Haskino: A Remote Monad for Programming the Arduino

Mark Grebe and Andy Gill

The University of Kansas

Haskino

- Haskino provides a mechanism for programming the Arduino microcontroller in Haskell, instead of C.
- We provide two complementary methods:
  - A method which uses an Arduino tethered to a host computer.
  - A method which out sources entire groups of commands and control idioms, and allows the Arduino to run stand-alone.
- But first some background…

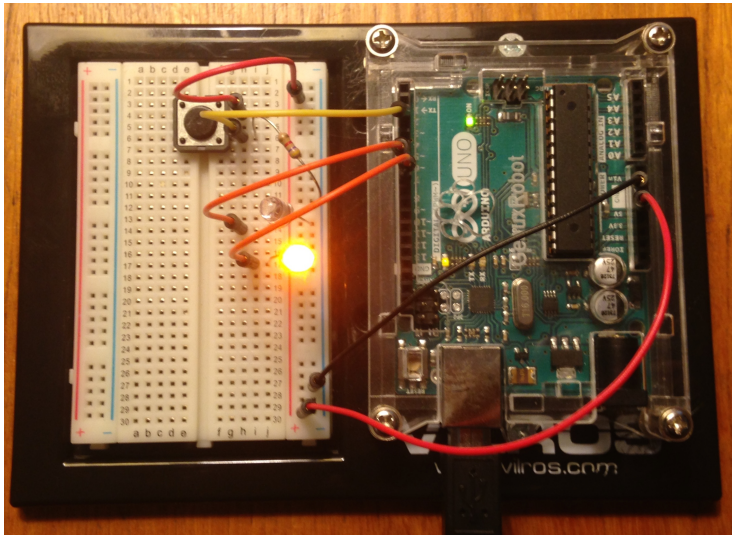
Monads!

- Haskell uses monads as the principal way of expressing side-effecting computation
  - The **`IO`** monad is the way of talking to the outside world
  - The **`Maybe`** monad is a way of expressing exceptions
  - etc, etc.
- Monads are composable effects

```
  return :: a -> IO a
   (>>=) :: IO a -> (a -> IO b) -> IO b
readFile :: FilePath -> IO String
writeFile :: FilePath -> String -> IO ()
```

Controlling an Arduino

```
setPinMode   :: Word8 -> PinMode -> IO ()
digitalWrite :: Word8 -> Bool -> IO ()
…
```



```
do setPinMode 2 OUTPUT
   digitalWrite 2 True
   …
```

I/O operations are often added directed as monadic **`IO`** functions

- This API only supports a single **`Arduino`**
- The ability to control the **`Arduino`** is given to everyone
- No statically enforced initialization
- The API does not reflect that the **`Arduino`** is a remote peripheral

The Remote Monad Design Pattern

```
send          :: ArduinoConnection ->
                 Arduino a -> IO a
setPinMode    :: Word8 -> PinMode -> Arduino ()
digitalWrite :: Word8 -> Bool -> Arduino ()
…
```

If you want to change the pin mode:

```
send conn (setPinMode 2 OUTPUT)
```

If you want to write an output to the pin:

```
send conn (digitalWrite 2 True)
```

In this remote monad, I/O operations are added as monadic **`Arduino`** functions

- This API supports multiple devices
- The ability to control a specific **`Arduino`** is now first class
- **send**, or the act of creating the **`ArduinoConnection`**, can enforce initialization
- The API reflects that the **`Arduino`** is a remote peripheral

The Key Remote Monad Idea

If you want to change the pin mode:

```
send conn (setPinMode 2 OUTPUT)
```

If you want to write an output to the pin:

```
send conn (digitalWrite 2 True)
```

If you want to change the pin mode **and** write output to the pin

```
send conn (setPinMode 2 OUTPUT >> digitalWrite 2 True)
```

Can we bundle **`setPinMode`** and **`digitalWrite`** into a single transaction?

