Swift/T Guide

Data Science and Learning

Swift/T Guide

About Argonne National Laboratory

Argonne is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357. The Laboratory's main facility is outside Chicago, at 9700 South Cass Avenue, Argonne, Illinois 60439. For information about Argonne and its pioneering science and technology programs, see www.anl.gov.

DOCUMENT AVAILABILITY

Online Access: U.S. Department of Energy (DOE) reports produced after 1991 and a growing number of pre-1991 documents are available free at OSTI.GOV (<http://www.osti.gov/>), a service of the U.S. Dept. of Energy's Office of Scientific and Technical Information

Reports not in digital format may be purchased by the public from the National Technical Information Service (NTIS):

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312
www.ntis.gov
Phone: (800) 553-NTIS (6847) or (703)
605-6000 Fax: (703) 605-6900
Email: **orders@ntis.gov**

Reports not in digital format are available to DOE and DOE contractors from the Office of Scientific and Technical Information (OSTI):

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
www.osti.gov
Phone: (865) 576-8401
Fax: (865) 576-5728
Email: **reports@osti.gov**

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor UChicago Argonne, LLC, nor any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of document authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof, Argonne National Laboratory, or UChicago Argonne, LLC.

Swift/T Guide

prepared by
Justin M. Wozniak

Data Science and Learning, Argonne National Laboratory

April 1, 2019

Swift/T Guide

May 2018

Table of Contents

- [1. Support](#)
- [2. Installation](#)
 - [2.1. For Debian-based systems \(including Ubuntu\)](#)
 - [2.2. From Spack](#)
 - [2.3. From source](#)
- [3. Usage](#)
- [4. Program structure](#)
- [5. Comments](#)
- [6. Modules](#)
- [7. Variable declarations](#)
- [8. Dataflow evaluation](#)
- [9. Composite functions](#)
- [10. Types](#)
 - [10.1. Primitive types](#)
 - [10.2. Files](#)
 - [10.3. Blobs](#)
 - [10.4. Arrays](#)
 - [10.5. Nested arrays](#)
 - [10.6. Structs](#)
 - [10.7. Defining new types](#)
 - [10.8. Global variables](#)
- [11. Control structures](#)
 - [11.1. Conditionals](#)
 - [11.2. Iteration](#)
 - [11.3. Explicit execution ordering](#)
 - [11.4. Scoping blocks](#)
- [12. Operators](#)
 - [12.1. Numeric operators](#)
 - [12.2. Boolean operators](#)
 - [12.3. String operators](#)
- [13. Standard library](#)
 - [13.1. General](#)
 - [13.2. Type conversion](#)
 - [13.3. Output](#)
 - [13.4. String functions](#)
 - [13.5. Math](#)
 - [13.6. System](#)
 - [13.7. Files](#)
 - [13.8. Blobs](#)
 - [13.9. Location](#)
 - [13.10. Unix tools](#)
- [14. Defining leaf functions](#)
 - [14.1. Swift extension functions](#)
 - [14.2. Dispatch and work types](#)
 - [14.3. App functions](#)
 - [14.4. Remote job execution with Coasters](#)
 - [14.5. Custom App Executors \(Advanced\)](#)
 - [14.6. External scripting support](#)
 - [14.7. Parallel tasks: Libraries](#)
 - [14.8. Parallel tasks: External programs](#)
- [15. More about functions](#)
 - [15.1. Function call annotations](#)
 - [15.2. Advanced function topics](#)
- [16. Optimizations](#)
- [17. Running in Turbine](#)
 - [17.1. Architecture](#)
 - [17.2. Concurrency](#)
 - [17.3. Invocation](#)
 - [17.4. Build configuration](#)
 - [17.5. Performance enhancements](#)
 - [17.6. Building standalone executables with mkstatic.tcl](#)
 - [17.7. Debugging Swift/T runs](#)
 - [17.8. Profiling with the Message Passing Environment \(MPE\)](#)
 - [17.9. Startup/shutdown hooks](#)
- [18. Model exploration workflows](#)
- [19. Developers' guide](#)
- [20. Publications](#)

Swift/T is a hierarchical, parallel programming system that allows you to rapidly develop many-task workflows that call into sequential code concurrently. It is implemented with a compiler and runtime system; the Swift/Turbine Compiler (STC) allows you to write Swift programs and run them using Turbine. At runtime, Swift/T programs are MPI programs capable of utilizing very large supercomputers.

1. Support

An overview of Swift/T may be found at the main Swift site:

<http://swift-lang.org/Swift-T>

You can contact our Google Group at this email address:

swift-t-user@googlegroups.com

You can subscribe to the Google Group here:

<https://groups.google.com/forum/#!forum/swift-t-user>

The Swift/T issue tracker is here:

<http://github.com/swift-lang/swift-t/issues>

2. Installation

Swift/T may be installed via binary Debian (and Ubuntu) packages, from [Spack](#), or from source.

Our GitHub repository is here:

<http://github.com/swift-lang/swift-t>

2.1. For Debian-based systems (including Ubuntu)

1. Obtain the Swift/T Debian packages from the [Downloads page](#) or via `wget`:

```
$ wget http://swift-lang.github.io/swift-t-downloads/1.3/swift-t-debs-1.3.tar.gz
```

2. Unpack and enter package directory:

```
$ tar xzf swift-t-debs-1.3.tar.gz
$ cd swift-t-debs
```

3. Install:

```
$ ./install-debs.sh
```

2.2. From Spack

Simply clone Spack and install STC.

```
$ git clone https://github.com/llnl/spack.git
$ . spack/share/spack/setup-env.sh
$ spack install stc
```

This will install all necessary dependencies.

2.3. From source

Writing and running Swift/Turbine programs requires multiple packages. This section provides generic instructions for installing Swift/T on a range of systems. We first cover locating and/or installing prerequisite software packages, then we cover building Swift/T from a source package.

The [Turbine Sites Guide](#) is a accompanying resource for configuration settings and preinstalled software for specific systems.

2.3.1. Prerequisites

All Swift/T prerequisites can be installed using the appropriate package manager for your system.

1. Install or locate MPI implementation (MPICH, OpenMPI, etc.)
 - On maintained compute clusters, an MPI implementation will almost certainly be pre-installed.
 - Many operating systems provide packages with MPI implementations that are usable, but often outdated. E.g. the `mpich` package on Debian/Ubuntu.
 - See [MPICH Guides](#) for information on installing the latest version of MPICH.
 - Installing MPICH from source is not difficult.
 - Other MPI implementations are [supported](#) as well.

Swift/T attempts to use MPI 3.0 functionality by default. If you are using an MPI implementation that does not support the MPI 3.0 standard, you must set `MPI_VERSION=2` (if using the `build-all.sh` build process), or provide the `--enable-mpi-2` configure option (if using the [manual build process](#)).

2. Install or locate Tcl 8.6.
 - You may need to install an additional Tcl development package in addition to the standard Tcl package, e.g., `tcl8.6` plus `tcl8.6-dev` on Debian/Ubuntu (APT) systems.
 - Source distributions are available at the [Tcl web site](#)
 - Tcl 8.5 is also acceptable. (Swift/T optionally uses Tcl 8.6 features to produce cleaner error messages.)
3. Ensure you have these third-party software development tools (APT packages are denoted in parentheses): SWIG (`swig`), ZSH (`zsh`), Apache Ant (`ant`), a Java Development Kit (`default-jdk`), Make (`make`), and GCC for C (`gcc`).

STC is compatible OpenJDK and IBM Java, requiring Java 1.6. Turbine is compatible with GCC, ICC, Clang, and XLC. If you have difficulties with compiler compatibility, please [contact us](#).

2.3.2. Installation of Swift/T from Source

Once you have found all prerequisites, we can continue with building Swift/T from source.

1. Obtain the Swift/T source package from the [Downloads page](#) or via `wget`:

```
wget http://swift-lang.github.io/swift-t-downloads/1.3/swift-t-1.3.tar.gz
```

2. Unpack and enter package directory

```
tar xzf swift-t-1.3.tar.gz
cd swift-t-1.3
```

3. Create a settings file by running:

```
./dev/build/init-settings.sh
```

4. Edit the settings file: `dev/build/swift-t-settings.sh`

At a minimum, you must set the install directory with `SWIFT_T_PREFIX` (the default will install to `/tmp`). On a standard system, no further configuration may be needed. In many cases, however, you will need to modify additional configuration settings so that all prerequisites can be correctly located and configured (see Section [Build configuration](#)).

A range of other settings are also available here: enabling/disabling features, debug or optimized builds, etc.

Tip | Save your `swift-t-settings.sh` when you download a new package

5. Run the setup script

```
dev/build/build-all.sh
```

See `dev/build/README.md` for more notes and build options.

If `build-all.sh` does not succeed on your system, see Section [Build configuration](#) below.

Tip | if you want more control than `build-all.sh` provides, you can build Swift/T with the [manual configure/make workflow](#).

6. Add Turbine and STC to your paths

```
PATH=${PATH}:/path/to/swift-t-install/turbine/bin
PATH=${PATH}:/path/to/swift-t-install/stc/bin
```

3. Usage

Swift code is conventionally written in `*.swift` files. Turbine code is stored in Tcl files with extension `*.tic` (for Turbine Intermediate Code). After writing the Swift program `program.swift`, run:

```
stc program.swift
```

This will compile the program to `program.tic`. A second, optional argument may be given as an alternate output file name.

Then, to run the program, use Turbine:

```
turbine -n 4 program.tic
```

You may compile and run in one step with:

```
swift-t -n 4 program.swift
```

In this case, `program.tic` is created in a temporary file and then deleted after execution.

Provide `-h` to any command (`swift-t`, `stc`, `turbine`) to obtain help.

`swift-t` accepts arguments for both STC and Turbine. If there is a conflict between the `stc` argument and the `turbine` argument, `stc` receives the argument. Use:

```
swift-t -t <flag>[:<value>]+
```

to pass `-<flag> [<value>]` to `turbine`

STC accepts the following arguments:

- A *name=value*
Set a command-line argument at compile-time. This may be found at runtime using the Swift [argument processing](#) library. This option enables these arguments to be treated as compile-time constants for optimization.
- D *macro=value*
Define a C preprocessor macro.
- E
Just run the C preprocessor: do not compile the program. The output goes into the STC output file (the second file name argument).
- I
Add a directory to the import and include search path.
- O
Set optimization level: 0, 1, 2, or 3. See [Optimizations](#).
- j
Set the location of the `java` executable.
- p
Disable the C preprocessor.
- u
Only compile if output file is not up-to-date.
- v
Output version number and exit.
- V
Verbose output.

STC runs as a Java program. You may use `-j` to set the Java VM executable. This Java VM must be compatible with the `javac` used to compile STC.

By default, STC runs the user script through the C preprocessor (`cpp`), enabling arbitrary macro processing, etc. The `-D`, `-E`, `-I`, and `-p` options are relevant to this feature.

Additional arguments for advanced users/developers:

- C
Specify an output file for STC internal representation
- l
Specify log file for STC debug log
- L
Specify log file for more verbose STC debug log
- f
Enable a specific optimization. See [Optimizations](#)
- F
Disable a specific compiler optimization. See [Optimizations](#)

See the [Turbine section](#) for more information about arguments for run time.

Use `swift-t -t <flag>[:<value>]` to pass-through to `turbine`. A common use case for this is `-f`, which is a compiler option flag for `stc` (analogous to `gcc -f`) and a runtime flag for `turbine` (analogous to `mpirun -f`). For example, to pass a MPICH hosts file from `swift-t` to `turbine`, use:

```
swift-t -t f:hosts.txt workflow.swift ...
```

4. Program structure

Swift is a language with C-like syntax. Hello world is written as:

```
import io;

printf("Hello world");
```

The newline is supplied by `printf()`.

