The System Services Interface (SSI) to LAM/MPI
SSI Version 1.0.0

Jeffrey M. Squyres
Brian Barrett
Andrew Lumsdaine
http://www.lam-mpi.org/

Open Systems Laboratory
Pervasive Technologies Labs
Indiana University
CS TR575

August 4, 2003

Contents

1 Overview **3**
- 1.1 Terminology . . . 3
- 1.2 LAM's Role in SSI . . . 4
  - 1.2.1 Statically vs. Dynamically Linked Modules . . . 4
- 1.3 LAM SSI and MPI SSI . . . 5
- 1.4 Module Basic Requirements . . . 5
- 1.5 Notable Side Effect: Long Public Symbol Names . . . 6
- 1.6 Organization . . . 6

2 Directory Layout **7**

3 Configuring the Module **7**
- 3.1 Generating `configure` Scripts . . . 7
  - 3.1.1 Using `autogen.sh` on a Single Directory . . . 8
  - 3.1.2 Changing `autogen.sh`'s Default Behavior . . . 8
  - 3.1.3 Utilizing Common LAM GNU Auto Tool Elements . . . 8
- 3.2 Running `configure` Scripts . . . 15
  - 3.2.1 Module-Specific Parameters . . . 15
  - 3.2.2 Returning Values From `configure` . . . 16
  - 3.2.3 When the `configure` Script *Must* Succeed . . . 17

4 Building the Module **17**

5 Installing the Module **17**

6 Selecting the Module at Run Time **19**
- 6.1 Terminology . . . 19
- 6.2 General Scheme . . . 20
  - 6.2.1 MPI Selection Algorithm . . . 20
- 6.3 Specific Selection Mechanisms . . . 21

7 Module Source Code **21**
- 7.1 Base SSI API . . . 22
  - 7.1.1 Module Open Function . . . 22
  - 7.1.2 Module Close Function . . . 23
  - 7.1.3 The Base SSI Datatype: `lam_ssi_t` . . . 23
  - 7.1.4 Example Usage: the rpi SSI Kind . . . 24
- 7.2 Help Messages . . . 26
- 7.3 Verbosity . . . 26

8 Passing Run-Time Parameters to the Module **27**

9 Unresolved Issues **27**

10 Acknowledgements **28**

References **28**

1 Overview

The LAM System Services Interface (SSI) is a two-tiered modular framework for collections of system interfaces. The goal of SSI is to enable multiple instances of interfaces (each from their own respective collection) to be available at run time. Run-time decisions can be made about which interface instance to use, and to enable passing of tunable parameters to target instances. Figure 1 shows this relationship graphically.

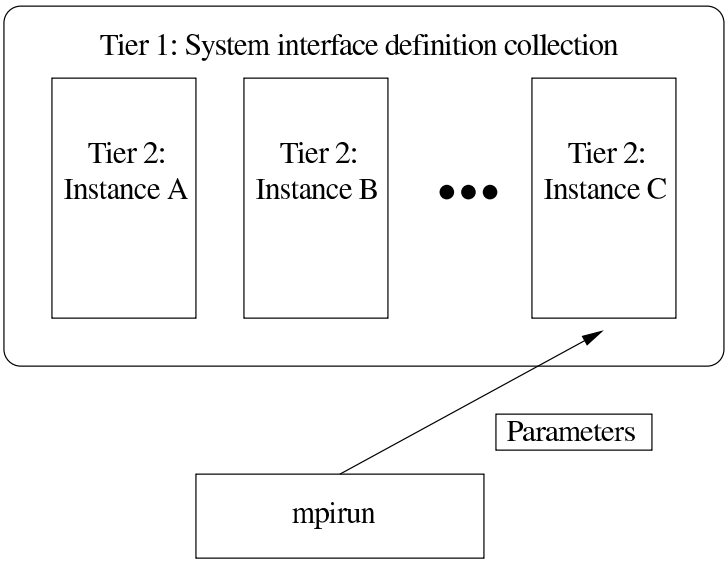

Figure 1: LAM may export one or more instances of any given system interface. At run time, the decision is made as to which instance will be selected. Optionally, parameters may also be passed to the interface instances at run time. In this example, `mpirun` has selected to use instance C, and has passed some parameters to it.

