# Table of Contents

Preface ................................................................................................................................. i  
  Conventions .................................................................................................................. i  
  Important announcements ......................................................................................... i  
  Typographical conventions used in this book ...................................................... ii  

## 1. Datatypes and global variables .............................................................................. 1  
  Datatypes ..................................................................................................................... 1  
    Simple types ........................................................................................................... 1  
    Dynamic messages ................................................................................................. 1  
    Reference-counted types ....................................................................................... 4  
    Signal values ........................................................................................................... 5  
    Speculation resolutions ......................................................................................... 9  
    Time types ............................................................................................................. 11  
    User-defined types ............................................................................................... 13  
  Global variables ....................................................................................................... 14  
  Global constants ...................................................................................................... 14  

## 2. Core API functions ................................................................................................. 15  
  Code generation directives ...................................................................................... 15  
  Codepoint functions ............................................................................................... 15  
  Data manipulation functions .................................................................................. 16  
  Event functions ....................................................................................................... 16  
  Instance accessors .................................................................................................. 17  
  Looping constructs .................................................................................................. 18  
  Method and query functions .................................................................................. 19  
  Miscellaneous functions ....................................................................................... 20  
  Port functions ......................................................................................................... 20  
  Reporting functions ............................................................................................... 23  
  Scheduling functions ............................................................................................... 24  

## 3. Emulation Interface ................................................................................................. 26  
  Datatypes, structure fields, and constants .............................................................. 26  
  Core emulation functions ....................................................................................... 29  
  Functions available with disassemble ...................................................................... 34  
  Functions available with operandval ........................................................................ 35  
  Functions available with operandinfo ................................................................. 36  
  Functions available with speculation ................................................................. 36  

## 4. Domain construction APIs ..................................................................................... 38  
  Macros without arguments ..................................................................................... 38  
  Python variables ..................................................................................................... 38  
  Other macros ........................................................................................................... 38  

## 5. Error Messages ...................................................................................................... 40  
  Build-time error messages ...................................................................................... 40  
  Link-time error messages ....................................................................................... 40  
  Run-time error messages ....................................................................................... 41  

### A. API names ......................................................................................................... 42
List of Tables

1-1. Datatypes .................................................................................................................................1
1-2. Signal value constants (of type LSE_signal_t) ......................................................................5
1-3. Resolution class values (of type LSE_resolution_class_t) .......................................................9
1-4. Speculation resolution type (LSE_resolution_t) elements .......................................................9
1-5. Time constants .........................................................................................................................12
3-1. Emulation interface datatypes ..................................................................................................26
3-2. Emulation interface structure fields .........................................................................................26
3-3. Emulation interface constants ................................................................................................29
3-4. Emulation interface structure attributes ................................................................................29
A-1. Core API identifier list ...........................................................................................................42
A-2. Emulator API identifier list ....................................................................................................47
A-3. Domain construction API identifier list ...................................................................................50
Preface

This book defines the Application Programming Interface (API) presented to the writers of modules and configurations.

Conventions

API function call descriptions have the following format:

```
returnType funcName(parameterType parameterName, ...);
```

**Availability:** What pieces of code?

**Evaluated at:** At what time?

A description of the function, including how `parameterName` is used.

When a parameter of a function does not have a type, that parameter is a literal; its value must either be a literal in the original code or be resolved to a literal through a parameter at code generation time. The value of the parameter cannot be provided through C macro processing and cannot change at runtime.

The **Availability:** line indicates what kinds of code can use the function, e.g. control functions and funcheaders. "Everywhere" indicates .clm files, control functions, user functions, funcheaders, and data collectors.

The optional **Additional Availability:** line indicates additional environments in which the API can be used. These correspond to triple-angle-brackets in lss and could include structadds, query definitions, method definitions, type definitions, user point definitions, and event definitions. When this line indicates that an API call is available in literal parameters, it indicates that any use of the API call in literal parameters is properly analyzed for rebuild conditions and scheduling.

The **Evaluated at:** line indicates when the function’s value is evaluated, e.g. runtime. Some functions are actually "called" at code generation time, so their results can be used in C macro processing. Note that there can be a difference between the time at which literal parameters must be fixed and the time at which the function is called. This is because the literal parameters are used during code generation.

Emulator interface functions also have a **Capability:** line indicating which emulator capability is required for the function to be available.

Important announcements

**Usage note for all API functions:** All API functions should be treated syntactically like C pre-processor macros which expand to unknown amounts of text. This involves being careful about two things:

1. Do not surround the API name with parenthesis, i.e., do not do:
   ```
   (LSE_api_do_neat_stuff)(arguments).
   ```
2. Do not use a "bare" API call after an if statement without wrapping it in curly braces, i.e., do not do:

```c
if (mycondition)
    LSE_api_do_neat_stuff(arguments);
```

3. Do not use expressions with side effects as arguments, as they may be evaluated more than once.

**Warning**

This document should list all of the supported identifiers in the core and emulation APIs. Any internal identifiers present in header files or generated code are subject to change without warning. A very easy way to tell whether an identifier is meant for use is this rule: if it begins with `LSE_`, you may use it. If the identifier begins with `LSExy_`, where `xy` are any two characters, you may not use it. There are a few additional identifiers you may use (e.g. `PARM`); these are all listed in this manual.

**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. Datatypes and global variables

This chapter describes datatypes and global variables available to users of the Liberty Simulation Environment. It also describes constants and API functions closely related to particular datatypes.

Datatypes

The following table lists all of the datatypes available in the core Liberty Simulation Environment.

Table 1-1. Datatypes

<table>
<thead>
<tr>
<th>Datatype name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>Boolean type</td>
</tr>
<tr>
<td>types defined in stdint.h</td>
<td>Integers of fixed sizes</td>
</tr>
<tr>
<td>standard C types</td>
<td>—</td>
</tr>
<tr>
<td>LSE_type_none</td>
<td>The C type of a port with the none type</td>
</tr>
<tr>
<td>LSE_dynid_num_t</td>
<td>Identification number of a dynamic message</td>
</tr>
<tr>
<td>LSE_dynid_t</td>
<td>A dynamic message</td>
</tr>
<tr>
<td>LSE_refcount_t</td>
<td>A base reference-counted type</td>
</tr>
<tr>
<td>LSE_resolution_class_t</td>
<td>The kind of speculation resolution message</td>
</tr>
<tr>
<td>LSE_resolution_t</td>
<td>A speculation resolution message</td>
</tr>
<tr>
<td>LSE_signal_t</td>
<td>Value of signals on a port</td>
</tr>
<tr>
<td>LSE_time_numticks_t</td>
<td>Number of timesteps</td>
</tr>
<tr>
<td>LSE_time_t</td>
<td>Global time; consists of cycle and phase number</td>
</tr>
<tr>
<td>User-defined types</td>
<td>Any types defined by the user in .lss files</td>
</tr>
</tbody>
</table>

These data types and the API functions which can be used to access them are described in more detail in the following subsections.

Simple types

All standard C types as well as those defined in stdint.h are available to all module and user code. In addition, a boolean type is defined for Boolean values. This type has constants TRUE and FALSE.

There is also a type LSE_type_none which is a dummy type used for ports with no type. There is a constant value LSE_type_none_NULL which is a null value for this type.

Dynamic messages

The LSE_dynid_t type is the base type for messages sent over the data signals of ports. Instances of this
type are known as dynids. Whenever data is sent on a port, there must be a dynid present. Dynids have a unique identifying number; this number is of type \texttt{LSE\_dynid\_num\_t}. Identifying numbers are always assigned in increasing order.

Dynids have only one user-accessible element of their own. This element is the \textit{idno} element of type \texttt{LSE\_dynid\_num\_t}, which holds the identifying number of the dynid. However, both module instances and domain instances can add elements to the dynids. When module instances do so, they are called fields. When domain instances do so, they are called attributes. Fields and attributes are named using a special tag (\texttt{field:} or \texttt{attr:}), an optional instance name followed by a :, and an element name. When the instance name is left out, a default is used; the default is calculated relative to the module instance in which the code is located. For fields, the default is simply the instance, while for attributes, the default is the earliest domain instance defining the element in the instance’s domain search path.

Dynids are explicitly reference counted; you must register any dynid to which you hold a reference beyond the end of a time step and must cancel the reference when you no longer hold it. Failure to register can lead to very strange and hard-to-debug results if the memory for the dynid is reused. Failure to cancel results in a memory leak; because code which fails to cancel often runs many times, this usually results in running out of memory. Dynids are not immediately reclaimed upon release of all references; reclamation happens every \texttt{LSE\_garbage\_collection\_interval} timesteps. Reclamation can be forced at any time using \texttt{LSE\_memory\_reclaim}.

To help debug dynid reference counting problems, there are two global parameters. The first, \texttt{LSE\_debug\_dynid\_limit} sets a limit to the number of "live" dynids there can be after reclamation; if there are more unreclaimed dynids than this limit, simulation terminates with an error message and a list of all the unreclaimed dynids is printed. The second parameter, \texttt{LSE\_debug\_dynid\_refs} prints a message to \texttt{LSE\_stderr} for every dynid create, register, cancel, and release operation. Obviously, this will create a very lengthy output and should be used as a last resort.

The following API functions operate on dynids:

\begin{verbatim}
void LSE_dynid_cancel(LSE_dynid_t d);
\end{verbatim}

\textbf{Availability:} Available everywhere
\textbf{Evaluated at:} runtime

Cancel a reference to dynid \texttt{d}. If the reference count goes to zero, the dynid is guaranteed to not be freed or reused until after the current timestep has completed unless \texttt{LSE\_memory\_reclaim} is called.

\begin{verbatim}
LSE_dynid_t LSE_dynid_create(void);
\end{verbatim}

\textbf{Availability:} Available everywhere
\textbf{Evaluated at:} runtime

Returns a fresh dynid, either by removing it from the free list or by allocating it. The reference count is set to \texttt{1}; i.e., when you call this function, you own a reference to the created dynid. The fields of the dynid are filled with zero bytes.

\begin{verbatim}
void LSE_dynid_dump(void);
\end{verbatim}
Chapter 1. Datatypes and global variables

Availability: Available everywhere
Evaluated at: runtime

Print a list of all unreclaimed dynids.

```c
varies *LSE_dynid_element_ptr(LSE_dynid_t d, elementname);
```

Availability: Available everywhere
Evaluated at: runtime

Return the address of the field, attribute, or other element of dynid `d` identified by `elementname`. Fields are identified by `field:module instance:name`. Attributes are identified by `attr:domain instance:name`. Other elements are identified simply by their name.

```c
bool LSE_dynid_eq(LSE_dynid_t d1, LSE_dynid_t d2);
```

Availability: Available everywhere
Evaluated at: runtime

Returns `TRUE` if dynid `d1` and dynid `d2` are the same dynid, `FALSE` otherwise.

**Warning**

This function can return incorrect results if a holder of references to a dynid has not registered those references and as a result the dynid has been reused from the free list.

```c
varies LSE_dynid_get(LSE_dynid_t d, elementname);
```

Availability: Available everywhere
Evaluated at: runtime

Return the field, attribute, or other element of dynid `d` identified by `elementname`. Fields are identified by `field:module instance:name`. Attributes are identified by `attr:domain instance:name`. Other elements are identified simply by their name.

