Liberty Simulation Environment Internals Manual

The Liberty Research Group

Liberty Simulation Environment Internals Manual

by The Liberty Research Group

Version 1.0 Edition

Table of Contents

Preface

> **Are you prepared?**
>
> Don’t read this book unless you are a maintainer of LSE or excessively curious about how things work. It will hurt! Also, this book assumes a thorough knowledge of the material in the User’s Manual, the Developer’s Manual, and the API Reference Guide.

Typographical conventions used in this book

The following typefaces are used in this book:

- Normal text
- *Emphasized text*
- `The name of a program variable`
- `The name of a constant`
- **The name of an LSE module**
- **The name of a package**
- *The name of an domain class*
- **The name of an domain implementation**
- **The name of an attribute in a domain implementation description file**
- **The name of an emulator**
- *The name of an emulator capability*
- *The name of a module parameter*
- **The name of a module port**
- `Literal text`
- *`Text the user replaces`*
- `The name of a file`
- The name of an environment variable
- *The first occurrence of a term*

Chapter 1. How simulators get built

This chapter describes the process by which a simulator gets built. It documents the scripts and files used. This chapter also describes the format of the "info" files used by LSS to communicate with the simulator build process.

The build process

The main build script is **ls-build**. This script takes as inputs a configuration description file written in LSS, generates code, compiles the code, and outputs a collection of libraries (within a big directory structure) for the simulator. These libraries are later linked with emulators by **ls-link** to form a simulator executable.

The steps of **ls-build** are:

1. Parse arguments.
2. Create a directory called `machines` if it is not present.
3. Backup an old machine directory, if present and requested to do so.
4. Remove old machine directory, if requested to do so.
5. Run **lss**. This program does the following:
    a. Read configuration files and determine instance structure of the configuration. Determine all types, parameter values, code functions, etc.
    b. Create the output machine directory and an `database` subdirectory.
    c. Place "information" files into the *`output_directory`*`/database` subdirectory. The format of these files is given in the Section called *Info file formats*.
    d. Additional subdirectories with additional code tarballs or files may be created, but the instance hierarchy subdirectories should not be. `SIM_instance_info.m4` can indicate a command to use to move additional code into the instance hierarchy.
6. Create the design database by reading the info files. APIs which are evaluated at code generation time are evaluated as part of the database build for all code pieces but `.clm` files; many other APIs are checked for the same code pieces.
7. Determine whether a clean rebuild is necessary. It is necessary if the user asked for it, the previous build's design database cannot be found, or the previous build failed while generating files. The way the last condition is determined is that a file named `changing_files` is used as a "lock" for generated files; it is created, then the files are created, then it is deleted. If this file is found at this point, then the previous build failed and a clean rebuild is necessary. Reads the previous design database if a clean rebuild is not needed.
8. Determine whether a new execution schedule is necessary and call **ls-schedule** to generate it if this is the case. If a new schedule was needed, run schedule post-processing.

9. Figure out how code blocks are to be invoked.
10. Determine whether the `include` directory, `framework` directory, or modules need to be rebuilt.
11. Lock the generated files by creating `changing_files`.
12. Create the `include` and `framework` directories if they need to be rebuilt. Use SHA hashes of the tarballs to double check for changes.
13. Create the instance directory structure (called `MODULES`).
14. For each module, if it needs to be rebuilt, clean out its directory, and detar the module tarball or run the command specified in `SIM_instance_info.py`. Uses SHA hashes of the tarballs to double check for changes.
15. Write the design database to `database/SIM_schedule.dat`
16. Create `SIM_inst_name.m4` files in each instance source directory needing to be rebuilt indicating the instance name.
17. Find all instance `*.mk` files for instances needing to be rebuilt and rename them to `Makefile`; if an instance does not have such a file, create a default `Makefile` for it.
18. Remove orphaned module directories
19. Write a file named `quick_make.sh` which contains make commands for each directory needing to be rebuilt.
20. Unlock the directories by deleting `changing_files` .
21. Create `Makefile` for each intermediate level of the directory structure and the top level.
22. Generate code from .clm or .m4 templates if the .c files are out of date with respect to the .clm or .m4. This step is done "early" here (as the Makefiles can do it as well) so that the design database does not need to be reloaded once for every file generated.
23. Generate a makefile defining special compilation flags required by domains. This file is called `include/Make_for_this.mk`.
24. Create the top-level `Makefile` .
25. Write the emulator style file `emulator_style`. This file indicates to **ls-link** what kind of emulator has been used so that it may look for the right set of command-line parameters. The possible values are `compile` and `interpret`.
26. Create `include/SIM_config.h` with top-level parameter macros.
27. Substitute configuration information into `include/Make_include.mk`.
28. If there is a `quick_make.sh` file and there is not a `make_aborted` file, run `quick_make.sh` to generate and compile framework and module code, otherwise run **make -e** in the machine directory. The `-e` option is used on calls to make so that environment variables can override defaults set in the makefile.

    The `make_aborted` file is used as a lock file to detect incomplete makes when using `quick_make.sh`. It is created before running `quick_make.sh` and then deletd afterwards.

The steps of **ls-link** are:

1. Parse arguments. The exact arguments needed depend upon whethat type of emulators are to be used, as given in the `emulator_style` file. Compiled emulators do not need binary names; interpreted emulators do.

   Open Issue

   How do we deal with multiple emulators with multiple binaries since the typical simulator command-line cannot?

2. Ensure that LIBERTY_SIM_LIB_PATH contains the default emulators and extensions directories
3. Copy benchmark binary if needed or discover compiled-code library for the benchmark.
4. Find all object libraries in the machine directory.
5. Find emulation libraries. The `emulator_style` lists libraries by name or `-l`; it may also include `-L` to indicate library search paths. Any `-l` or `-L` words are passed through to the linker, while other words are treated as library names and LIBERTY_SIM_LIB_PATH is used as a search path to find them.
6. Find libraries listed using the `-link_lib` option. The word following a `-link_lib` is treated just as emulation libraries are; `-l` and `-L` are passed through to the linker, while others are treated as library names and LIBERTY_SIM_LIB_PATH is used as a search path to find them.
7. Figure out linking options needed by a compiled-code benchmark.
8. Link.

Files used in the build process

This section describes various files used as inputs to the build process or generated by the build process. It begins with files which are installed by the LSE installation process (i.e. **make install**). It then proceeds in the order in which files are generated during the building of a simulator; this order is generally `include` directory, `framework` directory, and module directories. Note, however, that some files are seeded in each of these directories by the build scripts, as described in the Section called *The build process*.

Files installed by LSE

The following files are installed from the `framework` directory into *`install_dir`*`/share/lse/framework`.

`framework_code.tar`	- tarball for the framework directory
`framework_inc.tar`	- tarball for the include directory
`parmdecl.lss`	- top-level parameter definitions
`scheduler/LSE_schedule.py`	- Python code for manipulating schedules
`Make_include.mk`	- Template for common makefile rules
`SIM_analysis.py`	- Python code for analyzing the design
`SIM_apidefs.py`	- Python code for core APIs

`SIM_codegen.py`	- Python code for helping to generate C code
`SIM_database.py`	- Python code for manipulating machine data structures
`SIM_database_build.py`	- Python code to build the Python database
`SIM_debug_database.py`	- **Python** script to interactively debug the database
`SIM_rebuild.py`	- Python code for determining whether a rebuild is necessary
`SIM_tokenizer.py`	- Python code for simple parsing of C code

A program called `ls-schedule` is built and installed into *`install_dir`*`/bin`. This program does schedule analysis. The source code used for this program is in `framework/scheduler` and is named:

`scheduler.h`	- scheduler data structures
`scheduler.c`	- scheduler code
`develmain.c`	- main program for text-based (debugging) entry
`realmain.c`	- main program for entry from design databases

The following files are installed from the `scripts` directory into *`install_dir`*`/bin`.

