simpPRU

BeagleBoard.org Foundation
Jun 29, 2025

Table of contents

Chapter 1

simpPRU Basics

The PRU is a dual core micro-controller system present on the AM335x SoC which powers the BeagleBone. It is meant to be used for high speed jitter free IO control. Being independent from the linux scheduler and having direct access to the IO pins of the BeagleBone Black, the PRU is ideal for offloading IO intensive tasks.

Programming the PRU is a uphill task for a beginner, since it involves several steps, writing the firmware for the PRU, writing a loader program. This can be a easy task for a experienced developer, but it keeps many creative developers away. So, I propose to implement a easy to understand language for the PRU, hiding away all the low level stuff and providing a clean interface to program PRU.

This can be achieved by implementing a language on top of PRU C. It will directly compile down to PRU C. This could also be solved by implementing a bytecode engine on the PRU, but this will result in waste of already limited resources on PRU. With this approach, both PRU cores can be run independent of each other.

simpPRU

Intuitive language for PRU which compiles down to PRU C.

1.1 What is simpPRU

- simpPRU is a procedural programming language.
- It is a statically typed language. Variables and functions must be assigned data types during compilation.
- It is type-safe, and data types of variables are decided during compilation.
- simpPRU codes have a `.sim` extension.
- simpPRU provides a console app to use Remoteproc functionality.

Chapter 2

Build from source

2.1 Dependencies

- flex
- bison
- gcc
- gcc-pru
- gnuprumcu
- cmake

2.2 Build

```
git clone https://github.com/VedantParanjape/simpPRU.git
cd simpPRU
mkdir build
cd build
cmake ..
make
```

2.3 Install

```
sudo make install
```

2.4 Generate debian package

```
sudo make package
```

Chapter 3

Install

3.1 Dependencies

- gcc-pru
- gnuprumcu
- config-pin utility (for autoconfig)

3.2 Installation

For Instructions head over to Installation

3.3 Requirements

Currently this only supports am335x systems: PocketBeagle, BeagleBone Black and BeagleBone Black Wireless:

- `gcc-pru`
- `gnuprumcu`
- beaglebone image with official support for remoteproc: `ti-4.19+ kernel`
- `config-pin` utility

3.4 Build from source

For Instructions head over to Building from source

```
simppru-console
```

For detailed usage head to Detailed Usage

3.5 amd64

```
wget https://github.com/VedantParanjape/simpPRU/releases/download/1.4/
↪simppru-1.4-amd64.deb

sudo dpkg -i simppru-1.4-amd64.deb
```

3.6 armhf

```
wget https://github.com/VedantParanjape/simpPRU/releases/download/1.4/
↪simppru-1.4-armhf.deb

sudo dpkg -i simppru-1.4-armhf.deb
```

3.7 Issues

- For full source code of simPRU visit
- To report a bug or start a issue visit

Chapter 4

Language Syntax

- simpPRU is a procedural programming language.
- It is a statically typed language. Variables and functions must be assigned data types during compilation.
- It is type-safe, and data types of variables are decided during compilation.
- simPRU codes have a `.sim` extension.

4.1 Datatypes

- `int` - Integer datatype
- `bool` - Boolean datatype
- `char` / `uint8` - Character / Unsigned 8 bit integer datatype
- `void` - Void datatype, can only be used a return type for functions

4.2 Constants

- `<any_integer>` - Integer constant. Integers can be decimal, hexadecimal (start with 0x or 0X) or octal (start with 0)
- `'<any character>'` - Character constant. These can be assigned to both int and char/uint8 variables
- `true` - Boolean constant (True)
- `false` - Boolean constant (False)
- `Px_yz` - Pin mapping constants are Integer constant, where x is 1,2 or 8,9 and yz are the header pin numbers.

4.3 Operators

- `{,}` - Braces
- `(,)` - Parenthesis
- `/,*,+,-,%` - Arithmetic operators
- `>,<,==,!=,>=,<=` - Comparison operators
- `~,&,|,<<,>>` - Bitwise operators: not, and, or and bitshifts

- `not,and,or` - Logical operators: not, and, or
- `:=` - Assignment operator
- Result of Arithmetic and Bitwise operators is Integer constant.
- Result of Comparison and Logical operators is Boolean constant.
- Characters are treated as integers when used in Arithmetic expressions.
- Only Integer constants can be used with Arithmetic and Bitwise operators.
- Only Integer constants can be used with Comparison operators.
- Only Boolean constants can be used with Logical operators.
- Operators are evaluated following these precedence rules.

```
Correct: bool out := 5 > 6;
Wrong:   int yy := 5 > 6;
```

4.4 Variable declaration

- Datatype of variable needs to be specified during compile time.
- Variables can be assigned values after declarations.
- If variable is not assigned a value after declaration, it is set to `0` for `integer` and `char/uint8` and to `false` for `boolean` by default.
- Variables can be assigned other variables of same datatype. `int`s and `char`s can be assigned to each other.
- Variables can be assigned expressions whose output is of same datatype.

4.4.1 Declaration

```
int var;
char char_var;
bool test_var;
```

4.4.2 Assignment during Declaration

```
int var := 99;
char char_var := 'a';
uint8 short_var := 255;
bool test_var := false;
```

4.4.3 Assignment

```
var := 45;
short_var := var;
test_var := true;
```

- Variables to be assigned must be declared earlier.
- Datatype of the variables cannot change. Only appropriate expressions/constants of their respective datatypes can be assigned to the variables.

- Integer and Character variable can be assigned only Integer expression/Integer constant/Character constant.
- Boolean variable can be assigned only Boolean expression/constant.

4.5 Arrays

- Arrays are static - their size has to be known at compile time and this size cannot be changed later.
- Arrays can be used with bool, int and char.
- Arrays do not support any arithmetic / logical / comparison / bitwise operators, however these operators work fine on their elements.

4.5.1 Declaration and Assignment

- The data type has to be specified as data_type[size].
- Array of char can be initialized from a double quoted string, where the length of the array would be at least the length of the string plus 1.

```
int[16] a; /* array of 16 integers */
char[20] string1 := "I love BeagleBoards";
```

4.5.2 Indexing:

- Arrays are zero-indexed.
- The index can be either a char or an int or an expression involving chars and ints.
- Accessing elements of an array:

```
int a := arr[4]; /* Copy the 5th element of arr to a */
```

- Changing elements of an array:

```
arr[4] := 5; /* The 5th element of arr is now 5 */

int i := 4;
arr[i] := 6; /* The 5th element of arr is now 6 */

char j := 4;
arr[j] := 7; /* The 5th element of arr is now 7 */

arr[i+j] := 1; /* The 9th element of arr is now 1 */

/* Declaring and initializing an array with all zeros */
int[16] arr;
for: i in 0:16 {
    arr[i] := 0;
}
```

4.6 Comments

- simpPRU supports C style multiline comments.

```
/* This is a comment */

/* Comments can span
multiple lines */
```

4.7 Keyword and Identifiers

Table 4.1: Reserved keywords

“true“	**“read_counter“**	**“stop_counter“**
“false“	**“start_counter“**	**“pwm“**
“int“	**“delay“**	**“digital_write“**
“bool“	**“digital_read“**	**“def“**
“void“	**“return“**	**“or“**
“if“	**“and“**	**“not“**
“elif“	**“continue“**	**“break“**
“else“	**“while“**	**“in“**
“for“	**“init_message_channel“**	**“send_message“**
“receive_message“	**“print“**	**“println“**

4.7.1 Valid identifier naming

- An identifier/variable name must be start with an alphabet or underscore (_) only, no other special characters, digits are allowed as first character of the identifier/variable name.

  `product_name, age, _gender`

- Any space cannot be used between two words of an identifier/variable; you can use underscore (_) instead of space.

  `product_name, my_age, gross_salary`

- An identifier/variable may contain only characters, digits and underscores only. No other special characters are allowed, and we cannot use digit as first character of an identifier/variable name (as written in the first point).