Returning Remote Results

```
send         :: ArduinoConnection -> Arduino a -> IO a
setPinMode   :: Word8 -> PinMode -> Arduino ()
digitalWrite :: Word8 -> Arduino ()
digitalRead  :: Word8 -> Arduino Bool
```

Using result inside **`Arduino`**

```
send conn $ do
      input <- digitalRead 3
      digitalWrite 2 (not input)
```

Returning remote result

```
res <- send conn (digitalRead 3)
```

- The monadic commands inside **send** are executed in a remote location
- The results of those executions need to be made available for use locally

Remote Monad Laws

Use the monad-transformer lift laws, also known as the monad homomorphism laws.

```
send_c :: forall a . Remote a -> Local a

send_c (return a) = return a
send_c (m >>= k)  = send_c m >>= send_c . k
```

- **`send`** is a natural transformation from a remote effect to a local effect
- The laws give us the freedom to choose bundling strategy

The Command Design Pattern

*A remote **command** is a request to perform an action for remote effect, where there is no result value*

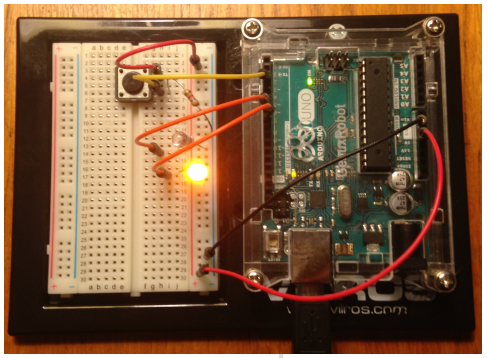
```
data Command =
  SetPinMode Word8 PinMode
| DigitalWrite Word8 Bool
    deriving Show

digitalWrite :: Word8 -> Bool -> Arduino ()
digitalWrite p v = Command $ DigitalWrite p v
```

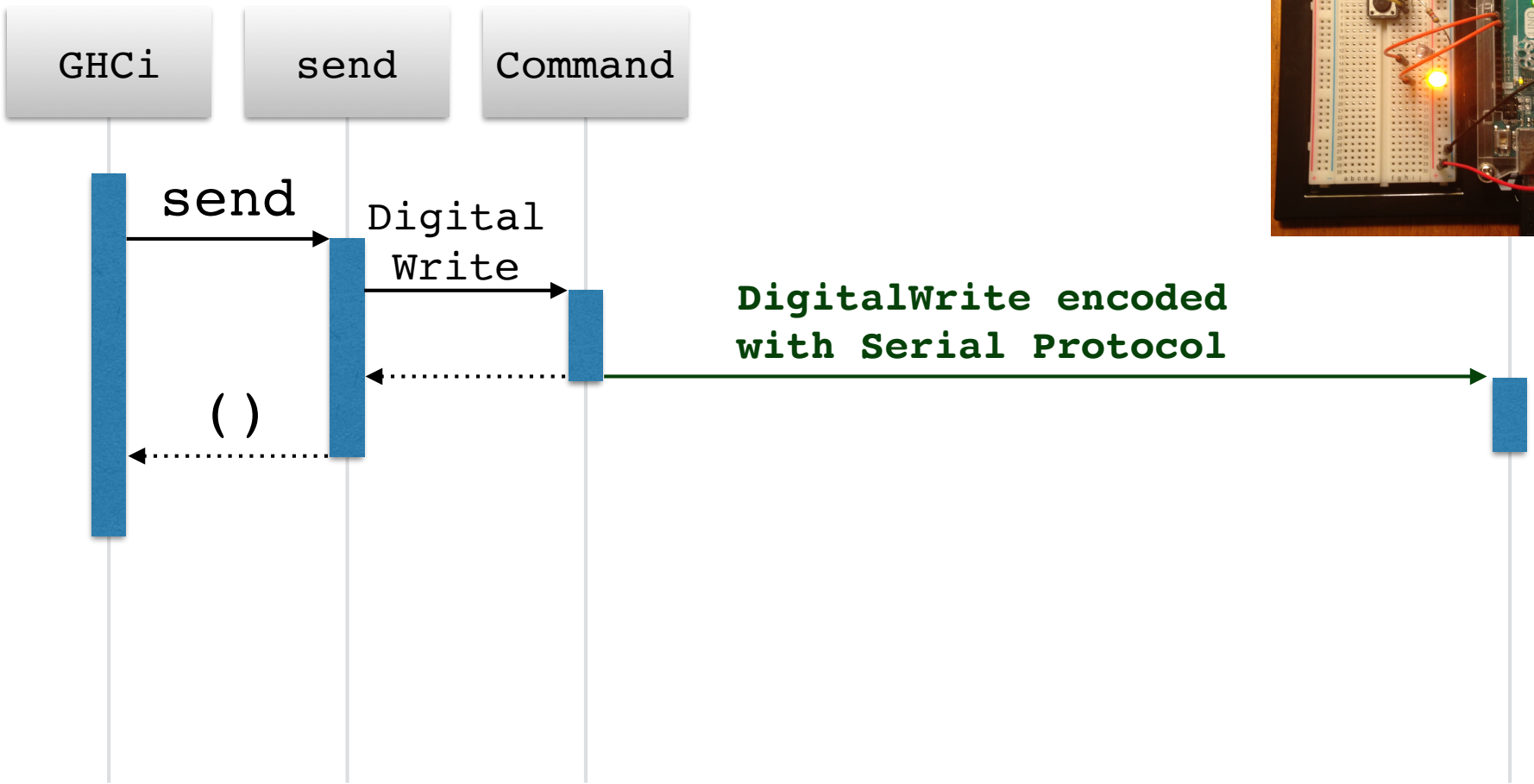
```
send :: ArduinoConnection -> Command -> IO ()
send conn cmd = do
  packCmd <- packageCommand cmd
  sendToArduino conn packCmd
```

```
GHCi> send conn (digitalWrite 2 True)
Arduino: LED on pin 2 turns on
```

