LPC11U68 user manual *UM10732* available from NXP. Below is a table of PLL MSEL and PSEL configuration combinations I have tested:

Input clock (MHz)	MSEL bits	M divider	Output SysClock (MHz)	PSEL bits	P divider	FCCO frequency (MHz)
12	011	4	48*	01	2	192
12	010	3	36	10	4	288
12	001	2	24	10	4	192
12	100	5	60	01	2	240
12	101	6	72	01	2	288
12	110	7	84	01	2	336
12	111	8	96	01	2	384

Overclocking?

The manufacturer NXP specifies 50MHz as the 'maximum' SysClock speed of the LPC11U68. In the *system_LPC11U6x.c* initialization file 'invalid' combinations of MSEL and PSEL are not allowed.

It was discovered however, that with a little modification of the system_LPC11U6x.c file, combinations leading to higher than 50 MHz can be used - *and it works on every Pokitto!*

Running at 72MHz

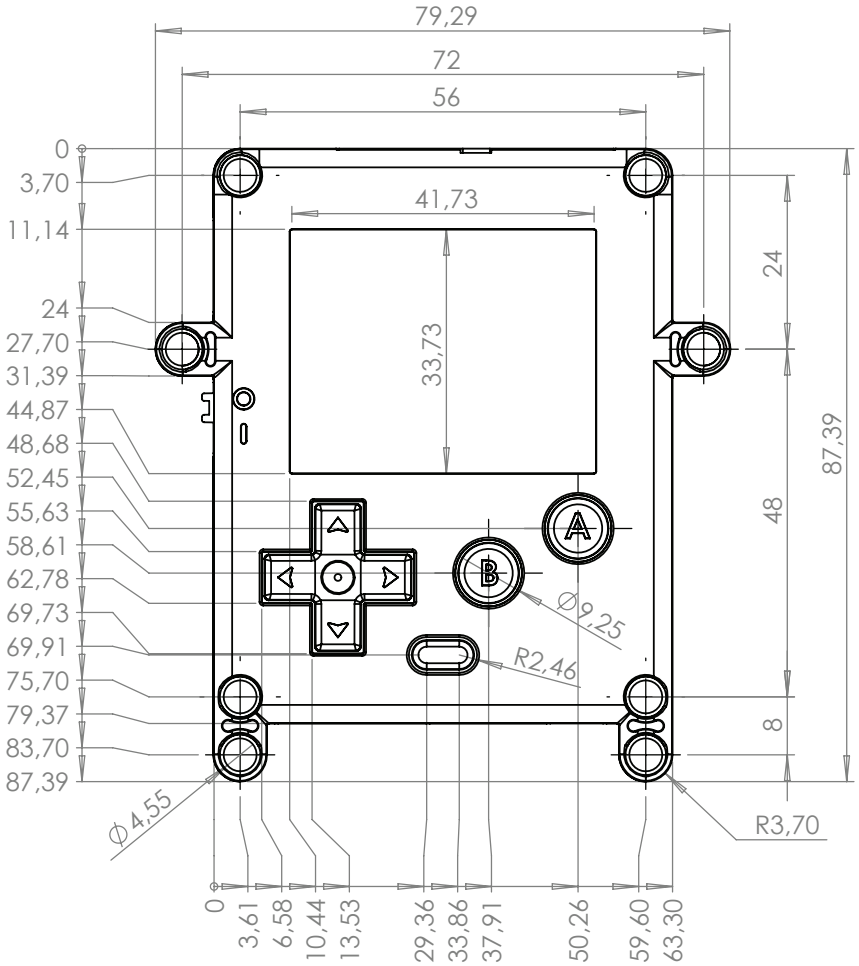
Overclocking Pokitto is simple. Just adding the following definition to your EmBitz / PlatformIO / FemtoIDE project will modify your clock speed to run at 72MHz

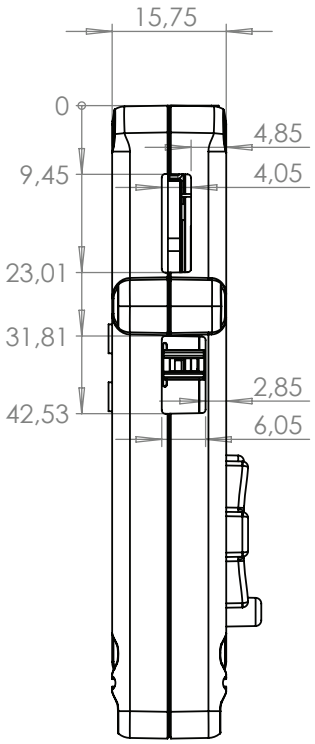
_OSCT=2

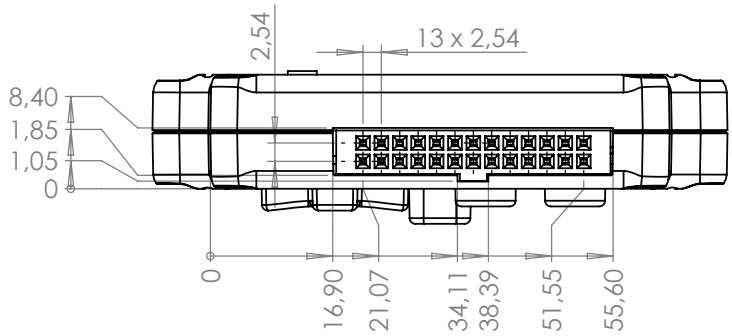
Our testing so far has shown that Pokitto runs stably at 72MHz! It will not break, the overclocking is completely safe, and works with 99% of code. Happy clocking!!

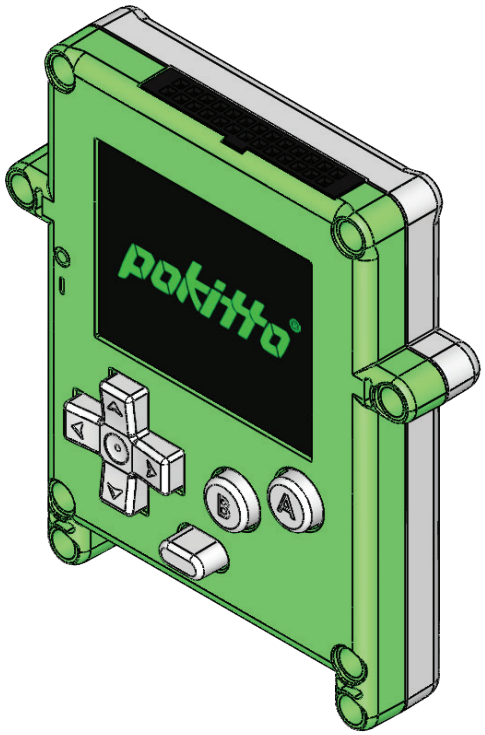
BONUS

Pokitto Dimensions









All dimensions are in millimeters.

Ordinate dimensions refer to aperture (openings) in the case.

Distances between pegs are according to Lego modular dimensions (divisible by 8mm)

pokitto

Game Jam Competition

JAVA OF THE DEAD

NOTHING WAKES THEM LIKE JAVA

Language: Java

Platform: Pokitto Hardware/Emulator

Theme: The Undead

Time: Starts Friday 19th July 2019
Ends Friday 13th Sept. 2019

For more information:
www.pokitto.com/javajam

Illustration made by @RaginROnin