The overall intent for SSI is to allow a modular "plug-n-play" approach to the system services that LAM and MPI must use. Specifically, each tier 2 instance in Figure 1 is a self-contained set of source code. It has its own directory structure and can configure, build, and install itself. The LAM frameworks will automatically detect each instance and invoke the corresponding hooks during the outer-level configuration, building, and installation phases.

This type of architecture will make it easy to add "drop-in" modules to the LAM framework and have them be seamlessly integrated into the run-time environment of LAM/MPI. From a source code perspective, this architecture creates a logical abstraction barrier between each instance and the rest of LAM. Not only does such a modular approach make is easier to add new interface instances, it can also facilitate development of interface instances by third parties.

1.1 Terminology

Each collection of interfaces (i.e., "tier 1" in Figure 1) is a component type (frequently abbreviated as "type," or referred to as a "kind"). LAM strictly defines each component type: the API that it exports, the mechanism used to determine run-time instance selection, etc. Table 1 lists the currently available kinds of APIs. Each is documented in a separate programmer's guide [3, 4, 5, 6]. Future SSI component types are also planned; they will be documented as they are created.

Name	Location	Documentation	Description
boot	LAM	See [4]	The boot SSI is used to start processes on remote nodes without the use of LAM daemons. It is used by `lamboot`, `recon`, `wipe`, and `lamgrow`. The boot SSI allows LAM to use other run-time environments to start itself (such as the ubiquitous `rsh`/`ssh` startup methods).
coll	MPI	See [5]	The coll SSI is used to provide back-end algorithms for the MPI collective operations. This kind allows third-party authors to provide new algorithms for MPI collectives based on specific hardware, or new theoretical models without needing to understand the intricacies of the internals in LAM.
cr	LAM and MPI	See [3]	Checkpoint-restart support. This kind is used to tie LAM/MPI to back-end checkpointing support to enable pausing and restarting parallel MPI jobs.
rpi	MPI	See [6]	The MPI Request Progression Interface (RPI). The RPI is used for all point-to-point MPI message traffic in LAM. It is the lowest layer in the point-to-point MPI communication stack and is responsible for actually moving bytes between MPI processes.

Table 1: Table of available SSI kinds.

Each interface / component instance (i.e., each of the "tier 2" boxes in Figure 1) is called a "module". Each module must conform to the component type's specifications, to include its exported API, run-time selection mechanisms, etc. Modules – although part of the larger SSI kind and LAM frameworks – are intended to be standalone, self-contained entities in terms of configuration, building, and installing themselves. More discussion on this subject is included later in this document.

1.2 LAM's Role in SSI

If SSI can be considered a collection of component instances, then LAM is the component architecture that provides the meta-framework that ties all the collections of instances together. Specifically, LAM provides the "glue" meta-framework for configuring, compiling, installing, and loading component modules.

The discussion in this document assumes a LAM development tree – not a LAM distribution package (e.g., typically not a `.tar.gz` or `.rpm` file). Development trees can be obtained via anonymous CVS checkout or from a nightly CVS snapshot. See `http://www.lam-mpi.org/cvs/` for details on both how to obtain a development tree as well as how to compile it.

1.2.1 Statically vs. Dynamically Linked Modules

The current LAM component framework only supports static linking. As such, all modules that are configured and compiled will become part of larger libraries, such as `liblam` and `libmpi`. Although multiple modules can be available at run time, the set of all possible modules is determined at compile time – it is not possible to add or remove modules at run-time.

This is mainly due to limitations in the GNU Libtool package. When the LAM/MPI 7.0 code base was "closed" in preparation for release, Libtool did not support dynamic modules on enough platforms (e.g.,

Mac OS X and AIX 5.1 were conspicuously absent) to justify dynamic modules. After LAM/MPI's formal release cycle was started, a new version of Libtool was released that included support for new platforms (including Mac OS X and AIX 5.1). Hence, although it was too late for 7.0, future versions of LAM/MPI will have an expanded SSI component framework that can utilize dynamically loaded modules as well. It is hoped that existing SSI modules will require little or no changes to be able to utilize these new features.

