I. F. F. dos Santos

THE TUIM PROJECT

Maceió
December 30, 2025

I. F. F. DOS SANTOS

The Tuim Project

Maceió
December 30, 2025

CONTENTS

1 INTRODUCTION

The Tuim Project was created by the Brazilian physicist Ismael and aims to increase code portability at source level and at binary level. The project comes with three levels of portability:

1. Development time – Different platforms have different Application Programing Interfaces (APIs) available for developers, making the development unique for each one since the implementation shall use what is available. Tuim handles this problem by using cross platform libraries for any application, in the core of these libraries are the libraries that abstracts the kernel, designed to have low overhead and to be fast.
2. Build time – The instruction that shall be followed to build an application are usually very distincts when developing for different platforms. However, the approach followed by Tuim minimizes that differences, decreasing the learning curve for cross-platform projects. This is allowed by a standadization of the development environment, with well defined command line tools and behavior.
3. Runtime – Finally, different platforms define different Application Binary Interfaces (ABIs) and therefore even if an application uses only cross-platform libraries and build system it needs to be recompiled to be ported to another platform. Tuim solves this problem by defining a cross-platform ABI and providing a loader capable to link dynamicaly and execute conforming applications.

1.1 PORTABILITY CHALLENGES

…

1.1.1 THE “HELLO, WORLD!” PROGRAM

…

```c
int puts(const char*);

int main(int argc, char **argv){
   int rc = puts(argc < 2 ? "Hello, World!" : argv[1]);
   if(rc < 0) return 1;
   return 0;
}
```

…

```
as libc-<triple>.asm -o libc.o
cc hello.c -o hello.o
```

…

```
ld -e _start libc.o hello.o -o hello
hello
```

…

```
make
./build/bin/hello
```

…

1.1.2 THE “SAY HELLO” PROGRAM

TODO: This is the right place to present the usage of portable APIs. The idea is to write a program that have they graphical window with the message `Click to say hello` and a button that sounds “Hello, World!” when clicked. For now I have no idea of what API I need to use.

```
make
./build/bin/say-hello
```

…

1.1.3 THE TUIM’S ELF INTERPRETER

…

```
make
tuim ./build/bin/say-hello
```

…

```
tuim hello-<arch>.elf
```

…

```
tuim say-hello-<arch>.elf
```

…

1.2 REFERENCE IMPLEMENTATION

…

1.2.1 GETTING PRE-BUILT BINARIES

…

1.2.2 BUILDING FROM SOURCE

…

Part I

Development Environment

2 IMPLEMENTATION

On The Tuim Project the developement environment defined here is implemented as scritps to wrap around LLVM. A typical installation may be done by copying the `src` directory from the project source code to one place easy to remember. For example, to install it at `~/dev`:

```
git clone https://github.com/ituim/the-tuim-project.git
cd the-tuim-project
mkdir ~/dev
cp -R dev/* ~/dev
```

In order to set up the environment for development the `start.sh` script shall be executed:

```
sh ~/dev/start.sh
```

The `start.sh` script set up environment variables, then `LIBRARY_PATH` defaults to `~/dev/lib`, `LD` defaults to `~/dev/bin/ld` and so on.

2.1 CROSS COMPILATION

When invoked as:

```
sh ~/dev/start.sh <triple>
```

the environment variable `LIBRARY_PATH` defaults to `~/dev/lib/<triple>` and `clang` uses the file `~/dev/share/<triple>.cfg` to get the correct flags for the target architecture.

Part II

System Development Kit

3 COMPILER RUNTIME LIBRARY

TODO: At the moment I can’t standardize the library calls generated by the compiler. The idea is to provide something similar to ARM EABI.

4 NATIVE KERNEL INTERFACE

The native kernel interface sevice the application with a interface to the kernel. The interface vary across distincts operating systems.

4.1 API

...

```
#include <syscall.h>
```

4.2 REFERENCE IMPLEMENTATION

5 TUIM'S KERNEL LIBRARY

The Tuim's kernel library is a library designed to abstract the operating system's kernel interface.

5.1 OBJECT MODEL

...

```
#include <tuim-object.h>
struct Class /* implementation defined */ ;
```

...

```
typedef /* implementation defined */ __tuim_obj_t;
```

5.2 IO

```
#include <tuim-io.h>

void program(void){
   const char str[] = "Hello, World!";
   write(stdout, str, sizeof(str));
}
```

5.3 REFERENCE IMPLEMENTATION

6 STANDARD LIBRARIES

Part III

Application Binary Interface

BIBLIOGRAPHY

APPENDIX A – MANUAL PAGES

A.1 ENVIRONMENT VARIABLES

At start of

A.2 LINKER

Synopsis

```
ld [options...] -r objects... -o output
ld [options...] -s objects... -o output
ld [options...] -e symbol objects... -o output
```

Description

The `ld` utility is a linker for Tuim's Development Environment. It combines one or more object files into a single object.

Outside a shell section programs shall use the value of the environment variable `LD` to get the linker path and the environment variable `LDFLAGS` shall be used to get user defined settings.