`ls-build`	- Script to build a simulator (generated from `ls-build.in`)
`ls-create-module`	- Script to create an outline of a module
`ls-link`	- Script to link simulators with emulators
`ls-prep-bench`	- Script to create a Liberty compiled-code emulator
`ls-run-bench`	- Script to run a simulator on a benchmark

TO DO

`ls-prep-bench` and `ls-run-bench` should be moved into the `local-scripts` CVS module in the near future. They are "blank" in the standard distribution.

The following files are installed from the `include` directory into *`install_dir`*`/include/lse`.

`SIM_clp_interface.h`	- Header defining types and function a command-line processor can use
`SIM_time.h`	- Time type and accessors
`SIM_types.h`	- Basic simulator types

Domain files

LSE relies upon the presence of the following two files installed by the `scripts` module:

- *`install_dir`* `/bin/l-create-domain-header` - a Python script to create the domain header file
- *`install_dir`* `/domains/LSE_domain.py` - Base Python class for domains and a few constants

Individual domain classes install at least two files into the *`install_dir`*`/share/domains` directory. The possible files are:

- *`domain`*`.py` - a Python module which includes, at a minimum, an object with a particular name (`LSE_DomainObject`) derived from the base domain class.
- *`domain`*`.lss` - a file defining a **lss** package for the domain.
- *`domain`*`.m4` - code to implement APIs and variables added by the domain. This file is optional.

- *domain* `/`*implementation*`.lss` - a package per domain class implementation which define implementation-specific types. These files are optional.

At present the only standard domain class is *LSE_emu*, which provides all the files. The class files are provided in the `emulib` module and the implementation files in the various emulator implementation modules.

Files in the built simulator `database` directory

The `include` directory begins with the "information" files created by **lss**. Further files are created by different steps of **ls-build** in the following order:

- `SIM_schedule.dat` - The Python database; created by using `SIM_database_build.py` and calling a number of Python functions to analyze it.
- `SIM_debug_database.py` - Python script to interactively debug the database. Copied from *`install_dir`*`/share/lse/framework`.
- `SIM_schedule.in` - Input to **ls-schedule**. Contains only the signal list and values of two parameters.
- `SIM_schedule.out` - Output of **ls-schedule**. Contains only the schedule.
- `SIM_schedule.txt` - Information about the schedule created while running **ls-schedule**.

The database file (`SIM_schedule.dat`) is created with the Python `cPickle` package. The `SIM_debug_database.py` script can be used with the **python -i** command to read the file so that the database can be interactively examined.

Files in the built simulator `include` directory

ls-build detars the include directory tarball previously installed to produce the following files:

`Make_module.mk`	- Default module makefile
`SIM_all_types.h`	- Include all the data types in the system
`SIM_allcb_api.m4`	- APIs visible in all scopes not already in .h files
`SIM_control.h.m4`	- Template for core data structure definitions
`SIM_database.m4`	- **m4** code to read the Python database
`SIM_domain_types.h.m4`	- Template for domain-dependent type definitions
`SIM_dynid.h`	- Dynamic instruction ID manipulation
`SIM_framework_inc.mk`	- renamed to `Makefile`
`SIM_prefix.m4`	- Prefix template for modules
`SIM_quotes.m4`	- Set up funny quotes and utility macros we use in **m4**
`SIM_refcount.h`	- A type for reference counting
`SIM_resolution.h`	- Resolution type and accessors
`SIM_time.h`	- Time type and accessors
`SIM_types.h`	- Basic simulator types
`SIM_user_types.h.m4`	- Template for user extensions to simulator datatypes

ls-build creates the following files:

- `Make_include.mk` - common makefile definitions. This file is produced by substituting text from the template of the same name in *`install_dir`*`/share/lse`.

- `Make_for_this.mk` - definition of **make** variables for this configuration. Currently contains compile flags for domain classes and instances.

The **make** command in the include directory transforms all the `*.h.m4` files into `*.h` files using **lm4**. (**ls-build** actually attempts to perform this transformation early, before **make**, as this will save database load time.) It also creates a file called `includedir` with the full path name of the include directory. This file is included by `SIM_database.m4` to find the correct directory to search for the design database.

Structure of the include files

The include files fall into two classes we call *type headers* and *variable declaration headers*. Type headers are used to declare types and method signatures upon those types. They may also include inlinable methods. Variable declaration headers are used to declare external variables and utility functions.

Most of the header files are type headers. The type header which all non-header C source files include (after code generation) is `SIM_all_types.h`, which simply pulls in all the other type headers in the proper order. No other type header should be included directly by any other source file. The other type headers do *not* include all the headers they need; they rely instead on `SIM_all_types.h` to include all the prerequisites. This is done to reduce build time in the C pre-processor. These prerequisites are normally listed in a comment at the type of the type header's code.

The only variable declaration header (at present) is `SIM_control.h`, which is included by most non-header C source files to declare the core simulator data structures. This header file does not include all the headers it needs; it relies upon the including file to have included `SIM_all_types.h` before.

Files in the built simulator `framework` directory

ls-build creates the `framework` directory and detars the framework directory tarball previously installed to produce the following files:

`SIM_control.c.m4`	- Definitions of core data structures and utility functions
`SIM_framework_code.mk`	- Renamed to `Makefile`
`SIM_initfinish.c.m4`	- Template for initialization/finalization/start/finish routines
`SIM_mainloop.c.m4`	- Template for simulator main loop

Files from the tarball described as templates which have two suffixes are passed through **lm4** to produce the file with the first suffix during the **make** command. (**ls-build** actually attempts to perform this transformation early, before **make**, as this will save database load time.)

Files in each module instance's `src` directory

Each module instance's tarball is detar'd by **ls-build**. The "head" file for each module has a "`.clm`" suffix. Modules can be split across multiple files, but only if the additional files are included into the head file using:

`#LSE include <`*`filename`*`>`

make knows how to handle `.clm` files because the module's Makefile file must include `Make_module.mk` from the `include` directory; this file (through its includes) provides the definition for handling `.clm` files. The way **make** proceeds is that a `.c` file is generated which contains a prefix, the module's code, and a suffix. The prefix and suffix templates are in the `include` directory. (**ls-build** actually attempts to perform this transformation early, before **make**, as this will save database load time.)

The prefix provides the following (order is not exact, as some things are intermingled):

- Read the database
- Include necessary type headers
- Prototypes for API implementation functions, control functions, data collectors, user functions, etc...
- Per-instance API definitions, including firing functions
- Control and user points
- Data collectors

The following is done to the module's `.clm` file:

- API calls are checked and mapped to actual function names needed
- Instance variable names and function names are resolved (generally through flattening of the instance name and prefixing)

The suffix provides the following:

- Query wrappers
- Wrappers for phase end/phase start/init/finish

Note that if a module wishes to create a "library" of functions to be shared among instances of the module, the best way to do this will be to create a domain implementation of the *library* domain class and install that library in the install area. The source code for this library should *not* be placed in the module tarballs and can only know about instance data through parameters of calls to the library.

Identifier construction

The generated code has a lot of identifiers in it, and there needs to be a way to guarantee uniqueness. This cannot be done in general without restricting the user's choice of identifiers in some way. The rules for the user are:

1. Do not start names with LSE or m4
2. Do not use two underscores in a row in a name or start a name with an underscore.

With these rules in place, the general way an identifier is constructed from a user's identifier is with four parts:

1. A prefix starting with `LSE` which indicates the subsystem or source of the identifier. Prefixes are listed below. The prefix usually ends with a single underscore (_).

