

MIT Proto Developers Guide

By the authors of MIT Proto

Last Updated: October 2, 2010

This is a guide to developing for the MIT Proto project. It outlines the various components of the project, how they fit together, and how to develop for them safely. For user documentation, begin with the **Proto Quick Start**. For installation instructions, see the **README**.

WARNING: This file is still highly incomplete.

Contents

1 Credits for Proto	1
2 Overview	2
3 Installed File Locations	3
4 Compiler	3
5 Libraries	4
6 ProtoKernel	4
7 Simulator	4
7.1 Distributions	5
7.2 Time Models	5
7.3 Layers	5
7.4 Adding Simulated Hardware	6
8 Adding Simulator Functionality	6
8.1 Layer Opcode Extensions	8
9 Regression Testing	8

1 Credits for Proto

The Proto language was created in partnership by Jonathan Bachrach and Jacob Beal. As they created the language, Jonathan created the first implementation of MIT Proto, including the first compiler, kernel, simulator, and embedded device implementations. Since that time, Jake and other contributors have built on the work begun by Jonathan.

MIT Proto also includes contributions from (alphabetically):

Anna Derbakova, Takeshi Fujiwara, Tony Grue, Tom Hsu, Joshua Horowitz, Kanak Kshetri, Dustin Mitchell, Omar Mysore, Maciej Pacula, Hayes Raffle, Dany Qumsiyeh, Omari Stephens, Mark Tobenkin, Dan Vickery

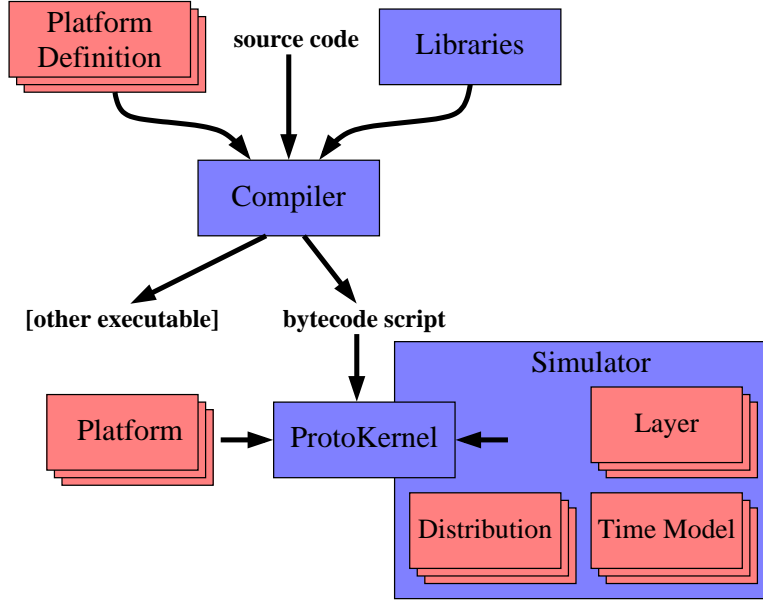


Figure 1: Relationship of MIT Proto components (blue) and extensions (red)

The Protobo platform code in `platforms/protobo/` also includes Topobo-related code from (alphabetically):

Mike Fleder, Limor Fried, Josh Lifton, Laura Yip

2 Overview

Proto is a language designed for implementing distributed systems on spatially-structured networks of static or mobile devices. The programmer describes a desired behavior for the continuous space occupied by the network, which can be translated into a program by which individual devices cooperate to produce an approximation of the desired behavior. Such programs can then be rendered into an executable that can run equivalently on different hardware platforms.

Components MIT Proto is an implementation of Proto which supplies four main components:

- A compiler which turns Proto code into executable code for individual devices
- A library of Proto code implementing commonly used functions
- An implementation of the ProtoKernel virtual machine, one means of executing programs on individual devices.
- An event-based simulator, based on ProtoKernel, for debugging and visualization of Proto programs executing on a network

MIT Proto is typically invoked through one of two executables, `p2b` and `proto`. The name `p2b` stands for “Proto to bytecode” and is a standalone invocation of the compiler. The `proto` executable, on the other hand, first compiles and then executes in the simulator.