  `length1, length2, _City_1`

Detailed info: https://www.includehelp.com/c/identifier-variable-naming-conventions.aspx

4.8 Expressions

4.8.1 Arithmetic expressions

```
=> (9 + 8) * 2 + -1;
33
=> 11 % 3;
2
=> 2 * 6 << 2 + 1;
96
=> ~0xFFFFFFFF;
0
```

4.8.2 Boolean expressions

```
=> 9 > 2 or 8 != 2 and not( 2 >= 5 or 9 <= 5 ) or 9 != 7;
true
=> 0xFFFFFFFF != 0XFFFFFFFF;
false
=> 'a' < 'b';
true
```

- **Note** : Expressions are evaluated following the *operator precedence <#operators>*

4.9 If-else statement

Statements in the if-block are executed only if the if-expression evaluates to `true`. If the value of expression is `true`, statement1 and any other statements in the block are executed and the else-block, if present, is skipped. If the value of expression is `false`, then the if-block is skipped and the else-block, if present, is executed. If elif-block are present, they are evaluated, if they become `true`, the statement is executed, otherwise, it goes on to eval next set of statements

4.9.1 Syntax

```
if : boolean_expression {
    statement 1
    ...
    ...
}
elif : boolean_expression {
    statement 2
    ...
    ...
    ...
}
else {
    statement 3
    ...
    ...
}
```

4.9.2 Examples

```
int a := 3;

if : a != 4 {
    a := 4;
}
elif : a > 4 {
    a := 10;
}
else {
    a := 0;
}
```

- This will evaluate as follows, since `a = 3`, if-block (`3!=4`) will evaluate to true, and value of a will be set to 4, and program execution will stop.

4.10 For-loop statement

For loop is a range based for loop. Range variable is a local variable with scope only inside the for loop.

4.10.1 Syntax

```
for : var in start:stop {
    statement 1
    ....
    ....
}
```

- Here, for loop is a range based loop, value of integer variable `var` will vary from `start` to `stop - 1`. Value of `var` does not equal `stop`. Here, `increment` is assumed to be 1, so `start` will have to less than `stop`.
- Optionally, `start` can be skipped, and it will automatically start from 0, like this:

```
for : var in :stop {
    statement 1
    ....
    ....
}
```

- Optionally, `increment` can also be specified like this. Here, `stop` can be less than `start` if `increment` is negative.

```
for : var in start:stop:increment {
    statement 1
    ....
    ....
}
```

- **Note** : **var** is a **integer**, and **start, stop, increment** can be **arithmetic expression, integer or character variable, or integer or character constant**.

4.10.2 Examples

```
int sum := 0;

for : i in 1:4 {
    sum = sum + i;
}
```

```
int mx := 32;
int nt;

for : j in 2:mx-10 {
    nt := nt + j;
}
```

```
int sum := 0;

for : i in in 10:1:-2 { /*10, 8, 6, 4, 2*/
    sum = sum + i;
}
```

4.11 While-loop statement

While loop statement repeatedly executes a target statement as long as a given condition is true.

4.11.1 Syntax

```
while : boolean_expression {
    statement 1
    ...
    ...
}
```

4.11.2 Examples

- Infinite loop

```
while true {
    do_something..
    ...
}
```

- Normal loop, will repeat 30 times, before exiting

```
int tag := 0;

while : tag < 30 {
    tag := tag + 1;
}
```

4.12 Control statements

- **Note** : `break` and `continue` can only be used inside looping statements

4.12.1 break

`break` is used to break execution in a loop statement, either `for loop` or `while loop`. It exits the loop upon calling.

Syntax

`break;`

Examples

```
for : i in 0:9 {
    if : i == 3 {
        break;
    }
}
```

4.12.2 continue

`continue` is used to continue execution in a loop statement, either `for loop` or `while loop`.

Syntax

```
continue;
```

Examples

```
for : j in 9:19 {
    if : i == 12 {
        continue;
    }
    else {
        break;
    }
}
```

4.13 Functions

4.13.1 Function definition

A function is a group of statements that together perform a task. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is such that each function performs a specific task. A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

- **Warning** : Function must be defined before calling it.

Syntax

```
def <function_name> : <data_type> : <data_type> <param_name>, <data_type>
↪<param_name>, ... {
    statement 1;
    ...
    ...

    return <data_type>;
}
```

Note: If return data type is void, then return statement is not needed, and if still it is added, it must be return nothing, i.e., something like this `return ;`

Warning: `return` can only be present in the body of the function only once, that too at the end of the function, not inside any compound statements.

Danger: `return` inside a compound statement, this syntax is not allowed.

```
def test : int : int a {
if : a < 4 {
return a;
}
}
```

- **Correct** : `return` is not inside compound statements, It should be placed only at the end of function definition

```
def test : int : int a {
int gf := 8;
if : a < 4
{
gf := 4;
}
return gf;
}
```

Examples

Examples according to return types

- **Integer**

```
def test_func : int : int a, int b
{
   int aa := a + 5;
   if : aa < 3 {
        aa : = 0;
   }

   return aa + b;
}
```

- **Character**

```
def next_char : char : char ch, int inc {
   char chinc := ch + inc;
   return chinc;
}
```

- **Boolean**

```
def compare : bool : int val {
   bool ret :=false;
      if : val < 0 {
         ret := true;
      }
      return ret;
   }
```

- **Void**

```
def example_func_v : void : {
    int temp := 90;

    return;
}
```

4.13.2 Function call

Functions can be called only if, they have been defined earlier. They return data types according to their definition. Parameters are passed by value. Only pass by value is supported as of now.

Syntax

```
function_name(var1, var2, ..);
```

Examples

- **Integer** `int a := 55; int ret_val := test_func(4, a);`
- **Character** `char a := 'a'; char b := next_char(a, 1);`
- **Boolean** `bool val := compare(22); compare(-2);`
- **Void** `example_func(false); example_func_v();`

4.13.3 Testing or Debugging

For testing or debugging code, use the –test or -t flag to enable print, println and stub functions. Use –preprocess to stop after generating the C code only. Then run the generated C code (at /tmp/temp.c) using `gcc`.

Print functions

print can take either a string (double quoted) or any `int` / `char` / `bool` identifier.

println is similar to print but also prints a newline (`\n`).

Examples

```
print(”Hello World!”);
int a := 2;
print(a);
a := a + 2;
print(a);
println(””);
```

Stub functions

PRU specific functions will be replaced by stub functions which print **function_name called with arguments arg_name** when called.

Chapter 5

IO Functions

- All Header pins are `constant integer variable` by default, with its value equal to respective R30/R31 register bit
  - Example: `P1_20` is an constant integer variable with value `16`, similarly `P1_02` is an constant integer variable with value `9`

5.1 Digital Write

`digital_write` is a function which enables PRU to write given logic level at specified output pin. It is a function with void return type and it's parameters are `integer` and `boolean`, first parameter is the pin number to write to or PRU R30 register bit and second parameter is `boolean` value to be written. `true` for HIGH and `false` for LOW.

5.1.1 Syntax

```
digital_write(pin_number, value);
```

Parameters

- `pin_number` is an integer. It must be a header pin name which supports output, or PRU R30 Register bit.
- `value` is a boolean. It is used to set logic level of the output pin, `true` for HIGH and `false` for LOW.

Return Type

- `void` - returns nothing.

5.1.2 Example

```
int a := 32;

if : a < 32 {
    digital_write(P1_29, true);
}
else {
    digital_write(P1_29, false);
}
```

If the value of a < 32, then pin P1_29 is set to HIGH or else it is set to LOW.

5.2 Digital Read

digital_read is a function which enables PRU to read logic level at specified input pin. It is a function with return type boolean and it's parameter is a integer whose value must be the pin number to be read or PRU R31 register bit.

5.2.1 Syntax

```
digital_read(pin_number);
```

Parameters

- pin_number is an integer. It must be a header pin name which supports input, or PRU R31 Register bit.

Return Type

- boolean - returns the logic level of the pin number passed to it. It returns true for HIGH and false for LOW.

5.2.2 Example

```
if digital_read(P1_20) {
    digital_write(P1_29, false);
}
else {
    digital_write(P1_29, true);
}
```

Logic level of pin P1_20 is read. If it is HIGH, then pin P1_29 is set to LOW, or else it is set to HIGH.

5.3 Delay

delay is a function which makes PRU wait for specified milliseconds. When this is called PRU does absolutely nothing, it just sits there waiting.

5.3.1 Syntax

```
delay(time_in_ms);
```

Parameters

- time_in_ms is an integer. It is the amount of time PRU should wait in milliseconds. (1000 milliseconds = 1 second).

Return Type

- `void` - returns nothing.