The prefixes used are:

`LSEan_` - Analysis: internal identifiers defined/generated in `SIM_analysis.py`
`LSEap_` - API definitions: internal identifiers defined/generated in `SIM_apidefs.py`
`LSEbl_` - Database build: internal identifiers defined/generated in `SIM_database_build.py`
`LSEcg_` - Code generation: internal identifiers defined/generated in `SIM_codegen.py`
`LSEdb_` - Database: internal identifiers defined/generated in `SIM_database.py`
`LSEdc_` - Domain class identifier (the class name is in the hierarchical path)
`LSEdi_` - Domain instance identifier (the instance name is in the hierarchical path)
`LSEdy_` - Dynamic ids: internal identifiers defined in `SIM_dynid.h`.
`LSEm4_` - A framework m4 macro
`LSEmi_` - Module instance identifier
`LSEpi_` - Port instance identifier
`LSEpy_` - A framework Python function
`LSEre_` - Resolutions: internal identifiers defined in `SIM_resolution.h`.
`LSEsc_` - Scheduling: internal identifiers defined/generated in `SIM_schedule.py`.
`LSErb_` - Rebuild: internal identifiers defined in `SIM_rebuild.py`.
`LSEti_` - Time: internal identifiers defined in `SIM_time.h`.
`LSEtk_` - Tokenizer: internal identifiers defined/generated in `SIM_tokenizer.py`.
`LSEty_` - Types: internal identifiers defined in `SIM_types.h`.
`LSEut_` - A user-defined type name or accessor
`LSEuv_` - A user-defined variable or runtime parameter name
`LSE_` - An identifier directly available to users in some way
`LSEfw_` - Any other identifier that is not supposed to be directly available to users

2. The purpose of the identifier. This part ends with a double-underscore (__). It is not always present.
3. A "flattened" hierarchical path to the name (if any). Flattening is the process of turning dots (`.`), the normal hierarchical separator, into double-underscores (__), the flat separator. If present, the path is usually followed by triple-underscores (___).

Open Issue

We might turn this path into a hash so that we do not make identifiers too long...

4. The name of something in the system, possibly flattened.

As an example of this construction, consider the name `LSE_CONTROL__foo__hey___out`. The prefix is `LSE_`. The purpose is `CONTROL`, which indicates that this is a control function. The flattened hierarchical path is `foo__hey`, so this is a control function for the `foo.hey` module instance. Finally, the name is `out`. So, this is the `out` control function of instance `foo.hey`.

Python identifiers that are to remain local to a module are prepended with an underscore (_). They do not always follow the naming convention because they do not have global scope.

API Identifier implementation

Possibly the nastiest thing to understand in the code generation process is how API identifiers actually get implemented. This section attempt to demystify it a little bit.

The first necessary concept is that of *code provenance*. There are three possible sources for code:

1. Fixed code within the framework (e.g. `SIM_types.h`.) This code is just normal C code; nothing more really needs to be said about it.
2. Generated code within the framework. This is code which has to change due to differences in the configuration. A good example of this is the static schedule for a timestep. This code is always generated by Python code embedded into a file which is expanded when the file is processed by m4.
3. User-supplied code. Users can supply code in configurations, in module `.clm` files, and in domain class files (both Python and macro). This is the hard-to-understand part of implementation and will be the focus of the rest of this section.

All user-supplied code *must* undergo parsing and translation . This applies to more code than you might expect; for example, any "external" type in LSS is user-supplied code and must be translated. The reason for this is that any user-supplied code could make references to API identifiers. These identifiers must be translated into appropriate code and also often cause dependencies between module instances, which can imply either rebuild conditions or even data dependencies affecting scheduling.

This translation step is accomplished via a "tokenizer" module in Python. This module is a generalization of m4 for C and C-like code; it can parse argument lists respecting square and curly braces, and it can output not only text, but also simple parse trees. Furthermore, the dictionaries of macros can be manipulated as a search list, allowing us to easily map definitions in and out in bulk.

These tokenizer features allow us to have dictionaries which hold API definitions. For core APIs, the dictionary entries generally define "tokenizer" macros — macros which use parse trees as input. This makes it easier to extract instance names and port names from arguments in the presence of parenthesis. Certain APIs also have "analysis" or "build" definitions which call the underlying API macro to do argument parsing and then mark data dependencies or rebuild conditions. Domain APIs are handled in a similar fashion, but because domain APIs must refer to a a domain class or instance, there is an additional indirection. We also use a different dictionary for them.

We define a domain API dictionary to hold all identifiers from domains. For every identifier defined by a domain, an entry is placed in the dictionary which points to a Python function which looks at the first argument and determines whether it is a domain instance reference; otherwise, it searches for the identifier in the domain search path. In either case it properly translates the reference to the unique backend name for the identifier.

Part of the identifier definition is a description of how to create the backend identifier. This varies dependencing upon the kind of identifier. LSE is responsible for creating the backend identifiers from these descriptions in appropriate header/code files. Any of these descriptions can also be `None`, which indicates to LSE that the definition of the backend identifier is in the macrofile for the domain. These definitions can include **m4** macros, which makes things interesting, as the m4 macro definitions need to be placed into the domain API dictionary as well.

So, what ends up happening is that the domain macro definitions get tokenized with the domain API dictionary at the head of the dictionary list, causing them to be placed into the domain API dictionary. This happens before any other user-supplied code is parsed. Then all other user-supplied code is parsed with this dictionary and the "core" API dictionary, as well as any additional dictionaries inserted to "hook" APIs for analysis of rebuild conditions or data dependencies.

Conversely, no code other than user-supplied code is automatically parsed and translated; other code simply goes through normal m4 expansion, which actually needs few macros other than the m4 builtins. (Those few macros are found in `SIM_common.m4`.) It is possible to parse and translate other code through the tokenizer by creating a suitable tokenizer and then forcing the text through it; it is even

possible to make the tokenizer look like an output stream that you can just write to and it will tokenize for you. We do this in a few places where it seems easier to tokenize in pieces than to create large strings to pass to the tokenizer.

Scope of types

One of the most difficult problems to deal with when generating code is the problem of resolving types. Most types are declared and used within a specific scope in LSS, but the C type scope of the generated code may be different. Keeping track of which types are in scope and which are not for a particular chunk of generated code is a tricky business, and implementing it is even worse, as C does not allow us to easily remove types from scope. This section describes how we deal with this problem.

Originally this section listed all the places in which types are used and what we wanted to be available at each place. But after listing that, we were able to come up with some very simple rules:

1. All types defined to LSS are visible within LSS using LSS's scoping rules. The `${}` notation is used to access them in places where program text (such as user point values) is enclosed in triple-angle-brackets (<<<>>>).
2. The LSS interpreter generates unique global names for each of the types, ensuring that a simple alias of a type to a new name does *not* create a new global type and outputs a list of these, along with their definitions. It also causes references to its types (in triple-angle-brackets) to translate to a particular macro (`LSEut_ref`) so the backend can easily translate the reference.
3. Some LSS types are made visible to the `.clm` file by exporting them with a specific local name to the back end; these types are *not* visible to the LSS file using the local name. These types are visible *only* to the `.clm` file. LSS outputs a list of these *local type aliases*.
4. The backend hashes the type definitions to create stable names for the types to reduce the number of forced rebuilds when types change. It also translates LSS type references and local type aliases.
5. For now, types can be defined in module and instance funcheader sections and data collector decl sections, but these will eventually be removed. Such types are visible to the user points (funcheaders), the .clm file (module funcheaders only), and data collectors (funcheaders and data collector decls). These types may have greater visibility than that specified here, which may lead to name clashes.

Supporting incremental build

The main problem for incremental rebuild is figuring out when things need to be rebuilt. Here's documentation of the cases where it is needed (or not needed) and why.

Rerun conditions for ls-schedule

1. The signal list changed in signal names, types, numbers, context numbers, or reduced dependencies (not real dependencies). Note that names and types are only required so that `SIM_schedule.txt` is consistent when outputting the schedule.
2. Either of the two parameters used in **ls-schedule** changes.