Swift programs are composed of *composite* functions containing Swift code, *top-level* Swift code outside of composite functions, and *leaf* functions that wrap non-Swift code, for example native code library functions or external application programs.

In typical Swift programs, [leaf functions](#) do the computational heavy lifting, while Swift code provides the "glue" to compose leaf functions into a complete workflow. From the perspective of the Swift programmer, leaf functions are atomic operations that wait for input variables and set

output variables.

The definition of a function can come before or after the usage in the Swift source code. E.g. a function defined on line 100 of a Swift source file can be called at line 10.

STC input is preprocessed by [cpp](#), the C preprocessor, [by default](#).

5. Comments

Swift supports C/C++-style comments:

```
// This is a comment
/* This is a
comment */
/** Also a
comment */
```

Additionally, if the preprocessor is disabled, single-line comments starting with `#` are supported:

```
# This will work if source file is not preprocessed
```

6. Modules

Swift has a module system that allows you to import function and variable definitions into your source file. Importing a module will import all function and variable definitions from that module into your program.

```
import io;
import mypackage.mymodule;
```

The mechanisms for locating source files is as follows:

- STC searches a list of directories in order to find a Swift source file in the correct directory with the correct name.
- The standard library is always first on the search path, and the current working directory is last.
- Additional directories can be added with the `-I` option to STC.
- Swift source files must have a `.swift` suffix. E.g. `import io;` looks for a file called `io.swift`.
- In the case of a multi-part import name, E.g. `import mypackage.mymodule`, then, it looks for `mymodule.swift` in subdirectory `mypackage`.

The alternative `#include` statement textually includes an entire file using the C preprocessor at the point of the statement. Note that `#include` will only work if the preprocessor is enabled on the current file. In contrast to `import`, `#include` will run the C preprocessor on any included modules. `import` is recommended over `#include` unless the imported module requires preprocessing.

```
#include <mypackage/mymodule.swift>
```

7. Variable declarations

Swift is a statically and strongly typed language. Thus, every variable in the program must have a fixed type at compile time and types are, with limited exceptions, not automatically converted.

To declare a variable in Swift code, you must declare it with one of following variations of the variable declaration syntax:

```
<type> <variable name>;
<type> <variable name> = <expression>;
<variable name> = <expression>;
```

For example:

```
int x;
int x = 1;
x = 1;
```

In the last two cases, the variable is declared and assigned in the same statement. In the first two cases, the type of the variable is explicitly specified by the programmer. In the last case, where the type of the variable is omitted and there was no prior declaration of the variable, then the type of the variable is automatically inferred based on the type of the expression on the right hand side of the assignment.

The scope of a variable is limited to the code block in which the declaration appears. Two variables with the same name cannot be declared in the same block. **Shadowing** of variables, where a variable has the same name as a variable in an outer scope, is also not allowed in Swift. This language design decision is intended to eliminate some common programming errors.

This is shadowing and will result in a compile error:

```
int x;
{
    int x;
}
```

This is not shadowing, because the scopes of the two declarations of `x` do not overlap, and is allowed:

```
{
    int x;
}
{
    int x;
}
```

Variables, functions, and types share the same namespace so it is possible for a variable's name to shadow a function's name. This is allowed in the specific case when a local variable has the same name as a global function (although it will not be possible to use the function within the scope where it has been shadowed).

8. Dataflow evaluation

Swift expressions are evaluated in *dataflow* order. This allows the system to run workflows with the maximum possible concurrency, limited only by the data dependencies in the workflow and available resources (workers and CPUs).

Consider the following code:

```
int a=1,b=2;
f(a);
g(b);
```

In this case, `f(a)` and `g(b)` are eligible to run at the same time. They will be distributed to different workers in the system and run in parallel. If all workers are currently busy, they will be queued for execution as soon as workers are available.

This dataflow mechanism allows Swift/T to execute loops, recursive function calls, and so on with maximal concurrency.

Consider the following code:

```
int a;
int b = f(a);
g(b);
```

In this case, a simple data dependency on `b` forces `g()` to wait for `b` to be set before it can be released for execution.

Consider this more complicated example:

```
int z1,z2;
int y;
int x = f(y);
y = g(2);
z1 = h(x,y,1);
z2 = h(x,y,2);
int output = r(z1,z2);
```

In this example, `g()` runs first, because it is dependent only on a literal. After `y` is set, `f()` runs, setting `x`. Then, two invocations of `h()` execute. Finally, `z1` and `z2` are set, allowing `r()` to run.

Variables may be assigned only once. Multiple assignment is often detected at compile time, and will always be detected at run time, resulting in a run time error. If variable is not assigned, expressions that depend on the variable cannot execute. If the variable is never assigned during the course of program execution, these expressions will never execute. Upon program completion, Swift/T will report the error and print debug information about any unexecuted expressions and identifiers of corresponding unassigned variables.

In summary, there is no instruction pointer in Swift/T. All statements in the workflow are eligible to run as soon as their input data is available.

Some applications have side effects, such as creating or modifying files that another task must wait for. It is a best practice to represent these data items with the [Swift/T file type](#). But if you need to enforce a certain order that you cannot express with dataflow, see the section on [explicit execution order](#).

9. Composite functions

Composite functions provide a way to organise and reuse Swift code. They have the form:

```
[(<output list>)] function_name [(<input list>)]
{
    statement;
    statement;
    ...
}
```

An empty input or output list may be omitted or written as `()`.

The output list may have more than one entry. Thus, assignments may be written as:

```
x1, x2 = f(i1, i2);
// or equivalently:
(x1, x2) = f(i1, i2);
```

- **Note:** If a composite function named `main` is provided, it is automatically run at the start of the program.

10. Types

Swift provides a similar range of primitive types to many other programming languages. Files are a primitive type in Swift, unlike in many other languages, and have a number of special characteristics that merit special mention. Two basic kinds of data structure are provided: arrays and structs.

10.1. Primitive types

Swift has the conventional types:

`string`

A complete string (not an array of characters).

`int`

A 64-bit integer.

`float`

A 64-bit (double-precision) floating point number.

`boolean`

A boolean (true/false).

`file`

A file (see [Section Files](#)).

`blob`

External byte data (see [Section Blobs](#)).

Literals for these types use conventional syntax:

- `int` literals are written as decimal numbers, e.g. `-1234`

- `float` literals are written as decimal numbers with a decimal point, e.g. `5493.352` or `1.0`. Scientific notation may be used, as in `2.3e-2` which is equivalent to `0.023`. The literals `NaN` and `inf` may be used. In some contexts `int` literals are promoted automatically to `float`.
- `boolean` literals `true` and `false` may be used.
- `string` literals are enclosed in double quotes, with a range of escape sequences supported:
 - `\\` for a single backslash
 - `\"` for a quote
 - `\n` for newline
 - `\t` for tab
 - `\a` (alarm)
 - `\b` (backspace)
 - `\f` (form feed)
 - `\r` (carriage return)
 - `\v` (vertical tab)
 - octal escape codes, e.g. `\001`
 - hexadecimal escape codes, e.g. `\xf2`
 - For more information: [ASCII control codes](#).
- Multi-line strings may be used in two syntaxes:
 - Python-style:

```
string s =
"""
line data 1
line data 2
""";
```

- AsciiDoc-style: like Python-style but use 4 dashes instead of 3 quotes.
- **Note:** Multi-line strings are somewhat incompatible with the C preprocessor: if you try to compile a Swift program using multi-line strings with the preprocessor enabled, you will likely see warnings or strange behavior. To disable the C preprocessor, use the `-p` option to STC.

10.2. Files

A file is a first-class entity in Swift that in many ways can be treated as any other variable, and passed to and from [app functions](#). The main difference is that a file is **mapped** to path in a filesystem. Assigning to a mapped file variable results in a file being created in the file system at the specified path. File paths can be arbitrary Swift expressions of type `string`. Absolute paths or relative paths are specified, with relative paths interpreted relative to the path in which `turbine` was run. File variables can also be written by assigning a pre-existing file using the `input()` function.

For example, if `/home/user/in.txt` is a file with some data in it, the following Swift program will copy the file to `/home/user/out.txt`.

```
file x = input("/home/user/in.txt");
file y <"home/user/out.txt">; // Declare a mapped file
y = x; // Do the copy
```

A range of functions to work with files are provided in the [files library module](#).

```
// Initialize an array of files from a range of files on disk with glob
file f[] = glob("directory/*.txt");

// Read the contents of a file with read
string filename = "in.txt";
string contents = read(input(filename));
trace("Contents of " + filename + ":\n" + contents);

// Write directly to an unmapped file
file tmp = write("first line\nsecond line");

// Find the name of a file with filename
trace("Temporary filename is: " + filename(tmp));
```

Temporary files are created as necessary if unmapped files are written to, for example, the file `tmp` in the code snippet above. This file is created in `/tmp` by default unless [overridden by `SWIFT\_TMP`](#).

The way this feature is implemented depends on the Tcl version:

8.5

Implemented by calling system command `mktemp`. The directory may be set with environment variable `TMPDIR` which defaults to `/tmp`, or `SWIFT_TMP`. The filename will contain `.turbine` in the middle or as a suffix, depending on `mktemp` implementation.

8.6

Implemented with a Tcl builtin. The default directory is `/tmp`, set environment variable `TMP` or `SWIFT_TMP` to change this. The file always has suffix `.turbine`.

Swift/T assumes that the file system is shared among all nodes. Set `SWIFT_TMP` to a shared file system for best results when running across nodes. (The directory `/tmp` is usually a fast local file system and cleared on system reboot.)

If a file is only specified in a return type of a [composite function](#), that specification not declare the file, create a temporary file, or assign to it, e.g.:

```
(app o) f() { /* Error: o is not assigned. */ }
```

The body of `f()` should assign to `o`.

Note

The syntax

```
file f<"f.txt"> = g();
```

is allowed but

```
file f<"f.txt">=g();
```

results in a parse error: the `>=` sequence is tokenized as *greater-than-or-equal-to*.

10.3. Blobs

Blobs represent raw byte data. They are primarily used to pass data to and from native code libraries callable from Swift. They are like Swift strings but may contain arbitrary data.

Swift provides multiple builtin functions to create blobs, convert blobs to and from Swift types, and pass blobs to leaf functions.

10.4. Arrays

Arrays can be declared with empty square brackets:

```
int A[];
```

Arrays with empty square brackets have integer indices. It is also possible to declare integers with other index types, such as strings:

```
string dict[string];
```

They are dynamically sized, expanding each time an item is inserted at a new index. Arrays are indexed using square brackets.

```
int A[string];
int B[];
B = function_returning_array();
A["zero"] = B[0];
A["one"] = B[1];
```

Each array index can only be assigned to once.

A given array variable must be assigned either *in toto* (as a whole) or *in partes* (piece by piece). In this example, `B` is assigned in toto and `A` is assigned in partes. Code that attempts to do both is in error.

Arrays may be used as inputs or outputs of functions.

Arrays are part of Swift dataflow semantics. An array is closed when all possible insertions to it are complete.

```
(int B[]) f(int j)
{
    int A[];
    A = subroutine_function(1);

    // Error: A has already been assigned in toto:
    A[3] = 4;

    // OK: assigning to output variable
    B = subroutine_function(2);
}
```

Array literals may be expressed using the range operator:

```
int start = 0;
int stop = 10;
int step = 2;
// Array of length 10:
int A[] = [start:stop-1];
// Array of length 5, containing only even numbers:
int B[] = [start:stop-1:step];
```

Array literals may also be expressed with list syntax:

```
int C[] = [4,5,6];
```

10.5. Nested arrays

Swift allows arrays of arrays: nested arrays. They can be declared and assigned as follows:

```
// An array of arrays of files with string keys
file A[string][string];
A["foo"]["bar"] = input("test.txt");
A["foo"]["gux"] = input("test2.txt");
```

Note: there is currently a limitation in assignment of nested arrays that a given array can only be assigned at a single "index level". If `A` is a 2D array, for example, then you cannot mix assignments specifying one index (e.g. `A[i] = ...`) with assignments specifying three indices (e.g. `A[i][j] = ...`).