**Note:** If the type of the accessed element is a reference-counted type, the reference count is not incremented.

```c
void LSE_dynid_recreate(LSE_dynid_t d);
```

Availability: Available everywhere
Evaluated at: runtime

Fill the fields of dynid `d` with zero bytes and assign the dynid a new identifying number. This API call is generally used to "reuse" the memory associated with a dynid without the overhead of cancelling the old one and creating the new one.
Warning
Recreating a dynid to which there is more than one outstanding reference will probably confuse the other holders of the reference.

void LSE_dynid_register (LSE_dynid_t d);
Availability: Available everywhere
Evaluated at: runtime
Register a reference to dynid d.

void LSE_dynid_set (LSE_dynid_t d, elementname, varies value);
Availability: Available everywhere
Evaluated at: runtime
Sets the field, attribute, or other element of dynid d identified by elementname to value. Fields are identified by field:module instance:name. Attributes are identified by attr:domain instance:name. Other elements are identified simply by their name.

Note: If the type of the accessed element is a reference-counted type, the reference count of the new value is not incremented and the reference count of the old value is not decremented.

Reference-counted types
Very rudimentary support is provided for reference-counted types. The base type is LSE_refcount_t. Users may use this type directly or may "embed" it within another type (as do LSE_dynid_t and LSE_resolution_t.)
Reference counting is not performed automatically; all reference counting must be done explicitly using LSE_refcount_register and LSE_refcount_cancel calls.
Reference counted objects can be allocated either dynamically or statically; statically allocated objects are never freed or placed onto a free list. It is possible to choose whether dynamically allocated objects are to be freed or to be placed onto a free list.
The API functions for reference-counted types are:

LSE_refcount_t *LSE_refcount_alloc(size_t size, LSE_refcount_t **freelist);
Chapter 1. Datatypes and global variables

**Availability:** Available everywhere

**Evaluated at:** runtime

Returns the head of the free list pointed to by `freelist`, if the list is non-NULL and non-empty, removing the head from the list as a side effect. Otherwise, dynamically allocates a reference-counted structure of size `size` and returns a pointer to it.

```c
void LSE_refcount_cancel(LSE_refcount_t *s, LSE_refcount_t **freelist);
```

**Availability:** Available everywhere

**Evaluated at:** runtime

Decrements the reference count of the structure pointed to by `s`. If the count reaches zero and the `freelist` argument is non-NULL, the structure is added to the free list pointed to by `freelist`. If the count reaches zero and the `freelist` argument is NULL, the structure is immediately freed.

```c
unsigned int LSE_refcount_get(LSE_refcount_t *s);
```

**Availability:** Available everywhere

**Evaluated at:** runtime

Returns the reference count of the structure pointed to by `s`.

```c
void LSE_refcount_init(LSE_refcount_t *s);
```

**Availability:** Available everywhere

**Evaluated at:** runtime

Mark that the structure pointed to by `s` is statically allocated.

```c
void LSE_refcount_register(LSE_refcount_t *s);
```

**Availability:** Available everywhere

**Evaluated at:** runtime

Increments the reference count of the structure pointed to by `s`.

## Signal values

The signal values of a port have type `LSE_signal_t`. This type is rather unusual; it can hold the values of all signals on a port simultaneously and is often used for this purpose. This is accomplished by placing each signal’s values in different fields of the type and providing different constants and accessor macros for each signal value.

The basic signal value constants are:

**Table 1-2. Signal value constants (of type LSE_signal_t)**

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
</table>

5
### Chapter 1. Datatypes and global variables

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSE_signal_unknown</td>
<td>Indicates that the signal value is not resolved (i.e. not known) yet.</td>
</tr>
<tr>
<td></td>
<td>Applicable to any signal.</td>
</tr>
<tr>
<td>LSE_signal_something</td>
<td>A &quot;yes&quot; value for the data signal. The dynamic message and any</td>
</tr>
<tr>
<td></td>
<td>other associated data on the port are valid.</td>
</tr>
<tr>
<td>LSE_signal_nothing</td>
<td>A &quot;no&quot; value for the data signal. The dynamic message and any</td>
</tr>
<tr>
<td></td>
<td>other associated data on the port are not valid.</td>
</tr>
<tr>
<td>LSE_signal_ack</td>
<td>A &quot;yes&quot; value for the ack signal.</td>
</tr>
<tr>
<td>LSE_signal_nack</td>
<td>A &quot;no&quot; value for the ack signal.</td>
</tr>
<tr>
<td>LSE_signal_enabled</td>
<td>A &quot;yes&quot; value for the enable signal.</td>
</tr>
<tr>
<td>LSE_signal_disabled</td>
<td>A &quot;no&quot; value for the enable signal.</td>
</tr>
<tr>
<td>LSE_signal_all_yes</td>
<td>A &quot;yes&quot; value for all signals on a port.</td>
</tr>
<tr>
<td>LSE_signal_all_no</td>
<td>A &quot;no&quot; value for all signals on a port.</td>
</tr>
</tbody>
</table>

Signal value constants may be combined by using the bit-wise OR operator (\(\mid\)). Do not attempt to use the bit-wise OR operator to perform a boolean OR of two signal values other than constants. Likewise, do not attempt to use the bit-wise AND (\(\&\)), bit-wise negation (\(\sim\)), or bit-wise exclusive-OR (\(^\ast\)) operators on signal values. There are special OR, AND, and negation operations provided by the API.

There are several functions defined by the API which can be used to test values of the signals. It is better to use these functions than to extract signal values and compare them because common compilers (i.e. gcc) are better able to optimize the implementations of the API functions in some cases.

The following API functions are used to access and manipulate signal values:

```c
bool LSE_signal_ack_known(LSE_signal_t sig);
Availability: Available everywhere
Evaluated at: runtime
Tests the ack signal value recorded in sig. Returns TRUE if the value is not
LSE_signal_unknown, FALSE otherwise.
```

```c
bool LSE_signal_ack_present(LSE_signal_t sig);
Availability: Available everywhere
Evaluated at: runtime
Tests the ack signal value recorded in sig. Returns TRUE if the value is "yes" (LSE_signal_ack),
FALSE otherwise.
```

```c
LSE_signal_t LSE_signal_ack2data(LSE_signal_t sig);
Availability: Available everywhere
Evaluated at: runtime
Returns a signal value constructed by extracting the value of the ack signal recorded in sig and
moving it to the data signal.
```
LSE_signal_t LSE_signal_ack2enable(LSE_signal_t sig);
Availability: Available everywhere
Evaluated at: runtime

Returns a signal value constructed by extracting the value of the ack signal recorded in sig and
moving it to the enable signal.

LSE_signal_t LSE_signal_and(LSE_signal_t sig1, LSE_signal_t sig2);
Availability: Available everywhere
Evaluated at: runtime

Return the boolean AND of each matched pair of signals recorded in sig1 and sig2. In other
words, sig1.data is ANDed with sig2.data, etc.

bool LSE_signal_data_known(LSE_signal_t sig);
Availability: Available everywhere
Evaluated at: runtime

Tests the data signal value recorded in sig. Returns TRUE if the value is not
LSE_signal_unknown, FALSE otherwise.

bool LSE_signal_data_present(LSE_signal_t sig);
Availability: Available everywhere
Evaluated at: runtime

Tests the data signal value recorded in sig. Returns TRUE if the value is "yes"
(LSE_signal_something), FALSE otherwise.

LSE_signal_t LSE_signal_data2ack(LSE_signal_t sig);
Availability: Available everywhere
Evaluated at: runtime

Returns a signal value constructed by extracting the value of the data signal recorded in sig and
moving it to the ack signal.

LSE_signal_t LSE_signal_data2enable(LSE_signal_t sig);
Availability: Available everywhere
Evaluated at: runtime

Returns a signal value constructed by extracting the value of the data signal recorded in sig and
moving it to the enable signal.

bool LSE_signal_enable_known(LSE_signal_t sig);
Availability: Available everywhere
Evaluated at: runtime

Tests the enable signal value recorded in sig. Returns TRUE if the value is not
LSE_signal_unknown, FALSE otherwise.
bool LSE_signal_enable_present(LSE_signal_t sig);

**Availability:** Available everywhere

**Evaluated at:** runtime

Tests the enable signal value recorded in `sig`. Returns TRUE if the value is “yes” (LSE_signal_enabled), FALSE otherwise.

LSE_signal_t LSE_signal_enable2ack(LSE_signal_t sig);

**Availability:** Available everywhere

**Evaluated at:** runtime

Returns a signal value constructed by extracting the value of the enable signal recorded in `sig` and moving it to the `ack` signal.

LSE_signal_t LSE_signal_enable2data(LSE_signal_t sig);

**Availability:** Available everywhere

**Evaluated at:** runtime

Returns a signal value constructed by extracting the value of the enable signal recorded in `sig` and moving it to the `data` signal.

LSE_signal_t LSE_signal_extract_ack(LSE_signal_t sig);

**Availability:** Available everywhere

**Evaluated at:** runtime

Returns a signal value constructed by extracting just the value of the `ack` signal from `sig`.

LSE_signal_t LSE_signal_extract_data(LSE_signal_t sig);

**Availability:** Available everywhere

**Evaluated at:** runtime

Returns a signal value constructed by extracting just the value of the `data` signal from `sig`.

LSE_signal_t LSE_signal_extract_enable(LSE_signal_t sig);

**Availability:** Available everywhere

**Evaluated at:** runtime

Returns a signal value constructed by extracting just the value of the enable signal from `sig`.

LSE_signal_t LSE_signal_not(LSE_signal_t sig);

**Availability:** Available everywhere

**Evaluated at:** runtime

Return the boolean negation of the signals recorded in `sig`.

LSE_signal_t LSE_signal_or(LSE_signal_t sig1, LSE_signal_t sig2);
Chapter 1. Datatypes and global variables

Availability: Available everywhere
Evaluated at: runtime

Return the boolean OR of each matched pair of signals recorded in \( sig1 \) and \( sig2 \). In other words, \( sig1.data \) is ORed with \( sig2.data \), etc.

```c
void LSE_signal_print(FILE *fp, LSE_signal_t sig);
```

Availability: Available everywhere
Evaluated at: runtime

Print the signal values recorded in \( sig \) to file \( fp \).

Speculation resolutions

The \( \text{LSE}_\text{resolution}_t \) type holds information about the resolution of a particular speculation decision. Instances of this type are known as \emph{resolutions}.

Resolutions are of a particular class which indicates their meaning. The possible classes are listed in the enumerated type \( \text{LSE}_\text{resolution}_\text{class}_t \) and are:

Table 1-3. Resolution class values (of type \( \text{LSE}_\text{resolution}_\text{class}_t \))

<table>
<thead>
<tr>
<th>Class name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{LSE}<em>\text{resolution}</em>\text{confirm} )</td>
<td>Resolution confirms a speculative decision; all instructions listed in the ( ids ) element are not speculative due to the speculation performed at the resolved instruction (though they may still be speculative).</td>
</tr>
<tr>
<td>( \text{LSE}<em>\text{resolution}</em>\text{redecide} )</td>
<td>Resolution indicates an incorrect speculative decision; all instructions listed in the ( ids ) element were incorrectly executed due to the mis-speculation.</td>
</tr>
</tbody>
</table>

Resolutions have several user-accessible elements. These elements are listed in the following table:

Table 1-4. Speculation resolution type (\( \text{LSE}_\text{resolution}_t \)) elements

<table>
<thead>
<tr>
<th>Element name</th>
<th>Element type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ids )</td>
<td>( \text{LSE}_\text{dynid}_t ) *</td>
<td>A list of dynids representing instructions affected by the resolution</td>
</tr>
<tr>
<td>( \text{num}_\text{id}s )</td>
<td>int</td>
<td>The number of dynids affected by the resolution</td>
</tr>
<tr>
<td>( \text{rclass} )</td>
<td>( \text{LSE}<em>\text{resolution}</em>\text{class}_t )</td>
<td>The kind of resolution</td>
</tr>
<tr>
<td>( \text{resolved}_\text{inst} )</td>
<td>( \text{LSE}_\text{dynid}_t )</td>
<td>A dynid representing the instruction being resolved</td>
</tr>
</tbody>
</table>

Both module instances and domain instances can add elements to the resolutions. When module instances do so, they are called \emph{fields}. When domain instances do so, they are called \emph{attributes}. Fields and attributes are named using a special tag (\texttt{field:} or \texttt{attr:}), an optional instance name followed by a \texttt{:}, and an element name. When the instance name is left out, a default is used; the default is calculated...
relative to the module instance in which the code is located. For fields, the default is simply the instance, while for attributes, the default is the earliest domain instance defining the element in the instance’s domain search path.