5.3.2 Example

```
digital_write(P1_29, true);
delay(2000);
digital_write(P1_29, false);
```

Logic level of pin `P1_29` is set to `HIGH`, PRU waits for *2000 ms = 2 seconds*, and then sets the logic level of pin `P1_29` to `LOW`.

5.4 Start counter

`start_counter` is a function which starts PRU's internal counter. It counts number of CPU cycles. So it can be used to count time elapsed, as it is known that each cycle takes 5 nanoseconds.

5.4.1 Syntax

`start_counter()`

Parameters

- n/a

Return Type

- `void` - returns nothing.

5.4.2 Example

```
start_counter();
```

5.5 Stop counter

`stop_counter` is a function which stops PRU's internal counter.

5.5.1 Syntax

`stop_counter()`

Parameters

- n/a

Return Type

- `void` - returns nothing.

5.5.2 Example

```
stop_counter();
```

5.6 Read counter

`read_counter` is a function which reads PRU's internal counter and returns the value. It counts number of CPU cycles. So it can be used to count time elapsed, as it is known that each cycle takes 5 nanoseconds.

5.6.1 Syntax

```
read_counter()
```

Parameters

- n/a

Return Type

- `integer` - returns the number of cycles elapsed since calling `start_counter`.

5.6.2 Example

```
start_counter();

while : read_counter < 200000000 {
    digital_write(P1_29, true);
}

digital_write(P1_29, false);
stop_counter();
```

while the value of hardware counter is less than 200000000, it will set logic level of pin `P1_29` to `HIGH`, after that it will set it to `LOW`. Here, 200000000 cpu cycles means 1 second of time, as CPU clock is 200 MHz. So, LED will turn on for 1 second, and turn off after.

5.7 Init message channel

`init_message_channel` is a function which is used to initialise communication channel between PRU and the ARM core. It is sets up necessary structures to use RPMSG to communicate, it expects a init message from the ARM core to initialise. It is a necessary to call this function before using any of the message functions.

5.7.1 Syntax

```
init_message_channel()
```

Parameters

- n/a

Return Type

- `void` - returns nothing

5.7.2 Example

```
init_message_channel();
```

5.8 Receive message

`receive_message` is a function which is used to receive messages from ARM to the PRU, messages can only be `integers`, as only they are supported as of now. It uses RPMSG channel setup by `init_message_channel` to receive messages from ARM core.

5.8.1 Syntax

`receive_message()`

Parameters

- n/a

Return Type

- `integer` - returns integer data received from PRU

5.8.2 Example

```
init_message_channel();

int temp := receive_message();

if : temp >= 0 {
    digital_write(P1_29, true);
}
else {
    digital_write(P1_29, false);
}
```

5.9 Send message

There are six functions which are used to send messages to ARM core from PRU, messages can be `integers`, `characters`, `bools`, `integer arrays`, `character arrays`, and `boolean arrays`. It uses RPMSG channel setup by `init_message_channel` to send messages from PRU to the ARM core.

For sending arrays, arrays are automatically converted to a string, for example, [1, 2, 3, 4] would become “1 2 3 4”.

5.9.1 Syntax

- `send_int(expression)`
- `send_char(expression)`
- `send_bool(expression)`
- `send_ints(identifier)`
- `send_chars(identifier)`
- `send_bools(identifier)`
- `send_message` is an alias for `send_int` to preserve backwards compatibility.

Parameters

- For `send_int` and `send_char`, `expression` would be an arithmetic expression.
- For `send_bool`, `expression` would be a boolean expression
- For `send_ints`, `identifier` should be an identifier for an integer array.
- For `send_chars`, `identifier` should be an identifier for a character array.
- For `send_bools`, `identifier` should be an identifier for a boolean array.

5.9.2 Example

```
init_message_channel();

if : digital_read(P1_29) {
    send_bool(true);
}
else {
    send_int(0);
}
```

Chapter 6

Usage(simppru)

```
simppru [OPTION...] FILE

      --device=<device_name> Select for which BeagleBoard to compile
                             (pocketbeagle, bbb, bbbwireless, bbai)
      --load                 Load generated firmware to /lib/firmware/
  -o, --output=<file>        Place the output into <file>
  -p, --pru=<pru_id>         Select which pru id (0/1) for which program is␣
↪to
                             be compiled
      --verbose              Enable verbose mode (dump symbol table and ast
                             graph)
      --preprocess           Stop after generating the intermediate C
                             file (located at /tmp/temp.c)
  -t  --test                 Use stub functions for PRU specific functions␣
↪and
                             enable the print functions, useful for testing␣
↪and debugging
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or␣
↪optional
for any corresponding short options.
```

simppru autodetects BeagleBoard model and automatically configures pin mux using config-pin. This functionality doesn't work on BeagleBone Blue and AI.

Say we have to compile a example file called `test.sim`, command will be as follows:

```
simppru test.sim --load
```

If we only want to generate binary for pru0

```
simppru test.sim -o test_firmware -p 0
```

this will generate a file named `test_firmware.pru0`

Chapter 7

Usage(simppru-console)

simppru-console is a console app, it can be used to send/receive message to the PRU using RPMSG, and also start/stop the PRU. It is built to facilitate easier way to use rpmsg and remoteproc API's to control and communicate with the PRU

- **Warning** : Make sure to stop PRU before exiting. Press `ctrl+c` to exit

7.1 Features

Use arrow keys to navigate around the textbox and buttons.

7.1.1 Start/stop buttons

Use these button to start/stop the selected PRU. If PRU is already running, on starting simppru-console, it is automatically stopped.

7.1.2 Send message to PRU

Use this text box to send data to the PRU, only *Integers* are supported. On pressing enter, the typed message is sent.

PRU0 is running echo program, whatever is sent is echoed back.

7.1.3 Receive message from PRU

The large box in the screen shows data received from the PRU, It runs using a for loop, which checks if new message is arrived every 10 ms.

- PRU is running echo program, whatever is sent is echoed back.

- PRU is running countup program, it sends a increasing count every 1 second, which starts from 0

7.1.4 Change PRU ID

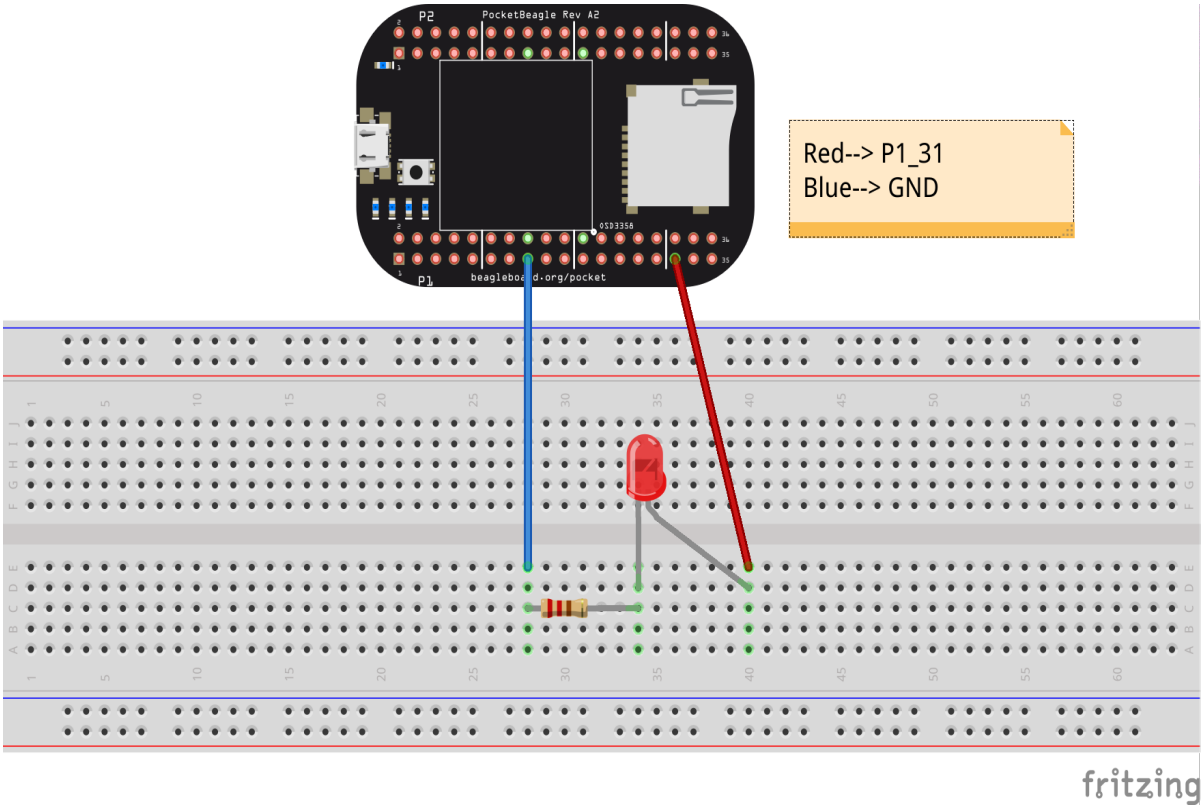
Using the radio box in the upper right corner, one can change the PRU id, i.e. if one wants to use the features for PRU0 or PRU1

Chapter 8

simpPRU Examples

These are the examples which have been tested on simpPRU.These examples will serve as a guide for the users to implement.