The Command Design Pattern

```
GHCi> send conn (digitalWrite 2 True)
Arduino: LED on pin 2 turns on
```





Remote Procedures

*A remote **procedure** is a request to perform an action for its remote effects, where there is a result value or temporal consequence*

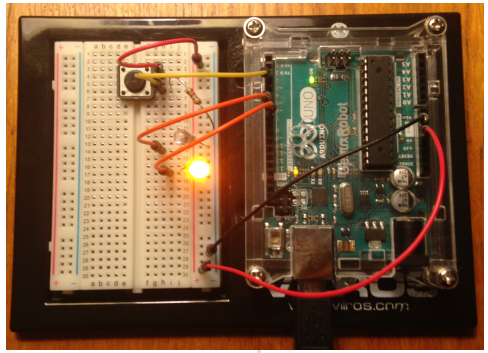
```
data Procedure :: * -> * where
  DigitalRead :: Word8 -> Procedure Bool
  DelayMillis :: Word32 -> Procedure ()
```

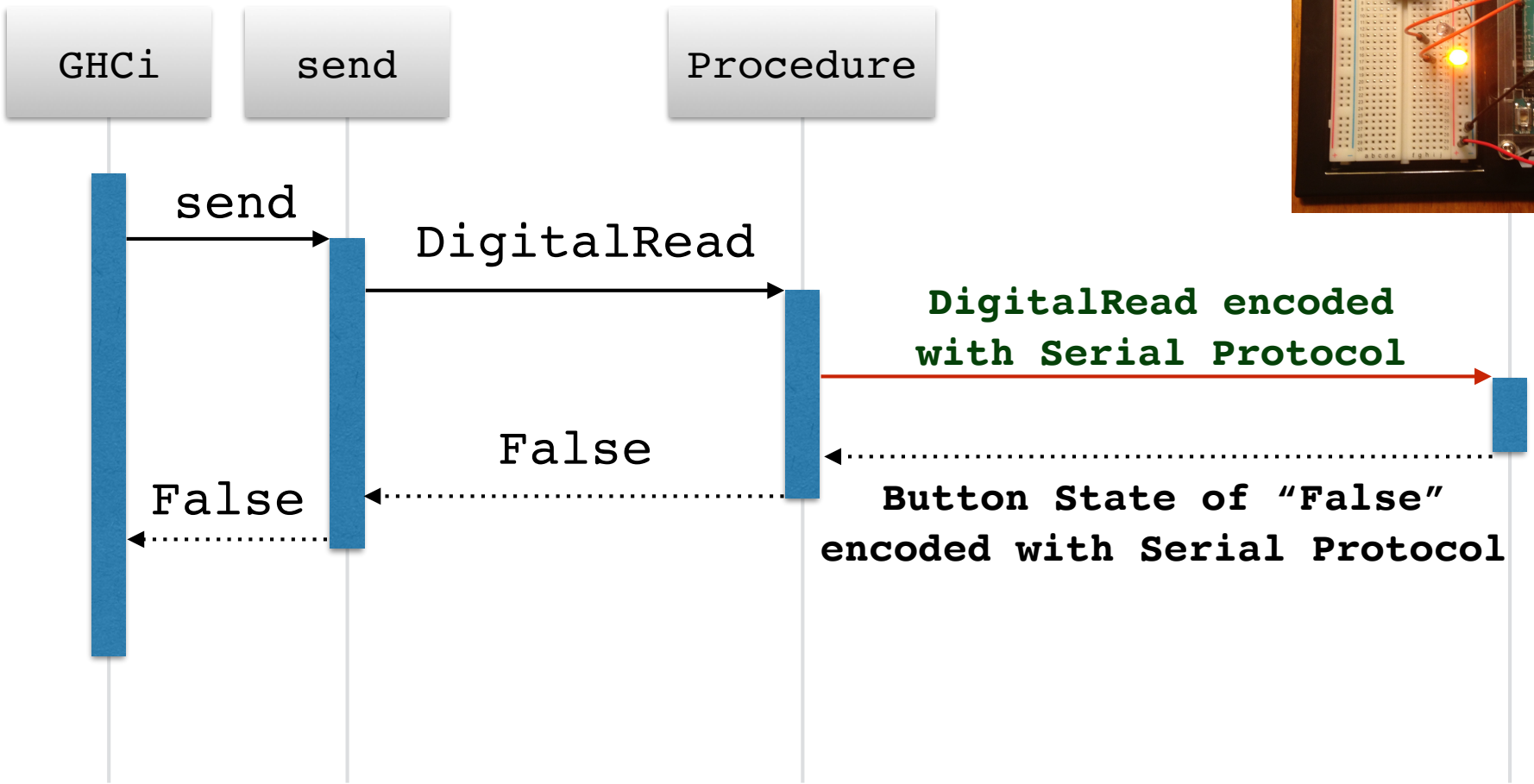
```
send :: ArduinoConnection -> Procedure a -> IO a
send conn p = do
    packP <- packageProcedure p
    sendToArduino conn packP
    rsp <- waitResponse conn p
    return rap
```

```
GHCi> send conn DigitalRead 3
```