Resolutions are explicitly reference counted; you must register any resolution to which you hold a reference beyond the end of a time step and must cancel the reference when you no longer hold it. Failure to register can lead to very strange and hard-to-debug results if the memory for the resolution is reused. Failure to cancel results in a memory leak. Resolutions are not immediately reclaimed upon release of all references; reclamation happens every $LSE_{\text{garbage\_collection\_interval}}$ timesteps. Reclamation can be forced at any time using $LSE_{\text{memory\_reclaim}}$.

To help debug resolution reference counting problems, there is one global parameter. This parameter, $LSE_{\text{debug\_resolution\_limit}}$, sets a limit to the number of “live” resolutions there can be after reclamation; if there are more unreclaimed resolutions than this limit, simulation terminates with an error message and a list of all the unreclaimed resolutions is printed.

The following API functions operate on resolutions:

```c
void LSE_resolution_add_dynid(LSE_resolution_t r, LSE_dynid_t d, boolean register);

Availability: Available everywhere
Evaluated at: runtime

Add a dynid $d$ to the list of affected instructions in resolution $r$. If $register$ is TRUE, increments the reference count for $d$; otherwise, the reference is “transferred” to the resolution and the caller should not cancel the reference.

void LSE_resolution_cancel(LSE_resolution_t r);

Availability: Available everywhere
Evaluated at: runtime

Cancel a reference to resolution $r$. If the reference count goes to zero, the resolution is guaranteed to not be freed or reused until after the current timestep has completed unless $LSE_{\text{memory\_reclaim}}$ is called.

LSE_resolution_t LSE_resolution_create(LSE_dynid_t rinst, LSE_resolution_class_t rclass);

Availability: Available everywhere
Evaluated at: runtime

Returns a fresh resolution instance, either by removing it from the free list or by allocating it. The reference count is set to 1; i.e., when you call this function, you own a reference to the created resolution. The resolution class is set to $rclass$ and the resolved instruction is set to $rinst$. The fields of the resolution are filled with zero bytes.

void LSE_resolution_dump(void);

Availability: Available everywhere
Evaluated at: runtime

Print a list of all unreclaimed resolutions.
varies *LSE_resolution_element_ptr (LSE_resolution_t r, elementname);

**Availability:** Available everywhere

**Evaluated at:** runtime

Return the address of the field, attribute, or other element of resolution \( r \) identified by

\( \text{elementname} \). Fields are identified by field:module instance:name. Attributes are

identified by attr:domain instance:name. Other elements are identified simply by their name.

---

varies LSE_resolution_get (LSE_resolution_t r, elementname);

**Availability:** Available everywhere

**Evaluated at:** runtime

Return the field, attribute, or other element of resolution \( r \) identified by \( \text{elementname} \). Fields are

identified by field:module instance:name. Attributes are identified by attr:domain

instance:name. Other elements are identified simply by their name.

**Note:** If the type of the accessed element is a reference-counted type, the reference count is

not incremented.

---

void LSE_resolution_register(LSE_resolution_t r);

**Availability:** Available everywhere

**Evaluated at:** runtime

Register a reference to resolution \( r \).

---

void LSE_resolution_set(LSE_resolution_t r, elementname, varies

value);

**Availability:** Available everywhere

**Evaluated at:** runtime

Sets the field, attribute, or other element of resolution \( r \) identified by \( \text{elementname} \) to \( \text{value} \).

Fields are identified by field:module instance:name. Attributes are identified by

attr:domain instance:name. Other elements are identified simply by their name.

**Note:** If the type of the accessed element is a reference-counted type, the reference count for

the new value is not incremented and the reference count for the old value is not decremented,

except for the resolved_inst element.
Time types

A time has two parts: a cycle count and a phase count. There are a constant number of phases per cycle; this number is set by the `LSE_phases` parameter. The normal type for time is `LSE_type_t`. With this type, times can be compared. A value of `LSE_type_t` can be constructed from a number of cycles and a number of phases and the cycles and phases can be extracted from the time. It is also possible to add, subtract, and compare times.

The type used for both the number of cycles and the number of phases is `LSE_time_numticks_t`. This type is also used to provide a "flat" time value that is compatible with integer operations.

Open Issue

We do not actually support multiple phases per cycle in our standard library (though one could do it in a custom library). The way in which cycles and phases are used is subject to change, particularly when we move to multiple clock domains.

There are three time constants:

Table 1-5. Time constants

<table>
<thead>
<tr>
<th>Constant name</th>
<th>Datatype</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSE_time_zero_ticks</td>
<td>LSE_time_numticks_t</td>
<td>A zero time value</td>
</tr>
<tr>
<td>LSE_time_one_cycle</td>
<td>LSE_time_t</td>
<td>One timestep</td>
</tr>
<tr>
<td>LSE_time_zero</td>
<td>LSE_time_t</td>
<td>A zero time value</td>
</tr>
</tbody>
</table>

The current simulation time is stored in the variable `LSE_time_now`. Do not attempt to modify this variable.

The following API functions manipulate time values. Any of them may be implemented as macros, so arguments with side effects should not be used with them.

```c
LSE_time_t LSE_time_add(LSE_time_t ta, LSE_time_t tb);
```

**Availability:** Available everywhere  
**Evaluated at:** runtime

Returns the sum of time `ta` and time `tb`.

```c
LSE_time_t LSE_time_construct(LSE_time_numticks_t cycles, LSE_time_numticks_t phases);
```

**Availability:** Available everywhere  
**Evaluated at:** runtime

Constructs a time value equal to `cycles` cycles plus `phases` phases. If `phases` is greater than `LSE_phases`, it is converted correctly to number of cycles and phases, thus allowing use of this function to convert from `LSE_time_numticks_t` to `LSE_time_t`.  

12
boolean LSE_time_eq(LSE_time_t ta, LSE_time_t tb);
boolean LSE_time_ge(LSE_time_t ta, LSE_time_t tb);
boolean LSE_time_gt(LSE_time_t ta, LSE_time_t tb);
boolean LSE_time_le(LSE_time_t ta, LSE_time_t tb);
boolean LSE_time_lt(LSE_time_t ta, LSE_time_t tb);
boolean LSE_time_ne(LSE_time_t ta, LSE_time_t tb);

**Availability:** Available everywhere

**Evaluated at:** runtime

Returns **TRUE** if the relation implied by the name (e.g. less than) holds between **ta** and **tb**, **FALSE** otherwise.

LSE_time_numticks_t LSE_time_get_cycle(LSE_time_t ta);

**Availability:** Available everywhere

**Evaluated at:** runtime

Returns the number of cycles in time **ta**.

LSE_time_numticks_t LSE_time_get_phase(LSE_time_t ta);

**Availability:** Available everywhere

**Evaluated at:** runtime

Returns the number of phases in time **ta**.

**Literal** LSE_time_print_args(LSE_time_t ta);

**Availability:** Available everywhere

**Evaluated at:** runtime

This API call is a macro which expands to an argument list which can be used by printf(3) to print out the value of time **ta** in a standard format. For example:

```c
LSE_time_t t;
...
printf(LSE_time_print_args(t));
```

LSE_time_t LSE_time_sub(LSE_time_t ta, LSE_time_t tb);

**Availability:** Available everywhere

**Evaluated at:** runtime

Returns the difference of time **ta** and time **tb**.

LSE_time_numticks_t LSE_time_ticks(LSE_time_t ta);

**Availability:** Available everywhere

**Evaluated at:** runtime

Returns the number of time steps represented by time **ta**.
Chapter 1. Datatypes and global variables

User-defined types

Types defined by the user in .lss files are available for use inside triple-angle-brackets in .lss files by use of the dollar-curly ($()) notation. In addition, types which are exported from lss to the module using the export statement are available to all code of instances of the module using the name given in the export statement.

Values of user-defined enumerated types must be referred to using dollar-curly notation inside of triple-angle-brackets. For exported types, the values should be accessed using the LSE_enum_value macro, which has the following definition:

\[
\text{varies LSE_enum_value} (\text{typename}, \text{valuename});
\]

**Availability:** Everywhere within a module instance

**Evaluated at:** code generation

Returns the enumerated value \textit{valuename} of type \textit{typename}. The type name must be a type name which has been exported to the module from lss.

Global variables

The global variables available are:

- \textbf{int LSE_sim_exit_status} - the exit status which the simulator will return when it finishes. This variable is initialized to 0.

- \textbf{int LSE_sim_terminate_now} - When this variable is set to a non-zero value, simulation terminates at the end of the current timestep. Positive numbers are considered to be "normal" terminations, while negative numbers are used for "error" terminations. This variable is initialized to 0.

- \textbf{int LSE_sim_terminate_count} - This variable is used by domains to indicate "normal" termination of the simulation, particularly by the emulation domain class. The variable is used as a counter of the number of "live" domain instances; when it reaches zero, simulation ends. However, this condition is not enabled if there are no domains that actually use this variable; LSE knows that they do because the domain class files define an attribute for this. The variable is initialized to 0.

- \textbf{FILE \* LSE_stderr} - a file for diagnostic messages. All diagnostic output should go to this file. The command-line processor normally initializes this variable.

- \textbf{LSE_time_t LSE_time_now} - the current time. Do not attempt to modify this variable.

Global constants

There is one additional global constant beyond those mentioned previously in this chapter. This constant is called \textbf{LSE_instance_name}. This constant is a string, with double-quotes, containing the name of the instance whose code is being generated. It is available everywhere, but is most useful in data collectors, where LSS cannot give you the instance name directly in some cases.
Chapter 2. Core API functions

This chapter describes functions available to users of the Liberty Simulation Environment. Functions closely related to datatypes are given in Chapter 1.

Code generation directives

There are several directives defined which parallel C pre-processor directives but which are evaluated at code generation time instead of compile time. Code generation directives are marked by the token \#LSE before them. All are available everywhere. The directives are:

- \#LSE if - works like a C pre-processor if evaluated at code generation.
- \#LSE else - works like a C pre-processor else evaluated at code generation.
- \#LSE endif - works like a C pre-processor endif evaluated at code generation.

Codepoint functions

LSE_signal_t LSE_controlpoint_call_default(void);
Availability: control points
Evaluated at: runtime
   Calls the default for the control function currently being executed, using the current value of the arguments to the control function.

LSE_signal_t LSE_controlpoint_call_empty(void);
Availability: control points
Evaluated at: runtime
   Call the "empty" control function (i.e. one that passes through its signals), using the current value of the arguments to the control function.

varies LSE_userpoint_call_default(...);
Availability: user points
Evaluated at: runtime
   Calls the default for the user point currently being executed, using the arguments given in the call.

boolean LSE_userpoint_defaulted(userpoint);
Chapter 2. Core API functions

Availability: Everywhere within a module instance
Evaluated at: code generation

Returns **TRUE** if user point `userpoint` has its default value, **FALSE** otherwise.