8.1 Delay example



8.1.1 Code

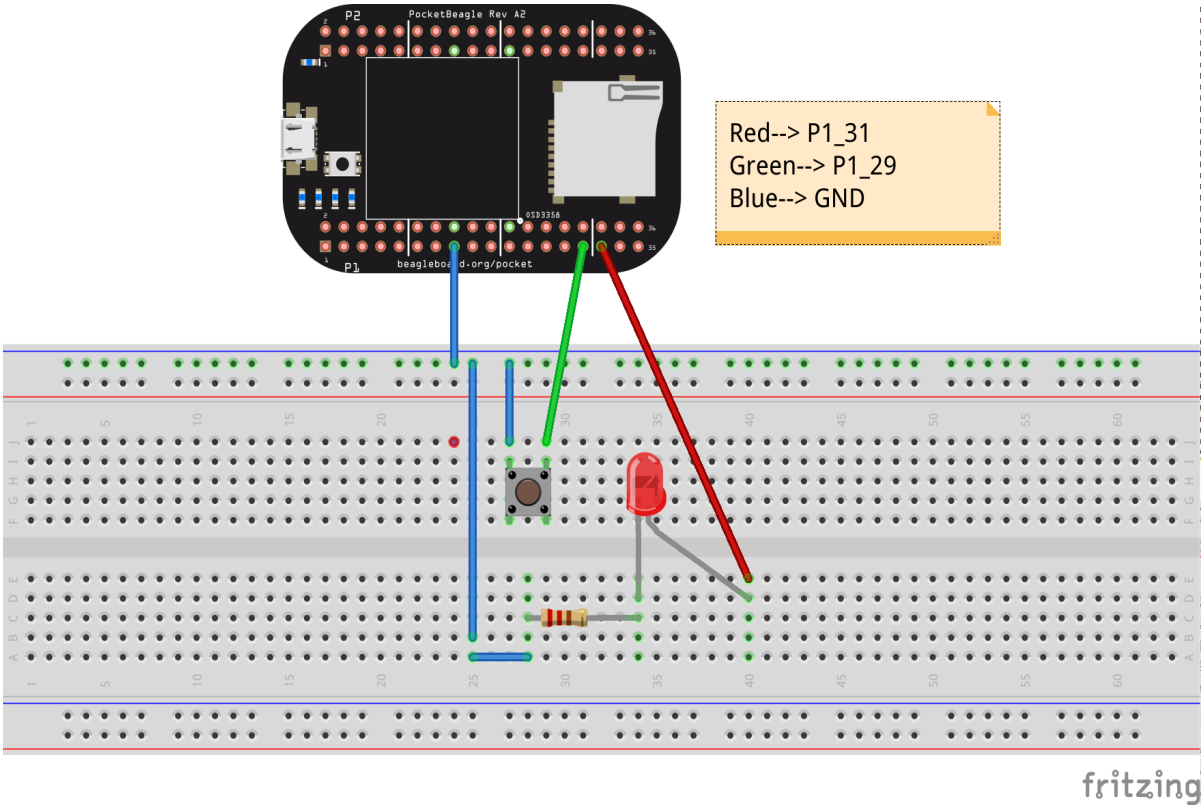
```
digital_write(P1_31, true);
delay(2000);
digital_write(P1_31, false);
delay(5000);
digital_write(P1_31, true);
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.1.2 Explanation

This code snippet writes HIGH to header pin P1_31, then waits for 2000ms using the `delay` call, after that it writes LOW to header pin P1_31, then again waits for 5000ms using the `delay` call, and finally writes HIGH to header pin P1_31.

8.2 Digital read example



8.2.1 Code

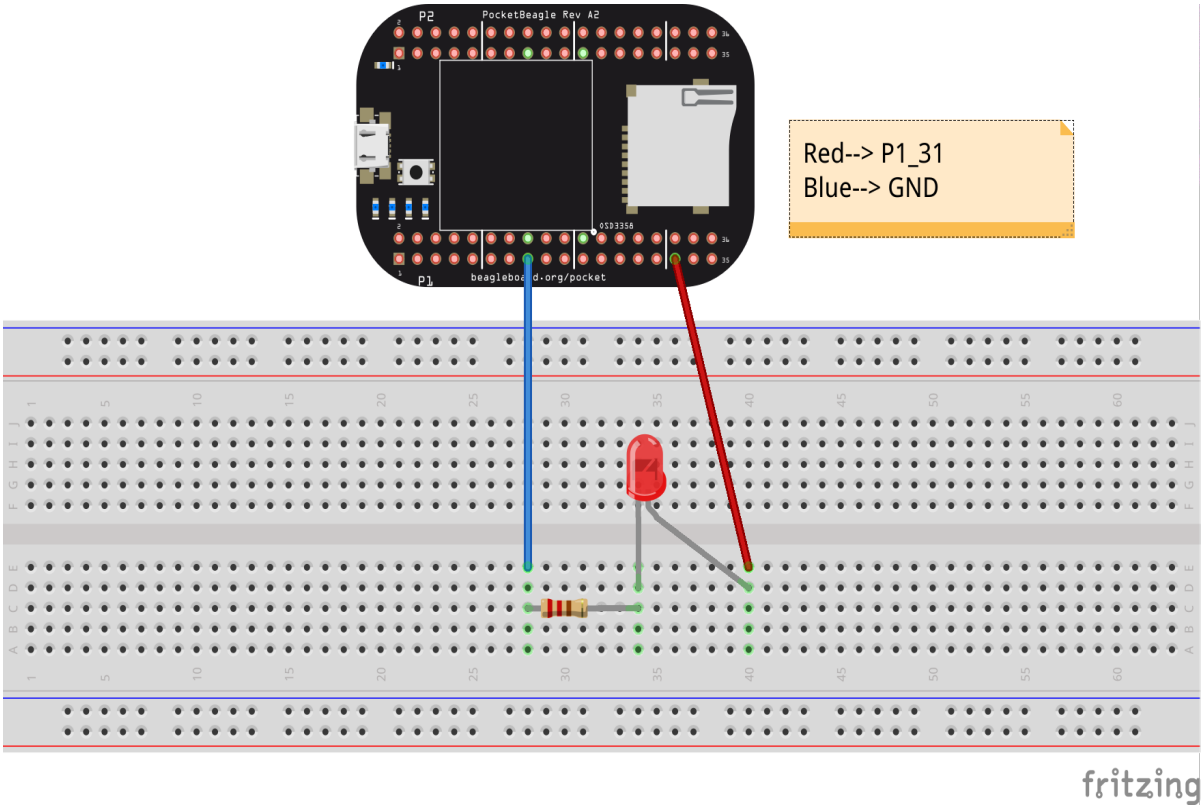
```
while : true {
    if : digital_read(P1_29) {
        digital_write(P1_31, false);
    }
    else {
        digital_write(P1_31, true);
    }
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.2.2 Explanation

This code runs a never ending loop, since it is `while : true`. Inside `while` it checks if header pin P1_29 is HIGH or LOW. If header pin P1_29 is HIGH, header pin P1_31 is set to LOW, and if header pin P1_29 is LOW, header pin P1_31 is set to HIGH.