Remote Procedures

```
GHCi> send conn DigitalRead 3
```

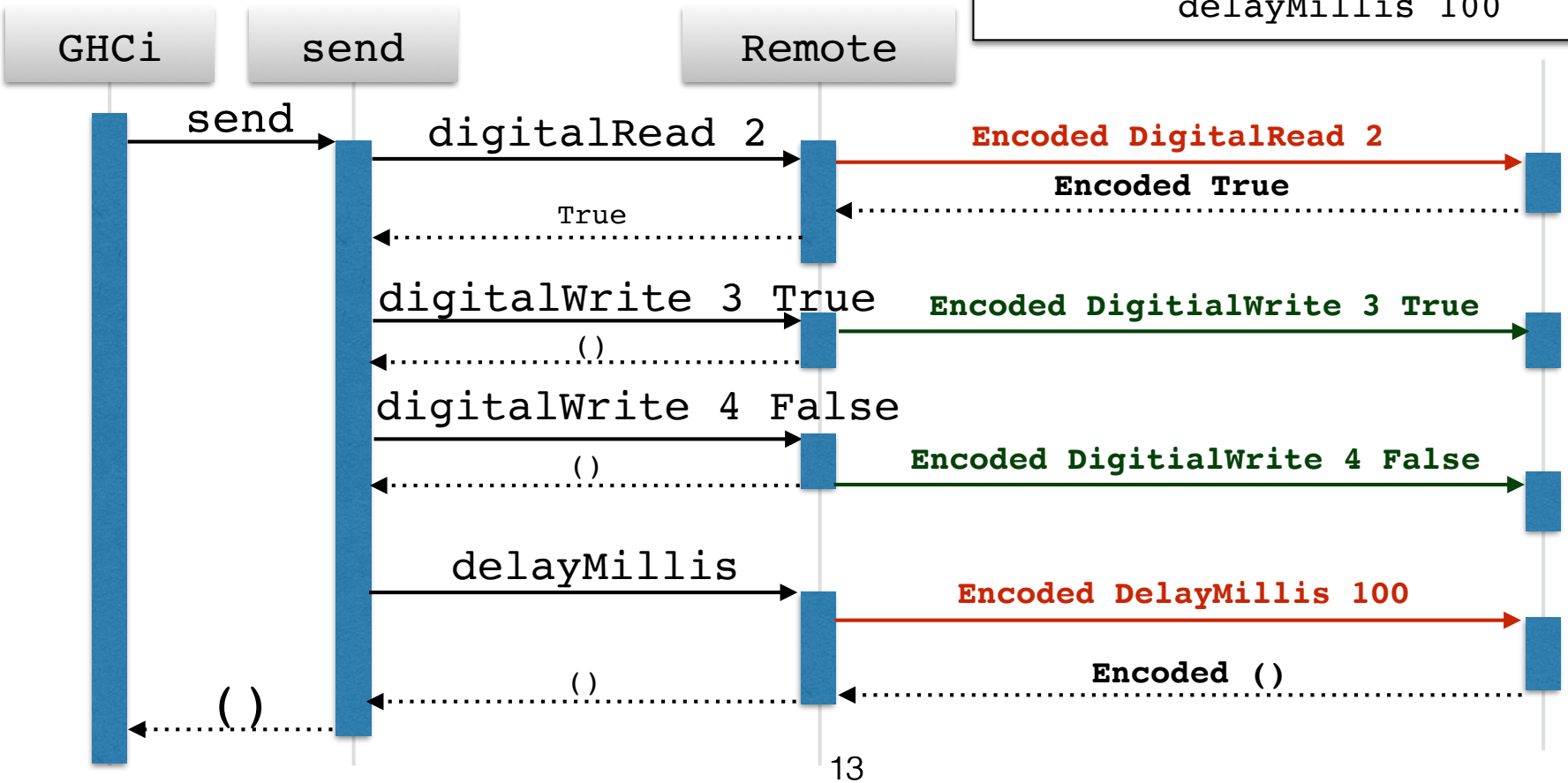




The Weak Remote Monad

*A **weak remote monad** is a remote monad that sends each of its remote calls individually to a remote interpreter*