```c
boolean LSE_userpoint_empty(userpoint);
```

Availability: Everywhere within a module instance
Evaluated at: code generation

Returns **TRUE** if user point `userpoint` is empty, **FALSE** otherwise.

```c
varies LSE_userpoint_invoke(userpoint, ...);
```

Availability: Everywhere within a module instance
Evaluated at: runtime

Calls the user point `userpoint` with the given parameters.

---

**Data manipulation functions**

```c
void LSE_data_cancel(type, type *datap);
```

Availability: Everywhere
Evaluated at: runtime

If `type` is a reference-counted type, cancel a reference to the value pointed to by `datap`.

```c
void LSE_data_copy(type, type *dest, type *src);
```

Availability: Everywhere
Evaluated at: runtime

Copy the value of type `type` pointed to by `src` to `dest`, incrementing the reference count of the value if the type is a reference-counted type.

```c
void LSE_data_move(type, type *dest, type *src);
```

Availability: Everywhere
Evaluated at: runtime

Copy the value of type `type` pointed to by `src` to `dest`, doing no reference counting operations.

```c
void LSE_data_register(type, type *datap);
```

Availability: Everywhere
Evaluated at: runtime

If `type` is a reference-counted type, register a reference to the value pointed to by `datap`. 
Event functions

boolean LSE_event_filled(eventname);
Availability: Everywhere
Evaluated at: code generation

Returns TRUE if there is a data collector for event eventname of the module instance whose code is being generated, FALSE otherwise.

void LSE_event_record(eventname, ...);
Availability: Everywhere
Evaluated at: runtime

Record that event eventname for the module instance whose code is being generated has occurred with the arguments given.

Instance accessors

literal FUNC(funcname, ...);
Availability: Everywhere within a module instance
Evaluated at: code generation

Return the module-instance-unique identifier for instance method funcname. The remaining arguments are appended as an argument list. Can be used both for method definition and calls.

literal FUNCPTR(funcname);
Availability: Everywhere within a module instance
Evaluated at: code generation

Return the module-instance-unique identifier for instance method funcname.

literal GLOB(varname);
Availability: Everywhere within a module instance
Evaluated at: code generation

Return the name of the module-instance-unique variable varname.

GLOBDEF(type, varname);
Availability: Everywhere within a module instance
Evaluated at: code generation

Define a module-instance-unique variable varname with type type.
**Chapter 2. Core API functions**

```plaintext
literal HANDLER(portname, ...);

Availability: .clm files
Evaluated at: code generation
```

Return the module-instance-unique identifier for a handler on port `portname`. The remaining arguments are appended as an argument list. Can be used both for handler definition and calls.

```plaintext
boolean LSE_parm_constant(instance : parameter);

Availability: Everywhere
Additional Availability: structadds
Evaluated at: code generation
```

Returns `TRUE` if the value of parameter `parameter` of module instance `instance` is a constant (i.e. it has not been made a run-time parameter). If the instance name is left out, parameters of the instance whose code is being generated are accessed. Top-level parameters cannot be examined with this API. An instance name cannot be specified within a .clm file.

**Note:** Some modules may define parameters conditionally; i.e. the parameter only exists if another parameter has a specific value. For such modules, you may only look at the conditional parameters if you have guaranteed that they exist, whether by knowledge of the configuration or by guarding the use with `#LSE if` directives.

```plaintext
varies PARM(instance : parameter);

Availability: Everywhere
Additional Availability: structadds
Evaluated at: code generation/runtime
```

Return the value of parameter `parameter` of module instance `instance`. If the instance name is left out, parameters of the instance whose code is being generated are accessed. Top-level parameters cannot be accessed in this fashion; use `@{} notation to do so. An instance name cannot be specified within a .clm file.

This function is normally evaluated at code generation; however, if the simulator configuration has declared the parameter to be a runtime parameter, this function is evaluated at runtime.

**Note:** Some modules may define parameters conditionally; i.e. the parameter only exists if another parameter has a specific value. For such modules, you may only use the conditional parameters if you have guaranteed that they exist, whether by knowledge of the configuration or by guarding the use with `#LSE if` directives.
Chapter 2. Core API functions

Looping constructs

A looping construct to loop over all instances of a port is provided through the macro
LSE_LOOP_OVER_PORT { ... } LSE_LOOP_END. The use of this looping construct is demonstrated in the following code snippet:

LSE_LOOP_OVER_PORT(porti,myoutport) {
  ...
} LSE_LOOP_END;

The first argument is the loop index variable while the second argument is the port name. Looping constructs are available within .clm files, control points, user points, data collectors, and funcheaders at code generation time. The port referenced can be a port of another module instance (using the instance : port notation) outside of .clm files.

Warning
Failure to use the curly braces will result in hard-to-debug errors during build.

Method and query functions

varies LSE_method_call(instname : methodname, ...);

Availability: control points, data collectors, user points
Evaluated at: runtime

Calls method methodname of module instance instname with the remaining arguments. If the instance name is left out, the instance whose code is being generated is used. An instance name cannot be specified within a .clm file.

boolean LSE_method_used(queryname);

Availability: Everywhere within a module instance
Evaluated at: code generation

Returns TRUE if method methodname of the the module instance whose code is being generated is called by some control point, user point, or data collector, FALSE otherwise.

varies LSE_query_call(instname : queryname, int qinstno, ...);

Availability: control points, data collectors, user points
Evaluated at: runtime

Calls query queryname of module instance instname with the arguments listed after qinstno. If the instance name is left out, the instance whose code is being generated is used. An instance name cannot be specified within a .clm file. The argument qinstno is used as an identifying number for the query call so that later calls that are intended to be the same call can be matched with information from the previous call.
Chapter 2. Core API functions

boolean LSE_query_used(queryname);
Availability: Everywhere within a module instance
Evaluated at: code generation

Returns TRUE if query queryname of the the module instance whose code is being generated is called by some control point, user point, or data collector, FALSE otherwise.

void LSE_query_results_changed(void);
Availability: .clm files, funcheaders
Evaluated at: runtime

See the entry for this function in the Section called Scheduling functions.

Miscellaneous functions

literal LSE_eval(expression);
Availability: Everywhere
Evaluated at: code generation

Evaluates expression as a Python expression and returns the "stringified" value of the expression as a literal. This function should principally be used to create conditions for #if and #LSE if.

boolean LSE_nameequal(name1, name2);
Availability: Everywhere
Evaluated at: code generation

Returns TRUE if name1 and name2 are the same text. Be careful not to put parenthesis and extra whitespace around the names.

Port functions

boolean LSE_port_connected(instance : port [ porti ]);  
Availability: Everywhere
Additional Availability: event definitions, method definitions, query definitions, structadds, user point definitions
Evaluated at: runtime/code generation

Returns TRUE if instance porti of port port of module instance instance is connected, FALSE otherwise. If the instance name is left out, the instance whose code is being generated is used. An instance name cannot be specified within a .clm file.

If the port index is left out, returns TRUE if any port instance is connected, FALSE otherwise.
This function is evaluated at runtime when the port index is specified and at code generation when it is left out. The locations where the function is available listed under “Additional Availability” are only available when no port index is specified.

LSE_signal_t LSE_port_get(port [ porti ]. signal, LSE_dynid_t *idp, varies **datapp);

Availability: .clm files, funcheaders, user points
Evaluated at: runtime

Returns the value of signal signal of instance porti of port port of the module instance whose code is currently being generated. If the .signal parameter is left out, all signals of the port are returned.

When the data signal’s value is returned, the dynamic message identifier and pointer to associated data are set in the variables pointed to idp and datapp, respectively, when these pointers are not NULL. Note that the dynamic message identifier and associated data are valid only when the data signal is LSE_signal_something. Also, the data is valid until just before the phase_end method of the module instance is called, unless the port is "independent", in which case the data is valid until the end of that method.

When the data signal’s value is not returned, idp and datapp must not be supplied.

Warning
Do not modify data pointed to by datapp

boolean LSE_port_get_flag(port [ porti ]);  

Availability: .clm files, funcheaders, user points
Evaluated at: runtime

Returns the flag value for port instance porti of port port of the module instance whose code is currently being generated. This flag is a simple flag modules can use as desired by the module author. The flag is automatically cleared to FALSE between time steps.

int LSE_port_num_connected(instance : port);

Availability: Everywhere
Additional Availability: event definitions, method definitions, query definitions, structadds, user point definitions
Evaluated at: code generation

Returns the number of connected port instances of the port port of module instance instance. If the instance name is left out, the instance whose code is being generated is used. An instance name cannot be specified within a .clm file.
Chapter 2. Core API functions

LSE_signal_t LSE_port_query(instance : port [ porti ] . signal,
LSE_dynid_t *idp, varies **datapp);

Availability: control points, data collectors, user points
Evaluated at: runtime

Returns the value of signal signal of instance porti of port port of module instance
instance. If the .signal argument is left out, all signals of the port are returned. If the instance
name is left out, the instance whose code is being generated is used.

LSE_port_query should be used for all access to signal values from control points. It should also
be used in user points when the queried module instance is not the calling instance or when the user
point is called from a block of code which does not have a data dependency, whether implied or
explicit, on the signal being queried. It can also be used within data collectors.

When the data signal’s value is returned, the dynamic message identifier and pointer to associated
data are set in the variables pointed to idp and datap, respectively, if these pointers are not
NULL. Note that the dynamic message identifier and associated data are valid only when the data
signal is LSE_signal_something. Also, the data is valid until just before the phase_end method
of the module instance is called, unless the port is "independent", in which case the data is valid
until the end of that method.

When the data signal’s value is not returned, idp and datap must not be supplied.

Warning

Do not modify data pointed to by datap

void LSE_port_set(port [ porti ] . signal, LSE_signal_t sigval,
LSE_dynid_t id, varies *datap);

Availability: .clm files, funcheaders, user points
Evaluated at: runtime

Sets the signal signal of instance porti of port port of module instance instance to
sigval. If the .signal parameter is left out, all output signals of the port are set. Extra signal
values encoded in sigval are ignored.

When the data signal is set to LSE_signal_something, dynamic message identifier id and the
pointer to associated data datap are stored as part of the port’s value. The parameters must be
given a value (even if it is simply NULL whenever the data signal is set to any value.

Any data pointed to by the datap signal must remain constant throughout the rest of the timestep.
It may not change until the phase_end method of the module is run.

If the signal has already been set in this timestep, additional calls to set the signal must set the same
signal value, the same dynamic message, and the same associated data pointer.

void LSE_port_set_flag(port [ porti ]);
Chapter 2. Core API functions

**Availability**: .clm files, funcheaders, user points

**Evaluated at**: runtime

Sets the flag value for port instance porti of port port of the module instance whose code is currently being generated to TRUE. This flag is a simple flag modules can use as desired by the module author. The flag is automatically cleared to FALSE between time steps.

```c
typedef LSE_port_type(instance : port);
```

**Availability**: Everywhere

**Additional Availability**: event definitions, method definitions, query definitions, structadds, user point definitions

**Evaluated at**: code generation

Returns the C type of the associated data on the port port of module instance `instance`. If the instance name is left out, the instance whose code is being generated is used. An instance name cannot be specified within a .clm file.