Rebuild conditions for the `include` directory

1. The directory tarball changed. This is checked by doing an SHA hash of the tarball.
2. The list of domain implementations changed or any domain changed identifiers, headers, compile flags, attributes, or parameters. `SIM_control.h` and `SIM_domain_types.h` contain variable, API, and hook definitions.
3. The list of instance names changed. This affects a variable declaration in `SIM_control.h`.
4. The list of structadds changed. Portions of the **LSE_dynid_t** and **LSE_resolution_t** structures are defined in `SIM_user_types.h`.
5. Any port names, types, widths, controlempty, or independence changed. All this information is used in generating the global message and status structures in `SIM_control.h`.
6. The list of user-defined types changed. User-defined types are defined in `SIM_user_types.h`.
7. Top-level parameters changed. These are used everywhere.
8. The list of user-defined variables changed. User-defined variables are declared in `SIM_control.h`.
9. A run-time variable definition changed. Run-time parameter variables are declared in `SIM_control.h`.

Rebuild conditions for the `framework` directory

1. The `include` directory tarball changed. This is checked by doing an SHA hash of the tarball.
2. The directory tarball changed. This is checked by doing an SHA hash of the tarball.
3. The list of domain implementations changed or any domain changed identifiers, headers, compile flags, attributes, or parameters. `SIM_control.c` contains variable, API, and hook definitions.
4. The list of instance names changed. This affects a variable declaration in `SIM_control.c`.
5. The list of structadds changed. Methods of **LSE_dynid_t** and **LSE_resolution_t** are defined in `SIM_control.c`.
6. Any port names, types, widths, independence, controlempty, or names changed. The global message and status structures whose types depend upon this information are instantiated in `SIM_control.c`.
7. The list of user-defined types changed. The global message structure definition and user-defined variables in `SIM_control.c` may depend upon the types which changed. Structadds may also depend on these types.
8. Top-level parameters changed. These are used everywhere.
9. The list of user-defined variables changed. User-defined variables are instantiated in `SIM_control.c`.
10. A run-time variable definition changed. Run-time parameter variables are instantiated in `SIM_control.c`.
11. The context list changed. `SIM_control.c` has a list of pointers to contexts.
12. The schedule changed. `SIM_control.c` has a structure containing the static schedule as well as dynamic section definitions.
13. The signal list changed in any way. May affect `SIM_control.c` or `SIM_mainloop.c`.
14. Top-level events/data collectors changed. The data collectors are instantiated in `SIM_mainloop.c`.

Rebuild conditions for all module instances together

1. The `include` directory tarball changed. This is checked by doing an SHA hash of the tarball.
2. The list of domain implementations changed or any domain changed identifiers, headers, compile flags, attributes, or parameters. `SIM_control.h` and `SIM_domain_types.h` contain variable, API, and hook definitions.
3. Any domain compilation flags changed
4. The list of structadds changed. Inlined methods of **LSE_dynid_t** and **LSE_resolution_t** which depend upon their structure are included by module instances. Also, attribute and field accessor macros depend upon structure. If fields of a module instance change or a module instance is deleted, must rebuild because we do not know what module instance might have accessed those fields. This rebuild condition is prevented when parameter *`LSE_use_direct_field_access`* is `FALSE`, as the accessor methods no longer need to know the structure, but use indirect accesses.
5. Top-level parameters changed. Not all parameters cause a rebuild; this is implemented by assuming that a change will cause a rebuild unless the parameter is on a list of exceptions or its name does not start with the magic `LSE_` prefix.

Rebuild conditions for individual module instances

1. The module instance is not in the old design.
2. The module tarball name or tarball contents changed. The contents check is done by doing an SHA hash of the tarball.
3. Any port names, types, widths, independence, controlempty, or direction changed. API names and implementations all depend upon this information. Also must rebuild if any port's handler status changed, because a firing function may call handlers directly (could be restricted to handler changes with control function non-empty).
4. Any user-defined types used (transitively) by the module instance changed. It is important that the transitive closure of type use be computed. Also, query calls imply use of their argument and return value types. Note that we know the text of type use within anything provided by **lss**: code functions, data collectors, structadds, query definitions, etc.; inside the .clm file we do not see it, but the .clm can only use mapped types, and we know those.
5. Any user-defined variables used by the module instance changed. Note that we know the text of variable use within code functions and data collectors; the .clm file cannot use these variables.
6. Any of a variety of module instance elements changed: parameters, phase/phase_end/phase_start flags, events/data collectors, funcheader, codepoints, domain search path, callers, queries, structadds
7. Signals driven by a control function of this module instance change constant status. If this happens, the control function contents must have changed, and that is part of the next condition. So.... this is a redundant condition.
8. Port connectivity or scheduling information for the module instance changed. This affects port API definitions and firing function generation.
9. Context numbers changed for the instance. This affects any calls/scheduling within the same instance; context numbers needed for scheduling between instances are always calculated through indirection in the port's global info structure. When parameter *`LSE_specialize_context_numbers`* is `FALSE`, within-instance context number calculation uses offsets from a global variable and this condition does not need to be checked.

10. Some items referenced by the module code across instances in code in triple-angle-brackets changes. These items are structure fields, query signatures, and port widths. Note that port widths are present because they are used to check arguments of port queries in the calling instance (not the callee) even though the `LSE_port_width` call translates to a constant. Port aliases must also be checked in this fashion.

    Just for completeness, the reason parameters and port types are not included in this are given here. Parameters are translated to constants in the codepoint text, which is compared separately. Port types are translated to a backend type, which is compared through both the codepoint and the type comparison logic.

11. The module contains an instance ref parameter parameter and anything accessable across instances by .clm code has changed in the referenced instance. At present only structure fields are in this category.
12. The module contains a literal parameter and anything accessable across instances by .clm code has changed in any instance. At present only structure fields are in this category. Note that this can cause many rebuilds for no good purpose; literal parameters should be avoided for this reason!

Furthermore, if a build is interrupted while the contents of the directories are being changed (i.e. before **make** begins), the directories and the database could be out of sync. It is necessary to detect this condition and force a complete rebuild on the next attempt to build the system. This is done by using a file called `changing_files` as a lock file to indicate that changes are in progress.

It would also be nice if we could use a "reduced" database when generating code for the individual module instances, as the whole database is time-consuming to read. However, the presence of port and general queries and field accessors complicates things: anywhere in the design is subject to query and API parameter checking needs to know at least the names available in the design. Some checking could be done at user/control function analysis (in fact, some already is), but there are still run-time checks to be made. Note that the only cross-instance pointer in the database at present is the pointer due to port connections.

Conditions that do not cause rebuilds.

1. Changes to the schedule or to the signal list do not cause rebuilds of all modules; only module instances whose scheduling behavior on the ports or actual port connectivity changed require a rebuild in this case. So if two different static schedules result because of a change in a control function somewhere, only the `framework` directory, the module instance with the control function that changed and potentially others connected to that port should need rebuilds.
2. Instances or ports disappearing do not cause rebuilds of all modules. This is because port and query analysis occurs on all rebuilds and will catch any references made by API calls in control or user functions to ports or queries that no longer exist.