1.3 LAM SSI and MPI SSI

The LAM software package has two major components: the LAM run-time environment and the MPI communications layer. Generally speaking, LAM provides the services and run-time backbone for the MPI communication layer. The MPI functionality therefore has a dependency on the LAM functionality; MPI must always be used with LAM, but LAM can be used independently of MPI. These layers are located in two different libraries; Figure 2 shows a simple view of the layers in a typical user MPI program.

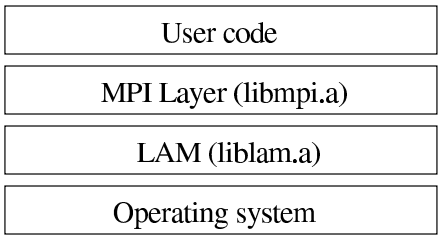

Figure 2: A simple view of the LAM and MPI layers.

This makes a natural division of SSI kinds – those designed to support LAM services and those designed to support MPI services. For example, the rpi SSI kind mentioned in Section 1.1 is used to support MPI services, and therefore will physically exist in the MPI library (`libmpi.a` or `libmpi.so`).

The same dependency exists between SSI kinds; MPI SSI kinds can use any services exported by both the MPI and LAM layers, but LAM SSI kinds can only use services exported by the LAM layer.

1.4 Module Basic Requirements

Each module is responsible for providing its own configuration, building, and installing mechanisms. LAM provides some helper tools and hooks into the main LAM configuration, building, and installing mechanisms.

In general, each module must supply the following:

- A top-level directory under `share/ssi/<kind>`, where `<kind>` is the name of the kind of SSI.[1]
- A directory name that confirms to C variable restrictions; this name is referred to as `<module>`, below.
- A `configure` script in the top-level module directory. This script will be run by the top-level LAM `configure` script. If the module's `configure` script succeeds, the following additional requirements must be met:
  - Several top-level `make` targets must be supplied. Each target is listed and briefly described below. For more specific information on these targets, see [2]:

[1] Note that the module names "`include`" and "`base`" are reserved and cannot be used.

    * `all`: build the module. At the end of this target, a GNU Libtool library named `lib<module>.la` must exist in the top-level module library.
    * `install`: install the module. This may be a no-op. For example, if the output GNU Libtool file `lib<module>.la` represents a static library (and it will, since dynamic modules are not yet supported; see Section 1.2.1), it will be folded into the LAM or MPI library directly, and does not need to be installed separately.
    * `uninstall`: uninstall the module. As with the `install` target, this may be a no-op for static libraries.
    * `clean`: remove all files generated by the `all` target.
    * `distclean`: do everything that the `clean` target does, and additionally remove all other generated files such that the directory tree is the same state that it was in when it was expanded from its distribution package.
    * `tags`: use the `etags` program to generate lists of tags.
  - An output GNU Libtool library (`lib<module>.la`) in the top-level module directory.
  - Export a public variable named `lam_ssi_<kind>_<module>_module` of type `lam_ssi_<kind>_t` pre-filled with relevant information.

1.5 Notable Side Effect: Long Public Symbol Names

It is necessary to prevent symbol name clashes not only between SSI modules themselves, the rest of LAM, and user programs. This is strictly necessary because of the current static linking model, but at least some platforms have a single, global symbol namespace even when used with dynamically-loaded modules. As such, all SSI modules must use prefixes for all public symbols. While guaranteeing to prevent name clashes with anything else, this has the side effect of making variable and function names rather lengthy.

Symbol prefixes are typically of the form "`lam_ssi_<kind>_<module>`" or "`lam-ssi-<kind>--<module>`" (depending on the context), but the case may be all upper case or all lower case (again, depending on context). This is referred to as "the prefix rule" throughout the rest of this document.

1.6 Organization

This document is intended as a programming guide to supply the necessary instructions and guidelines to write an SSI module. The specific SSI component type APIs are supplied elsewhere ([3, 4, 5, 6]); this document describes the meta-framework that is common to all SSI component types.