```c
int LSE_port_width(instance : port);
```

**Availability**: Everywhere

**Additional Availability**: event definitions, method definitions, query definitions, structadds, user point definitions

**Evaluated at**: code generation

Returns the width of the port port of module instance `instance`. If the instance name is left out, the instance whose code is being generated is used. An instance name cannot be specified within a .clm file.

### Reporting functions

**void LSE_report_err(const char *fmt, ...);**

**Availability**: Everywhere

**Evaluated at**: runtime

Prints a message prepended with the time and instance name of the module instance reporting the error to `LSE_stderr` and stops simulation at the end of the current time step. This function takes the same arguments as `printf(3)`.

**void LSE_report_err_at_codegen(...);**

**Availability**: Everywhere

**Evaluated at**: runtime

Prints the parameter list as if it were a code generation error message, including line number and file or piece of code where the function was called. Causes code generation to fail.

**void LSE_report_err_at_cpp(...);**
Chapter 2. Core API functions

Availability: Everywhere
Evaluated at: runtime

Prints the parameter list as if it were a C pre-processor error message. Causes C pre-processing of
the code of the module instance being compiled to fail.

\[
\text{void } \text{LSE_report_warn}(\text{const char }* \text{fmt}, ...); \\
\text{Availability: Everywhere} \\
\text{Evaluated at: runtime} \\
\]

Prints a message prepended with the time and instance name of the module instance reporting the
derror to LSE_stderr. This function takes the same arguments as printf(3).

\[
\text{void } \text{LSE_report_warn_at_codegen}(...); \\
\text{Availability: Everywhere} \\
\text{Evaluated at: runtime} \\
\]

Prints the parameter list as if it were a code generation warning message, including line number and
file or piece of code where the function was called.

\[
\text{void } \text{LSE_report_warn_at_cpp}(...); \\
\text{Availability: Everywhere} \\
\text{Evaluated at: runtime} \\
\]

Prints the parameter list as if it were a C pre-processor warning message.

Scheduling functions

\[
\text{void } \text{LSE_query_results_changed}(\text{void}); \\
\text{Availability: .clm files, funcheaders} \\
\text{Evaluated at: runtime} \\
\]

Indicates that the results of previously "unresolved" query return values might have become
resolved due to some computation within a phase method or handler. Should be called whenever
this has occurred so that any queries waiting for their return value to be resolved can be run. Note
that the module code cannot find out whether there actually are any queries waiting, but it can find
out whether particular queries are actually called by anyone; if no query is called by anyone, then
there is no need to call this function.

\[
\text{void } \text{LSE_sim_force_phase}(\text{void}); \\
\text{Availability: .clm files, funcheaders} \\
\text{Evaluated at: runtime} \\
\]

Force the phase method, if any, of the module instance calling this function to be run after all
phase_start methods have run. This function is only guaranteed to be effective when called in
the `phase_start` method or a function called by that method.

This function should be called in `phase_start` by any module which might compute an output signal without looking at input signals, but which needs to call a user point to help compute the output signal. In such a case, the user point could call a query or port query, but queries do not cause proper scheduling when called from `phase_start`. Instead, the computation of the output signal should happen in `phase` and this function should be called in `phase_start` to ensure that `phase` gets invoked.

```c
void LSE_sim_keep_alive(LSE_time_t t);
```

**Availability:** `.clm` files, control points, funcheaders, user points

**Evaluated at:** runtime

Ensure that the simulation will have a timestep no later than \( t \) units of time later and will not terminate for lack of scheduled time steps until after that time. Time steps may occur before that time, but are not guaranteed to occur.

While future optimizations may include causing the module to sleep for \( t \) time units, the module must not depend upon this behavior. All modules must assume that `phase` methods and handlers will be called multiple times and `phase_end` and `phase_start` methods will be called once for every time step that actually occurs.

This function should be called by any module which can foresee that any of its output signals change value given that its input signals do not. It is usually called during the `phase_end` method. It should also be called if the return value of any query would change, given the same input arguments. Also, if any externally callable method of the function returns a value dependent upon state which has changed, this function should be called.

This function should also be called by control points or user points which exhibit time-dependent behavior. If the return value of a user point or control point would change in the future given the same input arguments and results of its query calls, then the point should call this function. Similarly, if there is state-dependent behavior in the points and this state changes in such a way that the results might change in the future given the same input arguments and results of query calls, then the user point updating this state should call this function.

In short, if state changes and that change can be seen in any way on the signals, query return values, method return values, or if there is time-dependent behavior, this function must be called to inform LSE that there is something going on in the future.

Time in the simulator normally "skips" ahead to the earliest time recorded by any call to this function. It is important for performance, therefore, to avoid calling this function with a smaller time argument than is needed. However, you must make sure that you never call it with a larger time argument than is needed. If you must err, err on the small side.
Chapter 3. Emulation Interface

This chapter describes datatypes, structure fields, and API functions used to access emulator functionality in the Liberty Simulation Environment. Functions are grouped in this chapter by the capability which makes them available.

Datatypes, structure fields, and constants

The following tables list the datatypes and structure fields used in the emulation interface and what emulator capabilities are required for each field or datatype to exist. If no capability is listed, the datatype or structure field is always available.

Table 3-1. Emulation interface datatypes

<table>
<thead>
<tr>
<th>Datatype name</th>
<th>Capability required</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSE_emu_addr_t</td>
<td>—</td>
<td>An address (for instruction or data) in the ISA.</td>
</tr>
<tr>
<td>LSE_emu_contextno_t</td>
<td>—</td>
<td>A global context number. A number greater than 0 is a hardware context, a number less than 0 is a software context, and 0 is &quot;no context&quot;.</td>
</tr>
<tr>
<td>LSE_contextstate_t</td>
<td>—</td>
<td>Context state enumeration</td>
</tr>
<tr>
<td>LSE_emu_ctoken_t</td>
<td>—</td>
<td>An opaque context token provided by an emulator</td>
</tr>
<tr>
<td>LSE_emu_instr_info_t</td>
<td>—</td>
<td>Dynamic instruction instance information</td>
</tr>
<tr>
<td>LSE_emu_instrstep_name_t</td>
<td>—</td>
<td>Instruction step names</td>
</tr>
<tr>
<td>LSE_emu_operand_info_t</td>
<td>operandinfo</td>
<td>Instruction operand information operands</td>
</tr>
<tr>
<td>LSE_emu_operand_name_t</td>
<td>operandinfo</td>
<td>Instruction operand names</td>
</tr>
<tr>
<td>LSE_emu_operand_val_t</td>
<td>operandval</td>
<td>Instruction operand values</td>
</tr>
<tr>
<td>LSE_emu_spaceaddr_t</td>
<td>—</td>
<td>State space addresses</td>
</tr>
<tr>
<td>LSE_emu_spaceid_t</td>
<td>—</td>
<td>State space identifiers</td>
</tr>
<tr>
<td>LSE_emu_spacetype_t</td>
<td>—</td>
<td>State space type possibilities</td>
</tr>
</tbody>
</table>

Table 3-2. Emulation interface structure fields

<table>
<thead>
<tr>
<th>Field name</th>
<th>Field type</th>
<th>Capability required</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr</td>
<td>LSE_emu_addr_t</td>
<td>—</td>
<td>Address of the instruction</td>
</tr>
<tr>
<td>Field name</td>
<td>Field type</td>
<td>Capability required</td>
<td>Purpose</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------</td>
<td>---------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>branch_dir</td>
<td>int</td>
<td>branchinfo</td>
<td>Which potential next instruction to execute; 0 means the inline instruction</td>
</tr>
<tr>
<td>branch_num_targets</td>
<td>int</td>
<td>branchinfo</td>
<td>Number of potential next instruction, including the inline instruction</td>
</tr>
<tr>
<td>branch_targets</td>
<td>LSE_emu_addr_t[]</td>
<td>branchinfo</td>
<td>Addresses of potential next instructions, including the inline instruction (maximum number is LSE_emu_max_branch_targets)</td>
</tr>
<tr>
<td>hwcontextno</td>
<td>LSE_emu_contextno_t</td>
<td>—</td>
<td>Hardware context number of the instruction</td>
</tr>
<tr>
<td>iclasses</td>
<td>struct { ... }</td>
<td>—</td>
<td>Instruction classes (fields are of form boolean is_class;)</td>
</tr>
<tr>
<td>next_pc</td>
<td>LSE_emu_addr_t</td>
<td>—</td>
<td>Address of next instruction to execute</td>
</tr>
<tr>
<td>operand_dest</td>
<td>LSE_emu_operand_info_t[]</td>
<td>operandinfo</td>
<td>Destination operand information (maximum number is LSE_emu_max_operand_dest)</td>
</tr>
<tr>
<td>operand_src</td>
<td>LSE_emu_operand_info_t[]</td>
<td>operandinfo</td>
<td>Source operand information (maximum number is LSE_emu_max_operand_src)</td>
</tr>
<tr>
<td>operand_val_dest</td>
<td>LSE_emu_operand_val_t[]</td>
<td>operandval</td>
<td>Destination operand values (maximum number is LSE_emu_max_operand_dest)</td>
</tr>
<tr>
<td>operand_val_int</td>
<td>LSE_emu_operand_val_t[]</td>
<td>operandval</td>
<td>Intermediate operand values (maximum number is LSE_emu_max_operand_int and the field does not appear if this constant is 0)</td>
</tr>
</tbody>
</table>
### Field name | Field type | Capability required | Purpose
--- | --- | --- | ---
operand\_val\_src | LSE\_emu\_operand\_val\_t[] | operandval | Source operand values (maximum number is LSE\_emu\_max\_operand\_src)
operand\_written\_dest | boolean[] | operandval | Has the destination operand been written back? (maximum number is LSE\_emu\_max\_operand\_dest)
size | LSE\_emu\_addr\_t | — | Size (in address units)
swcontextno | LSE\_emu\_contextno\_t | — | Software context number of the instruction
swcontexttok | LSE\_emu\_ctoken\_t | — | Emulator context token of the instruction’s software context

**fields of LSE\_emu\_operand\_info\_t**

| Field name | Field type | Capability required | Purpose |
--- | --- | --- | ---
spaceaddr | LSE\_emu\_spaceaddr\_t | operandinfo | State space address
spaceid | LSE\_emu\_spaceid\_t | operandinfo | State space identifier. 0 indicates unused or immediate operand.
uses.reg.bits | uint64\_t | operandinfo | Bits accessed by the operand; a 1 bit indicates that the corresponding bit is accessed. Bit number x’s flag is uses.reg.bits[x/64] & (1LL<<(x%64)). Not valid for state spaces of memory type
uses.mem.flags | int | operandinfo | Flags describing a memory access (read vs. write, atomicity, ordering). Only valid for memory state spaces.
uses.mem.size | unsigned int | operandinfo | Size of the operand access (in bytes). Only valid for memory state spaces.

**fields of LSE\_emu\_spaceaddr\_t (a union)**

| Field name | Field type | Capability required | Purpose |
--- | --- | --- | ---
LSE | int | — | default address field used to signal immediate vs. invalid operands.
### Field name
<table>
<thead>
<tr>
<th>Field name</th>
<th>Field type</th>
<th>Capability required</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>statename</td>
<td>varies</td>
<td>—</td>
<td>Address within the state space named <code>statename</code>. Type is either an integer (of various lengths), an array of bytes, or a string.</td>
</tr>
</tbody>
</table>

### Table 3-3. Emulation interface constants

<table>
<thead>
<tr>
<th>Constant name</th>
<th>Capability required</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSE_emu_instrstep_name_step</td>
<td>—</td>
<td>Name for instruction step <code>step</code>.</td>
</tr>
<tr>
<td>LSE_emu_max_branch_targets</td>
<td><code>branchinfo</code></td>
<td>Maximum possibilities for next instruction.</td>
</tr>
<tr>
<td>LSE_emu_max_instrstep</td>
<td>—</td>
<td>Number of possible evaluation step names.</td>
</tr>
<tr>
<td>LSE_emu_max_operand_dest</td>
<td><code>operandinfo</code></td>
<td>Maximum destination operands in an instruction.</td>
</tr>
<tr>
<td>LSE_emu_max_operand_int</td>
<td><code>operandinfo</code></td>
<td>Maximum intermediate operands in an instruction.</td>
</tr>
<tr>
<td>LSE_emu_max_operand_src</td>
<td><code>operandinfo</code></td>
<td>Maximum source operands in an instruction.</td>
</tr>
<tr>
<td>LSE_emu_memaccess_atomic</td>
<td><code>operandinfo</code></td>
<td>Flag indicating memory access is atomic.</td>
</tr>
<tr>
<td>LSE_emu_memaccess_noaccess</td>
<td><code>operandinfo</code></td>
<td>Flag indicating memory access need not take place.</td>
</tr>
<tr>
<td>LSE_emu_memaccess_read</td>
<td><code>operandinfo</code></td>
<td>Flag indicating memory access is a read.</td>
</tr>
<tr>
<td>LSE_emu_memaccess_write</td>
<td><code>operandinfo</code></td>
<td>Flag indicating memory access is a write.</td>
</tr>
<tr>
<td>LSE_emu_num_statepaces</td>
<td>—</td>
<td>Number of state spaces.</td>
</tr>
<tr>
<td>LSE_emu_operand_name_oper</td>
<td><code>operandinfo</code></td>
<td>Name for operand <code>oper</code>.</td>
</tr>
<tr>
<td>LSE_emu_spaceid_spacename</td>
<td>—</td>
<td>Identifier for space named <code>spacename</code>.</td>
</tr>
<tr>
<td>LSE_emu_spacetype_creg</td>
<td>—</td>
<td>State space consists of control registers.</td>
</tr>
<tr>
<td>LSE_emu_spacetype_mapping</td>
<td>—</td>
<td>State space is a mapping between spaces.</td>
</tr>
<tr>
<td>LSE_emu_spacetype_mem</td>
<td>—</td>
<td>State space consists of memory.</td>
</tr>
<tr>
<td>LSE_emu_spacetype_reg</td>
<td>—</td>
<td>State space consists of simple registers.</td>
</tr>
</tbody>
</table>

### Table 3-4. Emulation interface structure attributes

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Attribute type</th>
<th>Capability required</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>attributes of LSE_dynid_t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instr_info</td>
<td>LSE_emu_instr_info_t</td>
<td>—</td>
<td>Instruction information</td>
</tr>
<tr>
<td></td>
<td>attributes of LSE_resolution_t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>next_addr</td>
<td>LSE_emu_addr_t</td>
<td>—</td>
<td>Address of the next instruction to execute</td>
</tr>
</tbody>
</table>
Core emulation functions

