Different Approaches to control GPIO Pins of the Raspberry Pi using Haskell

Eliane I. Schmidli
OST – Eastern Switzerland University of Applied Sciences
MSE Project 2
Supervisor: Prof. Dr. Farhad Mehta
Spring 2023

Abstract

The functional programming language Haskell allows the writing of elegant code and reduces the likelihood of runtime errors. This advantage can also be used when programming the General Purpose Input/Output (GPIO) pins on the Raspberry Pi. This paper presents three approaches in Haskell for controlling GPIO pins. The first method uses the GPIO sysfs interface, the second sends commands via sockets to the Python library gpiozero, and the third uses the Foreign Function Interface (FFI) to call functions from the C library bcm2835. The approaches described have different strengths and weaknesses, and it depends on the use case which approach is most suitable.

1 Introduction

The implementation of functional programming languages in developing embedded systems has numerous benefits. The usage of Haskell, for example, can lead to a reduction of runtime errors due to its strong type system. Additionally, Haskell allows for the creation of elegant, readable, and maintainable code. This paper demonstrates the application of Haskell's advantages in embedded system programming using the GPIO pins of the Raspberry Pi.

The GPIO pins are a powerful feature allowing the Raspberry Pi to interface with external electronic components such as LEDs, sensors, or buttons. Using software programming, the pins can be set as input or output and the state of the pin can be manipulated. In this way, electrical signals can be controlled and read to interact with the external components. For example, an LED can be turned on and off by setting the voltage level on the connected pin high (3.3V) or low (0V).

The following sections introduce three libraries and methods to enable the interaction between Haskell and GPIO pins. Figure 1 provides an overview of the three approaches. The method described in Section 2 uses the GPIO sysfs interface [Wal15b] and accesses the corresponding GPIO files and directories directly. In Section 3 commands are sent via sockets to the Python library gpiozero [NJ21], which provides an easy-to-use, high-level GPIO interface. Section 4 uses the FFI [Mar10] to call functions from the low-level C library bcm2835 [McC21]. To demonstrate the functionality of the libraries, an example is provided for each. An LED connected to GPIO 17 (pin 11) is made to blink five times.

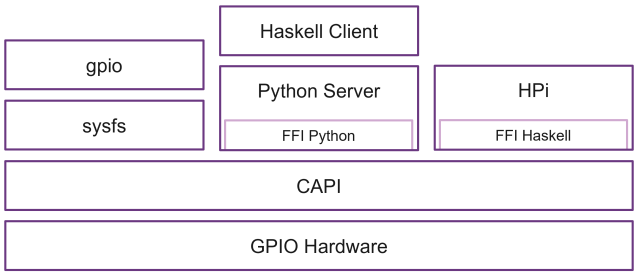

Figure 1. Overview of the three presented approaches to interact with the GPIO pins.

These three approaches on a single-board computer cover only a part of the possibilities for controlling hardware with Haskell. For instance, the utilization of Embedded Domain Specific Languages (EDSL) enables the compilation of Haskell into other languages that offer hardware interfaces on a microcontroller. A more detailed description of how this works can be found here [Sch23].

2 GPIO Sysfs Interface

The first library considered offers a straightforward approach to control the GPIO pins with Haskell, but it is no longer recommended due to more effective alternatives available. It accesses the GPIO pins using the GPIO sysfs interface. This is a pseudo filesystem under the path "/sys/class/gpio" in which the GPIOs are represented as files. So, it is possible to control the GPIOs using a shell. The library gpio [Ols17b] uses the IO monad to edit these files in Haskell. The following example illustrates the functions provided by the library in more detail.