Making a module usable in the LAM run-time environment and/or MPI communication layer involves the following steps:

1. Creating the module in the LAM/MPI development tree
2. Supplying source code for the module
3. Configuring the module
4. Building the module
5. Installing the module and/or any necessary support files
6. Being selected to execute at run time
7. Receiving run-time parameters (optional)

Each of these steps are covered in the sections that follow.

2 Directory Layout

SSI modules are intended to be implemented as "drop-in" directories in the LAM directory tree. Each module should be self-contained tree.[2] The overall goal is that an SSI module can be inserted in the LAM directory structure, and the LAM SSI framework will automatically detect it, allowing the module to be configured, built, and used. No code changes are required outside the module's own directory structure to enable this to happen.

A module's implementation may have whatever directory structure it wishes – it only needs to have its top-level directory appear in a specific place in the LAM directory tree: `share/ssi/<kind>/<module>`. The `<kind>` name is strictly defined by LAM; the `<module>` name must be the same as the module's name. For example, the LAM provided TCP RPI module is rooted at `share/ssi/rpi/tcp`. Module directory names may be any string adhering to the following requirements:

- The directory name must adhere to filesystem requirements. Some filesystems are not case-sensitive, so all-lowercase names are recommended.
- The directory name will be used as the name of the module itself, and will be used as [part of] a variable name in C code. Hence, the directory name must adhere to C variable name standards (cannot include whitespace, cannot include punctuation, etc.). The module name will not be the first part of the variable name (it will be prefixed with `lam_ssi_<kind>_`), so the name may begin with a number.
- Some SSI kinds reserve particular directory names. Unless otherwise specifically noted, the names "`include`" and "`base`" are reserved and should not used as module names for any kind.

Note that the special file "`.lam_ignore`" may be placed in a module's top-level directory. This will cause LAM to ignore the entire directory tree – it will not be configured, compiled, installed, etc.

3 Configuring the Module

Since SSI modules should be designed to run on as many POSIX systems as possible, SSI modules are configured before they are built or installed. There are two steps of configuration:

1. Generating `configure` scripts and related files (e.g., `Makefile.in` files)
2. Running `configure` scripts

3.1 Generating `configure` Scripts

The main LAM tree and the LAM Team-supplied SSI modules all use the GNU Autoconf and Automake tools to generate `configure` scripts. This entails a non-trivial sequence of steps that must be followed to generate these files. As such, the process is automated.

Invoking the `autogen.sh` in the LAM top-level directory will run the proper sequence of commands to generate the top-level `configure` script and as well as all necessary `Makefile.in` files throughout the base LAM development tree. Note that `autogen.sh` is not *necessary* – it is simply a convenience script to run all the required GNU Auto tools in the proper order.

[2] The typical exception to this guidelines is when the module chooses to use some reference code from the "outer" LAM tree, such as top-level `m4` Autoconf configuration macros.

Since the process is non-trivial, SSI modules may also wish to utilize this automated functionality. `autogen.sh` will also look for any `configure.in` [3] files under `share/ssi/<kind>` (for each valid `<kind>` name). If `autogen.sh` finds `configure.in`, it will run all the proper GNU Autoconf and Automake commands to generate a `configure` script in that directory. If this functionality is sufficient, SSI module authors do not need to provide their own `autogen.sh` scripts.

3.1.1 Using `autogen.sh` on a Single Directory

While developing and debugging module `configure.in` scripts, it may be necessary to regenerate the `configure` script many times. Invoking the top-level `autogen.sh` script may actually take several minutes to run because it has to examine the entire LAM tree (including each SSI module in the directory tree). The "`-l`" parameter can be specified to `autogen.sh` indicating that it should only run locally (i.e., the present working directory), and should not attempt to look for `configure.in` scripts anywhere else.