Info file formats

There are eight "information" files which contain information about the configuration. These files are placed in the *`machinename`*`/include` subdirectory. These files are:

File name: `SIM_codepoint_info.py`

Information about: code points

File name: `SIM_domain_info.py`
Information about: domains (not yet generated by LSS)

File name: `SIM_event_info.py`
Information about: events and data collectors

File name: `SIM_instance_info.py`
Information about: module instances

File name: `SIM_parm_info.py`
Information about: parameters

File name: `SIM_port_info.py`
Information about: ports and connections

File name: `SIM_query_info.py`
Information about: queries

File name: `SIM_struct_info.py`
Information about: user extensions to datatypes

File name: `SIM_type_info.py`
Information about: type definitions and mapping

File name: `SIM_var_info.py`
Information about: variables

The format of each of the files is described in the following sections. All of the files are parsed by Python. The formats specify whether single quotes or raw-triple-quotes (i.e. `r"""` *`text`* `"""`) are to be used for some parameters. Raw-triple-quotes are always acceptable where single-quotes are required, but single-quotes are not acceptable where raw-triple-quotes are required

`SIM_codepoint_info.py`

This file is parsed by Python. It is read once while the Python database is being built. It consists of a number of function calls; the function definitions are in `framework/SIM_database_build.py`. The functions and their parameters are:

```
add_funcheader_to_inst(inst, text)
```

inst	Instance name (hierarchy denoted with '.'), single-quoted.
text	The funcheader text, raw-triple-quoted

```
add_codepoint_to_inst(inst, cpname, kind, text, dtext, rtype, params)
```

inst	Instance name (hierarchy denoted with '`.`'), single-quoted.
cpname	Codepoint name, single-quoted
kind	One of `LSE_PointControl`, `LSE_PointUser`, or `LSE_PointDecode`.
text	The text of the code point, raw-triple-quoted.
dtext	The default text of the code point, raw-triple-quoted.
rtype	The return type of the code point, raw-triple-quoted. This should only be supplied for user points, not decode or control points.
params	The parameter list of the code point, raw-triple-quoted. This should only be supplied for user points, not decode or control points.

`SIM_domain_info.py`

This file is parsed by Python. It is read once while the Python database is being built. It consists of a number of function calls; the function definitions are in `framework/SIM_database_build.py`. The functions and their parameters are:

```
add_domain_instance(dclassname, dimplname, dinstname, args)
```

dclassname	Domain class name, single-quoted
dimplname	Domain implementation name, single-quoted
dinstname	Domain instance name, single-quoted
args	Run-time arguments formatted as a raw-triple-quoted string

```
add_domain_searchpath(iname,lpath,hpath)
```

iname	Full module instance name (hierarchy denoted with '`.`'), single-quoted
lpath	Domain class search path specified in the module definition, given as a Python list of two-tuples. The first element of the tuple is the single-quoted domain class name and the second element of the tuple is the single-quoted domain instance name.
hpath	Additional domain search path inherited hierarchically, given as a Python list of two-tuples. The first element of the tuple is the single-quoted domain class name and the second element of the tuple is the single-quoted domain instance name.

Domain instance names should be unique. Numeric names are acceptable. Parameters of type domain instance will have values equal to these names.

SIM_event_info.py

This file is parsed by Python. It is read twice while the Python database is being built; the first pass reads events and the second one reads collectors. It consists of a number of function calls; the function definitions are in `framework/SIM_database_build.py`. The macros and their parameters are:

```
add_event_to_inst(iname, ename, nstring, tstring)
```

iname	Full instance name (hierarchy denoted with '.'), single-quoted
ename	Event name, single-quoted
nstring	Colon-separated list of data names, raw-triple-quoted
tstring	Colon-separated list of data types, raw-triple-quoted

```
add_collector_to_inst(iname, ename, decl, init, record, report)
```

iname	Full instance name (hierarchy denoted with '.'), single-quoted
ename	Event name, single-quoted
decl	Declaration text, raw-triple-quoted
init	Initialization text, raw-triple-quoted
record	Recording text, raw-triple-quoted
report	Reporting text, raw-triple-quoted

SIM_instance_info.py

This file is parsed by Python. It is read once while the Python database is being built. It consists of a number of function calls; the function definitions are in `framework/SIM_database_build.py`. The functions and their parameters are:

```
add_inst(iname, mtype, start, phase, end, strict, reactive, tarball)
```

iname	Full instance name (hierarchy denoted with '.'), single-quoted
mtype	Module type name, single-quoted (WHAT ABOUT PACKAGE PARTS OF NAME?)
start	Does the instance have a phase_start method? (1=yes, 0=no)

phase	Does the instance have a phase method? (1=yes, 0=no)
end	Does the instance have a phase_end method? (1=yes, 0=no)
strict	Is the instance strict? (1=yes, 0=no)
reactive	Is the instance reactive? (1=yes, 0=no)
tarball	Tarball file name; if it begins with '-', it is taken to be a command to run to copy/generate module source files. This command will be run in the final source file directory.

```
add_dep_annotation(iname, annotation)
```

iname	Full instance name (hierarchy denoted with '.'), single-quoted
annotation	annotation information; this is a list of 3-tuples. The elements of the tuple are: a string indicating the source of the potential dependency, a string indicating the target of the potential dependency (i.e. target depends upon source), a string which is an expression comparing the port indices (held in isporti/osporti).

SIM_parm_info.py

This file is parsed by Python. It is read once while the Python database is being built. It consists of a number of Python function calls; the function definitions are in `framework/SIM_database.py`. The calls and their parameters are:

```
add_parm_to_inst(inst, name, value, type, runtimed, clname, desc)
```

inst	Instance name (hierarchy denoted with '.'), single-quoted; use the empty string for top-level parameters.
name	Parameter name, single-quoted
value	The value of the parameter, triple-quoted as necessary
type	The type name of the parameter, single-quoted
runtimed	1 if the parameter is to be made run-time changeable, 0 otherwise.
clname	Command-line argument used to set the parameter if it is run-timed, ignored otherwise
desc	Description of the command-line argument used to set the parameter if it is run-timed, ignored otherwise

SIM_port_info.py

This file is parsed by Python. It is read twice while the Python database is being built. In the first pass ports are read, and in the second pass connections are read. It consists of a number of function calls; the function definitions are in `framework/SIM_database_build.py`. The functions and their parameters are:

```
add_port_to_inst(iname, pname, width, direction, datatype, indepP, handlerP)
```

iname	Full instance name of module (hierarchy denoted with `.`), single-quoted
pname	Port name, single-quoted
width	Port width
direction	Port direction (`input` or `output`), single quoted
datatype	Port datatype, single-quoted. The type should be the global datatype name
indepP	Is the port independent? (1=yes, 0=no)
handlerP	Does the port have a handler? (1=yes, 0=no)

```
connect_ports(from, to)
```

from	Source port of connection in **`hierarchical_instance_name`**: **`port_name`**[**`port_num`**] format (hierarchy denoted with '`.`'), single-quoted
to	Destination port of connection in **`hierarchical_instance_name`**: **`port_name`**[**`port_num`**] format (hierarchy denoted with '`.`'), single-quoted

```
alias_ports(hname, dir, rports)
```

hname	Port name in hierarchical "wrapper" instance in **`hierarchical_instance_name port_name`** format, single-quoted
dir	Port direction (`input` or `output`), single quoted
rports	A Python list of equivalent "real" port instances where each port instance is denoted in **`hierarchical_instance_name`**: **`port_name`**[**`port_num`**] format (hierarchy denoted with '`.`'), with each port instance being single-quoted. An unconnected port instance is denoted with `None`.

Because it is difficult in LSS to distinguish what ports are real due to adapters and what ports are true aliases at the time at which the aliases are being output, it is legal to output `alias_ports` calls

for hierarchical ports which are actually defined by adapters; the backend ignores these aliases.

