Introduction and Fundamentals

Uartix

First Edition

Jonathan Eldy I. Baldivicio • 2024

Table of Contents

Chapter 1 - Why Uartix?

Despite its unconventional and dynamic behavior, as well as its lack of fundamental APIs, Uartix can still serve as a general-purpose programming and scripting language. The following points outline the reasons behind the development of Uartix:

- The Raspberry Pi Pico, with its dual-core ARM Cortex-M0+ processor, offers an irrelevant yet affordable platform for hardware-level computation.
- Uartix provides a specialized environment where developers can perform mathematical operations directly on the hardware.
- Compared to many external coprocessors, Uartix running on a Raspberry Pi Pico presents a highly cost-effective solution. The affordability of the Pico reduces overall project costs while still delivering substantial computational power for a variety of applications.
- The Raspberry Pi Pico, with its energy-efficient ARM Cortex-M0+ cores, offers a low-power solution for performing mathematical calculations which is perfect for Uartix runtime execution.

1.1 Features

- **Rich expression and statement constructs**

  The language includes standard control flow constructs such as `if`, `while`, and `loop`, as well as more specialized ones like `unless`, `when`, and `random`. These constructs enable developers to write clear and concise code that directly expresses their intent, reducing the need for verbose and boilerplate code.

- **Support for multiple numerical bases**

  One of the standout features of Uartix is its support for multiple numerical bases, including binary, trinary, octal, and hexadecimal, in addition to standard decimal numbers.

- **In-code testing units**

  Uartix introduces an innovative `test` statement that facilitates in-code testing and validation. This feature allows developers to embed tests directly within their codebase, providing a streamlined approach to verifying functionality and ensuring code correctness. The `test` statement is particularly useful for unit testing, where individual components of the code can be tested in isolation.

- **Control Flow as Expressions**

  A distinctive feature of Uartix is its treatment of control flow constructs as expressions rather than traditional statements. This design choice enhances the language's expressiveness and flexibility, allowing control flows to be used as part of larger expressions.

- **Boolean Constructs: `true`, `false`, and `maybe`**

  Uartix introduces unique boolean constructs `true`, `false`, and `maybe`, which add a layer of versatility and unpredictability to the language's logic handling.
  While `true` and `false` are standard boolean values, `maybe` is a distinctive feature that represents an uncertain or probabilistic state, which is resolved at runtime.

1.2 Your First Program

The Hello World program in Uartix demonstrates the basic syntax and functionality of the language, showcasing how to define and call a function that generates a simple greeting message. This example introduces key concepts such as function definition, string concatenation, and function invocation, providing a foundation for more complex programs.

hello.utx

```
# Hello world example

greet = func(name)
    render "Hello, " + name;
greet("world");
```

The line `greet = func(name)` defines a function named greet that takes a single parameter, name. In Uartix, the `func` keyword is used to declare a function. The assignment `greet = func(name)` binds this function to the identifier `greet`, allowing it to be called later in the program. The parameter `name` represents the input that the function will receive when it is called.

Within the function body, which is indicated by the indentation following the function definition, there is a single statement: `render "Hello, " + name;`. This line uses the `render` keyword, which is a command in Uartix to output text to the console or display. The expression `"Hello, " + name` demonstrates string concatenation, where the literal string `"Hello, "` is combined with the value of the `name` parameter. The resulting string is then rendered or displayed as the output of the function.

Finally, the line `greet("world");` calls the `greet` function with the argument `"world"`. When this line is executed, the function `greet` is invoked, and the string `"world"` is passed as the parameter `name`. Inside the function, this parameter is used to create the greeting message. Consequently, the `render` statement outputs the concatenated string `"Hello, world"`, which is the final result of the program.

Chapter 2 - Getting Started

Before installing Uartix, make sure you have JDK 22 (or OpenJDK) installed on your system. Follow the steps below to get started on different operating systems and to build various components from the source.

2.1 Installing Uartix

2.1.1 Linux

1. **Download the `.deb` File**: Go to the release page (github.com/nthnn/Uartix/releases) and download the latest `*.deb` file for Uartix.
2. **Install Uartix**: Open your terminal and navigate to the directory where the `.deb` file is located. Run the following command to install Uartix:

   ```
   sudo dpkg -i uartix_*.deb
   ```

3. **Running Uartix**: After successfully install the `.deb` package, you can now run the command uartix on your terminal.