```c
void LSE_emu_call_extra_func(funcname, varies *returnp, ...);
```

**Capability:** Always present

Call the extra function `funcname` in an emulator, putting the return value in the location pointed to by `returnp` and passing it the remaining arguments. The parameter `funcname` must be resolved at code generation time.

```c
int LSE_emu_create_context(LSE_emu_contextno_t cno, boolean automap);
```

**Capability:** Always present

Creates a new context with context number `cno` and automap flag `automap` in emulator instance `emulator`. If the context already exists, sets the automap flag without recreating the context. If `cno` is a hardware context number, a hardware context is created; if `cno` is a software context number, a software context is created. If `automap` is `TRUE`, the context may be mapped by this function. Returns non-zero if unable to create the context.

```c
void LSE_emu_do_instrstep(LSE_dynid_t id, LSE_emu_instrstep_name_t sname);
```

**Capability:** Always present

Perform the step of execution named `sname` for instruction `id`. Instruction information fields of `id` are used and updated as documented by the emulator used. Side effects can occur if the instruction is classified as an instruction with side effects. If needed source operands are unfetched or steps are called in an invalid order, the results are undefined and may cause simulator crashes. Performing the steps in numeric order is always valid.

**Open Issue**

Return values and exceptions

```c
void LSE_emu_doback(LSE_dynid_t id);
```

**Capability:** Always present

Perform the "back end" steps of execution for instruction `id` as defined by the emulator. These steps normally involve operand fetch, evaluation, and writeback. The emulator does not refetch the instruction. Instruction information fields of `id` are used and updated as documented by the emulator used.

Though it is always present, this function may not work with some emulators. Such emulators should include a statement to this effect in their documentation.
Chapter 3. Emulation Interface

Open Issue
Return values and exceptions

void \texttt{LSE\_emu\_docommit}(LSE\_dynid\_t \textit{id});
\textbf{Capability:} Always present

Commit instruction \textit{id}. After this function is called, it is no longer possible to undo the effects of the instruction.

Though it is always present, this function may have restrictions when used with some emulators. Such emulators should include a statement to this effect in their documentation.

Open Issue
Return values and exceptions

void \texttt{LSE\_emu\_dofront}(LSE\_dynid\_t \textit{id});
\textbf{Capability:} Always present

Perform the "front end" steps of execution for instruction \textit{id} as defined by the emulator. These steps normally involve instruction fetch and decode. Instruction information fields of \textit{id} are used and updated as documented by the emulator used.

Though it is always present, this function may not work with some emulators. Such emulators should include a statement to this effect in their documentation.

Open Issue
Return values and exceptions

\texttt{varies LSE\_emu\_dynid\_get}(LSE\_dynid\_t \textit{id}, \textit{fieldname});
\textbf{Capability:} Always present

Get the instruction information field \textit{fieldname} from \textit{id}'s \texttt{instr\_info} attribute.

\texttt{boolean LSE\_emu\_dynid\_is}(LSE\_dynid\_t \textit{id}, \textit{classname});
\textbf{Capability:} Always present

Returns \texttt{TRUE} if the instruction \textit{id} is of class \textit{classname}, \texttt{FALSE} otherwise.
void LSE_emu_dynid_set(LSE_dynid_t id, fieldname, value);

**Capability:** Always present

Set the instruction information field `fieldname` in `id`'s `instr_info` attribute to `value`.

LSE_emu_contextno_t LSE_emu_get_context_mapping(LSE_emu_contextno_t cno);

**Capability:** Always present

Also used by the command-line processor.

If `cno` is a hardware context number, returns the software context number it is mapped to. If `cno` is a software context number, returns the hardware context number it is mapped to. In either case, if the context is unmapped, returns 0.

LSE_emu_contextno_t LSE_emu_get_contextno(boolean wanthw);

**Capability:** Always present

If `wanthw` is TRUE, returns the lowest hardware context number which has not yet been created. If `wanthw` is FALSE, returns the lowest (in absolute value) software context number which has not yet been created.

LSE_emu_addr_t LSE_emu_get_start_addr(LSE_emu_contextno_t cno);

**Capability:** Always present

Returns the address of the first instruction in the context specified by the context number `cno`. May abort simulation on error.

unsigned int LSE_emu_get_statespace_bitsize(LSE_emu_contextno_t cno, LSE_emu_spaceid_t sid);

**Capability:** Always present

Returns the number of bits required for an address in the state space `sid` in context `cno`.

const char * LSE_emu_get_statespace_name(LSE_emu_spaceid_t sid);

**Capability:** Always present

Returns the name of the state space `sid`.

unsigned int LSE_emu_get_statespace_size(LSE_emu_contextno_t cno, LSE_emu_spaceid_t sid);

**Capability:** Always present

Returns the size of the state space `sid` in context number `cno`. Returns -1 if the state space size is too large to fit in a 32-bit unsigned integer.

LSE_statespace_type_t LSE_emu_get_statespace_type(LSE_emu_spaceid_t sid);
Chapter 3. Emulation Interface

**Capability:** Always present
Returns the type of the state space \( sid \).

```c
int LSE_emu_get_statespace_width(LSE_emu_spaceid_t sid);
```

**Capability:** Always present
Returns the width of elements in the state space \( sid \).

```c
boolean LSE_emu_has_capability(capability);
```

**Capability:** Always present
Returns TRUE if the emulator has capability \( capability \), FALSE otherwise. This function is evaluated at code generation time.

```c
boolean LSE_emu_has_instr_class(instrclass);
```

**Capability:** Always present
Returns whether the emulator has instruction class \( instrclass \). This function is evaluated at code generation time. It is intended to be used in default decode functions to alert the user that the emulator does not have a required class.

```c
void LSE_emu_init_instr(LSE_dynid_t id, LSE_emu_contextno_t cno, LSE_emu_addr_t addr);
```

**Capability:** Always present
Prepare instruction \( id \) to be used with the emulator. The global context number in which the instruction executes is \( cno \) and its address is \( addr \). Sets the corresponding fields of the instruction information structure.

```c
int LSE_emu_load_context(LSE_emu_contextno_t cno, int argc, char *argv[], char **envp);
```

**Capability:** Always present
Load a program specified by \( argv[0] \) into context \( cno \) in an emulator-defined fashion, setting up program arguments and environment to be \( argc, argv \), and \( envp \). Returns non-zero if an error is encountered.

```c
int LSE_emu_map_context(LSE_emu_contextno_t hwcno, LSE_emu_contextno_t swcno);
```

**Capability:** Always present
Map software context \( swcno \) to hardware context \( hwcno \). Returns non-zero if an error is encountered.

```c
void LSE_emu_set_context_automap(LSE_emu_contextno_t cno, boolean automap);
```
capability: Always present

set the automap flag of context cno to automap. If automap is TRUE and the previous value of
the automap flag was FALSE, attempts to map the context.

void LSE_emu_set_start_addr(LSE_emu_contextno_t cno, LSE_emu_addr_t addr);
capability: Always present

sets the starting address of the context specified by the global context number cno to be addr.

boolean LSE_emu_spaceref_equ(LSE_emu_spaceid_t id1, LSE_emu_spaceaddr_t addr1, LSE_emu_spaceid_t id2, LSE_emu_spaceaddr_t addr2);
capability: Always present

returns TRUE if id1 equals id2 and addr1 equals addr2, FALSE otherwise.

boolean LSE_emu_statespace_has_capability(LSE_emu_spaceid_t sid, capability);
capability: Always present

returns TRUE if the state space sid has the capability capability, FALSE otherwise.

varied LSE_emu_instr_info_get(LSE_emu_instr_info_t *ii, fieldname);
capability: Always present

get the field fieldname from ii.

boolean LSE_emu_instr_info_is(LSE_emu_instr_info_t *ii, classname);
capability: Always present

returns TRUE if the instruction information ii indicates that the instruction is of class
classname, FALSE otherwise.

void LSE_emu_instr_info_set(LSE_emu_instr_info_t *ii, fieldname, value);
capability: Always present

set the field fieldname in ii to value.

functions available with disassemble

void LSE_emu_disassemble(LSE_dynid_t id, FILE *outfile);
Capability: disassemble

Disassemble the instruction \textit{id} and print the results on \textit{outfile}. The instruction must have already been fetched.