10.6. Structs

In Swift, structs are defined with the `type` keyword. They define a new type.

```
type person
{
    string name;
    int age;
    int events[];
}
```

Structs are accessed with the `.` syntax:

```
person p;
p.name = "Abe";
p.age = 90;
```

It is possible to have arrays of structs, with some restriction on how they can be assigned. Each struct in the array must be assigned *in toto* (as a whole). For example, the following code is valid:

```

person people[], p1, p2;

p1.name = "Thelma";
p1.age = 31;

p2.name = "Louise";
p2.age = 29;

people[0] = p1;
people[1] = p2;

```

However, attempting to assign the structs in the following way is currently unsupported:

```

people[2].name = "Abe";    // Not supported!
people[2].age = 90;        // Not supported!

```

It is possible to construct a struct by invoking the type name as if it is a function:

```

person p3 = person("Nguyen", 42);

```

10.7. Defining new types

Swift has two ways to define new types based on existing types.

The first is `typedef`, which creates a new name for the type. The new type and the existing type will be completely interchangeable, since they are simply different names for the same underlying type. The new type name simply serves to improve readability or documentation.

```

typedef newint int;

// We can freely convert between int and newint
newint x = 1;
int y = x;
newint z = y;

```

The second is with `type`, which creates a new type that is a specialization of an existing type. That is, it is a distinct type that is not interchangeable. A specialized type can be converted into the original type, but the reverse transformation is not possible. This means that you can write functions that are more strictly typechecked, for example, you can ensure that certain `app` functions will only receive particular types of files.

```

typedef sorted_file file;
app (sorted_file out) sort (file i) {
    "/usr/bin/sort" "-o" out i
}

// The uniq utility requires sorted input
app (file o) unique (sorted_file i) {
    "/usr/bin/uniq" i @stdout=o
}

file unsorted = input("input.txt");
sorted_file sorted <"sorted.txt"> = sort(unsorted);
file u <"unique.txt"> = unique(sorted);

// Can convert from sorted_file to file
file result2 = sort(unsorted);

// This would cause a type error
// sorted_file not_sorted = unsorted;

```

10.8. Global variables

Variables defined at the top level of a Swift program are in global scope and can be used anywhere Swift supports a basic feature for defining globally visible constants.

You can use global variables to define program-wide constants: the single assignment semantics of Swift mean that they variables cannot be reassigned elsewhere in the program.

```

string HELLO = "Hello World";
float PI_APPROX = 3.142;
int ONE = 1;

trace(HELLO, PI_APPROX, ONE);

```

- **Note:** You can also define global constants with the syntax `global const <name> = <value>;`, e.g. `global const X = 1;`. This syntax offers no advantages and is deprecated.

11. Control structures

Swift provides control structures that may be placed as statements in Swift code.

11.1. Conditionals

11.1.1. If statement

If statements have the form:

```

if (<condition>)
{
    statement;
    ...
}
else
{
    statement;
    ...
}

```

As required by dataflow processing, neither branch of the conditional can execute until the value of the condition expression is available.

11.1.2. Switch statement

```
int a = 20;
switch (a)
{
    case 1:
        int c;
        c = a + a;
        b = a + 1;
    case 20:
        b = 1;
    case 2000:
        b = 2;
    default:
        b = 2102 + 2420;
}
printf("b: %i\n", b);
```

Note: there is no fall-through between cases in switch statements.

11.2. Iteration

Iteration is performed with the `foreach` and `for` statements.

11.2.1. Foreach loop

The `foreach` loop allows for parallel iteration over an array:

```
string A[];
foreach value, index in A
{
    printf("A[%i] = %s\n", index, value);
}
```

The `index` and `value` variables are automatically declared. The `index` variable may be omitted from the syntax.

A special case of the `foreach` loop occurs when combined with the array range operator. This is the idiomatic way to iterate over a range of integer values in Swift. The STC compiler has special handling for this case that avoids constructing an array.

```
foreach i in [start:stop:step] {
    ...
}
```

11.2.2. For loop

The `for` loop allows for sequential iteration. This example implements a counter based on the return values of a function that accepts integers:

```
int N = 100;
int count;
for (int i = 0, count = 0; i < N; i = i+1, count = count+c)
{
    int c;
    if (condition_function(i))
    {
        c = 1;
    }
    else
    {
        c = 0;
    }
}
```

The general form is:

```
for ( <initializer> ; <condition> ; <updates> )
{
    statement;
    ...
}
```

The initializer is executed first, once. The initializer is a comma-separated list of statements. The body statements are then executed. Then, the assignments are performed, formatted as a comma-separated list. Each is a special assignment in which the left-hand-side is the variable in the *next* iteration of the loop, while the right-hand-side is the variable in the *previous* loop iteration. Then, the condition is checked for loop exit. If the loop continues, the body is executed again, etc.

Variables declared in the initializer are only visible in the loop body scope. Variables used on the right-hand-side of updates must be assigned (but not necessarily declared) in the initializer, or in the loop body.

Performance Tip: use the `foreach` loop instead of `for` if your loop iterations are independent and can simply be executed in parallel.

11.3. Explicit execution ordering

In general, execution ordering in Swift/T is implicit and driven by data dependencies. In some cases it is useful to add explicit data dependencies, for example if you want to print a message to indicate that variable was assigned. It is possible for the programmer to express additional execution ordering using three constructs: the `wait` and `wait deep` statements and the `=>` chaining operator.

In a `wait` statement, a block of code is executed after one or more variables are closed.

```
x = f();
y = g();
wait (x) {
    trace("x is closed!");
}
wait(x, y) {
    trace("x and y are closed!");
}
```

In a `wait deep` statement, a block of code is executed after all elements of an array have been closed:

```
int A[];
A = [f(), g()];
```

```
wait deep (A) {
  trace("All elements of A are closed!");
}
```

The chaining operator chains statements together so that a statement only executes after the previous statement's output value is closed. This is a more concise way to express dependencies than the `wait` statement.

```
sleep(1) =>
  x = f() =>
    int y = g() =>
      trace("DONE!");
```

Chaining is based on the **output values** of a statement. In the simple case of a function call `f()` $\Rightarrow$ `...`, the output values are the output values of the function. In the case of an assignment `x = f()` $\Rightarrow$ `...` or a declaration, `int y = g()` $\Rightarrow$ `...`, then the next statement is dependent on the assigned values, or the declared values. Some functions such as `sleep` have `void` output values so that they can be used in this fashion.

11.4. Scoping blocks

Arbitrary scoping blocks may be used. In this example, two different variables, both represented by `b`, are assigned different values.

```
{
  int b;
  b = 1;
}
{
  int b;
  b = 2;
}
```

12. Operators

12.1. Numeric operators

+ (plus), - (minus), * (times), / (divide), %/ (integer divide), %% (modulus), ** (power).

== (equals), != (not equals), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to).

- (negate).

12.2. Boolean operators

&& (boolean and), || (boolean or).

`xor()` is a builtin function.

Swift boolean operators are not short-circuited (to allow maximal concurrency). For conditional execution, use an `if` statement.

The following unary operators are defined:

! (boolean not).

12.3. String operators

- String concatenation is also performed with + (plus).
- == and != may also be used on strings.
- Operator `s1/s2` is equivalent to `s1+"/"+s2`. Cf. [dircat\(\)](#).
- The Python-style string format operator `format%(arg1,arg2,...)` is equivalent to

```
sprintf(format, arg1, arg2, ...)
```

See [sprintf\(\)](#). The parentheses may be omitted for single arguments.

13. Standard library

Each category of function is shown with the required import statement, if necessary.

Functions that accept an input of any type are denoted [anything](#). Functions that accept variable numbers of arguments (including zero) are denoted with ellipsis (...).

A function that accepts more than one type is denoted as `f(int|string)`.

If a function is described below an **Import:** label, be sure to `import` that package.

13.1. General

`xor(boolean,boolean) → boolean`

Exclusive logical or

`propagate(int|float|string|boolean|void|file...) → void`

Create a void value.

(`make_void()` and `makeVoid()` are deprecated aliases for this function.)

`size(A[]) → int`

Obtain the size of array A

`contains(A[], key) → boolean`

Test that future `A[key]` exists. This function blocks until A is closed. Consumers of `A[key]` may block again until `A[key]` is stored.

13.1.1. Pick functions

Select items from an array.

`pick_integer_string(string A[], int indices[]) → string result[]`

Select the `indices` from A. The `result` may be reordered.

`pick_stable_integer_string(string A[], int indices[]) → string result[]`

Select the `indices` from A. The ordering of `result` will be the same as in A.

`pick_regexp(string pattern, string A[]) → string result[]`

Select strings from A matching regular expression `pattern`. (This function is implemented with the Tcl regular expressions [documented here](#).)

13.2. Type conversion

`int2string(int) → string`
Convert integer to string

`string2int(string) → int`
Convert string to integer (alias: `parseInt()`)

`float2string(float) → string`
Convert float to string

`string2float(string) → float`
Convert string to float (alias: `parseFloat()`)

`int2float(int) → float`
Convert integer to float

`float2int(float) → int`
Convert float to integer
(Retains integer part, rounds toward 0. Implemented with `Tcl int()`)

`boolean2string(boolean) → string`
Convert boolean to string

`string2boolean(string) → int`
Convert string to boolean. Case-insensitive. Accepts for true: "true", "yes", "y", any non-zero number as a string. Accepts for false: "false", "no", "n", "0".

`repr(*) → string`
Convert any type to internal string representation (exact format not guaranteed to be consistent, even from call to call)

`array_repr(*[[]]) → string[]`
Convert array of any type to internal string representation (exact format not guaranteed to be consistent, even from call to call)

13.3. Output

`trace(anything, anything, ...)`
Report the value of any variable

Import: `io`

`printf(string format, int|float|string|boolean...)`
As `printf()` in C

13.4. String functions

`strcat(string|int|float...) → string`
Returns the concatenation of all arguments as strings. Equivalent to the Swift/T plus (+) operator.

`length(string) → int`
Obtain length of string.

Import: `string`

`substring(string s, int start, int length) → string`
Obtain substring of given string `s` starting from character `start` of length `length`.

`find(string s, string substring, int start_index, int end_index) → int`
Find the index of the first occurrence of the string `substring` within the string `s` between the indices `start_index` and `end_index`. Here an index of -1 passed to `end_index` results in `end_index` being treated as the length of the string `s`. `find` returns -1 in case there is no occurrence of `substring` in `s` in the specified range.

`string_count(string s, string substring, int start_index, int end_index) → int`
Counts the occurrences of the string `substring` within the string `s` between the indices `start_index` and `end_index`. Here an index of -1 passed to `end_index` results in `end_index` being treated as the length of the string `s`

`is_int(string s) → boolean`
Returns true if string `s` is a number, else false.

`replace(string s, string substring, string rep_string, int start_index) → string`
Obtain the string created by replacing the first occurrence of the string `substring` within string `s`, after the index `start_index`, with the string `rep_string`. In case there is no such occurrence of the string `substring` in string `s`, the original string `s` is returned unmodified.

`replace_all(string s, string substring, string rep_string, int start_index) → string`
Obtain the string created by replacing all the occurrences of the string `substring` within string `s`, after the index `start_index`, with the string `rep_string`. In case no such occurrence of `substring` exists in `s`, the original string `s` is returned unmodified.