2.1.2 Windows

1. **Download the `.zip` File**: Go to the release page (github.com/nthnn/Uartix/releases) and download the latest `.zip` file for Windows.
2. **Extract the File**: Extract the contents of the `.zip` file to `C:\uartix`.
3. **Set Environment Path**: Add `C:\uartix\bin` to your Environment Path variables to ensure you can run Uartix from any command prompt.

2.1.3 Firmware Installation

To install the Uartix firmware on your Raspberry Pi Pico, follow these steps:

1. **Enter Flash Mode**: Connect your Raspberry Pi Pico to your system while holding the `BOOTSEL` button to enter flash mode.
2. **Download the UF2 Binary**: Download the UF2 binary of the Uartix firmware from the release page (github.com/nthnn/Uartix/releases).
3. **Install the Firmware**: Drag and drop the downloaded UF2 file into the Raspberry Pi Pico storage that appears on your computer.

2.2 Building from Source

2.2.1 Interpreter

To build the interpreter:

1. **Open in IntelliJ**: Open the Uartix repository in IntelliJ IDEA.
2. **Build Artifacts**: From the menu, go to `Build` menu item and select `Build Artifacts > Build`.

2.2.2 Launcher

On Ubuntu, to build the Uartix launcher, ensure you have Rust and cargo installed on your system. Follow these steps:

1. **Install Dependencies:**

```
sudo apt-get install mingw-w64
rustup target add x86_64-pc-windows-gnu
```

2. **Build the Launcher**: Run the following commands to build the launcher:

```
cargo build –release
cargo build --release --target x86_64-pc-windows-gnu
```

2.2.3 Firmware

To build the Uartix firmware from source, simply follow the steps below.

1. **Installing Raspberry Pi Pico on Arduino IDE**: Install the Raspberry Pi Pico boards on your Arduino IDE by following the steps here: https://randomnerdtutorials.com/programming-raspberry-pi-pico-w-arduino-ide/
2. **Open in Arduino IDE**: Open the file picoware/picoware.ino (https://github.com/nthnn/Uartix/blob/main/picoware/picoware.ino) in your Arduino IDE.
3. **Build & Upload**: Connect your Raspberry Pi Pico board on flash mode then upload and build the Picoware on your Arduino IDE.

2.2.4 Running from CLI

After successfully installing Uartix, you can interact with it via the command-line interface. The following guide outlines the basic usage and options available when running Uartix scripts from the CLI.

```
$ uartix -h
```

```
usage: uartix [-h] [-p {ttyACM0}] [-t]
[files [files ...]]

Execute Uartix script files.

positional arguments:
  files          List of files to execute.

named arguments:
  -h, --help    Show this help message.
  -p {port}, --port {port}
   Serial port device of the co-processor.
  -t, --test    Run test units.
```

2.2.4.1.1 Position Arguments

- **files** — A list of Uartix script files to be executed. This argument can accept multiple file paths, allowing you to run several scripts sequentially.

2.2.4.1.2 Named Arguments

- **help** — Displays the help message, providing an overview of available commands and options. Use this option if you need quick guidance on how to use the CLI.
- **port** — Specifies the serial port device connected to the co-processor. This option is useful if your co-processor is connected to a different serial port or if you have multiple devices connected.

- **test** — Runs test units within the provided script files. This is useful for verifying the functionality of your scripts or modules. By default, this option is set to false, meaning test units are not executed unless explicitly requested.

2.2.4.2 Examples

- To run a Uartix script file named `example.utx`, you can use the following command:

```
$ uartix example.utx
```

- If your co-processor is connected to a different port, specify the port using the `-p` or --port option:

```
$ uartix -p /dev/ttyUSB0 example.utx
```

- To execute test units defined in your script, use the `-t` option:

```
$ uartix -t example.utx
```

Chapter 3 – Grammar Backus-Naur Form Definition

The Uartix programming language features a comprehensive and intricate grammar as described in its Backus-Naur Form (BNF). The grammar delineates the syntax rules for various constructs within the language, ranging from fundamental data types to complex expressions and control structures. Uartix supports multiple numeric bases including binary (`0b`), trinary (`0t`), octadecimal (`0c`), and hexadecimal (`0x`), providing a versatile foundation for numerical operations. These are encapsulated under the DIGIT rule, which allows for a broad range of numeric representations, enhancing the language's flexibility in handling computational tasks.

The `global` rule is a sequence of `statements`, indicating that a Uartix program is essentially a series of statements executed sequentially. Statements in Uartix can be simple control flow directives like `use`, `test`, `break`, `continue`, return (`ret`), and `throw`, each followed by a semicolon. These basic statements enable control over the program's execution flow, allowing developers to implement loops, conditional logic, and error handling effectively.