SIM_query_info.py

This file is parsed by Python. It is read once while the Python database is being built. It consists of a number of function calls; the function definitions are in `framework/SIM_database_build.py`. The functions and their parameters are:

```
add_query_to_inst(iname, qname, rtype, params)
```

iname	Full instance name (hierarchy denoted with '.'), single-quoted
qname	Query name, single-quoted
rtype	C return type, raw-triple-quoted
params	C parameter list, raw-triple-quoted

```
add_method_to_inst(iname, mname, rtype, params, locked)
```

iname	Full instance name (hierarchy denoted with '.'), single-quoted
mname	Method name, single-quoted
rtype	C return type, raw-triple-quoted
params	C parameter list, raw-triple-quoted
locked	Is the method locked to the instance? (1=yes, 0=no)

SIM_struct_info.py

This file is parsed by Python. It is read once while the database is being built. It consists of a number of function calls; the function definitions are in `framework/SIM_database_build.py`. The functions and their parameters are:

```
add_to_struct(sname, iname, fields)
```

sname	Name of structure to extend, single-quoted
iname	Full instance name (hierarchy denoted with '.'), single-quoted

fields	A Python list where each element is a (type, name) tuple with each element raw-triple-quoted. The values should be such that "*`type name;`*" would be a valid C structure field definition. For example: `[ ( r"""int""", r"""foo""" ), ( r"""struct { float himo }""", r"""mystuff""" ) ]`

SIM_type_info.py

This file is parsed by Python. It is read once while the Python database is being built. It consists of a number of Python function calls; the function definitions are in `framework/SIM_database_build.py`. The calls and their parameters are:

`add_type(name, level, def)`

name	Global type name, single-quoted. The global type name must be unique and be of the form `LSEut_#`.
level	Instance name at which the type was originally defined (hierarchy denoted with '`.`'), single-quoted; use the empty string for top-level type definitions . *Now obsolete*.
def	C type definition, raw-triple-quoted. Any use of user types in the definition must be wrapped as a call to `LSEut_ref`. Array definitions must be wrapped in calls to `LSEut_arraydef`. Structure definitions must be wrapped in calls to `LSEut_structdef`.

`add_type_mapping_to_inst(inst, localname, globalname)`

inst	Instance name (hierarchy denoted with '`.`'), single-quoted.
localname	Local type name, single-quoted.
globalname	Global type name, single-quoted.

All types referred to in any C type definition (e.g. structure fields) must be defined through a call to `add_type`, except for the *system types* and *domain types*, which should not be defined through such a call. The following are the system types:

- **LSE_type_none** - the type of ports with no associated data
- **LSE_dynid_t** - dynamic message identifier

- **LSE_resolution_t** - resolution messages
- types defined by ANSI-C in <`stdint.h`> (e.g. **uint16_t**)
- all atomic C data types (e.g. **char**)
- **boolean** - boolean data type

The `add_type` calls must be output in an order such that all types referenced inside a given type definition have already been added. For example, if type A is a structure with a field of type B, type B must be added before type A. Also, global types referred to in `add_type_mapping_to_inst` calls must be added before the call.

Domain types (e.g. **LSE_emu_addr_t**) should be output as fully-qualified domain type names (e.g. `LSE_emu_addr_t([`*`emuinst`*`])`). Note that users are not in general required to use ${} within <<<>>> to get domain types unless the string ends up at a lower level of hierarchy where the primary domain object for the domain has changed. Usually the backend will find the correct default object and there will be no problem.

Value names of enumerated types in the *`def`* parameter of `add_type` should be defined using `LSEut_enumdef(`*`name`*`)`. Dollar-sign-curly-brace evaluation of string constants in LSS should output `LSEut_enumref(`*`type`*`,`*`value name`*`)`.

Array types are defined using the `LSEut_arraydef` macro inside of their definition. This macro takes two arguments; the first is the size of the array (which must be a positive integer), while the second is the name of the type (wrapped in `LSEut_ref` if it is a user-defined type) of the array elements.

Structure types are defined using the `LSEut_structdef` macro inside of their definition. This macro takes one argument per field in the structure plus an extra empty argument at the end; each non-empty argument must be made up of two words. The first word is the name of the type of the field in parenthesis (and it should be an `LSEut_ref` call if a user-defined type). The second word is the field name.

SIM_var_info.py

This file is parsed by Python. It is read once while the Python database is being built. It consists of a number of Python function calls; the function definitions are in `framework/SIM_database_build.py`. The calls and their parameters are:

```
add_var(globalname, username, type)
```

globalname	Variable name, single-quoted; the name should be of the form `LSEut_#` and be globally unique.
username	Variable name provided by the user, single-quoted.
type	Global type name, single-quoted.

Rules for curly brace resolution.

There are three special rules for resolving curly braces in LSS:

1. Type references should be wrapped in a `LSEut_ref` macro call. The call takes the global type name as its sole argument. The type name does not need to be m4-quoted as the backend parses these at databse build time.
2. Variable references should be wrapped in a `LSEuv_ref` macro call. This call takes the global variable name as its sole argument. The argument does not needs to be m4-quoted as the backend parses these at database build time.
3. Enumerated type values should be wrapped in a `LSEut_enumref` macro call. This call takes two arguments: the global type name and the enumerated value name. The arguments do not need to be m4-quoted as the backend parses these at database build time.
4. Runtime parameter values should be wrapped in a `LSEuv_rp_ref` macro call. This call takes one arguments: the command-line option name.

Adding APIs

We do not recommend that you add APIs directly to the framework; it is better to add them using the official method for extending the APIs: domains. However, we do want to document how APIs are added for maintenance purposes.

Important: If you add an identifier of any type, do not forget to update the API Reference Manual.

Adding types

Adding types is a simple process. There are several cases:

1. The type is to be visible to the user in all code and does not depend upon instance information in any way. This case is the simplest; add the type to `SIM_types.h`. If the type is also to be visible in **lss**, it should be added to `LSS_builtins.lss`.

   If runtime parameters can have this type, there is one additional complication; in `SIM_initfinish.c.m4` you need to add the type to the `type2scanner` Python mapping so that it can be properly parsed on the command line. It may be necessary to add a new scanning routine to match the type if one of the existing ones does not fit.

2. The type is to be visible to the user and depends upon instance information. Here the type is added to `SIM_user_types.h.m4` and will require use of embedded Python code to generate the correct definition. It should be added before the types from LSS. If the type is also to be visible in **lss**, it should be added to `LSS_builtins.lss`.

It is unlikely that runtime parameters will have such a type, but if they do, they are handled as in the previous case.

3. The type is not to be visible to the user but needs to be available to all code. Add such types to `SIM_control.h.m4` after the port structures. If the type depends upon instance information, you will need to use embedded Python code. The type name should have a `LSEfw_` prefix.
4. The type is not to be visible to the user but needs to be available to only a single instance. Add such types to `SIM_prefix.m4` after the domain macro definitions. the type depends upon instance information (as it probably will), you will need to use embedded Python code. The type name should have a `LSEmi_` prefix.

Note that in all cases some cleverness may be needed when the new type is a structure to get incomplete structure definitions (i.e. `struct blah;`) into place in the right order to make everything work.

Refcounted types

The above descriptions assume that the type is not reference-counted. If it is, then some additional things must be done:

1. The type name *must* end in `_t`.
2. The type's structure definition (and it must be a structure) must begin with a field called *`super`* of type **SIM_refcount_t**.
3. Functions to cancel, register, and create the type must be added to `SIM_all_types.h` after `SIM_refcount.h` has been included.See the corresponding routines in `SIM_resolution.h` for examples. For a type **LSE_foo_t**, the function names must be `LSE_foo_cancel`, `LSE_foo_register`, and `LSE_foo_create`.

   These functions can also be put in a new file included at the same location in `SIM_all_types.h`. In such a case, be certain to add the new file name to the `framework_inc_TARSTUFF` variable in `framework/Makefile.am`.

4. Add return statements containing calls to the cancel and register functions in `SIM_apidefs.py` in `_LSEap_data_cancel`, `_LSEap_data_copy`, and `_LSEap_data_register`. They should look like the ones done for **LSE_resolution_t**.
5. In `SIM_control.h.m4`, duplicate the lines referring to **LSE_resolution_t** in the "Datatype refcounting" section, changing **LSE_resolution_t** to the new type name.