`split(string s, string delimiter=" ") → string[]`
Tokenize string `s` with given delimiter.
The `delimiter` is optional, the default is a space (" ").

`trim(string s) → string`
Remove leading and trailing whitespace from `s`

`strlen(string) → int`
Obtain the length of the given string (as `length()`).

`hash(string) → int`
Hash the string to a 32-bit integer

`sprintf(string format, int|float|string|boolean...) → string`
As `sprintf()` in C. In the `format` string, use `%i` for int, `%f` for float, `%s` for string. Use `%i` for boolean to obtain "1" for true and "0" for false. Use the `%%` escape to obtain a plain %. Most other ANSI C `sprintf()` features are supported.

This function is implemented by calling to the Tcl `format` command.

`join(string A[], string separator=" ") → string`
Join strings in `A` with given `separator`. The `separator` may be the empty string.
The `separator` is optional, the default is a space (" ").

`join_args(string separator, string|int+float...) → string`
Join given arguments with given `separator`. The `separator` may be the empty string.

`dircat(string...) → string`
directory-concatenate. Concatenate arguments with / as separator. Cf. [\[String operators\]](#).

13.5. Math

`max|min_integer(int,int) → int`
Obtain maximum or minimum integer, respectively

```
max|min_float(float,float) → float
    Obtain maximum or minimum float, respectively
```

```
pow_integer(int b,int x)
    Obtain bx
pow_float(float b,float x)
    Obtain bx
```

Import: math

```
floor(float) → float
    Round down
ceil(float) → float
    Round up
round(float) → float
    Round nearest
ln(float) → float
    Natural logarithm
log10(float) → float
    Base-10 logarithm
log(float x, float b) → float
    Base-b logarithm of x
exp(float) → float
    Natural exponentiation: ei
sqrt(float) → float
    Square root
is_nan(float) → boolean
    Check for NaN
abs_integer(int) → int
    Absolute value
abs_float(float) → float
    Absolute value
```

Import: random

```
random() → float
    Obtain random number
randint(int start, int end)
    Obtain random integer from start, inclusive, to end, exclusive
```

Import: stats

```
sum_integer(int[]) → int
    Sum
avg(int|float[]) → float
    Average
std(float[]) → float
    Standard deviation
```

13.6. System

Import: sys

```
getenv(string) → string
    Obtain an environment variable
sleep(float) → void
    Delay for the given number of seconds.
clock() → float
    Obtain time since the Unix epoch in seconds.
clock_seconds() → int
    Obtain time since the Unix epoch in whole seconds.
INT_MAX() → int
    Obtain the largest integer representable on the current platform.
system(string[]) → int
    Run the command (string array), return its exit code.
system1(string) → int
    Run the command (simple string), return its exit code.
```

13.6.1. Command line

Consider this command line:

```
turbine -l -n 3 program.tic -v -a=file1.txt file2.txt --exec="prog thing1 thing2" -
-help file4.txt
```

The arguments to `program.tic` are just the tokens after `program.tic`

```
args() → string
    Obtain all arguments as single string
    E.g., "-v -a=file1.txt file2.txt --exec="prog thing1 thing2" --help file4.txt"
```

The remaining functions are convenience functions oriented around Swift conventions. Under these conventions, the example command above has *flagged* arguments `v`, `a=file.txt`, `exec="prog thing1 thing2"`, and `help`. The command has *unflagged* arguments `file2.txt` and `file4.txt`

```
argc()
    Get count of unflagged arguments
argv(string, [default])
    (argument-value) Given a string, returns the flagged argument with that key:
    argv("a") → file1.txt
    In addition to regular run-time arguments, the STC compile-time arguments feature allows
    argv() arguments to be provided at compile time. This allows a specialized, optimized
    version of code to be compiled for a particular set of arguments. See the -A name=value
    argument to stc. Note that if the argument is re-specified at run-time, an error will occur.
    If argument a is not provided, the default value is used. So:
    argv("b", "f0.txt") → "f0.txt"
argp(int)
    (argument-positional) Given an integer, returns the unflagged argument at that index:
```

```
argp(2) → file4.txt
Given 0, returns the program name,
argp(0) → /path/to/program.tic
argv_accept(string...)
If program is given flagged command line arguments not contained in given list, abort. E.g.,
argv_accept("x") would cause program failure at run time
argv_contains(string) → boolean
Test if the command line contains the given flagged argument:
argv_contains("v") → true
```

13.6.2. Debugging

Import: assert

```
assert(boolean condition, string message)
If condition is false, report message and exit immediately.
```

13.6.3. Turbine information

```
adlb_servers() → int
Number of ADLB servers
turbine_workers() → int
Number of Turbine workers
```

13.7. Files

```
filename(file) → string
Obtain the name of a file
input(string) → file
Obtain a file. At run time, the filesystem is checked for the given file name
input_file(string) → file
Alias for input()
input_url(string) → file
Obtain a file. Some automatic operations and optimizations are disabled
urlname(file) → string
Obtain the name of a file created with input_url()
```

Import: files

13.7.1. I/O with files

```
read(file) → string
Read file as a string
write(string) → file
Write string to file
write_array_string(string[] A, int chunk=100, boolean indices=false) → file
Write strings in A to output file, chunk at a time (chunk defaults to 100). There is no ordering
on the output. If indices==true, the integer indices of A will be put on each line as a prefix.
This is more efficient than write(join(A)) for large A.
write_array_string_ordered(string A[]) → file
Write strings in A to output file in order. This is more efficient than write(join(A)) for large
A, but slower than write_array_string() (which is not ordered).
file_lines(file, comment="#") → string[]
Reads the whole file, returning each line as a separate entry in the output array. The
comment argument is optional and defaults to "#". Comments are excised, leading and
trailing whitespace is trimmed, and blank lines are omitted. If comment="", comments are
disabled (that is, you will receive the whole file, nothing will be excised).
```

13.7.2. Finding files

```
glob(string) → file[]
Perform glob operation, returning files that match. Available glob symbols include:


- : any character sequence (including the zero-length sequence)
- ?: any character
- [chars]: any of the given characters
- \x: character x
- {a,b,c,...} any of a, b, c, etc.


file_exists(string) → boolean
Attempt to find a file with the given name in the filesystem: return true if found, else false.
```

13.7.3. File metadata

```
file_mtime(string) → int
Attempt to find a file with the given name in the filesystem: return its POSIX modification
time in seconds since the Unix epoch.
file_type(file) → string
Returns a string giving the type of file, which will be one of "file", "directory",
"characterSpecial", "blockSpecial", "fifo", "link", or "socket".
file_type_string(string) → string
As file_type() but accepts a string.
```

13.7.4. File name manipulation

```
dirname_string(string) → string
Returns the directory part of the given string path.
dirname(file) → string
Returns the directory part of the given file path.
basename_string(string) → string
Returns the file part of the given string path.
basename(file) → string
Returns the file part of the given file path.
rootname_string(string) → string
Returns the root part of the given string path (filename without extension).
rootname(file) → string
Returns the root part of the given file path (filename without extension).
extension_string(string) → string
Returns the filename extension part of the given string path.
```


`extension(file) → string`

Returns the filename extension part of the given file path.

13.7.5. Temporary files

`mktemp() → file`

Obtain new temporary file.

`mktemp_string() → string`

Obtain new temporary file, return its name as a string.

13.8. Blobs

Import: `blob`

`blob_size(blob) → int`

Obtain the size of a blob in bytes.

`blob_null() → blob`

Obtain an empty blob of size 0.

`string2blob(string) → blob`

Convert a string into a blob.

`blob2string(blob) → string`

Convert a blob into a string. If the blob is not NULL-terminated, this function appends the NULL-terminator.

`floats2blob(float[]) → blob`

Convert an array of Swift floats (implemented as doubles) to blob containing the C-formatted array of doubles .

`blob2floats(blob) → float[]`

Convert blob containing the C-formatted array of doubles to an array of Swift floats (implemented as doubles).

`ints2blob(int i[]) → blob`

Convert blob containing the C-formatted array of ints to an array of Swift ints (implemented as 64-bit integers).

`blob_read(file) → blob`

Reads whole file, returning it as a blob.

`blob_write(blob) → file`

Writes whole file with given blob.

13.9. Location

See the section about [location](#).

Import: `location`

`rank2location(int) → location`

Convert the rank integer to a `location` variable compatible with `@location` with `HARD`, `RANK`. (Alias: `locationFromRank()`.)

`randomWorker() → location`

Obtain a worker at random with `HARD`, `RANK`.

`randomWorkerRank() → int`

Obtain a random worker rank.

`hostmapList() → string[]`

Obtain the whole hostmap as an array with integer keys and string hostname values.

`hostmapOne(string) → location`

Lookup the string as a host in the hostmap and return one rank running on that host with `HARD`, `RANK`.

`hostmapOneWorkerRank(string) → int`

Lookup the string as a host in the hostmap and return one of the worker ranks running on that host.

`hostmapLeaders() → int[]`

Obtain ranks of all "leaders"- the lowest worker rank on each node of the run. Can be used to send a task to each node.

13.10. Unix tools

Swift/T provides a small number of standardized [app](#) functions for common shell tools. Here, "via `tool`" means that the program `tool` is simply used to perform the task.

These functions may be found in `turbine/export/unix.swift`, and may be easily copied out and extended to serve other purposes.

Import: `unix`

`cp(file) → file`

Copy file to file (via `cp`).

`catp(file f[])`

(*cat-print*) Print files to standard output (via `cat`).

`cat(file f[]) → file`

Concatentate all files to output file (via `cat`).

`sed(file, string command) → file`

Perform the given `sed` `command`, output to file (via `sed`).

`touch() → file`

Apply `touch` to the file (via `touch`).

`printenv()`

Print the environment to standard output (via `printenv`).

`echo(string) → file`

Write the string to the file (via `echo`).

`sleep(int i) → void`

Sleep for given number of seconds (via `sleep`).

`mkdir(string) → void`

Make the given directory (via `mkdir`).

14. Defining leaf functions

In typical Swift applications, the computationally intensive parts of the application are not written in the Swift language. Rather, the work is done by *leaf functions* that are *composed* together with Swift code. The three key leaf function types are:

Extension functions: Functions that call to Tcl and/or native code. These functions primarily

operate on in-memory data, and are appropriate for high-performance computing.

app functions: Functions that call to a command-line program (the shell). These functions primarily operate on files, and are appropriate for ordinary workflows.

External scripting functions: Functions that call into an in-memory interpreter in another scripting language, such as [Python](#), [R](#), or [Julia](#). These functions allow you to integrate code from these other languages and run them without forking an interpreter, allowing you to run these languages on high-performance computers (including the Blue Gene and Cray).

The Swift runtime, Turbine, is built on Tcl, a language which was designed to to make it easy to call C/C++/Fortran functions. The Swift/T [standard library](#) is implemented as extension functions in Tcl, some of which wrap C functions.

14.1. Swift extension functions

Swift/T extension functions connect Swift semantics to a Tcl function. Tcl has [excellent support](#) for wrapping native C/C++ functions, so this provides an excellent way to call C/C++ functions from Swift.

Several components are required to implement a Swift native code function:

- Tcl bindings to your function.
- The requisite files required to build a Tcl package (e.g `pkgIndex.tcl`)
- Swift declarations for the function that specify the type of the function and the Tcl implementation.

14.1.1. Simple Tcl fragment example

In this example, the Swift program will simply use Tcl to output a string:

```
() my_output (string s) "turbine" "0.0" [
    "puts <<s>>"
];

my_output("HELLO");
```

`puts` is the Tcl builtin for screen output, like `puts()` in C.