The blink example uses GPIO 17. The library gpio represents pins with the following type:

```
1  data Pin = P2 | P3 | P4 | P17 | P27 | P22 | P10 |
2     P9 | P11 | P5 | P6 | P13 | P19 | P26 | P14 |
3     P15 | P18 | P23 | P24 | P25 | P8 | P7 | P12 |
4     P16 | P20 | P21
```

The structure for the blink example is as follows. First `led` is initialized and the function `void` discards the resulting value. Then the blink program lets the LED blink five times. At the end, the pin is closed.

```
led :: Pin
led = P17

main = do
   void (initWriterPin led)
   blink 5
   reattachToWriterPin led >>= closePin
```

To access the GPIO 17 directory, it must first be created. For this purpose, "17" is written into the file "/sys/class/gpio/export". Then the pin can be set as input or output by writing "in" or "out" into the newly created file "/sys/class/gpio/gpio17/direction".

In gpio, these tasks are accomplished in the function `initReaderPin` respectively `initWriterPin`.

```
initReaderPin :: (MonadCatch m, MonadIO m) => Pin
      -> m (ActivePin 'In)
initWriterPin :: (MonadCatch m, MonadIO m) => Pin
      -> m (ActivePin 'Out)
```

The type `ActivePin` is defined as follows:

```
data ActivePin (a :: Direction) where
   ReaderPin :: Pin -> ActivePin 'In
   WriterPin :: Pin -> ActivePin 'Out
```

To control the pin, the function `reattachToWriterPin` is used. It checks if the pin is initialized as output and throws an error if not. The function `reattachToReaderPin` works analogously with input pins.

```
reattachToWriterPin :: (MonadCatch m, MonadIO m)
    => Pin -> m (ActivePin 'Out)
reattachToReaderPin :: (MonadCatch m, MonadIO m)
    => Pin -> m (ActivePin 'In)
```

At the end of the program, the pin should be deactivated. By writing "17" into the file "/sys/class/gpio/unexport", the directories of GPIO 17 are removed. In gpio, this can be done with the function `closePin`.

```
closePin :: (MonadCatch m, MonadIO m) =>
    ActivePin a -> m ()
```

The following code section shows the blink program. It switches the LED on, waits for one second, and switches it off again. One second later the program calls itself recursively. The parameter `n` specifies how often the program should be called.

```
blink :: Int -> IO ()
blink 0 = return ()
blink n = do
   reattachToWriterPin led >>= writePin HI
   threadDelay 1000000
   reattachToWriterPin led >>= writePin LO
   threadDelay 1000000
   blink (n-1)
```

To switch the LED on and off the voltage level of the pin must be set to high or low. For this, the values "1" (high) or "0" (low) are written into the file "/sys/class/gpio/gpio17/value". The function `writePin` does the same in the gpio library. For the value zero `LO` is used and for one `HI`.

```
writePin :: (MonadCatch m, MonadIO m) => Value ->
      ActivePin 'Out -> m ()
```

To wait a certain time between switching on and off the function `threadDelay` from the concurrency extension for Haskell [oG23a] can be used. The time is specified in microseconds.

The gpio library provides a simple way to control the GPIO pins. But the sysfs interface is no longer being developed and has been replaced with the GPIO character device interface [Wal15a]. Furthermore, it is recommended to use the existing standard kernel drivers [Wal21] for common GPIO tasks instead of accessing the interfaces directly. For example, there is an LEDs driver for GPIOs.

3 Socket Connection to Python

A better way to control the GPIO pins is the Python library gpiozero [NJ21]. It allows controlling the GPIO pins with simple commands and without low-level programming. For example, it offers pre-built objects for commonly used components. These objects include LEDs, buttons, buzzers, and various other hardware components that can be connected to GPIO pins. So, developers can easily control and interact with these hardware components without having to handle low-level GPIO configurations themselves.

An easy way to connect Haskell to Python code is to establish a socket connection. This way a program sequence can be defined in Haskell sending specific commands to Python. The Python code can then control the corresponding GPIOs and return results if requested.

3.1 Haskell Client

The Haskell client should execute the blink program and send the commands to turn the LED on or off to the Python server. The client socket can be created using the Network library [YB23a, YB23b].