```c
void LSE_emu_disassemble_addr(LSE_emu_contextno_t cno, LSE_emu_addr_t addr, FILE *outfile);
```

Capability: disassemble

Fetch and disassemble the instruction \textit{id} located at address \textit{addr} in context \textit{cno} and print the results on \textit{outfile}.

Functions available with \textit{operandval}

```c
void LSE_emu_fetch_operand(LSE_dynid_t id, LSE_emu_operand_name_t oname);
```

Capability: \textit{operandval}

Fetch a source operand (read state) for instruction \textit{id}. The operand’s name is given by \textit{oname}. Updates the instruction information for \textit{id} by placing the operand’s value into the \textit{operand_val_src[oname].data} field and setting \textit{operand_val_src[oname].valid} to \textit{TRUE}. Some emulators may require that certain operands be fetched before others can be. Violating these requirements results in undefined behavior which may include simulator crashes.

Open Issue

- Return values and exceptions

```c
void LSE_emu_fetch_remaining(LSE_dynid_t id);
```

Capability: \textit{operandval}

Fetch source operands (read state) for instruction \textit{id} whose valid flags in the \textit{operand_val_src} array of the instruction information are not set. Updates the \textit{operand_val_src} array.

Open Issue

- Return values and exceptions
Chapter 3. Emulation Interface

void LSE_emu_writeback_operand(LSE_dynid_t id, LSE_emu_operand_name_t oname);

Capability: *operandval*

Writeback (update current state for) a destination operand for instruction *id*. The operand’s name is given by *oname*. Values are taken from the instruction information for *id*, using field *operand_val_dest*[oname].data. If the destination value is not valid, undefined results (which may include simulator crashes) occur.

Open Issue

Return values and exceptions

void LSE_emu_writeback_remaining(LSE_dynid_t id);

Capability: *operandval*

Writeback (update current state for) destination operands in instruction *id* whose flags in the *operand_written_dest* array of the instruction information are not set. Updates the *operand_written_dest* array.

Open Issue

Return values and exceptions

**Functions available with *operandinfo***

boolean LSE_emu_spaceref_is_constant(LSE_emu_contextno_t cno, LSE_emu_spaceid_t id, LSE_emu_spaceaddr_t addr);

Capability: *operandinfo*

Returns TRUE if the address *addr* in state space *id* is always a constant.

int LSE_emu_spaceref_to_int(LSE_emu_contextno_t cno, LSE_emu_spaceid_t id, LSE_emu_spaceaddr_t addr);

Capability: *operandinfo*

Translates the space address *addr* within state space *id* into an integer which is not larger than the state space size. State spaces which are larger than $2^{31} - 1$ locations return an undefined result. This function is intended to simplify working with register state spaces which have string addresses.
Functions available with *speculation*

```c
void LSE_emu_rollback_dynid(LSE_dynid_t id);
```
**Capability: speculation**

Unmodify state modified by the instruction `id`.

```c
void LSE_emu_rollback_resolution(LSE_resolution_t res);
```
**Capability: speculation**

Unmodify state modified by the dependent instructions in `res`. 
Chapter 4. Domain construction APIs

This chapter describes macros available to writers of domain classes.

There are several macros which are used when writing a domain class `m4` file. These macros are available in no other code and are expanded at code generation time.

Macros without arguments

There are two macros without arguments:

- `LSE_domain_class_name` - the name of the current domain class as a literal. This macro is of limited use because the writer of a domain class already knows the domain class name.
- `LSE_domain_inst_name` - the name of the current domain instance as a literal. It is empty in per-class sections of the `m4` macro file.

Python variables

There are two Python variables available:

- `LSE_domain_class` - the domain class object.
- `LSE_domain_inst` - the domain instance object. It equals `None` in per-class sections of the `m4` macro file.

Other macros

```
literal CLASSID(identifier);
Availability: domain macrofiles
Evaluated at: code generation
Expands to a unique name for domain class identifier identifier.
```