Adding variables

Variables are also simple to add. The declarations of variables should be added to `SIM_control.h.m4`, after the port structures. The definitions of variables should be added to `SIM_control.c.m4`, after the port structures.

Adding functions

Adding a function is more complex, as there are many more options for how to do this. This section will not explain everything, particularly not the internal functions available for parsing arguments and putting together APIs. It will give you an idea of what must be done and where to look for examples.

The first decision which must be made is when to evaluate the function. The options are:

- At code generation time - such APIs can be used in definitions, can easily use global information, and can be used in `#LSE if` and `#if`, but must result in constants.
- At C pre-processing - such APIs can be used in `#if`, but generally have the worst argument checking (as it is buried in CPP macros) and some difficulty using global information and must result in constants.
- At run-time - such APIs have difficulty using global information, but the results can vary at runtime.

The actual translation of API function calls to final code can take place in one of several ways. They are:

- Completely translate the function call to a constant at code generation time. This approach must be taken for functions evaluated at code generation time. An example is `LSE_port_width`.
- Translate the function call to C code at code generation time. This approach is often taken for functions which are evaluated at run-time.
- Do not translate the function call at all; allow the C pre-processor and compiler to treat the call as a normal macro or function invocation. All of the functions having to do with **LSE_type_t** use this method.

Instructions for each method are given in the following sections. We will assume that we wish to add a function `LSE_num_ports` that returns the number of ports on a particular module instance.

Complete translation at code generation

1. Add an entry for the function to `LSEap_codePreDict` in `SIM_apidefs.py`. Try to keep alphabetical order for easier reading. It should look like this:

   ```
   "LSE_num_ports" :
        (LSEtk_Tok_macro, 0, _LSEap_num_ports, None),
   ```

2. Add Python function `_LSEap_num_ports` to `SIM_apidefs.py` in the section where other API macros are placed, keeping alphabetical order for easier reading of the file. This function should return either a string or tokens which correspond to the value desired. See `_LSEap_port_width` for a good example.

Translate to C code at code generation

1. Add an entry for the function to `LSEap_codePreDict` in `SIM_apidefs.py`. Try to keep alphabetical order for easier reading. It should look like this:

   ```
   "LSE_num_ports" :
        (LSEtk_Tok_macro, 0, _LSEap_num_ports, None),
   ```

2. Add Python function `_LSEap_num_ports` to `SIM_apidefs.py` in the section where other API macros are placed, keeping alphabetical order for easier reading of the file. This function should return either a string or tokens which correspond to the C code desired. Of course, in this example, the C code is really a constant. See `_LSEap_GLOBDEF` for an example.

If the C code involves C macro or function calls, there is additional work to be done. This depends upon whether the function or macro calls must be generated on a per-instance basis.

Not per-instance macro or function calls

For a C macro, add it to `SIM_control.h.m4`. For a C function, add its prototype to `SIM_control.h.m4` and its implementation to `SIM_control.c.m4`.

Per-instance macro or function calls

For an example of how to handle per-instance macro or function calls, see `LSE_sim_keep_alive`. The steps are:

1. Add a Python function `LSEcg_num_ports` to `SIM_codegen.py`. This function should take at least the database and instance as arguments and return a string with the text of the function or macro definition.
2. Add a line in the "internal per-instance API functions" section of `SIM_prefix.py` to instantiate the internal function's code. This line looks like:

   ```
   m4_pythonfile(print LSEcg_num_ports(LSE_db,
                                       LSEpy_instance))
   ```

3. If the function can be used in user points or data collectors at the top-level, then add a line to instantiate the code to the "internal API functions" section of `SIM_mainloop.c.m4`. This line looks like:

   ```
   m4_pythonfile(print LSEcg_num_ports(LSE_db,None))
   ```

If the API can access information beyond the local instance, the API must be registered with the rebuild analysis. To do this:

1. Modify function `_LSEap_num_ports` to change its argument `args[0]` to a tuple consisting of the original value of `args[0]` and any other information received from analysis; in our example, the only other information would be the instance name.
2. Add a Python function `_LSEbl_num_ports` to `SIM_database_build.py`. This function would look something like:

   ```
   def _LSEbl_num_ports(tinfo,args,typelog):
       rval = SIM_apidefs._LSEap_num_ports(tinfo,args,_LSEbl_env)
       if _LSEbl_env[1]:
           _LSEbl_env[1].instSeen[args[0][1]] = 1
       return rval
   ```

Note: This will only work if the API is accessing information already looked at for rebuild analysis. If it accesses more information, function `LSErb_referenced_changes` in `SIM_rebuild.py` must be modified.

Do not translate at all

If the function is to be a C macro, add it to `SIM_control.h.m4`, otherwise, add its prototype to `SIM_control.h.m4` and its implementation to `SIM_control.c.m4`.

Chapter 2. Domain Interfaces

This chapter describes nasty guts of how domains are accessed by the simulator. It also describes some emulator decisions. interfaces between the command-line processor, LSE, and domains.

Interface goals

The goals of the interfaces are to:

- Make it possible to embed Liberty into other systems (such as vertically-integrated systems like MILAN).
- Make it possible to use domains (such as emulators) from a variety of sources with limited modifications to their source code or even without source code availability.
- Recognize that some domains may have many different implementations and that different experimental conditions may involve different domain implementations. In such situations, recompilation should be minimized. For example, compiled-code emulators will produce a different library for each benchmark. Ideally, nothing would need recompiled when a different benchmark is to be run except for the emulator itself.
- Allow both domain instances and simulators to have run-time arguments to change some behaviors.
- Support domain implementations written in both C++ and C.

Design principles and decisions

There are some fundamental principles and design decisions driven by the goals:

- Simulator code is always generated. It can be specialized as needed to fit the situation. However, it cannot be specialized at link time, so the simulator builder must know about the domains to be used.
- Command-line parser code should be a library. This allows it to be easily replaced by another command-line parser, making it possible to embed Liberty into other systems.
- Linking should be accomplished through C++ to ensure that constructors for emulators written in C++ get called and that C++ system libraries are included as needed.
- All interface routines have "C" linkage.
- We are willing to have some inefficiency on entry to domain instances in return for flexibility, but expect not to lose much because of code specialization on the simulator side.

Termination conditions

How does the simulator or CLP know when to terminate? There are three ways to terminate:

1. An error condition occurs or a module or domain class or instance requests termination. In both cases, the variable `LSE_sim_terminate_now` is set to a non-zero value (a negative value in the case of errors).
2. The simulator has no more scheduled timesteps.
3. The termination count variable (`LSE_sim_termination_count`) reaches zero. This variable is initialized to zero at the beginning of `LSE_simulation_start`. Domain classes and instances can increment and decrement it. For example, an emulator may increment the count when there is a new execution context created and decrement it when a context finishes. When the counter reaches zero at the beginning of a time step, simulation terminates. Domains which change the termination count indicate that they do so in their Python class object. If there are no domains which change the termination count, it is initialized to `1` so that this termination condition does not cause immediate termination.

Implications for the build process

Multiple-definition identifiers

The most significant way in which the domain interfaces affect the build process is that many identifiers (i.e. constants, types, variables, API calls) do not have one fixed, global definition. Instead, these identifiers (such as **LSE_emu_addr_t**) depend upon which domain implementation (or even instance) is being referenced.

Figuring out what the proper definition is for a particular reference to an identifier is difficult. There are two ways it can be determined:

1. It can be stated explicitly by using a domain instance name as the first parameter of the API call or an additional parameter on other identifiers, e.g.: `LSE_emu_addr_t[`*`myinst`*`]`. The name can also be the name of a domain class for class-defined identifiers.
2. It can be found implicitly. This is possible unambiguously when there is no name conflict between identifiers in different domain instances or domain classes. When there is a conflict, the ambiguity is resolved using a "domain instance search path".