8.3 Digital write example



8.3.1 Code

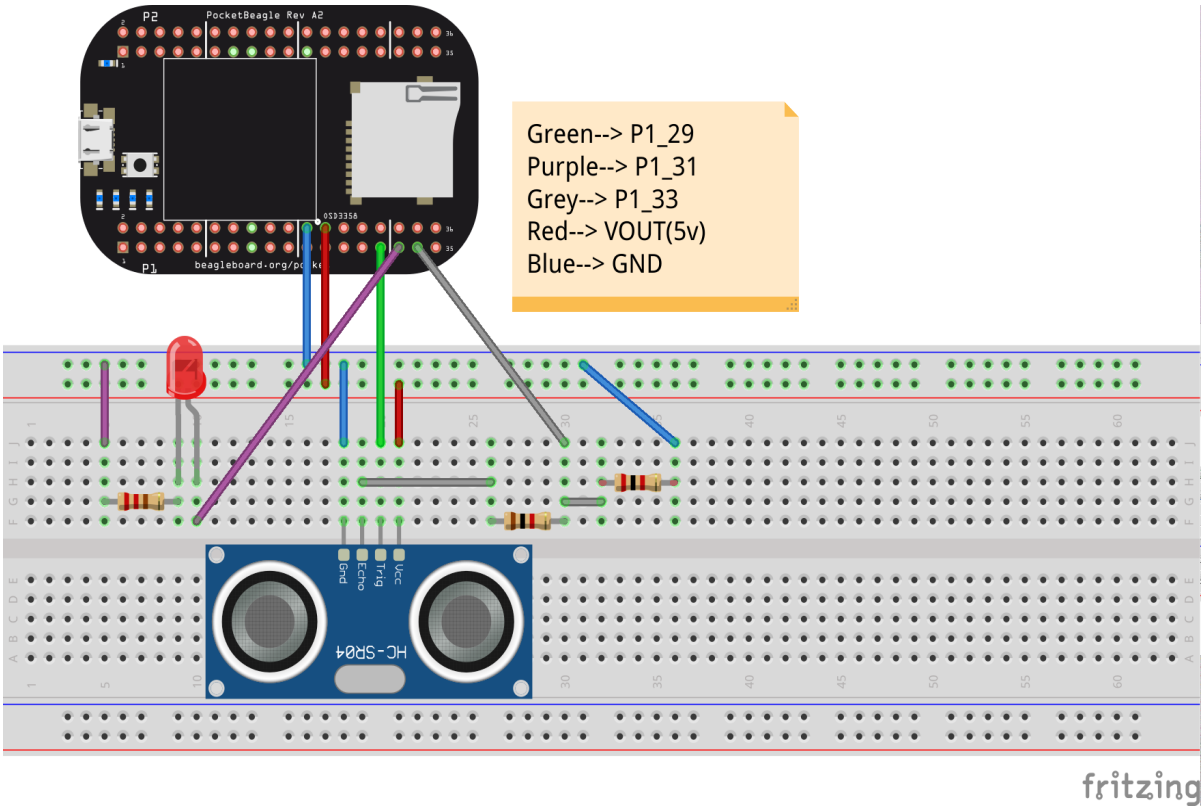
```
while : true {
    digital_write(P1_31, true);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.3.2 Explanation

This code runs a never ending loop, since it is `while : true`. Inside `while` it sets header pin P1_31 to HIGH.

8.4 HCSR04 Distance Sensor example (sending distance data to ARM using RPMSG)



8.4.1 Code

```
def measure : int : {
    bool timeout := false;
    int echo := -1;

    start_counter();

    while : read_counter() <= 2000 {
        digital_write(5, true);
    }
    digital_write(5, false);
    stop_counter();

    start_counter();
    while : not (digital_read(6)) and true {
        if : read_counter() > 200000000 {
            timeout := true;
            break;
        }
    }
    stop_counter();

    if : not(timeout) and true {
        start_counter();
        while : digital_read(6) and true {
            if : read_counter() > 200000000 {
                timeout := true;
```

(continues on next page)

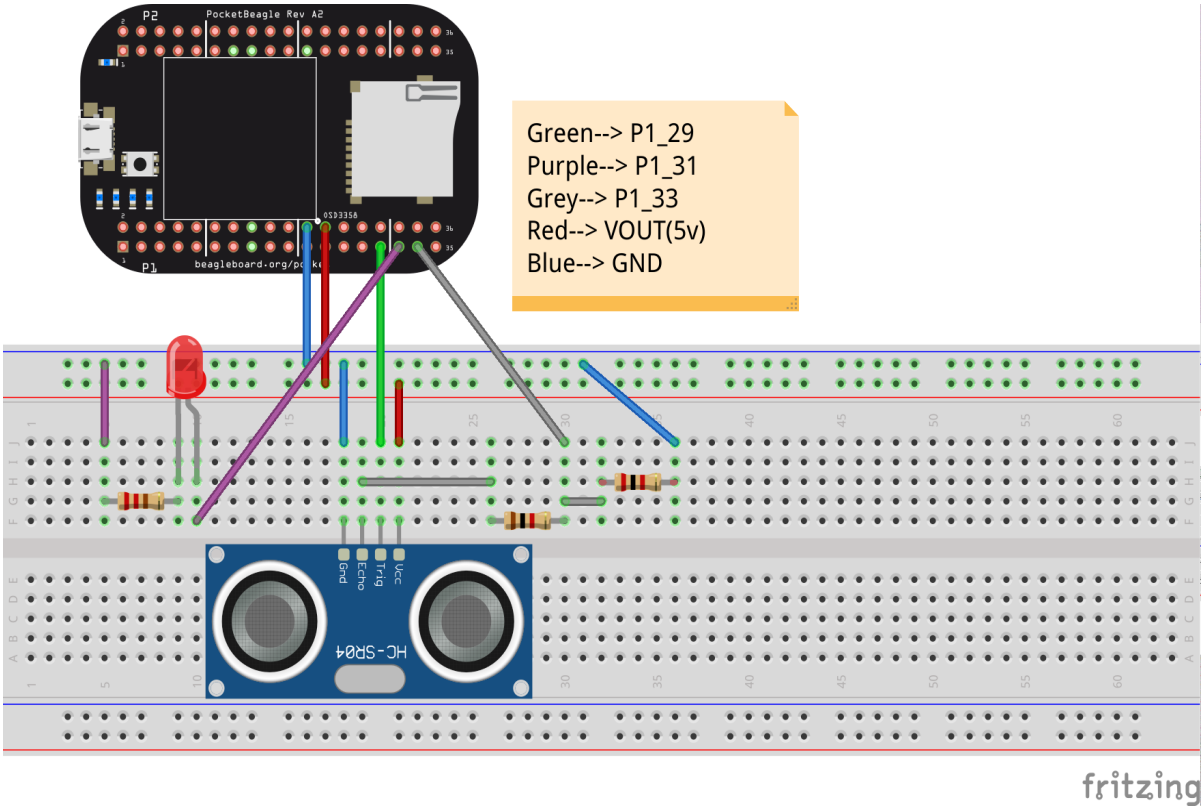
(continued from previous page)

```
                break;
            }
            echo := read_counter();
        }
        stop_counter();
    }

    if : timeout and true {
        echo := 0;
    }

    return echo;
}

init_message_channel();

while : true {
    int ping:= measure();

    send_message(ping);
    delay(1000);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.4.2 Explanation