Extensions MIT Proto has interfaces allowing this system to be extended in three ways:

- The compiler can emit code customized to different platforms
- ProtoKernel can incorporate a set of platform-specific opcodes
- The simulator supports plug-in:
 - Distributions for laying out the devices of a network (e.g. random, grid, torus).
 - Models for the evolution of time on individual devices (e.g. synchronous, drifting clocks)
 - Physics “layers” for simulating the interaction of sensors and actuators with aspects of a dynamic external environment (e.g. wireless communication, chemical diffusion, Newtonian kinetics, LEDs and beeps)

MIT Proto’s is copylefted such that any modification of its components is copylefted as well, but extensions need not be.

3 Installed File Locations

MIT Proto uses autotools to build and install on a machine. By default, the installation directory prefix is `/usr/local/` (or the OS-appropriate equivalent). From this base, the following default locations are used (reconfigurable via `./configure` when the project is being built):

- Executables are placed in `PREFIX/bin/`
- The library files in the build directories `lib/` and `/lib/core/` are both placed into `PREFIX/share/proto/lib`. This directory is part of the default search path for executions of the compiler.
- Opcode definition files for platform `PLATFORM` are expected to be found in `PREFIX/share/proto/platforms/PLATFORM`
- The header files needed for building extensions for MIT Proto are installed in `PREFIX/include/proto/`.
- The simulator searches for plug-ins in `PREFIX/lib/proto/`

4 Compiler

This document describes only the compiler’s interactions with other pieces of the installation and its major known bugs. The internal structure of the compiler will not be documented in detail at this time, because it is both highly complex and obsolete.

Compiling for a Platform The compiler loads platform-specific operations during its initialization. When given a `--platform PLATFORM` argument, it reads the opcode definition file `platform_ops.proto` from the platform directory for `PLATFORM` (see above). If given no `--platform` argument, it assumes the platform is `sim`, for the Proto simulator.

This is a Proto file and is expected to contain a set of `defop` expressions of the form:

`(defop ? function-name type ...)`

where *function-name* is the Proto function that will invoke the opcode. The first *type* expression specifies the return type, and any others that follow specify argument types for the operator. The *type* expressions are currently allowed only to be one of `scalar`, `boolean`, and `(vector 3)`.

In addition, for each `-L OPFILE` argument, the compiler loads the opcode definition file `OPFILE` (appending a `.proto` extension if needed) from the same directory.

Compilation and Files Compilation always starts with a Proto expression on the command line. Except for the smallest of programs, this expression is not self-contained. Whenever the compiler encounters a function *name* that it does not know, it searches for a file called *name.proto* in the directories listed in its path. By default, the path consists of the current directory and the installed library location. These two command line arguments:

`--path PATH`

`--basepath PATH`

add to the default path and override the default path, respectively.

Known Bugs Important known bugs in the current compiler are:

- Variables in a `letfed` that are not used within the body of the `letfed` are not updated correctly.
- Variables in `let` statements that are used only within a conditional further on in the code are inlined inside that conditional, which makes side effects of the computation in the `let` happen only in the domain of that branch of the conditional.

Neocompiler A complete rebuild of the compiler is in progress, but not currently checked into the repository. When complete, it will implement an enhanced version of Proto, fix a number of known compiler bugs, change how platform-specific opcodes are handled, and allow compilation for execution on systems other than ProtoKernel. **NEEDS DEVELOPER ATTENTION**

5 Libraries

MIT Proto includes two libraries: the functions in `lib/` are moderately complex programs that make useful building blocks, while the functions in `lib/core/` are language primitives implemented in Proto. They are mixed together in installation, but kept separate in the build so that maintenance of the “library” collection is not cluttered up by the large numbers of simple core functions like `log10` and `logn`. The core library exists because it helps minimize the number of opcodes needed without hardwiring definitions into the compiler.