The `runTCPClient` function creates a client socket with the corresponding host and port. The last argument is the function that will use the socket. In this code, the `blink'` function will send commands over the socket.

```haskell
main :: IO ()
main = runTCPClient "127.0.0.1" "10000" $ \s ->
    do
   blink' 5 s

runTCPClient :: HostName -> ServiceName -> (
    Socket -> IO a) -> IO a
runTCPClient host port client = withSocketsDo $
    do
   addr <- resolve
   bracket (open addr) close client
   where
     resolve = do
       let hints = defaultHints {addrSocketType
     = Stream}
       head <$> getAddrInfo (Just hints) (Just
     host) (Just port)
     open addr = bracketOnError (openSocket addr)
      close $ \sock -> do
       connect sock $ addrAddress addr
       return sock
```

The use of the `withSocketsDo` function is recommended for compatibility with older versions of the network library on Windows. This function initializes the networking subsystem.

The `resolve` function resolves the host address and port number. The resulting value of type `IO AddrInfo` can then be used to open the corresponding socket. To use a stream socket, the property `addrSocketType` of the `defaultHint` is overwritten.

The `open` function opens the socket with the corresponding address information and tries to connect to the server socket. This is done using the `bracketOnError` function from the Exception module [oG23b], which has the following type.

```haskell
bracketOnError
  -- run first ("acquire resource")
  :: IO a
  -- run last ("release resource"), only if an
     exception was raised
  -> (a -> IO b)
  --  run in-between
  -> (a -> IO c)
  -> IO c
```

If the connection to the server socket cannot be established, `bracketOnError` calls `close` to close the socket and throws an error. If the connection is successful, the socket is returned.

In the function `runTCPClient` the function `open` is called by `bracket`, which is also provided by the Exception module [oG23b]. After setting up the network, the resulting socket is passed to the `client` function (`Socket -> IO a`). In this example, it executes the blink program and sends the commands to the server socket. When the program is finished, `bracket` calls `close` to close the socket. Unlike `bracketOnError`, `bracket` always calls the last operation to release the resource.

```haskell
bracket
  -- run first ("acquire resource")
  :: IO a
  --run last ("release resource")
  -> (a -> IO b)
  -- run in-between
  -> (a -> IO c)
  -> IO c
```

The blink program works similarly to the previous section 2. Instead of executing the commands directly, they are sent to the Python service with `sendAll` of type `Socket -> ByteString -> IO()`. The string "`\n`" is chosen as a separator so that the messages can be divided.

```haskell
blink' :: Int -> Socket -> IO ()
blink' 0 _ = return ()
blink' n s = do
  sendAll s "led_on\n"
   threadDelay 1000000
   sendAll s "led_off\n"
   threadDelay 1000000
   blink' (n-1) s
```

3.2 Python Server

In this example, the Python service should act as a server and receive commands from the Haskell service. The server socket in the following program is created using the Python library socket [Fou23, McM23]. First, the server socket is initialized with the specified port number and host address. Once the connection is established, requests from the client can be processed. Here the function `handle` receives and executes the commands from the client socket.

```python
PORT = 10000
HOST = '127.0.0.1'
serversocket = initialize_server(HOST, PORT)
clientsocket = accept_client(serversocket)
handle(clientsocket)
```

The method `initialize_server` initializes a server socket in Python. The parameters of `socket` specify that it should be an INET streaming socket. Then the socket is bound to a host and port. The parameter in `listen` is used to specify how many pending connections can be queued up before refusing outside connections.

```python
def initialize_server(host, port):
    serversocket = socket.socket(socket.AF_INET,
     socket.SOCK_STREAM)
    serversocket.bind((host, port))
    serversocket.listen(1)
    return serversocket
```

To accept a connection from outside, the function `accept` is called.

```python
def accept_client(server):
    (clientsocket, address) = server.accept()
    return clientsocket
```

The function `handle` receives and processes the client's messages in an infinite loop. In this simple example, the

functions with the same names are called for the messages “led_on” and “led_off”.

```
def handle(clientsocket):
    while 1:
        messages = receive_data(clientsocket)
        if (not messages): return
     for m in messages:
        if m == "led_on": led_on()
        if m == "led_off": led_off()
```