The above definition has, from left to right, the output arguments (none), the name of the new Swift function, input arguments, the name of the Tcl package containing the file (here, none, so we use `turbine`), and the minimum version of that package (here, 0.0).

We tell the compiler how to call our Tcl function using inline Tcl code as a template with variable names surrounded by `<< >>` indicating where variables should be substituted.

14.1.2. Simple Tcl package example

In this first example we will implement a trivial Tcl extension function that doubles an integer.

Here is the Tcl code that will go in `myextension.tcl`:

```
namespace eval myextension {
    proc double { x } {
        return [ expr $x * 2 ]
    }
}
```

Here is the Swift function definition that will go in `myextension.swift`:

```
@pure
(int o) double (int i) "myextension" "0.0.1" [
    "set <<o>> [ myextension::double <<i>> ]"
];
```

We can also tell the Swift compiler a little about the function so that it can better optimize your programs. For example, `double` has no side-effects and produces the same result each time for the same arguments (i.e. is deterministic), so we can annotate it as a `@pure` function.

If your function has a long running time and should be dispatched to a worker process for execution, then you need to label the function as a worker function, for example:

```
@dispatch=WORKER
(int o) process (int i) "pkg" "0.0.1" [
    "set <<o>> [ pkg::process <<i>> ]"
];
```

Tcl code is conventionally placed into *packages*. In this example, `myextension.tcl` would be part of the package.

More information about building Tcl packages may be found [here](#). Ultimately, you produce a `pkgIndex.tcl` file that contains necessary information about the package.

To ensure that Swift can find your package, use

```
stc -r <package directory> ...
```

or set `SWIFT_PATH` at run time.

- **Tip:** advanced users can also create [standalone executables](#) with compiled code and Tcl code for the extension directly linked in.

14.1.3. Swift/Tcl data type mapping

If you are defining Tcl functions in the way above with inline Tcl code, Swift types are mapped to Tcl types in the following way:

- `int/float/string/bool` are converted to the standard Tcl representations.
- blobs are represented as a Tcl list with first element a pointer to the data, the second element the length of the data, and if the blob was loaded from the ADLB data store, a third element which is the ADLB ID of the blob.
- files are represented as a list, with the first element the file path, and the second element a reference count
- arrays are represented by Tcl dictionaries with keys and values represented according to their type.
- bags are represented by Tcl lists with elements in an arbitrary order.
- output voids are set automatically.

14.1.4. Calling native libraries from Swift

The first step is to test that you can successfully call your C/C++/Fortran function from a test Tcl script. If so, you will then be able to use the Swift-Tcl techniques to call it from Swift.

A popular tool to automate Tcl-C bindings is [SWIG](#), which will wrap your C/C++ functions and

help you produce a Tcl package suitable for use by Swift.

To call Fortran functions, first wrap your code with [FortWrap](#). Then, use SWIG to produce Tcl bindings.

14.1.5. Writing custom Tcl interfaces

It is possible to write a Tcl wrapper function that is directly passed references to data in Swift's global data store. In this case your function must manually retrieve/store data from/to the global distributed data store. In this case, you do not use the STC Tcl argument substitution syntax (`<<i>>`).

Consider this custom Swift→Tcl binding:

```
(int o) complex_function (int arr[]) "pkg" "0.0.1" "complex";
```

This function jumps into Tcl function `complex`, which must perform its own data dependency management.

See the [Swift/T Leaf Function Guide](#) for more information about this process.

14.2. Dispatch and work types

Each worker in Swift/T is devoted to executing a single type of work. There is a default work type that encompasses Swift/T script logic and CPU tasks.

There are two subtypes of the default work type that are treated differently by the Swift/T optimizer. `CONTROL`, the default, is intended for functions that run for a short duration, for example builtin functions such as `strcat()`, arithmetic. `WORKER`, the second subtype, is intended for functions that perform more computation and I/O, which may keep a worker busy for a while. Swift/T may bundle together multiple `CONTROL` tasks to reduce overhead, but will keep `WORKER` tasks separate to increase parallelism.

The work type is specified with a `@dispatch` annotation above leaf function definitions.

```
@dispatch=WORKER
(int o) process (int i) "pkg" "0.0.1" [
    "set <<o>> [ pkg::process <<i>> ]"
];
```

Alternative work-types include custom user-defined work types, along with tasks for external executors such as Coasters or GeMTC for remote command-line and GPU tasks respectively. It is also possible to define custom types of CPU leaf functions.

14.2.1. Custom work types

For some applications, it is useful to be able to divide up CPU workers into multiple categories that execute different kinds of work. For these scenarios, Swift/T provides the ability to define custom work types.

This sample program illustrates how to define a custom work type `foo_work` and define a function, `hello1()`, which executes on a `foo_work` worker. For comparison, we also include `hello2()`, which uses the default work type.

```
pragma worktypedef foo_work;

@dispatch=foo_work
hello1 (string msg) "turbine" "0.0" [
    "puts [ format {Hello %s} <<msg>> ]"
];

hello2 (string msg) "turbine" "0.0" [
    "puts [ format {Hello %s} <<msg>> ]"
];

// Hello Foo will be printed on a foo_work worker
hello1("Foo");

// Hello Bar will be printed on a regular worker
hello2("Bar");
```

In order to run the above script, we need to ensure that a worker is allocated to execute `foo_work` tasks. This is achieved by setting the environment variable `TURBINE_FOO_WORK_WORKERS` to the desired number of workers. The Swift/T workers are then divided between default workers and `foo_work` workers.

An example run is as follows:

```
$ export TURBINE_LOG=1
$ export TURBINE_FOO_WORK_WORKERS=2

$ swift-t -l -n 5 types.swift
[0]    0.000 WORK TYPES: WORK foo_work
[0]    0.000 WORKERS: 4 RANKS: 0 - 3
[0]    0.000 SERVERS: 1 RANKS: 4 - 4
[0]    0.000 WORK WORKERS: 2 RANKS: 0 - 1
[0]    0.000 foo_work WORKERS: 2 RANKS: 2 - 3
...
[0] Hello Bar
[3] Hello Foo
```

When logging is enabled, Swift/T reports the registered work types and their actual ranks. In the example, Swift/T runs on 5 ranks (`-n 5`) with rank numbers on the output (`-l`). As shown, ranks 0-1 are default workers. Ranks 2-3 are `foo_work` workers. (Rank 4 is the ADLB server.) Thus, "Hello Bar" is reported on rank 3 and "Hello Foo" is reported on rank 0.

14.3. App functions

App functions are functions that are implemented as command-line programs. These command-line programs can be brought into a Swift program as functions with typed inputs and outputs. An app function definition comprises:

- The standard components of a Swift function declaration: input and output arguments and the function name. Note that the output variable types are restricted to individual `files`.
- The command line, which comprises an initial string which is the executable to run, and then a series of arguments which are the command-line arguments to pass to the program.

App arguments can be:

- Literals such as numbers or strings.
- File variables (passed as file paths).
- Other variables, which are converted to string arguments. Arrays (including multi-

dimensional arrays) are expanded to multiple arguments.

- Arbitrary expressions surrounded by parentheses.

Standard input, output and error can be redirected to files via `@stdin=`, `@stdout=`, and `@stderr=` expressions. If used, these should point to a [file](#).

Here is an example of an app function that joins multiple files with the `cat` utility:

```
import files;

app (file out) cat (file inputs[]) {
  "/bin/cat" inputs @stdout=out
}

file joined <"joined.txt"> = cat(glob("*.txt"));
```

Here is an example of an app function that sleeps for an arbitrary amount of time:

```
app (void signal) sleep (int secs) {
  "/bin/sleep" secs
}

foreach time in [1:5] {
  void signal = sleep(time);
  // Wait on output signal so that trace occurs after sleep
  wait(signal) {
    trace("Slept " + fromint(time));
  }
}
```

14.3.1. App retry

Some user application codes fail for non-deterministic reasons. To have Turbine automatically retry to run app functions `N` times, simply set environment variable `TURBINE_APP_RETRIES` to `N`.

Consider this Swift script (`false.swift`):

```
app f() { "false" ; }
f();
```

This is the behavior with `TURBINE_APP_RETRIES` enabled:

```
$ export TURBINE_APP_RETRIES=3
$ export TURBINE_LOG=1
$ swift-t ~/mcs/ste/false.swift
0.061 exec: false {}
0.063 shell: false
0.068 shell: Command failed with exit code: 1: retries: 1/3 on: hostname
0.072 shell: false
0.076 shell: Command failed with exit code: 1: retries: 2/3 on: hostname
0.230 shell: false
0.234 shell: Command failed with exit code: 1: retries: 3/3 on: hostname
0.567 shell: false

Swift: app execution failed on: hostname
shell: Command failed with exit code: 256
command: false

Swift: Aborting MPI job...
```

A randomized, exponential backoff delay is applied between tries.

14.4. Remote job execution with Coasters

Note: Swift/T and Coasters integration is a work in progress and is currently best suited for advanced users. Planned future changes will make it easier to install and use.

Swift/T supports execution of command-line app functions on a wide range of clusters, clouds, and grids with the Coaster executor. In order for an app function to be executed through coasters, the annotation `@dispatch=COASTER` must be added to the app function definition:

```
@dispatch=COASTER
app (file out) echo (string args[]) {
  "echo" args @stdout=out
}
```

Running a Swift/T script with Coasters function requires a few steps to set up all components:

- Before starting, you must have Swift/T compiled with Coaster support, and the coaster service from Swift/K installed.
- Before running your Swift/T script, start a Coaster service from the shell, for example:

```
export WORKER_MODE=local
export IPADDR=127.0.0.1
export SERVICE_PORT=53363
export JOBSPERNODE=4
export LOGDIR=/home/user/swift-logs
export WORKER_LOG_DIR=/home/user/swift-logs

coaster-service -nosec -port ${SERVICE_PORT}
```

- Coaster configuration must be set in the the `TURBINE_COASTER_CONFIG` environment variable, for example:

```
export
TURBINE_COASTER_CONFIG="jobManager=local,maxParallelTasks=4,coasterServiceURL=${IPADDR}:${SERVICE_PORT}"
```

- Once the Coaster service is running and `TURBINE_COASTER_CONFIG` is set, you can run your Swift/T program in the normal way, and any coaster app tasks will be dispatched to the Coaster service for execution.

Configuration keys include:

`coasterServiceURL`
the url of the coaster service to submit tasks through, e.g. `localhost:63001`. Default is `127.0.0.1:53001`.

`jobManager`

the Coaster job manager the service should use to submit tasks. E.g. to execute jobs locally, the job manager can be set to `local` and to execute jobs on resources managed through a batch scheduler such as PBS or Slurm, the job manager should be set to `pbs`, `slurm`, or the appropriate scheduler.

`maxParallelTasks`

the maximum number of concurrent tasks per Coaster worker. Default is 256.

Other settings

additional configuration keys are passed through to the Coaster service.

For more information on configuring and using Coasters, please refer to the [Swift/K documentation](#).

14.5. Custom App Executors (Advanced)

It is possible to extend Swift/T with additional app executors. To implement an executor, you need to write C code implementing the `turbine_executor` interface (defined in `turbine/executors/exec_interface.h`). Your `turbine_executor` implementation is registered at runtime by calling the `turbine_add_async_exec` function. The details of the interface are documented in `exec_interface.h`.

You also need to register the executor with Swift by adding a `appexecdef` statement in your Swift source code, typically in a module that is imported to enable your executor. The `appexecdef` statement provides a Tcl template that is used to start jobs in your executor. You must implement this Tcl function as part of implementing the executor. For example, the coaster executor is defined as follows:

```
pragma appexecdef COASTER "turbine" "0.8.0"
    "turbine:async_exec_coaster <<cmd>> <<args>> <<stage_in>> <<stage_out>>
<<props>> <<success>> <<failure>>";
```

The arguments are:

`cmd`

a string with the executable to run

`args`

a list of command-line arguments to pass to the executable

`stage_in`

a list of files to stage in (currently unused)

`stage_out`

a list of files to stage out (currently unused)

`props`

a Tcl dictionary of properties to pass to the executor. This includes `stdin`, `stdout`, `stderr` for input/output redirects and any executor-specific options.