8.5 Ultrasonic range sensor example



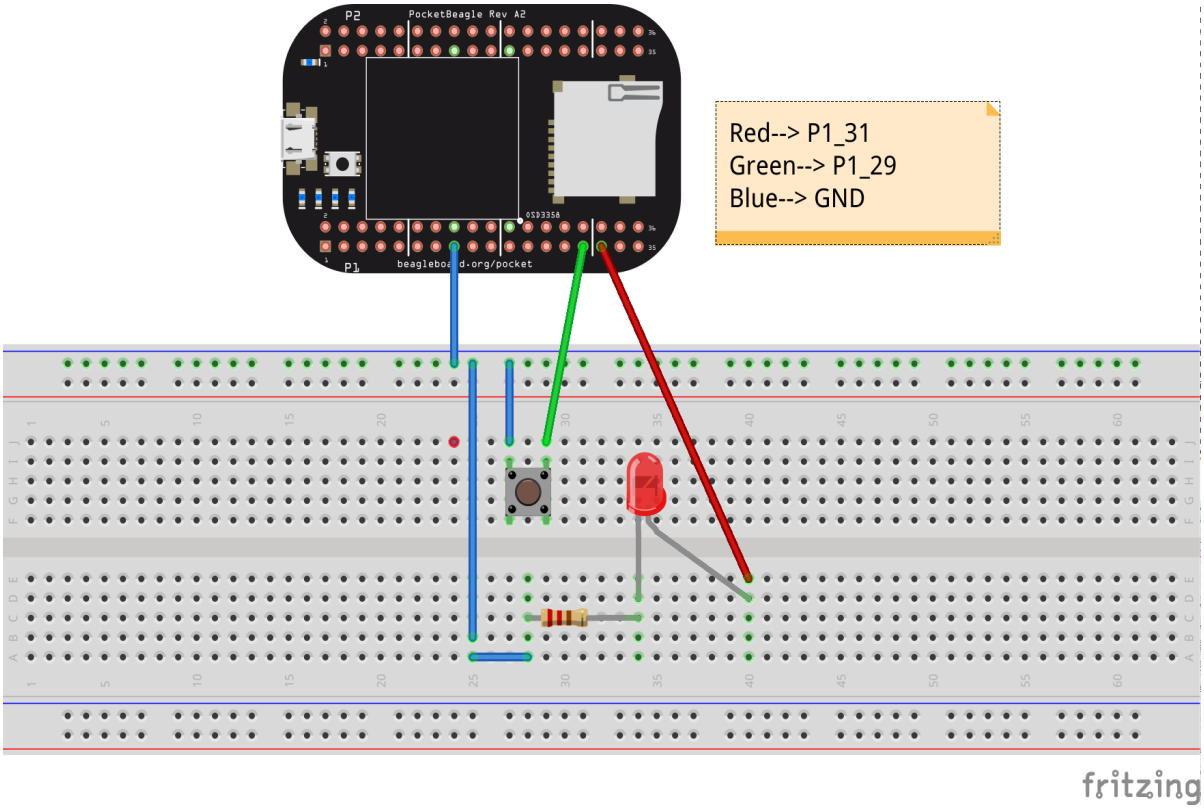
8.5.1 Code

```
def measure : int : {
    bool timeout := false;
    int echo := 0;

    start_counter();

    while : read_counter() <= 2000 {
        digital_write(7, true);
    }
    digital_write(7, false);
    stop_counter();

    start_counter();
    while : not (digital_read(1)) and true {
        if : read_counter() > 200000000 {
            timeout := true;
            break;
        }
    }
    stop_counter();

    if : not(timeout) and true {
        start_counter();
        while : digital_read(1) and true {
            if : read_counter() > 200000000 {
                timeout := true;
                break;
            }
            echo := read_counter();
        }
        stop_counter();
    }

    if : timeout and true {
        echo := 0;
    }

    return echo;
}

while : true {
    int ping:= measure()*1000;

    if : ping > 292200 {
        digital_write(4, false);
    }
    else
    {
        digital_write(4, true);
    }
    delay(1000);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.5.2 Explanation

8.6 Sending state of button using RPMSG



8.6.1 Code

```
init_message_channel();

while : true {
    if : digital_read(P1_29) {
        send_message(1);
    }
    else {
        send_message(0);
    }
    delay(100);
}
```

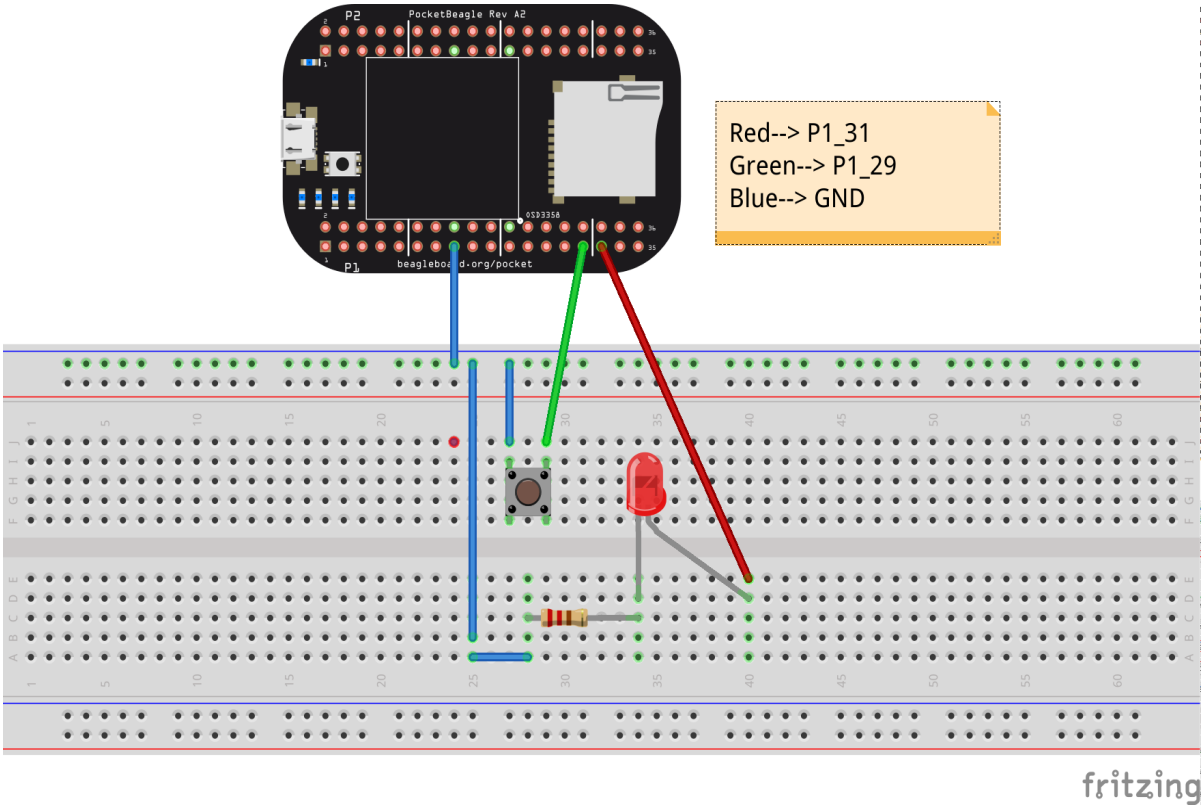
- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.6.2 Explanation

`init_message_channel` is needed to setup communication channel between ARM<->PRU. It only needs to be called once, before using RPMSG functions.

`while : true` loop runs endlessly, inside this, we check for value of header pin P1_29, if it reads HIGH, 1 is sent to the ARM core using `send_message` and if it is LOW, 0 is sent to ARM core using `send_message`. Then PRU waits for 100ms, and repeats the steps again and again.

8.7 LED blink on button press example



8.7.1 Code

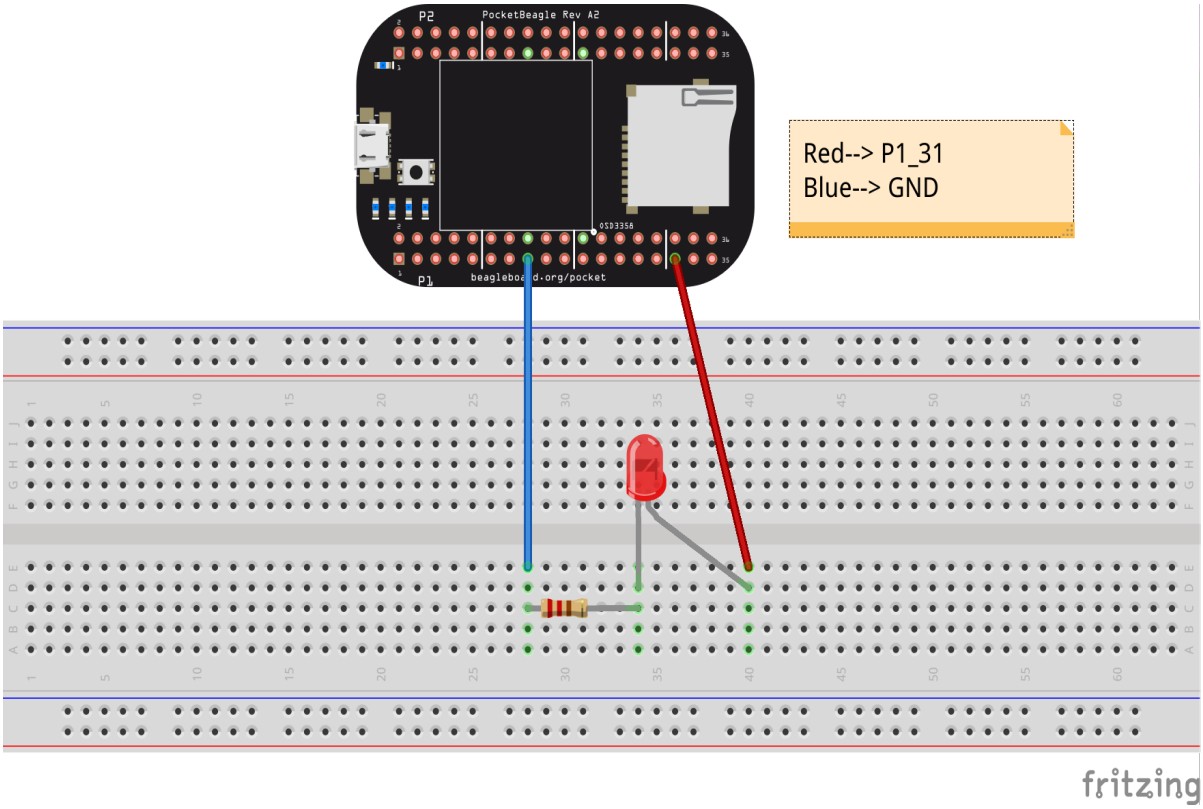
```
while : true {
    if : digital_read(P1_29) {
        digital_write(P1_31, false);
    }
    else {
        digital_write(P1_31, true);
    }
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.7.2 Explanation