To receive the messages from the client, the `recv` function is called. The constant value `MAX_LENGTH` defines the maximum amount of data to be received at once. The resulting bytes object is decoded to get the data as a string. The client and server can agree on a separator that will divide the messages.

```
MAX_LENGTH = 4096

def receive_data(clientsocket):
    buf = clientsocket.recv(MAX_LENGTH)
    if buf == '': return []
    data = buf.decode('utf-8')
    seperator = "\n"
    return data.split(seperator)
```

The library gpiozero provides a simple interface to address the LEDs. With the command `LED` the desired pin can be initialized as LED. Then the LED can be toggled with `on` and `off`.

```
LED17 = LED(17)

def led_on():
    LED17.on()

def led_off():
    LED17.off()
```

The advantage of the variant with the socket connection is that the two services are decoupled. So, it is easy to replace the Python service with something else without adapting the Haskell code. For example, the same Haskell code could be linked to another hardware interface. However, building a network is an additional overhead and error-prone. The network protocols and the serialization and deserialization of the data can also introduce latency and impact performance.

4 FFI to C

The library HPi [Hil20a] uses a more efficient solution than building a network. It directly calls the functions of the C library bcm2835 [McC21]. This library provides access to the GPIO pins on the Broadcom BCM 2835 chip used in earlier Raspberry Pis and its successors on the newer versions. To call the functions directly, the HPi uses FFI. FFI enables interaction with external libraries written in other languages like C or C++.

For example, HPi imports the function `bcm2835_gpio_write` to set the voltage level. It uses the function `ccall` for this. But this function has become deprecated and should be replaced by `capi` [Tea20].

```
foreign import ccall unsafe
    "bcm2835.h"
    "bcm2835_gpio_write"
    c_writePin :: CUChar -> CUChar -> IO ()
```

The call specifies where the function is located (here “bcm2835.h”), which function should be imported (`bcm2835_gpio_write`), and how the new type signature should look like (`c_writePin :: CUChar -> CUChar -> IO ()`). The `CUChar` type is a wrapper around the C `unsigned char` type. The function `c_writePin` can now be used in Haskell code to call the C function.

The use of the HPi library is also explained with the blink example. The function `withGPIO` of type `IO a -> IO` initializes the use of the GPIO pins with the bcm2835 library.

```
main = withGPIO $ do
    setPinFunction Pin11 Output
    blink'' 5
```

To set the pin 11 (GPIO 17) as input or output, the function `setPinFunction` of type `Pin -> PinMode -> IO ()` is called with the pin and the corresponding pin mode. The function calls the bcm2835 function `bcm2835_gpio_fsel`. The modes `Input` and `Output` correspond to the numbers zero and one.

The blink example works in the same way as the previous examples. It uses `writePin` which calls the `c_writePin` function described earlier. The function `writePin` is of type `Pin -> LogicLevel -> IO ()`. The `LogicLevel` type is a Boolean where the value `True` represents a high voltage level or one, while `False` represents a low voltage level or zero.

```
blink'' :: Int -> IO()
blink'' 0 = return ()
blink'' n = do
    threadDelay 1000000
    writePin Pin11 True
    threadDelay 1000000
    writePin Pin11 False
    blink'' (n-1)
```

Utilizing FFI eliminates the network overhead, resulting in improved performance. Also, the bcm2835 library, operating at a lower level, offers better performance compared to the higher-level gpiozero library. Nevertheless, employing the bcm2835 requires a more low-level understanding and entails a higher degree of complexity in its usage.

5 Conclusions

The libraries described before have their advantages and disadvantages. Using the sysfs interface is very simple but outdated and no longer recommended. The GitHub projects for the gpio and HPi libraries both have few contributors and have not been updated in over three years [Hil20b, Ols17a]. When using these libraries, small updates could be necessary to bring them up to date

with the latest standards and requirements. The gpiozero and network libraries are much better supported and are updated more regularly.