`success/failure`

Tcl code that is executed on success or failure. Your executor implementation only needs to save these values and return them once the app completes (this is documented in `exec_interface.h`).

14.6. External scripting support

14.6.1. Calling Python

You can evaluate arbitrary Python code from within Swift/T. For example, you can perform processing with a Python library. Once you have that working, you can use Swift/T to coordinate concurrent calls to that library.

Consider the following Swift script:

```
import io;
import python;

i = python("print(\"python works\")", "repr(2+2)");
printf("i: %s", i);
```

The `python()` function takes two string arguments, *code* and an *expression*. This simply executes the Python code, then returns the expression as a Swift string. The expression must evaluate to a Python string. The expression argument may be omitted if desired, in which case the return value is always the empty string. The expected output is shown below:

```
python works
i: 4
```

Swift multi-line strings may be used to enter more complex Python code without the explicit use of `\n`.

Additionally, you can call Python libraries such as [Numpy](#) if available on your system. The following code adds matrices $I_3 + I_3$ using Numpy arrays.

```
import io;
import python;
import string;

global const string numpy = "from numpy import *\n\n";

typedef matrix string;

(matrix A) eye(int n)
{
    string command = sprintf("repr(eye(%i))", n);
    matrix t = python_persist(numpy, command);
    A = replace_all(t, "\n", "", 0);
}

(matrix R) add(matrix A1, matrix A2)
{
    string command = sprintf("repr(%s+%s)", A1, A2);
    matrix t = python_persist(numpy, command);
    R = replace_all(t, "\n", "", 0);
}

matrix A1 = eye(3);
matrix A2 = eye(3);
matrix sum = add(A1, A2);
printf("2*eye(3)=%s", sum);
```

An Python script template is created that imports Numpy and performs some simple calculations.

This code is represented in a Swift string. The template is filled in by the Swift call to `sprintf()`. Then, the code is passed to Python for evaluation. The output is:

```
2*eye(3)=array([[ 2.,  0.,  0.],
                 [ 0.,  2.,  0.],
                 [ 0.,  0.,  2.]])
```

Python state

The `python()` function resets the Python interpreter at the end of each task. If you want to access state from a previous Python task, use `python_persist()`, which leaves the Python interpreter in memory. Global variables and other state in Python will be accessible from task to task. When using Numpy, always use `python_persist()`, as Numpy cannot be re-initialized.

Exceptions

Normally, uncaught exceptions are caught by Swift/T, reported via a Python stack trace, and cause Swift/T to abort. To run through these exceptions, provide an optional 3rd argument to `python()` or `python_persist()`: `exceptions_are_errors=false`. The exception stack trace will still be reported, but the function will return successfully with value `"__EXCEPTION__"`.

Note To use this, Turbine must be [configured and compiled with Python enabled](#). This feature is implemented by linking to Python as a shared library, enabling better performance than calling the `python` program (which may be done by using a normal Swift [app function](#)). Error messages for minor Python coding mistakes may be badly mangled and refer to missing Python symbols- refer to the first error in the Python stack trace. Due to the Python C API, error reports may be slightly better for the code argument and worse in the expression argument.

14.6.2. Calling R

Consider the following Swift script:

```
import io;
import string;
import R;

global const string template =
"""
  x <- %i
  a <- x+100
  cat("the answer is: ", a, "\\n")
""";

code = sprintf(template, 4);
s = R(code, "toString(a)");
printf("the answer was: %s", s);
```

An [R language](#) script template is placed in a Swift string. The template is filled in with the value 4 by the Swift call to `sprintf()` (note the `%i` conversion specifier). Then, the code is passed to R for evaluation. The output is:

```
the answer is: 104
the answer was: 104
```

As coded here, both R and Swift report the value of `a`.

The `R()` function takes two string arguments, `code` and an *expression*. This simply executes the R code, then returns the expression as a Swift string. The expression must evaluate to a R string. The expression argument may be omitted if desired, in which case the return value is always the empty string.

R state

State is always maintained in the R interpreter, so global variables can be accessed from task to task.

Exceptions

Normally, uncaught exceptions are caught by Swift/T, reported via a R stack trace, and cause Swift/T to abort. To run through these exceptions, provide an optional 3rd argument to `R()`: `exceptions_are_errors=false`. The exception stack trace will still be reported, but the function will return successfully with value `"__EXCEPTION__"`.

Note To use this, Turbine must be [configured and compiled with R enabled](#). This feature is implemented by linking to R as a shared library, enabling better performance than calling the `R` program (which may be done by using a normal Swift [app function](#)).

14.6.3. Calling Julia

Consider the following Swift script:

```
import io;
import julia;
import string;
import sys;

start = clock();
f =
"""
begin
  f(x) = begin
    sleep(1)
    x+1
  end
  f(%s)
end
""";
s1 = julia(sprintf(f, 1));
s2 = julia(sprintf(f, 2));
s3 = julia(sprintf(f, 3));
printf("julia results: %s %s %s", s1, s2, s3);
wait(s1, s2, s3) {
  printf("duration: %0.2f", clock()-start);
}
```

In this example, a [Julia](#) script is placed in string `f`. It is parameterized three times by `sprintf()`. Each Julia invocation runs concurrently (if enough processes are provided to Swift/T).

Note To use this, Turbine must be [configured and compiled with Julia enabled](#). This feature is implemented by linking to Julia as a shared library, enabling better performance than calling the `julia` program (which may be done by using a normal Swift [app function](#)).

14.6.4. Calling JVM languages

Many languages based on the Java Virtual Machine may be accessed directly from Swift/T, including Groovy, JavaScript, Scala, and Clojure.

Groovy

The `groovy()` function accepts one string argument, a fragment of Groovy code. The value returned to Swift/T is whatever was put on `stdout`:

```
s1 = groovy("println \"GROOVY WORKS\"");
trace(s1);
```

thus, `s1="GROOVY WORKS"`.

JavaScript

The `javascript()` function accepts one string argument, a fragment of JavaScript code. The value returned to Swift/T is whatever was put on `stdout`:

```
s2 = javascript("print(\"JAVASCRIPT WORKS\");");
trace(s2);
```

thus, `s2="JAVASCRIPT WORKS"`.

Scala

The `scala()` function accepts one string argument, a fragment of Scala code. The value returned to Swift/T is whatever was put on `stdout`:

```
s3 = scala("println(\"SCALA WORKS\")");
trace(s3);
```

Clojure

The `clojure()` function accepts two string arguments, a fragment of Clojure code that returns nothing and a fragment of Clojure code that returns a string value. The string value is returned to Swift/T:

```
s4 = clojure("\"CLOJURE SETUP\"", "\"CLOJURE WORKS\"");
trace(s4);
```

thus, the value `"CLOJURE SETUP"` is discarded by the interpreter and `s4="CLOJURE WORKS"`.

Note To use the JVM, Turbine must be [configured and compiled with the JVM languages module](#). This feature is implemented by linking to the JVM as a shared library, enabling better performance than calling the `java` program (which may be done by using a normal Swift [app function](#)).

14.7. Parallel tasks: Libraries

Swift/T can be used to invoke parallel libraries by creating a communicator for them with `MPI_Comm_create_group()` and passing this to a user Tcl function that accepts this communicator. See the [Swift/T Leaf Function Guide](#) for more details. This method works well for software that is easy to invoke as a library.

14.8. Parallel tasks: External programs

If you cannot invoke your parallel code as a library, you may use the `launch()` function to invoke a parallel code

15. More about functions

In this section we discuss more advanced features for defining and calling functions.

15.1. Function call annotations

Swift/T supports many annotations to influence the behavior of function calls.

15.1.1. Priority

Leaf tasks resulting from Swift dataflow may be prioritized by using the `@prio` annotation:

```
foreach i in [0:n-1] {
  @prio=i f(i);    // or
  int j = @prio=i f(i);
}
```

In this case, `f()` will operate on higher values of `i` first. Priority is best-effort; it is local to the ADLB server. The values of `i` may be any Swift integer.

This annotation is applied to the leaf task call.

15.1.2. Location

Leaf tasks resulting from Swift dataflow may be assigned to a given processing location by using the `@location` annotation:

```
foreach i in [0:n-1] {
  location L = locationFromRank(i);
  @location=L f(i);
}
```

In this case, each `f(i)` will execute on a different worker. This annotation is applied to the leaf task call.

The `location` type may constructed by the `location()` builtin, which has the signature:

```
location L = location(rank, HARD|SOFT, RANK|NODE);
```

- `rank` is the MPI rank.
- `HARD` indicates that the location specification must be met strictly.
- `SOFT` indicates that the location specification may be ignored if there is otherwise no work available for a worker. This is noticeable during startup and shutdown, when there is typically limited available work for workers.
- `RANK` indicates that the task must be run on the given `rank`.
- `NODE` indicates that the task may be run on any rank running on the same node as the given `rank`.

See the [Location library](#) for usage for helper functions to work with locations.

Hostmap

At startup, by default, Turbine obtains all hostnames used in the run and builds up a data structure called the hostmap to map hostnames to ranks. You may combine the location features with the hostmap features to send work to assigned hostnames. See the [Location library](#) for usage. The hostmap may be [debugged](#) or [disabled](#).

15.2. Advanced function topics

Swift/T provides various facilities to define functions with flexible input and output types that can enable writing cleaner and more generic code.

15.2.1. Optional and keyword arguments to functions

Swift/T functions can have optional arguments to functions, where a default value is used if the caller does not provide that function argument. The default values are specified in the function definition and must be literal constant expressions. For example, the following function has two optional arguments:

```
(string o) msg(string a, int b=0, float c=0.0) {
    o = "%s %i %f" % (a, b, c);
}
```

Optional arguments can be provided as **positional arguments** in the same way as non-optional arguments, e.g.:

```
trace(msg("test"));
trace(msg("test", 1));
trace(msg("test", 1, 2.0));
```

Optional arguments, unlike non-optional arguments, can also be provided as **keyword arguments**. For example:

```
trace(msg("test", y=2.0));
```

15.2.2. Variable-length argument lists

[Swift extension functions](#) support variable length argument lists, where the final argument can be repeated 0 or more times. The full list of arguments is passed into the extension function definition. For example, it is possible to define a function that takes any number of integers as arguments and returns the sum. Let us assume that a Tcl function `my_sum` is defined that computes the sum of its inputs. Then the following Tcl extension function definition is possible:

```
(int o) my_sum(int... vals) "my_pkg" "1.0" [
    "set <o> [ my_pkg:my_sum <<vals>> ]"
];
```

15.2.3. Function overloading

Swift/T supports *overloading* of functions, where multiple function definitions with the same name coexist in the program. If a function name is overloaded, Swift/T will decide which definition to use based on the argument types. In order to overload a function, the input types of the two definitions must be different enough to allow Swift/T to reliably determine which definition is meant. This means that you can only overload functions if:

- All input arguments have a single concrete type, e.g. `file` or `int[] []`. Union argument types and type variables are not supported in overloaded functions.
- No optional arguments are used (variable-length argument lists are supported)
- No list of input types could match multiple possible definitions, for example `f(1)` could match both `f(int x)` and `f(int x, float y...)`.

Swift/T will exit with a compile error if you break one of these rules.