Expressions form the backbone of Uartix's syntax, includes a wide array of operations and constructs. These include `type_expr` for type annotations, `block_expr` for grouping statements within braces, and `render_expr` for output operations. The `catch_expr` provides a mechanism for exception handling, encapsulating the `catch`, `handle`, and `then` keywords to manage errors gracefully. Control flow is further enriched with constructs like `do_expr` and `while_expr` for loop operations, `if_expr` for conditional branching, `random_expr` for probabilistic decision making, `loop_expr` for traditional for-loops, `unless_expr` for negated conditions, and `when_expr` for pattern matching.

```
binary           := "0b" ("0" | "1")*
trinary          := "0t" ("0" - "2")*
octadecimal      := "0c" ("0" - "7")*
hexadecimal      := "0x" (
           "0" - "9" |
           "a" - "f" |
           "A" - "F")*

DIGIT            :=
           ("0" - "9")*    |
           binary          |
           trinary         |
           octadecimal     |
```

```
           hexadecimal

global     := (statement)*

statement  :=
           use_stmt   |
           test_stmt  |
           break_stmt |
           continue_stmt    |
           ret_stmt         |
           throw_stmt       |
           expr_stmt

use_stmt   :=
      "use" expression
       [expression ("," expression)*]
      ";"

test_stmt  :=
      "test" "(" expression ")"
      expression ";"

break_stmt       := "break" ";"
continue_stmt    := "continue" ";"
ret_stmt         := "ret" expression ";"
throw_stmt       := "throw" expression ";"
expr_stmt        := expression ";"

expression       :=
      type_expr          |
      block_expr         |
      render_expr        |
      catch_expr         |
```

```
      do_expr          |
      while_expr       |
      if_expr          |
      random_expr      |
      loop_expr        |
      unless_expr      |
      when_expr        |
      func_expr        |
      maybe_expr       |
      array_expr       |
      logic_or_expr

type_expr        := "type" expression
block_expr       := "{" (statement)* "}"
render_expr      := "render" expression

catch_expr       :=
      "catch" block_expr
      "handle" <IDENTIFIER> block_expr
      "then" block_expr

do_expr          :=
      "do" expression
      "while" "(" expression ")"

while_expr       :=
      "while" "(" expression ")"
      expression

if_expr          :=
      "if" "(" expression ")" expression
      ["else" expression]
```

```
random_expr      :=
      "random" expression
      ["else" expression]

loop_expr        :=
      "loop" "("
            expression ";"
            expression ";"
            expression
      ")" expression

unless_expr      :=
      "unless" "(" expression ")"
expression
      ["else" expression]

when_expr        :=
      "when" "(" expression ")" "{"
      ["if" "(" expression ")" expression
            (","
                  "if" "(" expression ")"
                  expression)*
      ]
      ["else" expression]
      "}"

maybe_expr       := "maybe"

func_expr        :=
      "func" "("
       [<IDENTIFIER> ("," <IDENTIFIER>)*]
      ")"
      expression
```

```
array_expr             :=
     "["
      [expression ("," expression)*]
     "]"

logic_or_expr          :=
     logic_and_expr
     ["||" logic_and_expr]

logic_and_expr         :=
     bitwise_or_expr
     ["&&" bitwise_or_expr]

bitwise_or_expr        :=
     bitwise_xor_expr
     ["|" bitwise_xor_expr]

bitwise_xor_expr       :=
     bitwise_and_expr
     ["^" bitwise_and_expr]

bitwise_and_expr       :=
     null_coalesce_expr
     ["&" null_coalesce_expr]

null_coalesce_expr     :=
     equality_expr ["?" equality_expr]

equality_expr                :=
     comparison_expr
     [("==" | "!=" | "=")
          comparison_expr]
```

```
comparison_expr  :=
      shift_expr
      [("<" | "<=" | ">" | ">=")
      shift_expr]

shift_expr       :=
      term_expr [("<<" | ">>") term_expr]

term_expr        :=
      factor_expr
      [("+" | "-") factor_expr]

factor_expr      :=
      primary_expr
      [("*" | "/" | "%) primary_expr]

primary_expr     :=
      (("+" | "-" | "~") expression |
      "(" expression ")" |
      <IDENTIFIER> ("[" expression "]")* |
      literal_expr)
      (
            "("
             [expression ("," expression)*]
            ")" |
            "[" expression"]"
      )*

literal_expr     :=
      "true" | "false" | "nil" |
      <STRING> |
      <DIGIT>
```