This code runs a never ending loop, since it is `while : true`. Inside `while` if header pin P1_29 is HIGH, then header pin P1_31 is set to HIGH, waits for 1000ms, then sets header pin P1_31 to LOW, then again it waits for 1000ms. This loop runs endlessly as long as header pin P1_29 is HIGH, so we get a Blinking output if one connects a LED to output pin.

8.8 LED blink using for loop example



8.8.1 Code

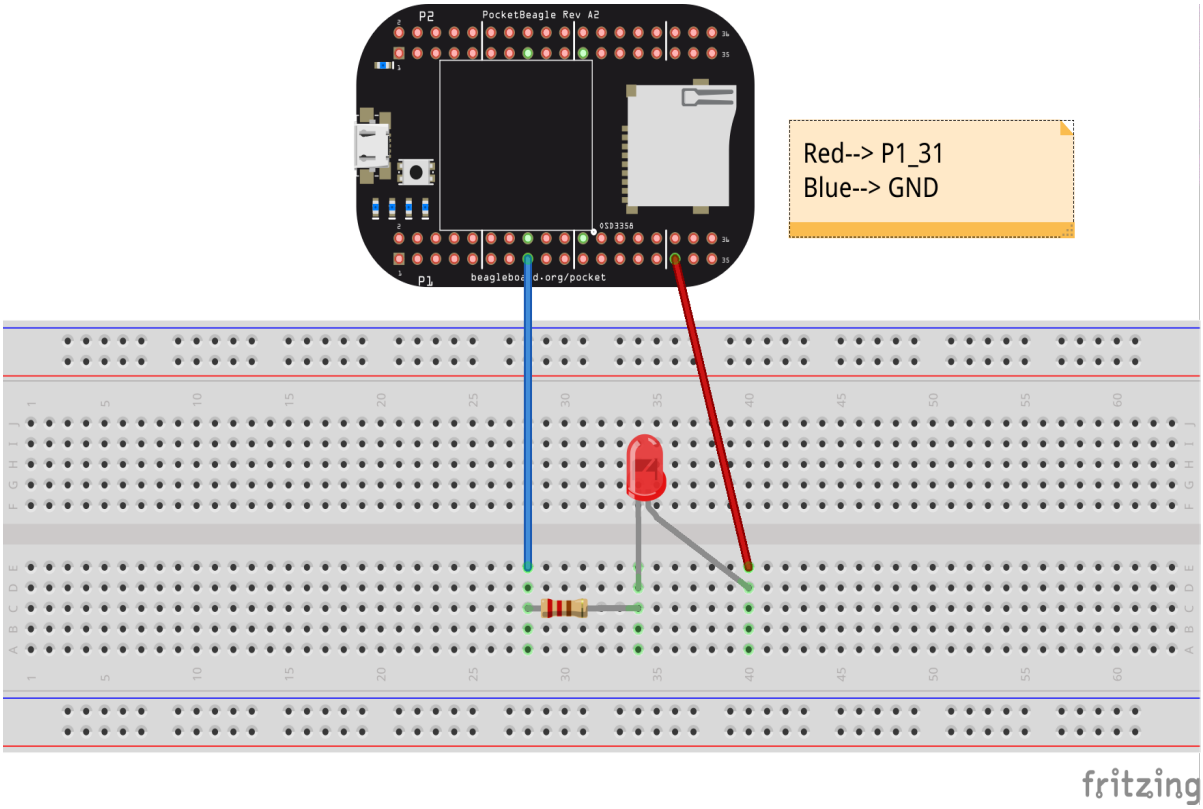
```
for : l in 0:10 {
    digital_write(P1_31, true);
    delay(1000);
    digital_write(P1_31, false);
    delay(1000);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.8.2 Explanation

This code runs for loop with 10 iterations, Inside `for` it sets header pin P1_31 to HIGH, waits for 1000ms, then sets header pin P1_31 to LOW, then again it waits for 1000ms. This loop runs endlessly, so we get a Blinking output if one connects a LED. So LED will blink 10 times with this code.

8.9 LED blink using while loop example



8.9.1 Code

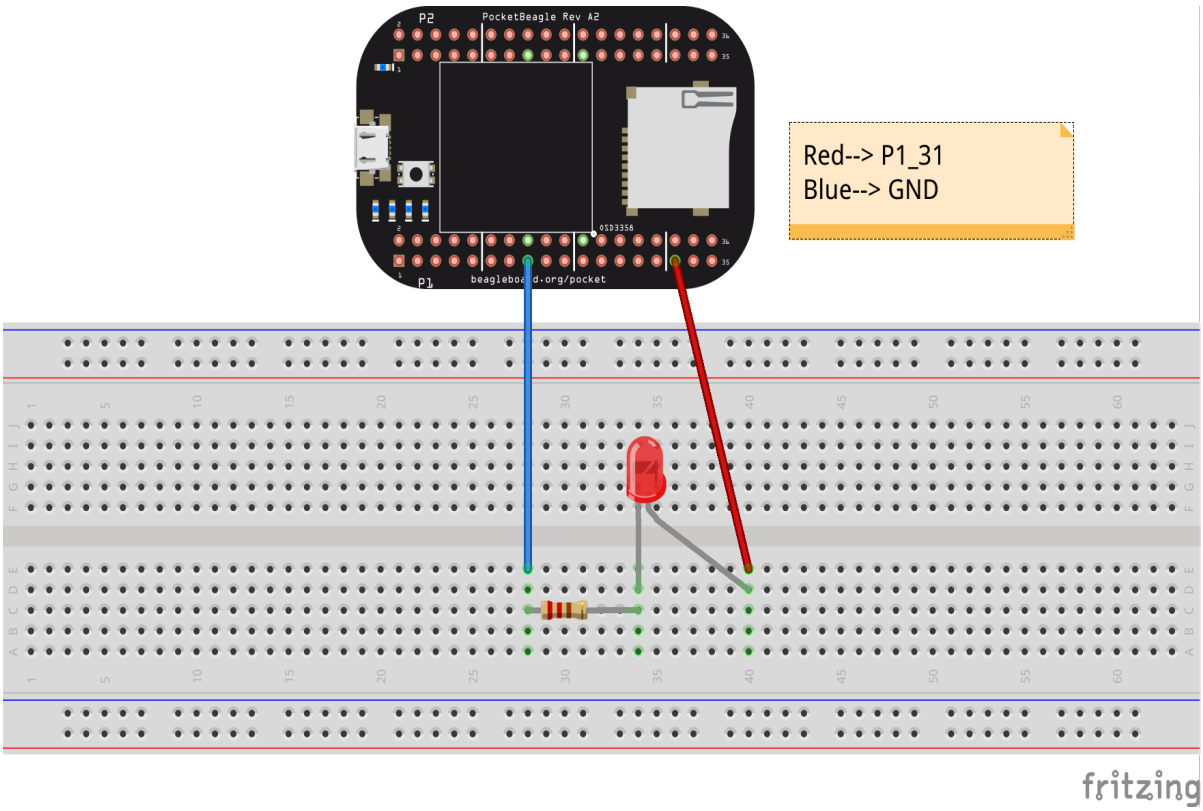
```
while : true {
    digital_write(P1_31, true);
    delay(1000);
    digital_write(P1_31, false);
    delay(1000);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.9.2 Explanation