For example, the following program will print `int: 1` and `float: 3.14`.

```
import io;

() print_num(int x) {
    printf("int: %i", x);
}

() print_num(float x) {
    printf("float: %0.2f", x);
}

print_num(1);
print_num(3.14);
```

15.2.4. Union argument types and type variables

[Swift extension functions](#) support additional argument types that can match multiple possible input types.

TODO

16. Optimizations

STC performs a range of compiler optimizations that can significantly speed up most Swift programs. The optimization level can be controlled by the `-O` command line option. The default optimization level `-O2`, or the increased optimization level `-O3` are usually the best choices. Some applications benefit markedly from `-O3`, while others do not, and compile times can increase slightly.

```
# No optimizations at all (not recommended)
stc -O0 example.swift
```



```
# Basic optimizations (not recommended)
stc -O1 example.swift

# Standard optimizations (recommended)
stc example.swift example.tcl
# OR
stc -O2 example.swift example.tcl

# All optimizations (also recommended)
stc -O3 example.swift example.tcl
```

Individual optimizations can be toggled on using `-f <opt name>` or off with `-F <opt name>`, but this typically is only useful for debugging. You can find an up-to-date list of optimizations in the `stc` command-line help:

```
stc -h
```

17. Running in Turbine

The following describes how to run Turbine programs.

17.1. Architecture

Turbine runs as an MPI program consisting of many processes. Turbine programs are ADLB programs. Thus, they produce and execute discrete tasks that are distributed and load balanced at run time.

Each process runs in a *mode*: *worker*, or *server*.

Workers

Evaluate the Swift logic. Produce tasks. Execute tasks.

Servers

Distributes tasks. Manages data.

Typical Swift programs perform compute-intensive work in leaf functions that execute on workers. Execution of the Swift logic is split and distributed among workers.

Servers distribute tasks in a scalable, load balanced manner. They also store Swift data (integers, strings, etc.).

17.2. Concurrency

The available concurrency and efficiency in your Swift script is limited by the following factors:

- The available concurrency in the Swift logic. Sequential dependencies will be evaluated sequentially. `foreach` loops and branching function calls may be evaluated concurrently
- The number of workers available to process leaf functions concurrently
- The number of servers available to control the Turbine run. Adding more servers can improve performance for applications with small tasks or complex data dependencies but ties up processes

17.3. Invocation

The form of a Turbine invocation for STC-generated `program.tic` is:

```
turbine <turbine arguments> <program.tic> <program arguments>
```

The program arguments are available to Swift ([\[argv\]](#)).

Turbine accepts the following arguments:

```
-f <file>
    Provide a machine file to mpiexec

-h
    Print a help message

-l
    Enable mpiexec -l ranked output formatting

-n <procs>
    The total number of Turbine MPI processes

-v
    Report the Turbine version number

-V
    Make the Turbine launch script verbose

-x
    Use turbine_sh launcher with compiled-in libraries instead of tclsh (reduces number of files
    that must be read from file system)

-X
    In place of of program.tic, run standalone Turbine executable (created by mkstatic.tcl)
```

17.3.1. Environment variables

The user controls the Turbine run time configuration through environment variables:

ADLB_SERVERS

Number of ADLB servers

The remaining processes are workers. These values are available to Swift ([Turbine information](#)).

TURBINE_<type>_WORKERS

Number of workers of specified type.

Any workers not allocated to specific types are general-purpose workers that execute Swift/T control code and CPU-based tasks. There must be at least one leftover worker to serve as a general-purpose worker.

Valid work types include:

- `TURBINE_COASTER_WORKERS::` for Coaster workers.
- A work type defined with `pragma worktypedef` in Swift. E.g. if you define `pragma worktypedef a_new_work_type;"` in Swift, the environment variable is `+TURBINE_A_NEW_WORK_TYPE_WORKERS`.

Generally you will need to allocate workers for any work type used in your Swift program.

TURBINE_LOG=0

Disable logging. `TURBINE_LOG=1` or unset enables logging, assuming logging was not disabled at configure time. Logging goes to standard output by default.

SWIFT_PATH

TURBINE_PATH

Space-separated list of Swift/T extension (Tcl) package locations. Either variable name may be used.

TURBINE_USER_LIB

Alias for SWIFT_PATH. Deprecated but retained for compatibility.

SWIFT_TMP

TMP

Directory location for [unmapped files](#) and `mktemp()`. SWIFT_TMP overrides TMP. The default is /tmp.

SWIFT_TMP_AUTODELETE=0

By default, Swift/T deletes [unmapped files](#) when they are no longer used. Setting this to 0 logs these temporary files when created and prevents Swift/T from automatically deleting them.

TURBINE_LOG_FILE=<file>

Set log file location. Defaults to standard output.

TURBINE_LOG_RANKS=1

Using `turbine -1` or equivalent prepends the MPI rank number to each output line. This works with typical MPICH or OpenMPI systems, however, this is not available on some systems, so set this to emulate the rank output on such systems.

ADLB_PRINT_TIME=1

Enable a short report of total elapsed time (via `MPI_Wtime()`)

ADLB_PERF_COUNTERS=1

Enable performance counters (printed at end of execution). The [Swift/T internals guide](#) has information about interpreting the output.

ADLB_EXHAUST_TIME

Time in seconds taken by ADLB task servers to shut down. May include a decimal point. Default 0.1. Setting this lower will reduce delay in detection exhaustion. Setting this higher will reduce overhead due to failed exhaust checks. The default setting is almost always adequate.

ADLB_REPORT_LEAKS=1

Enable reporting of any unfreed data in ADLB data store at end of execution.

ADLB_TRACE=true

ADLB_DEBUG=true

To print DEBUG/TRACE level information for ADLB (if ADLB was compiled with it enabled)

TURBINE_LAUNCH_OPTIONS

Provide other arguments to `mpirun`.

TURBINE_SRAND

If unset or empty, the random number generator seed will be set to the process rank for each process, giving reproducible results. If set to an integer *seed*, the random number generator seed for each process will be set to *seed* + *rank*.

For non-reproducible random results, use the following shell commands:

```
export TURBINE_SRAND=$( date +%s )
turbine ...
```

The seed is recorded in the log.

TURBINE_APP_RETRIES

If set to a positive integer, `app` functions will be retried that many times they produce exit code 0. Cf. [App retry](#).

TURBINE_APP_DELAY

If set to a positive floating point number *N*, `app` functions will be delayed a number of seconds drawn uniformly between 0 and *N*.

ADLB_DEBUG_RANKS=1

Enable a report showing the rank and hostname of each process. This allows you to determine whether your process layout on a given machine is as intended.

ADLB_DEBUG_HOSTMAP=1

Enable a report showing the hostmap, which maps hostnames to ranks for use with the location functionality.

ADLB_DISABLE_HOSTMAP=1

Prevent the hostmap from being constructed (avoiding some communication overhead).

ADLB_PLACEMENT

Change the placement policy. Alternatives are: random (place data on a random server - the default) and local (place data on the local server).

VALGRIND

Enable valgrind debugging. Cf. [valgrind](#).

GDB_RANK

Enable GDB debugging. Cf. [GDB](#).

TURBINE_MPI_THREAD=1

Enables `MPI_THREAD_MULTIPLE`. Allows exotic user tasks that use MPI functions while using threads.

17.4. Build configuration

The following describes how to run Swift/T programs in Turbine on more complex systems.

17.4.1. Build troubleshooting

If `build-all.sh` does not succeed, you may need to change how it tries to configure and compile Swift/T.

Troubleshooting a build problem can require a few steps. The first step is to determine why the build failed. `build-all.sh` will usually report the step at which configuration failed. For example, if it was unable to locate a valid Tcl install, it will report this. Then you can try these steps to resolve the problem:

1. If your system is covered by the [Sites Guide](#), check to see if the problem and solution are described there.
2. Inspect `swift-t-settings.sh` settings related to the reported problem. For example, if locating a Tcl install failed, setting the `TCL_INSTALL` and `TCL_VERSION` variables to the correct location and version may help.
3. If the options in `swift-t-settings.sh` do not give sufficient control to fix the problem, you may need to manually configure some components of Swift/T, as described in the next section.

17.4.2. Manual configuration

`build-all.sh` and `swift-t-settings.sh` provide a convenient way to install Swift/T from the downloadable package or from a Git clone. However, this method does not allow full control over the configuration. Swift/T is built with standard Ant (Java) and Autotools/Makefile (C, Tcl) techniques. You can more directly control the configuration when building through the arguments to `ant` or `configure`.

To perform the installation using `configure/make`, simply untar the distribution package or clone from [GitHub](#) and do:

```
cd c-utils/code
./configure ...
make install

cd ../../lb/code
./configure ...
make install

cd ../../turbine/code
./configure ...
make install

cd ../../stc/code
ant install -Ddist.dir=... -Dturbine.home=...
```

Note | Use `./configure --help` and the [Sites Guide](#) for further options.

To obtain the latest source from GitHub, do:

```
git clone https://github.com/swift-lang/swift-t.git
```

Makefile debugging

Use `make V=1` to get verbose output from our Makefiles.

17.4.3. Non-standard MPI locations

Sometimes simply specifying the MPI directory is not enough to configure Swift/T.

You can modify these settings in `swift-t-settings.sh` to more precisely define locations of MPI resources:

```
EXM_CUSTOM_MPI=1
MPI_INCLUDE=/path/to/mpi.h/include
MPI_LIB_DIR=/path/to/mpi_lib/lib
MPI_LIB_NAME=funny.mpi.a
```

If you are following the manual build process, configure Turbine with:

```
--enable-custom-mpi
--with-mpi-include=/path/to/mpi.h/include
--with-mpi-lib-dir=/path/to/mpi_lib/lib
--with-mpi-lib-name=funny.mpi.a
```

17.4.4. External scripting

Python

To build Swift/T with Python, either:

- When using `build-all.sh`: Set `ENABLE_PYTHON=1` in `swift-t-settings.sh`.
- When doing a [manual build](#) provide the `--enable-python` argument to `configure`. See `./configure --help` for further options.
- When running, you may need to set `PYTHONPATH` to the installation directory. For example:

```
export PYTHONPATH=$HOME/Python-2.7.6/lib/python2.7
```

R

To build Swift/T with R, either:

- When using `build-all.sh`: set `ENABLE_R=1` in `swift-t-settings.sh`.
- When doing a [manual build](#) provide the `--with-r` argument to `configure`. See `./configure --help` for further options.

Other notes:

- When installing R from a binary package, be sure to include the `devel` package.
- When installing R from source, configure R with `--enable-R-shlib`.
- When running, you may need to set the environment variable `R_HOME` to the directory containing the R installation. For the APT package, this is `/usr/lib/R`.
- When running, you may need to set the environment variable `LD_LIBRARY_PATH` to include the directory containing the R shared library. For a source R build on Linux, this is `R_HOME/lib/R/lib`.

Julia

To build Swift/T with Julia, either:

- When using `build-all.sh`: set `ENABLE_JULIA=1` in `swift-t-settings.sh`.
- When doing a [manual build](#) provide the `--enable-julia` argument to `configure`. See `./configure --help` for further options.