Options

- `-l name`
  Search for library `libname.a` in the library search paths and use it to satisfy undefined references.
- `-L dir`
  Add directory `dir` to the library search path for locating archive libraries specified with `-l`. Directories given with `-L` are searched before the one specified with `LIBRARY_PATH`.
- `-T script`
  Use linker script `script`. When provided, the script fully governs layout rules used by the linker.
- `-r`
  Write a object file suitable for linking by `ld`.
- `-s`
  Write a library suitable for dynamic linking at runtime.
- `-e symbol`
  Write a executable file with entry point `symbol`.
- `-o output`
  Write the linked output to `output`.
- `-g`
  Generate debugging information in the produced object file.

Environment Variables

- `LIBRARY_PATH`
  The path that the linker will search for library files. That directory is searched after those specified with `-L`.

A.3 ASSEMBLER

Synopsis

```
as [options...] input.s -o output.o
```

Description

The `as` utility is an assembler for Tuim's Development Environment. It translates assembly language source files into object files suitable for linking by `ld`. The assembler accepts GNU style assembly syntax.

Outside a shell section programs shall use the value of the environment variable `ASM` to get the assembler path and the environment variable `ASMFLAGS` shall be used to get user defined settings.

Options

- `-o output.o`
  Write the assembled output to `output.o`.
- `-g`
  Generate debugging information in the produced object file.

A.4 C COMPILER

Synopsis

```
cc [options...] input.c -o output.o
```

Description

The `c` utility is an C compiler for Tuim's Development Environment. It translates C source files into object files suitable for linking by `ld`. The compiler accepts the C23 language standard.

Outside a shell section programs shall use the value of the environment variable `CC` to get the compiler path and the environment variable `CCFLAGS` shall be used to get user defined settings.

Options

- `-D name[=value]`
  Define the macro `name` (with optional `value`) for the preprocessor. This is equivalent to inserting `#define name value` at the start of each input file and affects conditional compilation and macro substitutions.
- `-U name`
  Undefine the macro `name` for the preprocessor (as if `#undef name` were applied).
- `-I dir`
  Add `dir` to the header search path for locating header files. Directories given with `-I` are searched before the one specified with `C_INCLUDE_PATH`.
- `-E`
  Run only the C preprocessor stage and output the preprocessed source to standard output. No compilation is performed.
- `-o output.o`
  Write the compiled output to `output.o`.
- `-g`
  Generate debugging information in the produced object file.

Environment Variables

- `C_INCLUDE_PATH`
  The path that the preprocessor will search for header files. That directory is searched after those specified with `-I`. That directory shall contain at least the non-deprecated free-standing headers and the following headers: `<tuim/arch.h>`, `<tuim/kernel.h>`.

Synopsis

```
#include <tuim/arch.h>
```

Description

Macros to describe processor Instruction Set Architecture (ISA).

The `<tuim/arch.h>` header shall define the following macros:

Macro	Value	ISA
`tuim_unknown`	`0x0000`	unknown
`tuim_riscv32`	`0x0001`	32-bit RISC-V
`tuim_riscv64`	`0x0002`	64-bit RISC-V
`tuim_riscv128`	`0x0003`	128-bit RISC-V
`tuim_arm`	`0x0004`	ARM
`tuim_aarch64`	`0x0005`	ARM AArch64
`tuim_i386`	`0x0007`	x86 IA-32
`tuim_amd64`	`0x0008`	x86-64
`tuim_arch`	Someone above	target ISA

Synopsis

```c
#include <tuim/kernel.h>
```

Description

Macros useful to describe the operating system kernel.

The `<tuim/kernel.h>` header shall define the following macros:

Macro	Value	Operating system kernel
`tuim_unknown`	`0x0000`	unknown
`tuim_linux`	`0x0010`	Linux Kernel
`tuim_xnu`	`0x0020`	XNU
`tuim_windows_nt`	`0x0030`	Windows NT
`tuim_kernel`	Someone above	target kernel

A.5 MAKE SCRIPT WRAPPER

Synopsis

```
make [macro=value...] [command]
```

Description

The `make` utility is a wrapper to execute Make Scripts. Make Scripts are the standard way to handle complex tasks at build time, a Make Script is a shell script `Make.sh` that shall behave like the `make` utility when called as `sh Make.sh`.

The following table shows the macros that shall be supported by Make Scripts:

Macro	Default value	Meaning
`SOURCE_DIR`	`src`	source code path
`INCLUDE_DIR`	`include`	interface search path
`BUILD_DIR`	`build`	path for out of source build
`BUILD_TYPE`	`release`	`debug` or `release` build type
`STATIC_LIBRARY_PREFIX`	unspecified	Prefix for static libraries
`SHARED_LIBRARY_PREFIX`	unspecified	Prefix for shared libraries
`STATIC_LIBRARY_SUFFIX`	unspecified	Suffix for static libraries
`SHARED_LIBRARY_SUFFIX`	unspecified	Suffix for shared libraries
`EXECUTABLE_SUFFIX`	unspecified	Suffix for executables
`DESTDIR`	unspecified	
`PREFIX`	unspecified	
`BINDIR`	unspecified	
`DATADIR`	unspecified	

If a macro is specified more than once then the last definition shall be used, this allows the `make` utility to set defaults for unspecified values.

The following table shows the commands that shall be supported by Make Scripts:

Command	Behavior
`pack`	Package the application/library
`install`	Install the application/library
`clean`	Remove the build directory

If no command is used the default behavior is to build the application/library/document/… from source code.

A.6 PROGRAM INTERPRETER

Synopsis

```
tuim [-b <backend>] <command|file>
```

Description