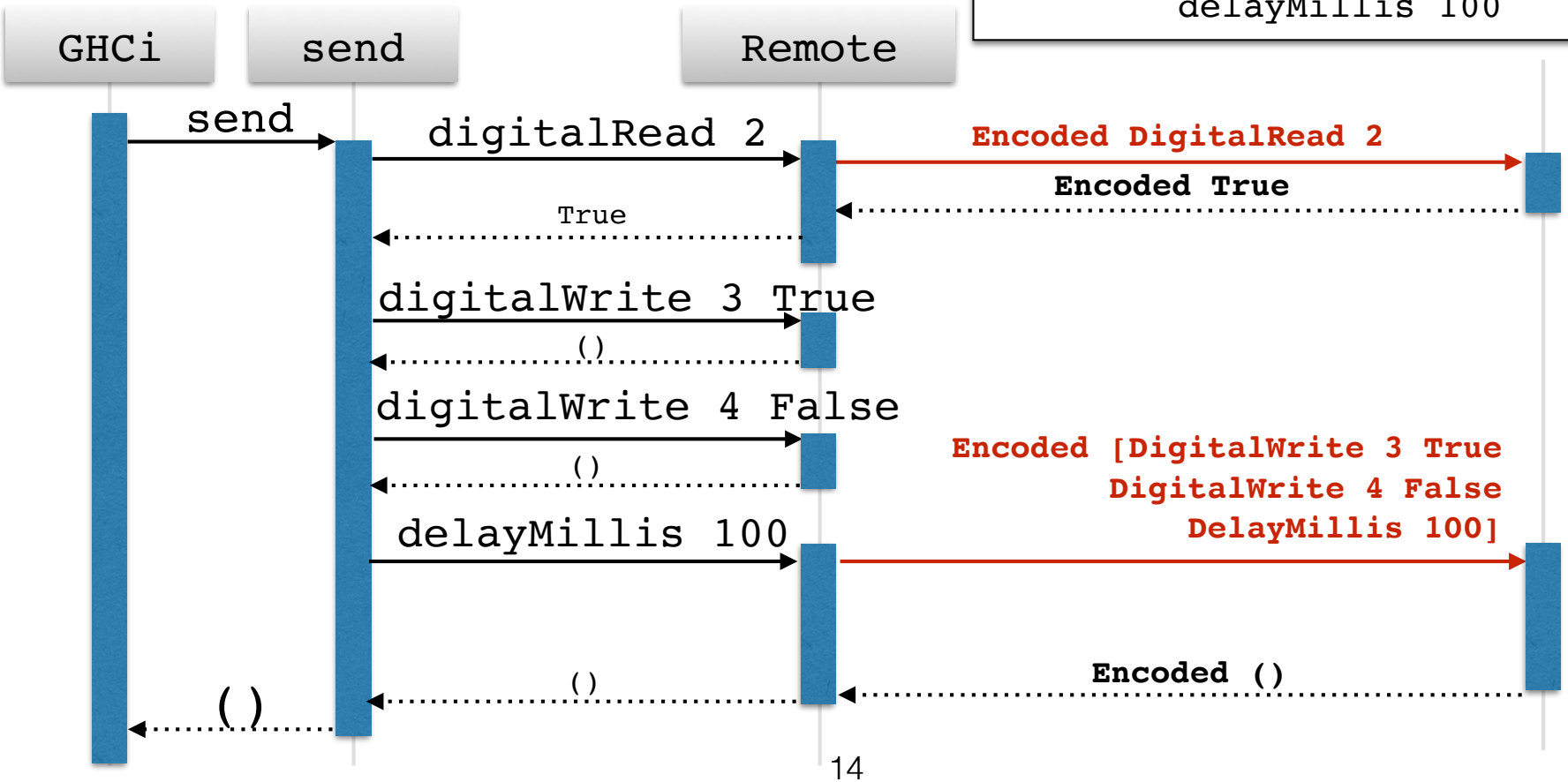
```
send conn $ do
          x <- digitalRead button
          digitalWrite led1 x
          digitalWrite led2 (not x)
          delayMillis 100
```



The Strong Remote Monad

*A **strong remote monad** is a remote monad that bundles all of its remote calls into packets of commands, punctuated by procedures, for remote execution*

```
send conn $ do
          x <- digitalRead button
          digitalWrite led1 x
          digitalWrite led2 (not x)
          delayMillis 100
```



Weak, Strong, and EDSL Versions

- Levent Erkök's hArduino package is an example of a weak remote monad, this was our starting point.
- The first version of Haskino extended hArduino by applying the strong remote monad concepts, to increase communications efficiency through bundling.
- To develop our second method of allowing stand-alone Arduino execution, required a deep embedding.

EDSL Modifications

- Add Expressions to the language.
- Add remote storage of computation results.
- Add Conditionals to the language
- Replace the Arduino firmware (which was called Firmata in hArduino and the initial version of Haskino).
  - Allows the firmware to handle interaction with an EDSL and optimizes communication.

Adding Expressions

The tethered Strong Remote Haskino uses commands and procedures such as:

```
digitalWrite :: Word8 -> Bool -> Arduino ()
analogRead :: Word8 -> Arduino Word16
```

To move to the deeply embedded version, we instead use:

```
digitalWriteE :: Expr Word8 -> Expr Bool ->
                 Arduino ()
analogReadE :: Expr Word8 ->
                 Arduino (Expr Word16)
```

Strong commands may be written in terms of Deep ones, i.e.:

```
digitalWrite p b =
   digitalWriteE (lit p) (lit b)
```

Expression Types

The Haskino EDSL provides **`Expr a`** parameterized over the following types:

- **`Word8`**
- **`Word16`**
- **`Word32`**
- **`Int8`**
- **`Int16`**
- **`Int32`**
- **`Bool`**
- **`Float`**
- **`[Word8]`**

- Numeric operations include addition, subtraction, division, multiplications, comparisons, and conversion between numeric types.
- Boolean operations include **not**, **and,** and **or**.
- Integer operations include standard bitwise operations.
- [Word8] operations include append and element retrieval.