The way in which this gets implemented is that each domain instance must state all of its identifiers to the backend. The backend creates a macro definition for each identifier. This definition translates to a call to a Python domain name resolution function, which checks the current domain search path to determine what the implicit domain instance should be for the identifier. This procedure works surprisingly well; domain search paths can be easily manipulated while generating include and framework files to only allow identifiers that should be in scope to be manipulated. Identifiers which are out of scope are ignored and passed through to the output code, which prevents problems where a "non-interesting" domain overrides an identifier in some module.

Implementing identifiers

How are identifiers implemented? Constants and types are simple enough; the domain class implementation gives a list of constants and types along with their definitions, which are placed in `SIM_domain_types.h`. APIs and m4 macros are a bit more difficult.

Clearly, we must have a list of APIs and macros so we can do the mapping, just as we had for constants and types. The trick is to have two ways to implement things: they can either be placed as identifiers with non-`None` implementation (just as constants and types usually are), or they can be defined in an m4 "macro" file.

The structure of the macro file is defined by keywords which indicate whether the following section of code is for classes or instances, and whether it is for headers, stand-alone (one-generation-only) code, and macros (which appear in headers, stand-alon code, and modules). Within each section, there are macro calls for defining per-class and per-inst macros and for translating identifiers into per-class or per-inst identifiers. During database build, after the Python module is read, the macro file (if any) is read and separated into sections.

> **Note:** This split occurs without going through m4 first, so the keywords cannot be commented out or qualified in any way.

Hooks

Hooks are known identifiers that a class or instance can fill with code.

Open Issue

At present, hooks must go in the macro file. Should we have a way to fill in their code separately?

Callbacks

There are interesting issues when domain implemenations are to call functions or access variables provided by the domain or by LSE (e.g. `LSE_stderr`. First, how do they name them, as functions/variables provided by the domain will have LSE-munged names? Second, how do you get a dynamic library to build and link properly with the main program when it uses variables that are outside?

The first issue could be resolved with macros supplied to domain implementers. The second is actually possible with the right linker command-line flags and use of libtool. But either one is ugly. A better way to go is to not attempt to statically name or link called-back functions or variables. Instead, the domain class passes a structure with pointers to the functions and variables to the domain implementation when it is initialized. This solution is simple, requires no special linking tricks and prevents domain implementations from accessing parts of the simulator they should not. The only downside is that use of callbacks requires an extra indirect reference.

Linking

The linker script must search for domain libraries, which may be test-case specific. There are some tricky things to worry about:

- There could be filename conflicts between test-case specific libraries.
- There can be symbol conflicts between domain libraries

The build process does at least know about the domain instances and the nominal library names.

Open Issue

- How do we get the linker to actually do all this? There will be symbol conflicts between domain instances from the same domain class. Will need to rename symbols. Can do this for static library with objcopy (awkwardly) or with special tool. For shared library, will need special tool (objcopy does not change dynamic symbol tables); may not work. dlopen can also be used, but some shared libraries cannot be dlopened and not all systems have it.
- Also need to get LD_LIBRARY_PATH right for copies of shared library domain implementations.
- Symbol naming in support libraries and multiple-versioning of support libraries. In particular, SimpleScalar cache/bpred models and ptrace support depend upon the datatypes. I guess as long as the support libraries are part of the domain implementation and the cache/bpred stuff becomes *module* code, there won't be an issue (since the module code gets compiled separately), but that's ugly. A better idea: let the domain implementation declare additional symbols to be exported to the simulator and let wrappers be created for those. This also affects things like the "is it in bounds" check routine for bliss....

Details of the *emulator* domain class

Emulators are an important domain class and have had much debate and thought invested in them. The goals of the emulator interface were:

- Make Liberty configurations as ISA-independent as possible.
- Support multi-processing-element systems with heterogeneous ISAs.
- Support weird and wonderful new ISA ideas and paradigms for sharing state between execution contexts.
- Allow switching between "pure" emulation and detailed simulation.
- Benchmarks should be choosable at link-time or run-time (link-time for compiled-code emulators, run-time otherwise).

Management of contexts

LSE is responsible for context management. The basic idea is that the simulator and emulators trade opaque tokens which allow them to notify each other of events to the context. For example, emulators call the context management library using the simulator token when they create new contexts or contexts finish. The simulator uses the emulator token when calling emulator functions.

There is no need here to repeat the information in the Developer's Manual and the User's Manual about the difference between hardware and software contexts. What should be pointed out is that LSE does need to check the inputs to context management API calls to confirm that the context numbers are valid.

Contexts can be identified both by this global context number and by an opaque emulator token (**LSE_emu_ctoken_t**). LSE must maintain a mapping from number to token for use by various emulator APIs, but need not maintain a backwards mapping. LSE does not attempt to classify relationships (e.g. kill children on parent death) among contexts.

LSE does not provide a scheduler for hardware contexts; emulators are responsible for maintaining mappings and informing LSE of changes to the mappings. LSE keeps its own copy of the global mappings. There are API calls (rather nasty ones) for trying to inform the emulator of how to manage the mappings.

Open Issue

Can we do away with LSE needing to know all the mappings (it could be cheaper and less confusing, but would require calls into the emulator to ask for the mapping, which does happen quite rapidly)?

Open Issue

Another difficult problem in context management is the allocation/freeing of resources. Either contexts must leak memory, or some sort of reference counting scheme must be used. Any dynid in a context constitutes a reference, but there are also references involved in potential dynid generators. How to deal with this is an open issue. (For now, we leak.)

Note: Contexts cannot be moved from one emulator instance to another.

Emulator instances

Emulator instances are domain instances and, as such, are "copies" of an emulator implementation's code and data. Different emulator instances are used mainly because some emulators may not be able to support multiple contexts internally. Copying the code and data allows the emulator to be used multiple times. Note, however, that when such copying occurs, sharing of data between contexts must pass through LSE; all of the instances but one must treat the shared state space as external.

The *operandval* capability.

This capability was a gut-wrencher. Originally we copied source operands into a source value array, stuck results in a destination value array, and then wrote back from the result array. This was awfully slow. We didn't think that mattered too much, until we learned that statistical sampling is a desirable methodology and that fast-forward speed matters immensely then (and found that the IA64 emulator didn't achieve 1 MIPS on a 733 MHz Pentium III).

So . . . after a lot of thinking, we decided to use pointers. Yes, it's a bit weird when you first think about it. Yes, not all operands are treated identically. But it makes data copy a microarchitectural thing, which is more like hardware, and happens only during detailed simulation. Furthermore, it makes it easier to handle register renaming; just change your destination pointers and don't write back normally! In fact, we had no way other than just a lot of slogging through data copying to make register renaming work before.

One final thing: the thought did occur that maybe we should just not spec this and let each emulator handle it on its own. After all, we don't seem to write generic modules to handle operand values. However, giving a spec gives emulator writers some guidance as to how we think they can best achieve both performance and flexibility.

Old stuff to figure out how to say

Stuff to do on emulator init - Create, load, and map initial contexts. Binaries (if needed) and program arguments are supplied for each context at this time.

Internal APIs for context manipulation (not documented elsewhere)

`LSE_emu_contextno_t` **`LSE_emu_context_alloc`**`(int` *`emuinstno`*`);`

> Create a new context in emulator instance number *`emuinstno`*. The context state is `LSE_contextstate_waiting`. Returns the global context number for the newly created context. This function may fail, returning `-1` if the emulator instance cannot create more contexts.

Things a command-line processor extension for emulators might do

Open Issue

- Specifying state sharing in the CLP.
- Extending the command-line stuff to support breakpoints and debugging in different emulators (if they support it).