```
literal INSTID(identifier);
Availability: domain macrofiles
Evaluated at: code generation
Expands to a unique name for domain instance identifier identifier.
```
Chapter 4. Domain construction APIs

LSE_domain_class_define(\texttt{id}entifier, \texttt{d}efinition);
\textbf{Availability:} domain macrofiles
\textbf{Evaluated at:} code generation

Defines a per-domain-class \texttt{m4} macro with the unique domain-class name \texttt{id}entifier and definition \texttt{definitions}.

\texttt{LSE_domain_hook}(\texttt{hook}\_name);
\textbf{Availability:} domain macrofiles
\textbf{Evaluated at:} code generation

Expands to a unique name for domain hook \texttt{hook}\_name.

LSE_domain_inst_define(\texttt{id}entifier, \texttt{d}efinition);
\textbf{Availability:} domain macrofiles
\textbf{Evaluated at:} code generation

Defines a per-domain-instance \texttt{m4} macro with the unique domain-instance name \texttt{id}entifier and definition \texttt{definitions}.

\texttt{LSE_domain_invoke}(\texttt{b}ackend\_name, \ldots);
\textbf{Availability:} domain macrofiles
\textbf{Evaluated at:} runtime

Expands to a function call to the backend function \texttt{backend}name with the remaining arguments.
Chapter 5. Error Messages

This chapter lists the error messages which may be generated by the Liberty Simulation Environment.

Build-time error messages

Because the simulator is mostly generated code, it can be difficult to track down where an error originated. The following hints may help:

- Errors in SIM_prefix.m4 generally originate in code functions applied to the module instance. The line number reported is usually useless. Report such errors to the development team; we hope to catch errors before they reach this point.

**Error on line # in lssfile: Element domainname on DomainInstanceMap has not been initialized.**

A module has defined a domain search path but no instance has been assigned to it for one of the domains.

**filename.c: line#: parse error ...**

Compiler parse errors can be caused by a variety of problems. Common ones are:

- C syntax errors in a module.
- Whitespace between an API call and the opening parenthesis of its argument list. If inspection of the generated file looks like an API call has mysteriously vanished without being replaced by anything else, but left its parameters in the file, this is probably the problem.

**TO DO**

Finish this

Link-time error messages
Run-time error messages

LSE: Simulator has no more timesteps at time $t_{ime}$
CLP: Error -1 returned from LSE_sim_engine

The simulator has run out of timesteps. This can occur when a module which is not purely reactive does not properly use LSE_sim_keep_alive to indicate that its output signals could have different values in the future.
# Appendix A. API names

This appendix lists all of the identifiers defined as part of the core API or emulation interface.

## Table A-1. Core API identifier list

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Kind</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>#LSE else</td>
<td>directive</td>
<td>the Section called <em>Code generation directives</em> in Chapter 2</td>
</tr>
<tr>
<td>#LSE endif</td>
<td>directive</td>
<td>the Section called <em>Code generation directives</em> in Chapter 2</td>
</tr>
<tr>
<td>#LSE if</td>
<td>directive</td>
<td>the Section called <em>Code generation directives</em> in Chapter 2</td>
</tr>
<tr>
<td>boolean</td>
<td>type</td>
<td>the Section called <em>Simple types</em> in Chapter 1</td>
</tr>
<tr>
<td>FALSE</td>
<td>constant</td>
<td>the Section called <em>Simple types</em> in Chapter 1</td>
</tr>
<tr>
<td>FUNC</td>
<td>function</td>
<td>the Section called <em>Instance accessors</em> in Chapter 2</td>
</tr>
<tr>
<td>FUNCPtr</td>
<td>function</td>
<td>the Section called <em>Instance accessors</em> in Chapter 2</td>
</tr>
<tr>
<td>GLOB</td>
<td>function</td>
<td>the Section called <em>Instance accessors</em> in Chapter 2</td>
</tr>
<tr>
<td>GLOBDEF</td>
<td>function</td>
<td>the Section called <em>Instance accessors</em> in Chapter 2</td>
</tr>
<tr>
<td>HANDLER</td>
<td>function</td>
<td>the Section called <em>Instance accessors</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_controlpoint_call_default</td>
<td>function</td>
<td>the Section called <em>Codepoint functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_controlpoint_call_empty</td>
<td>function</td>
<td>the Section called <em>Codepoint functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_data_cancel</td>
<td>function</td>
<td>the Section called <em>Data manipulation functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_data_copy</td>
<td>function</td>
<td>the Section called <em>Data manipulation functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_data_move</td>
<td>function</td>
<td>the Section called <em>Data manipulation functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_data_register</td>
<td>function</td>
<td>the Section called <em>Data manipulation functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_dynid_cancel</td>
<td>function</td>
<td>the Section called <em>Dynamic messages</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_dynid_create</td>
<td>function</td>
<td>the Section called <em>Dynamic messages</em> in Chapter 1</td>
</tr>
<tr>
<td>Identifier</td>
<td>Kind</td>
<td>Defined in</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LSE_dynid_dump</td>
<td>function</td>
<td>the Section called <em>Dynamic messages</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_dynid_element_ptr</td>
<td>function</td>
<td>the Section called <em>Dynamic messages</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_dynid_eq</td>
<td>function</td>
<td>the Section called <em>Dynamic messages</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_dynid_get</td>
<td>function</td>
<td>the Section called <em>Dynamic messages</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_dynid_num_t</td>
<td>type</td>
<td>the Section called <em>Dynamic messages</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_dynid_recreate</td>
<td>function</td>
<td>the Section called <em>Dynamic messages</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_dynid_register</td>
<td>function</td>
<td>the Section called <em>Dynamic messages</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_dynid_set</td>
<td>function</td>
<td>the Section called <em>Dynamic messages</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_dynid_t</td>
<td>type</td>
<td>the Section called <em>Dynamic messages</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_enum_value</td>
<td>function</td>
<td>the Section called <em>User-defined types</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_enum_value</td>
<td>function</td>
<td>the Section called <em>User-defined types</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_eval</td>
<td>function</td>
<td>the Section called <em>Miscellaneous functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_event_filled</td>
<td>function</td>
<td>the Section called <em>Event functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_event_record</td>
<td>function</td>
<td>the Section called <em>Event functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_instance_name</td>
<td>constant</td>
<td>the Section called <em>Global constants</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_LOOP_END</td>
<td>macro</td>
<td>the Section called <em>Looping constructs</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_LOOP_OVER_PORT</td>
<td>macro</td>
<td>the Section called <em>Looping constructs</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_method_call</td>
<td>function</td>
<td>the Section called <em>Method and query functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_method_used</td>
<td>function</td>
<td>the Section called <em>Method and query functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_nameequal</td>
<td>function</td>
<td>the Section called <em>Miscellaneous functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_parm_constant</td>
<td>function</td>
<td>the Section called <em>Instance accessors</em> in Chapter 2</td>
</tr>
</tbody>
</table>

Appendix A. API names
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Kind</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSE_port_connected</td>
<td>function</td>
<td>the Section called Port functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_port_get</td>
<td>function</td>
<td>the Section called Port functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_port_get_flag</td>
<td>function</td>
<td>the Section called Port functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_port_num_connected</td>
<td>function</td>
<td>the Section called Port functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_port_query</td>
<td>function</td>
<td>the Section called Port functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_port_set</td>
<td>function</td>
<td>the Section called Port functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_port_set_flag</td>
<td>function</td>
<td>the Section called Port functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_port_type</td>
<td>function</td>
<td>the Section called Port functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_port_width</td>
<td>function</td>
<td>the Section called Port functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_query_call</td>
<td>function</td>
<td>the Section called Method and query functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_query_used</td>
<td>function</td>
<td>the Section called Method and query functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_query_results_changed</td>
<td>function</td>
<td>the Section called Scheduling functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_refcount_alloc</td>
<td>function</td>
<td>the Section called Reference-counted types in Chapter 1</td>
</tr>
<tr>
<td>LSE_refcount_cancel</td>
<td>function</td>
<td>the Section called Reference-counted types in Chapter 1</td>
</tr>
<tr>
<td>LSE_refcount_get</td>
<td>function</td>
<td>the Section called Reference-counted types in Chapter 1</td>
</tr>
<tr>
<td>LSE_refcount_init</td>
<td>function</td>
<td>the Section called Reference-counted types in Chapter 1</td>
</tr>
<tr>
<td>LSE_refcount_register</td>
<td>function</td>
<td>the Section called Reference-counted types in Chapter 1</td>
</tr>
<tr>
<td>LSE_refcount_t</td>
<td>type</td>
<td>the Section called Reference-counted types in Chapter 1</td>
</tr>
<tr>
<td>LSE_report_err</td>
<td>function</td>
<td>the Section called Reporting functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_report_err_at_codegen</td>
<td>function</td>
<td>the Section called Reporting functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_report_err_at_cpp</td>
<td>function</td>
<td>the Section called Reporting functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_report_warn</td>
<td>function</td>
<td>the Section called Reporting functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_report_warn_at_codegen</td>
<td>function</td>
<td>the Section called Reporting functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_report_warn_at_cpp</td>
<td>function</td>
<td>the Section called Reporting functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_resolution_add_dynid</td>
<td>function</td>
<td>the Section called Speculation resolutions in Chapter 1</td>
</tr>
<tr>
<td>Identifier</td>
<td>Kind</td>
<td>Defined in</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LSE_resolution_cancel</td>
<td>function</td>
<td>the Section called <em>Speculation resolutions</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_resolution_class_t</td>
<td>type</td>
<td>the Section called <em>Speculation resolutions</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_resolution_confirm</td>
<td>constant</td>
<td>the Section called <em>Speculation resolutions</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_resolution_create</td>
<td>function</td>
<td>the Section called <em>Speculation resolutions</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_resolution_dump</td>
<td>function</td>
<td>the Section called <em>Speculation resolutions</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_resolution_element_ptr</td>
<td>function</td>
<td>the Section called <em>Speculation resolutions</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_resolution_get</td>
<td>function</td>
<td>the Section called <em>Speculation resolutions</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_resolution_redecide</td>
<td>constant</td>
<td>the Section called <em>Speculation resolutions</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_resolution_register</td>
<td>function</td>
<td>the Section called <em>Speculation resolutions</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_resolution_set</td>
<td>function</td>
<td>the Section called <em>Speculation resolutions</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_resolution_t</td>
<td>type</td>
<td>the Section called <em>Speculation resolutions</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_ack</td>
<td>constant</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_ack_known</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_ack_present</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_ack2data</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_ack2enable</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_all_no</td>
<td>constant</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_all_yes</td>
<td>constant</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_and</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_data_known</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_data_present</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_data2ack</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_data2enable</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_disabled</td>
<td>constant</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_enable_known</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_enable_present</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_enable2ack</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_enable2data</td>
<td>function</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_enabled</td>
<td>constant</td>
<td>the Section called <em>Signal values</em> in Chapter 1</td>
</tr>
<tr>
<td>Identifier</td>
<td>Kind</td>
<td>Defined in</td>
</tr>
<tr>
<td>-------------------------</td>
<td>---------</td>
<td>---------------------------------------------------</td>
</tr>
<tr>
<td>LSE_signal_extract_ack</td>
<td>function</td>
<td>the Section called Signal values in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_extract_data</td>
<td>function</td>
<td>the Section called Signal values in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_extract_enable</td>
<td>function</td>
<td>the Section called Signal values in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_nack</td>
<td>constant</td>
<td>the Section called Signal values in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_not</td>
<td>function</td>
<td>the Section called Signal values in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_nothing</td>
<td>constant</td>
<td>the Section called Signal values in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_or</td>
<td>function</td>
<td>the Section called Signal values in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_print</td>
<td>function</td>
<td>the Section called Signal values in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_something</td>
<td>constant</td>
<td>the Section called Signal values in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_t</td>
<td>type</td>
<td>the Section called Signal values in Chapter 1</td>
</tr>
<tr>
<td>LSE_signal_unknown</td>
<td>constant</td>
<td>the Section called Signal values in Chapter 1</td>
</tr>
<tr>
<td>LSE_sim_exit_status</td>
<td>variable</td>
<td>the Section called Global variables in Chapter 1</td>
</tr>
<tr>
<td>LSE_sim_force_phase</td>
<td>function</td>
<td>the Section called Scheduling functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_sim_keep_alive</td>
<td>function</td>
<td>the Section called Scheduling functions in Chapter 2</td>
</tr>
<tr>
<td>LSE_sim_terminate_now</td>
<td>variable</td>
<td>the Section called Global variables in Chapter 1</td>
</tr>
<tr>
<td>LSE_sim_terminate_count</td>
<td>variable</td>
<td>the Section called Global variables in Chapter 1</td>
</tr>
<tr>
<td>LSE_stderr</td>
<td>variable</td>
<td>the Section called Global variables in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_add</td>
<td>function</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_construct</td>
<td>function</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_eq</td>
<td>function</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_ge</td>
<td>function</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_get_cycle</td>
<td>function</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_get_phase</td>
<td>function</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_gt</td>
<td>function</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_le</td>
<td>function</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_lte</td>
<td>function</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_ne</td>
<td>function</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_now</td>
<td>variable</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_numticks_t</td>
<td>type</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_one_cycle</td>
<td>constant</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_print_args</td>
<td>function</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_sub</td>
<td>function</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_t</td>
<td>type</td>
<td>the Section called Time types in Chapter 1</td>
</tr>
</tbody>
</table>
### Appendix A. API names

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Kind</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSE_time_ticks</td>
<td>function</td>
<td>the Section called <em>Time types</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_zero</td>
<td>constant</td>
<td>the Section called <em>Time types</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_time_zero_ticks</td>
<td>constant</td>
<td>the Section called <em>Time types</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_type_none</td>
<td>type</td>
<td>the Section called <em>Simple types</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_type_none_NULL</td>
<td>constant</td>
<td>the Section called <em>Simple types</em> in Chapter 1</td>
</tr>
<tr>
<td>LSE_userpoint_call_default</td>
<td>function</td>
<td>the Section called <em>Codepoint functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_userpoint_defaulted</td>
<td>function</td>
<td>the Section called <em>Codepoint functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_userpoint_empty</td>
<td>function</td>
<td>the Section called <em>Codepoint functions</em> in Chapter 2</td>
</tr>
<tr>
<td>LSE_userpoint_invoke</td>
<td>function</td>
<td>the Section called <em>Codepoint functions</em> in Chapter 2</td>
</tr>
<tr>
<td>PARM</td>
<td>function</td>
<td>the Section called <em>Instance accessors</em> in Chapter 2</td>
</tr>
<tr>
<td>TRUE</td>
<td>constant</td>
<td>the Section called <em>Simple types</em> in Chapter 1</td>
</tr>
</tbody>
</table>

### Table A-2. Emulator API identifier list

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Kind</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSE_emu_addr_t</td>
<td>type</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_call_extra_func</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_contextno_t</td>
<td>type</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_create_context</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_ctoken_t</td>
<td>type</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_disassemble</td>
<td>function</td>
<td>the Section called <em>Functions available with disassemble</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_disassemble_addr</td>
<td>function</td>
<td>the Section called <em>Functions available with disassemble</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_do_instrstep</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_doback</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_docommit</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_dofront</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
</tbody>
</table>
## Appendix A. API names

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Kind</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSE_emu_dynid_get</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_dynid_is</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_dynid_set</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_fetch_operand</td>
<td>function</td>
<td>the Section called <em>Functions available with operandval</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_fetch_remaining</td>
<td>function</td>
<td>the Section called <em>Functions available with operandval</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_get_context_mapping</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_get_contextno</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_get_start_addr</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_get_statespace_bitsize</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_get_statespace_name</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_get_statespace_size</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_get_statespace_type</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_has_capability</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_has_instr_class</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_init_instr</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_instr_info_get</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_instr_info_is</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_instr_info_set</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_instr_info_t</td>
<td>type</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_instrstep_name_step</td>
<td>constant</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_instrstep_name_t</td>
<td>type</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>Identifier</td>
<td>Kind</td>
<td>Defined in</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>LSE_emu_load_context</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_map_context</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_max_branch_targets</td>
<td>constant</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_max_instrstep</td>
<td>constant</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_max_operand_dest</td>
<td>constant</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_max_operand_int</td>
<td>constant</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_max_operand_src</td>
<td>constant</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_memaccess_atomic</td>
<td>constant</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_memaccess_noaccess</td>
<td>constant</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_memaccess_read</td>
<td>constant</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_memaccess_write</td>
<td>constant</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_num_statespaces</td>
<td>constant</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_operand_info_t</td>
<td>type</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_operand_name_oper</td>
<td>constant</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_operand_name_t</td>
<td>type</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_operand_val_t</td>
<td>type</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_rollback_dynid</td>
<td>function</td>
<td>the Section called <em>Functions available with speculation</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_rollback_resolution</td>
<td>function</td>
<td>the Section called <em>Functions available with speculation</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_set_context_automap</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_set_start_addr</td>
<td>function</td>
<td>the Section called <em>Core emulation functions</em> in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_spaceaddr_t</td>
<td>type</td>
<td>the Section called <em>Datatypes, structure fields, and constants</em> in Chapter 3</td>
</tr>
</tbody>
</table>
## Appendix A. API names

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Kind</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSE_emu_spaceid_spacename</td>
<td>constant</td>
<td>the Section called Datatypes, structure fields, and constants in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_spaceid_t</td>
<td>type</td>
<td>the Section called Datatypes, structure fields, and constants in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_spaceref_equ</td>
<td>function</td>
<td>the Section called Core emulation functions in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_spaceref_is_constant</td>
<td>function</td>
<td>the Section called Functions available with operandinfo in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_spaceref_to_int</td>
<td>function</td>
<td>the Section called Functions available with operandinfo in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_spacetype_creg</td>
<td>constant</td>
<td>the Section called Datatypes, structure fields, and constants in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_spacetype_mapping</td>
<td>constant</td>
<td>the Section called Datatypes, structure fields, and constants in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_spacetype_mem</td>
<td>constant</td>
<td>the Section called Datatypes, structure fields, and constants in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_spacetype_reg</td>
<td>constant</td>
<td>the Section called Datatypes, structure fields, and constants in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_spacetype_t</td>
<td>type</td>
<td>the Section called Datatypes, structure fields, and constants in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_statespace_has_capability</td>
<td>function</td>
<td>the Section called Core emulation functions in Chapter 3</td>
</tr>
<tr>
<td>LSE_emu_writeback_operand</td>
<td>function</td>
<td>the Section called Functions available with operandval in Chapter 3</td>
</tr>
</tbody>
</table>

### Table A-3. Domain construction API identifier list

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Kind</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASSID</td>
<td>macro</td>
<td>the Section called Other macros in Chapter 4</td>
</tr>
<tr>
<td>INSTID</td>
<td>macro</td>
<td>the Section called Other macros in Chapter 4</td>
</tr>
<tr>
<td>LSE_domain_class</td>
<td>Python variable</td>
<td>the Section called Python variables in Chapter 4</td>
</tr>
<tr>
<td>LSE_domain_class_define</td>
<td>macro</td>
<td>the Section called Other macros in Chapter 4</td>
</tr>
<tr>
<td>LSE_domain_class_name</td>
<td>macro</td>
<td>the Section called Macros without arguments in Chapter 4</td>
</tr>
<tr>
<td>LSE_domain_hook</td>
<td>macro</td>
<td>the Section called Other macros in Chapter 4</td>
</tr>
<tr>
<td>LSE_domain_inst</td>
<td>Python variable</td>
<td>the Section called Python variables in Chapter 4</td>
</tr>
<tr>
<td>LSE_domain_inst_define</td>
<td>macro</td>
<td>the Section called Other macros in Chapter 4</td>
</tr>
<tr>
<td>LSE_domain_inst_name</td>
<td>macro</td>
<td>the Section called Macros without arguments in Chapter 4</td>
</tr>
<tr>
<td>LSE_domain_invoke</td>
<td>function</td>
<td>the Section called Other macros in Chapter 4</td>
</tr>
</tbody>
</table>