Remote Refs/Conditionals

```
class RemoteReference a where
    newRemoteRef    :: Expr a -> Arduino (RemoteRef a)
    readRemoteRef   :: RemoteRef a -> Arduino (Expr a)
    writeRemoteRef  :: RemoteRef a -> Expr a ->
                        Arduino ()
    modifyRemoteRef :: RemoteRef a ->
                       (Expr a -> Expr a) ->
                       Arduino ()
```

```
ifThenElse :: Expr Bool ->          -- If expression
              Arduino () ->         -- Then clause
              Arduino () ->         -- Else clause
              Arduino ()
while :: RemoteRef a ->             -- Loop Reference
         (Expr a -> Expr Bool) ->   -- Termination Test
         (Expr a -> Expr a) ->      -- Update Function
         Arduino () ->              -- Loop Body
         Arduino ()
loopE :: Arduino () ->              -- Loop Body
         Arudino ()
```

Remote Refs/Conditionals

```
class RemoteReference a where
    newRemoteRef    :: Expr a -> Arduino (RemoteRef a)
    readRemoteRef   :: RemoteRef a -> Arduino (Expr a)
    writeRemoteRef  :: RemoteRef a -> Expr a ->
                        Arduino ()
    modifyRemoteRef :: RemoteRef a ->
                       (Expr a -> Expr a) ->
                       Arduino ()
```

```
ifThenElse :: Expr Bool ->          -- If expression
              Arduino () ->         -- Then clause
              Arduino () ->         -- Else clause
              Arduino ()
while :: RemoteRef a ->             -- Loop Reference
         (Expr a -> Expr Bool) ->   -- Termination Test
         (Expr a -> Expr a) ->      -- Update Function
         Arduino () ->              -- Loop Body
         Arduino ()
loopE :: Arduino () ->              -- Loop Body
         Arudino ()
```

Firmata -> Haskino Firmware

- The firmware and serial communication protocol used with hArduino is Firmata.
- Firmata is based on MIDI, and has a strange, inefficient 7 bit encoding.
- Digital and Analog Reads are done via a continuous update mechanism in Firmata, which would not fit well with the Haskino architecture.
- Extending Firmata to handle expressions and conditions would have been very difficult.
- Instead, a new Haskino protocol and firmware were developed.

Strong Haskino Example

```
example :: IO ()
example =
     withArduino False "/dev/cu.usbmodem1421" $ do
          let button = 2
          let led1 = 6
          let led2 = 7
          setPinMode button INPUT
          setPinMode led1 OUTPUT
          setPinMode led2 OUTPUT
          loop $ do
               x <- digitalRead button
               digitalWrite led1 x
               digitalWrite led2 (not x)
               delayMillis 100
```

Deep Haskino Example

```
exampleE :: IO ()
exampleE =
     withArduino False "/dev/cu.usbmodem1421" $ do
          let button = 2
          let led1 = 6
          let led2 = 7
          setPinModeE button INPUT
          setPinModeE led1 OUTPUT
          setPinModeE led2 OUTPUT
          loopE $ do
               ex <- digitalReadE button
               digitalWriteE led1 ex
               digitalWriteE led2 (notB ex)
               delayMillisE 100
```