JVM languages module

- Using the JVM languages currently requires a [manual build](#).
- Clone the Swift/T JVM Engine with:

```
cd turbine/code
git clone https://github.com/isislab-unisa/swift-lang-swift-t-jvm-engine.git \
swift-t-jvm
```

- Build the Swift/T JVM Engine with:

```
cd swift-t-jvm
./bootstrap
./configure
make
```

- Build Turbine with:

```
cd turbine/code
./configure --enable-jvm-scripting
make install
```

- At runtime, set environment variable `LD_LIBRARY_PATH` to contain the JVM lib and the Swift/T JVM Engine lib (where `SWIFT_T` is your Swift/T source directory):

```
export LD_LIBRARY_PATH=
LD_LIBRARY_PATH+=/usr/lib/jvm/java-8-openjdk-amd64/jre/lib/amd64/server:
LD_LIBRARY_PATH+=SWIFT_T/turbine/code/swift-t-jvm/src/.libs
```

- At runtime, set environment variable `SWIFT_JVM_USER_LIB` to contain Swift/T JVM Engine classes (where `SWIFT_T` is your Swift/T source directory):

```
export SWIFT_JVM_USER_LIB=SWIFT_T/turbine/code/swift-t-jvm/swift-jvm/swift-
jvm-build/target/swift-jvm-build-0.0.1-bin/swift-jvm/classes
```

17.5. Performance enhancements

1. Disable logging/debugging via environment
2. Disable logging/debugging at configure/compile time
 - Configure c-utils with `--disable-log`
3. Specify `EXM_OPT_BUILD=1` in `swift-t-settings.sh` or configure everything with `--enable-fast`. This disables assertions and other checks.
4. When making performance measurements, always subtract 0.1 seconds (or the value of `ADLB_EXHAUST_TIME`) from the Turbine run time due to the ADLB shutdown protocol, which does not start until the system is idle for that amount of time.
5. Reduce the number of program files that must be read off the filesystem. This is particularly useful for parallel file systems and large scale applications. In increasing order of effectiveness, you can:
 - use the `turbine_sh` launcher in place of `tcsh` in submit script, or by specifying the `-x` argument to `turbine`
 - Use `mkstatic.tcl` to create a standalone executable with the Tcl main script and Tcl library code compiled in, and compiled code statically linked.

17.6. Building standalone executables with `mkstatic.tcl`

It is possible to build a fully self-contained executable, including all Tcl scripts and compiled code, provided that all dependencies support static linking. If not, it is also possible to build an executable with a subset of Tcl scripts and code linked in, providing some performance benefits.

The provided `mkstatic.tcl` utility can produce a C source file with Tcl scripts bundled in, which can then be compiled and linked with a C compiler. This is a multi-step process that can be automated as part of your build process.

Note Ensure that static versions of the `c-utils`, `lb`, and `turbine` libraries were built, typically with a `.a` suffix, e.g. `libadlb.a`. These are created by default, unless you specified `DISABLE_STATIC=0` or `--disable-static`. To build a fully standalone executable, you will also need to build a static version of Tcl (with the `--disable-shared` configure option), and static versions of any other libraries your own code needs to link with, such as your MPI distribution or application code.

1. Compile your Swift script

```
stc my.swift
```

producing the Turbine Tcl script `my.tic`.

2. Create a manifest file, e.g. `my.manifest`. This file describes the resources to be bundled, including the STC-generated code and any user libraries.
To do this, make a copy of `scripts/mkstatic/example.manifest` from the Turbine installation directory. This file contains examples and descriptions of all the possible settings. Note that an empty manifest file corresponds to the `turbine_sh` utility, which is a replacement for `tcsh` with required Turbine libraries statically linked in. For a simple Swift program with no user Tcl libraries, you only need to set `main_script = my.tic`.
3. Invoke `mkstatic.tcl` (found under `scripts/mkstatic/mkstatic.tcl` in the Turbine installation) to translate your Tcl script to a C main program (e.g., `my_main.c`) with Tcl source code included. The minimal invocation is

```
mkstatic.tcl my.manifest -c my_main.c
```

You will likely wish to include Tcl system libraries with `--include-sys-lib /home/example/tcl-install/lib --tcl-version 8.6`. The Tcl system library directory can be identified by the fact that it contains the file `init.tcl`. This directory must be specified with a special flag so that `mkstatic.tcl` can correctly replace the regular Tcl initialization process.

You can include additional libraries and packages with `--include-lib /home/example/tcl-lib/`. Any `.tcl` or `.tm` source files in the directory will be included. Source-only packages can generally be completely linked into the executable, but if a package loads shared libraries, only the `pkgIndex.tcl` file will be linked into the executable. A package with compiled code can be converted to support static linking by specifying a package init function, plus static library or object files in the manifest file.

4. Link together the compiled C main program with user libraries and Swift/T libraries to produce a final executable. The details of the process vary depending on the compiler and system: we assume GCC. You will need to provide the correct flags to link in all libraries required by Swift/T or your own user code.
 - **User code:** you must identify the libraries used by your application and ensure link flags are provided. If linking static libraries, ensure that any indirect dependencies of these libraries are also linked.
 - **Swift/T system:** The Turbine distribution includes a helper script, `turbine-build-config.sh`, that can be sourced to obtain linker flags for Swift/T dependencies.
 - **Link order:** In the case of static linking, if `libA` depends on `libB`, then the `-lA` flag must precede `-lB` on the command line. To actually do the linking, there are two further cases to consider:

- If building a fully static executable, you can provide the `-static` flag, plus all object files, plus `-L` and `-l` flags for all required library directories and libraries. This requires that all libraries have static archives (`*.a`), including Tcl (simply build Tcl with `--disable-shared`).

```
gcc -static script_main.c file1.o file2.o -L/path/to/lib/dir -lsomething ...
```

- If you are building an executable that depends on one or more shared libraries, you will need to provide the `-dynamic` flag, and then ensure that static libraries are linked statically. If a shared version of a library is available, `gcc` will use that in preference to a static version. You can override this behaviour by specifying `-Wl,-Bstatic` on the command line before the flags for the libraries you wish to statically link, then `-Wl,-Bdynamic` to reset to dynamic linking for any libraries after those.

We have described the most commonly-used options. A full list of options and descriptions can be obtained by invoking `mkstatic.tcl -h`. Additional options include:

```
--main-script      Specify Tcl main script (overrides manifest file)
-r                Specify non-standard variable prefix for C code
-v                Print verbose messages
--deps            Generate Makefile include for generating C file
--ignore-no-manifest Pretend empty manifest present
```

17.7. Debugging Swift/T runs

Applying the debugger allows you to debug native code linked to Swift/T from a normal debugger. This allows you to step through your leaf function code (and the Swift/T run time libraries).

- When using Swift/T dynamically with Tcl packages (the default), you need to attach to the `tclsh` process. This process loads your native code and calls into it.
- When using `mkstatic`, you generate a complete executable. You can debug this in the normal method for debugging MPI programs.

17.7.1. Valgrind

The Swift/T launcher scripts support [valgrind](#). Simply set the environment variable `VALGRIND` to the valgrind command you wish to use. A suppressions file is distributed with Turbine to ignore known issues. (Swift/T is valgrind-clean but there are some issues in the libraries we use.)

```
export VALGRIND="valgrind --suppressions=SWIFT_T/turbine/code/turbine.supp"
swift-t program.swift
```

17.7.2. GDB

The Turbine library provides a convenient attachment mechanism compatible with debuggers like GDB, Eclipse, etc. You attach to a Turbine execution by using the `GDB_RANK` variable:

```
$ export GDB_RANK=0
$ swift-t program.swift
Waiting for gdb: rank: 0 pid: 23274
...
```

Rank 0, running in process 23274, has blocked (in a loop) and is waiting for the debugger to attach. When you attach, set the variable `t=1` to break out of the loop. Then you can debug normally.

17.8. Profiling with the Message Passing Environment (MPE)

The Message Passing Environment ([MPE](#)) is a profiling library for MPI-based applications like Swift/T. By enabling MPE, you can obtain a wealth of profiling information about Swift/T internals and your user tasks.

MPE works closely with MPI, it intercepts MPI API calls to capture the profiling data.

17.8.1. Compiling for MPE

(Thanks to Azza Ahmed for producing these instructions.)

1. **Install MPE.** This starts by downloading the library, for example from the [official site](#). Installing from source is ideally straight forward, except that one also needs to generate a shared library file `libmpe.so`, which the Swift/T installer expects to find in the `mpe_installation_dir/lib`. The final command looks something like:

```
$ ./configure CC=mpicc F77=mpifort --prefix=<mpe_installation_dir> MPI_LIBS=-lpthread --enable-PIC
$ make
$ make install
$ make installcheck # 4 yeses is what you are looking for!
# Then, add <mpe_installation_dir>/bin to your PATH
```

Notes about the configuration parameters above:

- `CC` and `F77` are program to compile MPI C programs and Fortran programs, and they are mandatory parameters when building MPE from source. When working with default `mpicc` these are fine.
- `MPI_LIBS=-lpthread`: This parameter is an MPI requirement. On some systems machine, regular MPI programs do not compile successfully without this library. Therefore, you may need to include it in the MPE build process as well.
- `--enable-PIC` is necessary to build the shared library (`libmpe.so`), which the Swift/T installer expected to find in `installation_path/lib`.

Once MPE is installed properly, you will need to run code similar to this: [MPICH ticket #1104](#) to create the shared library (`.so`) file. This is a work around, as MPE does not support shared libraries, even if you configure with `--enable-shared`.

2. **Install Swift/T C-utils.** Install as usual.
3. **Install ADLB.** Configure with:

```
$ export CFLAGS=-mpilog
$ export LDFLAGS="-L/<mpe_installation_dir/lib> -lmpe \
```

```
-Wl,-rpath -Wl,/<mpe_installation_dir>/lib"
$ ./configure --prefix=... CC=mpecc \
              --with-mpe=<mpe_installation_dir> \
              --with-c-utils=...
$ make install
```

4. **Install Turbine.** Configure with:

```
$ unset CFLAGS LDFLAGS
$ ./configure --with-mpe=<mpe_installation_dir> <Turbine arguments...>
$ make install
```

5. **Install STC.** Install as usual.

17.8.2. Runtime usage

Now, when you run, ADLB will create a MPE CLOG file in `PWD`. You can process it with the normal MPE tools such as Jumpshot. It will contain customized ADLB-specific information. See ADLB [mpe-tools.c](#) for the list of events that are captured. These correspond to functions with the similar names in [adlb.h](#), such as task puts and gets, and data stores and retrieves.

17.8.3. Log processing

Swift/T provides some tools to process MPE logs. They are available and [documented here](#). See also [this paper](#).

17.9. Startup/shutdown hooks

Turbine has hooks that can be accessed to execute shell scripts, Tcl code, or C/C++ code on each node at startup and shutdown.

17.9.1. Shell or Tcl hooks

By simply setting environment variables, you can trigger fragments of Tcl or shell code to execute on each node of the Swift/T run at startup and/or shutdown. In this example:

```
$ export TURBINE_LEADER_HOOK_STARTUP=" puts [ exec $PWD/hook-startup.sh ]"
$ export TURBINE_LEADER_HOOK_SHUTDOWN="puts [ exec $PWD/hook-shutdown.sh ]"
$ swift-t workflow.swift
```

the Tcl fragments in double quotes will be run at startup and shutdown respectively. In this case, the Tcl fragments simply `exec` the given shell scripts.

These fragments are run on the *leader* ranks of each node. These are the lowest ranks on the node.

A typical use of this feature is to copy data into node-local storage before workflow execution, and remove it on workflow completion.

17.9.2. C/C++ hooks

If you are calling C/C++ native code from Swift/T, you can use the provided C/C++ shutdown hook to clean up at the end of the workflow. The function signature is in `turbine-finalizers.h`:

```
int turbine_register_finalizer(void (*func)(void*),
                              void* context);
```

The `func()` will be called once per rank at the end of the workflow. Multiple functions can be registered.

18. Model exploration workflows

The [EMEWS Tutorial](#) covers building and running advanced model exploration workflows in Swift/T.

19. Developers' guide

For more information, see the [Developers' Guide](#).

20. Publications

For more information, see the [Swift/T Publications](#).

Data Science and Learning

Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

www.anl.gov

Argonne National Laboratory is a U.S. Department of Energy
laboratory managed by UChicago Argonne, LLC