6 ProtoKernel

SECTION IN PROGRESS

Built purely in C (no C++) for minimum size and maximum portability across embedded platforms.

7 Simulator

SECTION IN PROGRESS

Event-driven simulation

```
class EventConsumer {
public:
    virtual BOOL handle_key(KeyEvent* key) {return FALSE;} // return if consumed
    virtual BOOL handle_mouse(MouseEvent* mouse) {return FALSE;} // same return
    virtual void visualize() {} // draw, assuming a prepared OpenGL context
    // evolve moves state forward in time to 'limit' (an absolute)
    virtual BOOL evolve(SECONDS limit) {} // return whether state changed
};
```

Time real time vs. simulation time vs. device time

Visualization

7.1 Distributions

```
class Distribution {
public:
    int n; Rect *volume;
    METERS width, height, depth; // bounding box of volume occupied
    Distribution(int n, Rect *volume) { // subclasses often take an Args* too
        this->n=n; this->volume=volume;
        width = volume->r-volume->l; height = volume->t-volume->b; depth=0;
        if(volume->dimensions()==3) depth=((Rect3*)volume)->c-((Rect3*)volume)->f;
    }
    // puts location in *loc and returns whether a device should be made
    virtual BOOL next_location(METERS *loc) { return FALSE; } // loc is a 3-vec
};
```

7.2 Time Models

```
// a bit of state attached to a device to say how its time advances
class DeviceTimer {
public: // both of these report delay from the current compute time
    virtual void next_transmit(SECONDS* d_true, SECONDS* d_internal)=0;
    virtual void next_compute(SECONDS* d_true, SECONDS* d_internal)=0;
    virtual DeviceTimer* clone_device()=0; // split the timer for a clone dev
};

// factory class for producing DeviceTimers
class TimeModel {
public:
    virtual DeviceTimer* next_timer(SECONDS* start_lag)=0;
    virtual SECONDS cycle_time()=0;
};
```

7.3 Layers

```
class Layer : public EventConsumer {
public:
    int id; // what number layer this is, for lookup during callbacks
    BOOL can_dump; // -ND[layer] is expected to turn off dumping for a layer
    SpatialComputer* parent;
    Layer(SpatialComputer* p);
    virtual ~Layer() {} // make sure that destruction is passed to subclasses
    virtual BOOL handle_key(KeyEvent* key) {return FALSE;}
    virtual void visualize() {}
    virtual BOOL evolve(SECONDS dt) { return FALSE; }
    virtual void add_device(Device* d)=0; // may add a DeviceLayer to Device
    virtual void device_moved(Device* d) {} // adjust for device motion
    // removal, updates handled through DeviceLayer
    virtual void dump_header(FILE* out) {} // field names in ""s for a data file
};
```

```

    virtual layer_type get_type() { return LAYER_OTHER; }
};

// this is the device-specific instantiation of a layer
class DeviceLayer : public EventConsumer {
public:
    Device* container;
    DeviceLayer(Device* container) { this->container=container; }
    virtual ~DeviceLayer() {}; // make sure destruction cascades correctly
    virtual void preupdate() {} // to called before computation
    virtual void update() {} // to called after a computation
    virtual void visualize() {} // to be called at visualization
    virtual BOOL handle_key(KeyEvent* event) { return FALSE; }
    virtual void copy_state(DeviceLayer* src)=0; // to be called during cloning
    virtual void dump_state(FILE* out, int verbosity) {}; // print state to file
};

// The Body/BodyDynamics is a layer that is stored and managed
// specially because it tracks the position of the device in space.
class Body : public DeviceLayer {
public:
    BOOL moved;
    Body(Device* container) : DeviceLayer(container) {}
    virtual ~Body() {}; // make sure destruction cascades correctly
    // on delete, a body should remove itself from the BodyDynamics
    virtual const flo* position()=0; // returns a 3-space coordinate
    virtual const flo* velocity()=0; // returns a 3-space vector
    virtual const flo* orientation()=0; // returns a quaternion
    virtual const flo* ang_velocity()=0; // returns a 3-space vector
    virtual void set_position(flo x, flo y, flo z)=0;
    virtual void set_velocity(flo dx, flo dy, flo dz)=0;
    virtual void set_orientation(const flo *q)=0;
    virtual void set_ang_velocity(flo dx, flo dy, flo dz)=0;
    virtual flo display_radius()=0; // bigger bodies get bigger displays
    virtual void render_selection()=0; // render for selection
    void copy_state(DeviceLayer* src) {} // required virtual is moot
};

// BodyDynamics is a special type of layer, implementing the base physics
class BodyDynamics : public Layer {
public:
    BodyDynamics(SpatialComputer* p) : Layer(p) {}
    virtual ~BodyDynamics() {} // make sure destruction is passed to subclasses
    virtual Body* new_body(Device* d, flo x, flo y, flo z)=0;
    void add_device(Device* d) {} // required virtual, replaced by new_body
};