This code runs a never ending while loop, since it is `while : true`. Inside `while` it sets header pin P1_31 to HIGH, waits for 1000ms, then sets header pin P1_31 to LOW, then again it waits for 1000ms. This loop runs endlessly, so we get a Blinking output if one connects a LED

8.10 LED blink example



8.10.1 Code

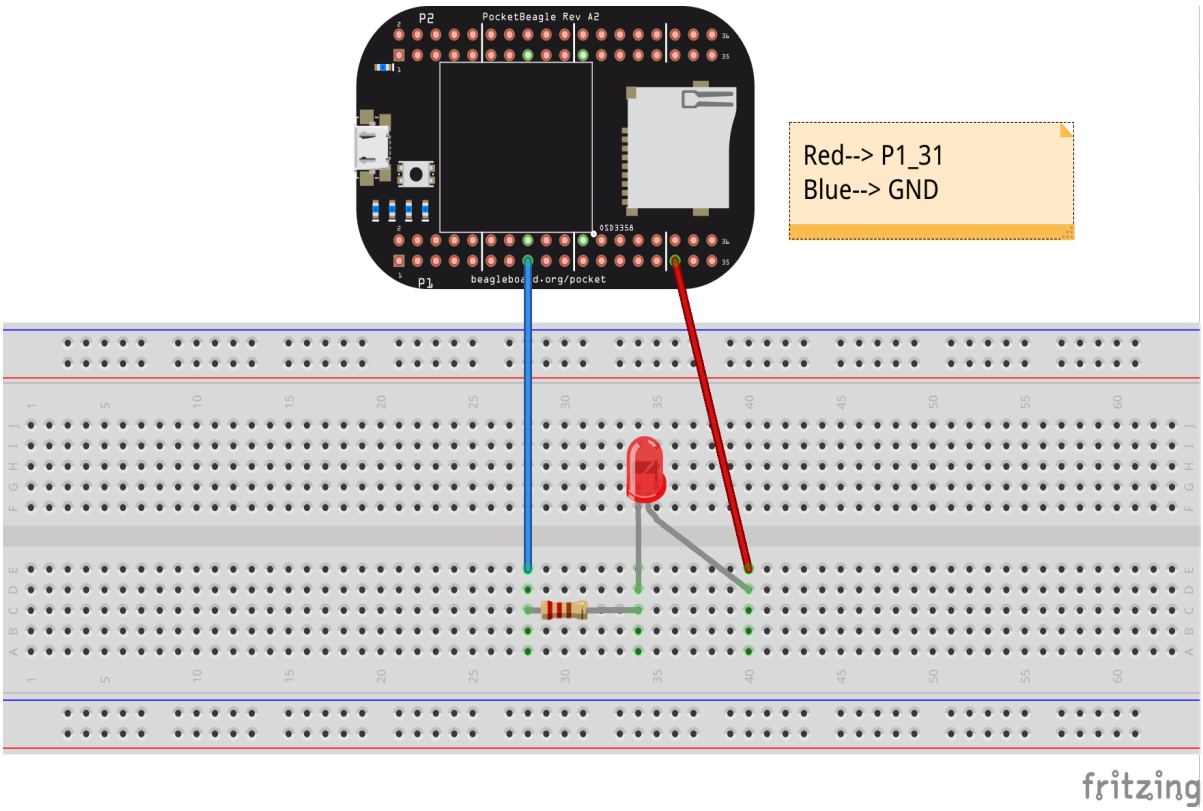
```
while : 1 == 1 {
    digital_write(P1_31, true);
    delay(1000);
    digital_write(P1_31, false);
    delay(1000);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.10.2 Explanation

This code runs a never ending loop, since it is `while : true`. Inside `while` it sets header pin P1_31 to HIGH, waits for 1000ms, then sets header pin P1_31 to LOW, then again it waits for 1000ms. This loop runs endlessly, so we get a Blinking output if one connects a LED

8.11 LED blink using hardware counter



8.11.1 Code

```
while : true {
    start_counter();
    while : read_counter() < 200000000 {
        digital_write(P1_31, true);
    }
    stop_counter();

    start_counter();
    while : read_counter() < 200000000 {
        digital_write(P1_31, false);
    }
    stop_counter();
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.11.2 Explanation

This code runs a never ending while loop, since it is `while : true`. Inside `while` it starts the counter, then in a nested while loop, which runs as long as `read_counter` returns values less than 200000000, so for 200000000 cycles, HIGH is written to header pin P1_31, and after the while loop ends, the counter is stopped.

Similarly counter is started again, which runs as long as `read_counter` returns a value less than 200000000, so for 200000000 cycles, LOW is written to header pin P1_31, and after the while loop ends, the counter is stopped.

This process goes on endlessly as it is inside a never ending while loop. Here, we check if `read_counter` is less than 200000000, as counter takes exactly 1 second to count this much cycles, so basically the LED is turned on for 1 second, and then turned off for 1 second. Thus if a LED is connected to the pin, we get a endlessly blinking LED.

8.12 Read hardware counter example

8.12.1 Code

```
start_counter();
while : read_counter() < 200000000 {
    digital_write(4, true);
}
stop_counter();
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.12.2 Explanation

Since, PRU's hardware counter works at 200 MHz, it counts up to 2 x 108 cycles in 1 second. So, this can be reliably used to count time without using `delay`, as we can find exactly how much time 1 cycle takes.

2 x 108 cycles/second.
1 Cycles = 0.5 x 10-8 seconds.

So, it can be used to count how many cycles have passed since, say we received a high input on pin 3. `start_counter` starts the counter, and `read_counter` reads the current state of the counter, and `stop_counter` stops the counter.

8.13 Using RPMSG to communicate with ARM core

8.13.1 Code

```
init_message_channel();

int count := receive_message();

while : true {
    send_message(count);
    count := count + 1;
    delay(1000);
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.13.2 Explanation

PRU has a functionality to communicate with the ARM core, it is called RPMSG. This examples show how to use RPMSG functionality to communicate with ARM core using RPMSG.

`init_message_channel` is needed to setup communication channel between ARM<->PRU. It only needs to be called once, before using RPMSG functions.

`int count := receive_message();` waits for a message from ARM Core, we need to send some integer to PRU with which to start the counting. So, say we send 3, then int variable count will be equal to 3.

After this, there is `while : true` block which runs endlessly. Inside the block there is a `send_message` call, this sends message back to the ARM Core.

So, inside the for loop we are sending value of count variable, after this we increase value of count by 1. Then we wait for 1000ms, and repeat the above steps again and again.

8.14 Using RPMSG to implement a simple calculator on PRU

8.14.1 Code

```
init_message_channel();

while : true {
    int option := receive_message();
    int a := receive_message();
    int b := receive_message();

    if : option == 1 {
        send_message(a+b);
    }
    elif : option == 2 {
        send_message(a-b);
    }
    elif : option == 3 {
        send_message(a*b);
    }
    elif : option == 4 {
        if : b != 0 {
            send_message(a/b);
        }
        else {
            send_message(a);
        }
    }
    else
    {
        send_message(a+b);
    }
}
```

- Following code works on PocketBeagle, to use on other boards, please change the pins accordingly.

8.14.2 Explanation

`init_message_channel();` starts the message channel for communication with ARM <-> PRU cores. Then `while : true` loops runs endlessly.

`int option := receive_message();` receives which operator to be executed and stores it in option variable. 1 for addition, 2 for subtractions, 3 for multiplication and 4 for division. `int a := receive_message();` receives the value of first operand, and `int b := receive_message();` receives the value of second operand.

if-elseif ladder checks if value of option is 1, 2, 3 or 4 and accordingly sends the value of operation back to ARM core using `send_message`. While division, it makes sure that divisor is not 0. If value of option is anything other than 1, 2, 3, 4, then it defaults to else condition, that is a+b.

This runs endlessly since it is inside a `while : true` loop.