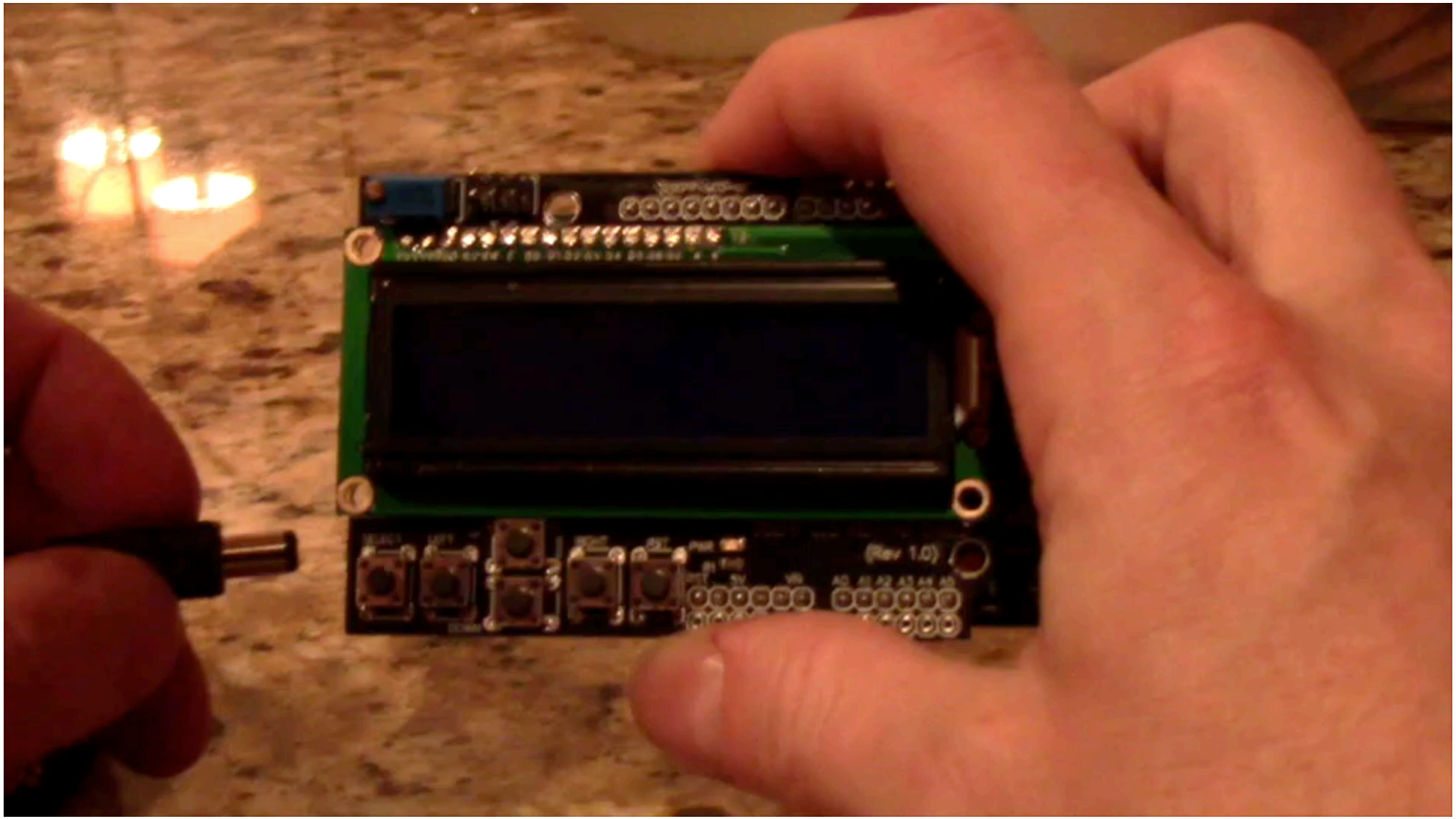
Cutting the Cord

- The firmware includes the notion of tasks, which are a monadic structure which can be scheduled to execute at a future time.
- These tasks are created using a createTaskE command which takes an **Arduino ()** monad as a argument.
- Additionally, a bootTaskE command allows one task to be stored in the Arduino's EEPROM.
- If the firmware finds a task stored in EEPROM upon boot, it will execute that task at startup, which provides our desired ability to execute a program written in Haskell stand-alone.

Comparison of Strong and Deep Versions

	Runtime-tethered	**Deeply-embedded**
Values Stored On	Host	Arduino
Binds Occur On	Host	Arduino
Conditionals on Target	No	Yes
Tasks Can Use Procedures	No	Yes
Maximum Program Size	Limited by Host Memory	Limited by Arduino Memory
Communication Overhead	Higher	Lower

A Larger Example



Conclusion

- Haskino provides two complimentary methods of using Haskell as a development environment for Haskell software
  - Strong remote monad provides a method for quick prototyping in a tethered environment.
  - Deep version of Haskino allows the programmer to bring the full power of Haskell to developing standalone software for the Arduino
- Future work
  - Add a third method of development, directly generating C code from the Arduino monad.
  - We want to extend the scheduling mechanism in Haskino to allow for interrupt processing and inter-task communication.
  - We want to explore using HERMIT to semi-automatically translate programs written in the tethered strong version into programs written using deep embedding.

github.com/ku-fpg/haskino