```

7.4 Adding Simulated Hardware

8 Adding Simulator Functionality

As previously noted, the simulator can be extended with new distributions for laying out device positions, models for the evolution of time on individual devices, and physics “layers” for simulating interaction with

an external environment.

Extensions are dynamically loaded from libraries in the plugin directory. Also living in that directory is a file, **registry.txt**, that contains the current known inventory of plugins. This file is a case-sensitive list of mappings in a simple key/value format:

- Each line has format **TYPE NAME = FILESTEM**. Whitespace (or its absence) will be ignored.
- Lines beginning with **#** are comments

This file is used to determine what dynamic library should be loaded to retrieve a desired plugin. The **TYPE** is one of **Layer**, **TimeModel**, or **Distribution**, and the **NAME** and **FILESTEM** are arbitrary alphanumeric tokens. When the spatial computer is being constructed, a component is looked up first by type, then by name (components of different type can have the same name without conflict), to find what library should be loaded. A system-appropriate extension is then added to the **FILESTEM** (e.g. **.dylib**, **.so**, or **.dll**) and the library loaded dynamically.

A library must contain two symbols, which will be invoked as C functions:

- `char* get_proto_plugin_inventory()`
- `ProtoPluginLibrary* get_proto_plugin_library()`

The `get_proto_plugin_inventory` function returns a C string containing its portion of the registry file.

The `get_proto_plugin_library` function returns an accessor object for the dynamically loaded library. This must be subclassed from the type `ProtoPluginLibrary`, which is defined:

```
class ProtoPluginLibrary {
public:
    // Used to get simulator Layers, Distributions, and TimeModels
    virtual void* get_sim_plugin(string type, string name, Args* args,
                                SpatialComputer* cpu, int n) { return NULL; }

    // Used to get compiler extensions
    virtual void* get_compiler_plugin(string type, string name, Args* args)
    {return NULL; }
};
```

Each of the functions in this interface is used to poll whether a desired extension object can be obtained from this plugin. The function should return the requested object if it can supply it, and **NULL** if not. The argument **type** is its category (e.g. “Layer”), **name** is an identifier the desired object, **args** is the current command line arguments, **cpu** is the model under construction, and **n** is the number of devices to be created. The function may consume command-line arguments, but must not modify the model directly.

The plugin manager also exposes its own plugin retrieval function, so that “meta-plugins” can combine together the functions of other objects. This consists of `get_sim_plugin` and `get_compiler_plugin` functions formatted identically to those of the `ProtoPluginLibrary`.

For example, a multi-radio communication model could use the `get_sim_plugin` function to fetch a layer to model each radio on the device.

Layers, time models, and distributions are invoked by the command line arguments

```
-L layer
-TM time-model
-DD distribution
```

During the construction of the spatial computer model, these command line arguments are used to find what object is desired. The simulator’s plugin manager reads the registry to determine what library the object should be found in, links in the library if it is not already, then uses the `get_sim_plugin` to retrieve the object.