In all libraries, it is possible to describe the behavior of the GPIO pins in elegant Haskell code. The gpiozero library offers a high-level interface and is better suited for beginners. The communication over sockets makes the application more modular. For example, the Python server can be replaced with another hardware control service.

To achieve good performance and fine-granular access to the GPIO pins, the bcm2835 library stands out as the most suitable choice. It is therefore recommended to use the FFI approach even if the HPi library needs to be updated.

References

[Fou23] Python Software Foundation. Lib: socket.py. https://docs.python.org/3/library/socket.html, 2023. Accessed: 2023-06-14.

[Hil20a] Wander Hillen. Hackage package: HPi. https://hackage.haskell.org/package/HPi, 2020. Accessed: 2023-06-14.

[Hil20b] Wander Hillen. Hpi. https://github.com/WJWH/HPi, 2020. Accessed: 2023-06-14.

[Mar10] Simon Marlow. Foreign function interface. https://www.haskell.org/onlinereport/haskell2010/haskellch8.html#x15-1490008, 2010. Accessed: 2023-06-14.

[McC21] Mike McCauley. bcm2835. https://www.airspayce.com/mikem/bcm2835/, 2021. Accessed: 2023-06-14.

[McM23] Gordon McMillan. Socket programming howto. https://docs.python.org/3/howto/sockets.html, 2023. Accessed: 2023-06-14.

[NJ21] Ben Nuttall and Dave Jones. gpiozero. https://gpiozero.readthedocs.io/en/stable/, 2021. Accessed: 2023-06-14.

[oG23a] The University of Glasgow. Hackage package: Control.concurrent. https://hackage.haskell.org/package/base-4.18.0.0/docs/Control-Concurrent.html, 2023. Accessed: 2023-06-14.

[oG23b] The University of Glasgow. Hackage package: Control.exception. https://hackage.haskell.org/package/base-4.18.0.0/docs/Control-Exception.html, 2023. Accessed: 2023-06-14.

[Ols17a] Tyler Olson. Github repository: gpio. https://github.com/TGOlson/gpio/tree/master, 2017. Accessed: 2023-06-14.

[Ols17b] Tyler Olson. Hackage package: gpio. https://hackage.haskell.org/package/gpio, 2017. Accessed: 2023-06-14.

[Sch23] Eliane I. Schmidli. Embedded Programming with Embedded Domain-Specific Languages (EDSLs) in Haskell. Rapperswil, July 2023. OST – Eastern Switzerland University of Applied Sciences.

[Tea20] GHC Team. The capi calling convention. https://downloads.haskell.org/ghc/9.0.1/docs/html/users_guide/exts/ffi.html?highlight=capiffi#extension-CApiFFI, 2020. Accessed: 2023-06-14.

[Wal15a] Linus Walleij. git commit: "gpio: Abi: mark the sysfs abi as obsolete". https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/Documentation/ABI/obsolete/sysfs-gpio?h=v5.16&id=fe95046e960b4b76e73dc1486955d93f47276134, 2015. Accessed: 2023-06-14.

[Wal15b] Linus Walleij. Gpio sysfs interface for userspace. https://www.kernel.org/doc/Documentation/gpio/sysfs.txt, 2015. Accessed: 2023-06-14.

[Wal21] Linus Walleij. Subsystem drivers using gpio. https://www.kernel.org/doc/Documentation/driver-api/gpio/drivers-on-gpio.rst, 2021. Accessed: 2023-06-14.

[YB23a] Kazu Yamamoto and Evan Borden. Hackage package: network. https://hackage.haskell.org/package/network-3.1.4.0, 2023. Accessed: 2023-06-14.

[YB23b] Kazu Yamamoto and Evan Borden. network documentation: Network.socket. https://hackage.haskell.org/package/network-3.1.4.0/docs/Network-Socket.html, 2023. Accessed: 2023-06-14.