There is currently no facility to handle plugins built with conflicting names (though objects of different type are each in their own namespace).

Note: GNU compilers have severe limitation with respect to linking libraries. In particular, the compiler requires that there must 'not' be any circular dependency between libraries or there must be a hierarchical dependency between any two libraries. Hence utmost care must be taken to 'not' introduce any circular dependencies between libraries and beware of the order in which libraries are specified in makefiles.

8.1 Layer Opcode Extensions

Simulator layers can also extend the simulator functionality by adding new opcodes to the kernel and new functions to the language. This is accomplished by means of the function:

```
SimulatedHardware::registerOpcode(OpHandlerBase* opHandler);
```

where the argument passed is an instance of the `OpHandler` template class:

```
template <class Tclass> class OpHandler : public OpHandlerBase {
private:
    void (Tclass::*fnp)(MACHINE* machine);
    Tclass* pThis;
public:
    OpHandler(Tclass* _pThis, void (Tclass::*_fnp)(MACHINE* machine), const char* _defop) {
        fnp = _fnp;
        pThis = _pThis;
        defop = _defop;
    }
    virtual void operator()(MACHINE* machine) {
        (*pThis.*fnp)(machine);
    }
};
```

Basically, an `OpHandler` is created specialized for a particular layer, and maps a declaration of the new Proto functions name and type to a C++ function invoked on that layer instance.

For example, the `SimpleDynamics` layer adds a new opcode for measuring the radius of a device by invoking

```
parent->hardware.registerOpcode(new OpHandler<SimpleDynamics>(this, &SimpleDynamics::radius_get_op,
```

where `parent` is the enclosing `SpatialComputer` and `radius_get_op` is its implementation of the opcode.

9 Regression Testing

MIT Proto comes with a built in regression test suite, located in `src/tests/`. The file `prototest.py` is a Python script for executing tests, and files ending in `.test` are particular test suites.

A test suite is invoked by running:

```
./prototest.py suite.test
```

Multiple test suites can be given; for example,

```
./prototest.py *.test
```

will execute every test in the directory.

The regression tester works by executing either `proto` or `p2b`, capturing its output as a file (by default stored in `src/tests/dumps/`), then comparing the contents of this file against values specified for the test.

The command to be executed for a test is specified as:

```
test: $(app) arguments
```


where *app* is either `PROTO` or `P2B`. The arguments `-D` and `-dump-stem` are supplied automatically if the test does not supply them explicitly, and `-headless` is added for tests invoking the simulator.

Following the executable specification is a series of lines specifying comparisons to perform on the output of the execution. Each line is formatted:

test line column expected-value [additional-args]

The *line* and *column* values specify which output token is to be compared, counting from zero, and splitting each line into “columns” on whitespace. The *column* can also have the special value `_`, meaning that the comparison should be performed against the entire line (trimming leading and following whitespace, but leaving the interior intact).

The *test* specifies how the *expected-value* is to be compared to the actual value found at *line* and *column*:

- `=`, `>`, `<`, `>=`, `<=`, and `!=` treat the value as a scalar number and perform the appropriate numerical comparison. Note that comparison with infinity can be done by giving an expected value of `Inf` or `-Inf`.
- `~=` treats the value as a number and tests for approximate equality, using an additional argument *tolerance* for how different the value can be from *expected-value*.
- `is_nan` checks whether the value is equal to the special floating point “not a number” value (needed because, by IEEE standards, NaN is not equal to itself). An *expected-value* must be supplied, but is ignored.
- `is` treats the value as a string and performs a case-insensitive comparison with *expected-value* (e.g. “OUTPUT”, “Output”, and “output” all match).

A test succeeds if the executable terminates normally (exit code 0) and if all comparisons return true.

For each file *suite.test*, a summary of test results is printed on standard out and full detail is output into a file *suite.test.RESULTS*.

Other notes:

- Lines beginning with `//` are treated as comments.
- Default paths and other options in `prototest.py` can be changed using various command line arguments. These options can be listed by running `./prototest.py -h`