For example, the following will generate a `configure` script and related `Makefile.in` files for just the tcp rpi SSI module:

```
shell$ cd share/ssi/rpi/tcp
shell$ ../../../../autogen.sh -l
```

3.1.2 Changing `autogen.sh`'s Default Behavior

The default behavior of `autogen.sh` can be changed in the following ways:

- If a file named `.lam_no_gnu` exists in a module's top-level directory, `autogen.sh` will not invoke the standard GNU tools to generate `configure` scripts. This may be desirable for third-party modules do not need to have their `configure` scripts and `Makefile.in` files regenerated.
- Having a file named `.lam_ignore` in a module's top-level directory has the same effect as the `.lam_no_gnu` file – the GNU Auto tools will not be invoked. However, unlike `.lam_no_gnu`, `.lam_ignore` will cause the entire directory (and therefore the entire module) to be ignored by the LAM infrastructure.
- If an executable script named `autogen.sh` exists in the module's top-level directory (with no corresponding `.lam_no_gnu` or `.lam_ignore` files), this script will be executed in lieu of running the GNU Auto tools.

3.1.3 Utilizing Common LAM GNU Auto Tool Elements

If a module uses the standard GNU Auto tools for configuration and buildings, the following notes may be helpful.

Using LAM Autoconf M4 macros. LAM provides a few Autoconf M4 macros that may be used by SSI modules for common functionality. Since an SSI module is embedded in the greater LAM directory structure, it may be convenient to establish `Makefile` macros that represent LAM's top source directory and top build directory, respectively. The following lines, placed in a module's `configure.in` script will enable the use of such macros:

[3] Recent versions of GNU Autoconf prefer the use of `configure.ac`. This document will use "`configure.in` " to mean "`configure.in` or `configure.ac` ."

```
top_lam_srcdir='$(top_srcdir)/../../../..'
top_lam_builddir='$(top_builddir)/../../../..'

AC_SUBST(top_lam_srcdir)
AC_SUBST(top_lam_builddir)
```

The Autoconf macros described below exists in one or more `.m4` files in LAM's top-level `config` directory. Each macro will list the `.m4` file that needs to be included (either by `acinclude.m4` or in the `configure.in` script itself), the input arguments, and the output results of the macro.

Using LAM's `Makefile.options` file. `Makefile.am` files may wish to include LAM's top-level `config/Makefile.options` file to include common Automake options. For example, if any of LAM's standardized Autoconf macros are used, the following line can be placed in a `Makefile.am`:

```
include $(top_lam_srcdir)/config/Makefile.options
```

Notice the use of the `$(top_lam_srcdir)` macro – the `config/Makefile.options` file is relative to LAM's top source directory, not the SSI module's top source directory.

Setup Autoconf Macro. The following macro should be invoked before all other LAM-supplied Autoconf macros in order to setup the environment and perform any other required initialization.

- Macro name: `LAM_CONFIGURE_SETUP`
- Summary: setup to use LAM's public M4 macros
- `sinclude` file: `lam_functions.m4`
- Input arguments:
  1. None
- Output:
  - None.
- Example usage:

```
sinclude(../../../../config/lam_functions.m4)
LAM_CONFIGURE_SETUP
```

Module version numbers. Each module has to have its own version number (described in Section 7.1.3). It may be desirable to maintain the version number in a separate file and dynamically assign this number at configuration / build time. LAM provides a script and corresponding Autoconf M4 macro to parse a specifically-formatted version number file and assign relevant version number components to environment variables. The following shows a typical version number file:

```
major=6
minor=6
release=0
alpha=0
beta=2
cvs=1
```

The LAM Autoconf macro `LAM_GET_VERSION` can be used to extract the various version number components from this file.

- Macro name: `LAM_GET_VERSION`
- Summary: Obtain the version number components from a text file.
- `sinclude` file: `lam_get_version.m4`
- Input arguments:
  1. Directory where the `get_lam_version` script resides.
  2. The filename of the version number file.
  3. The environment variable name prefix.
- Output:
  - `${var_prefix}_VERSION` environment variable containing all the relevant components put together in a single string. Components equaling 0 will be left out. If the alpha component is included, it will be prefixed with "`a`". If the beta component is included, it will be prefixed with "`b`".
  - `${var_prefix}_MAJOR_VERSION` environment variable containing the major version number component
  - `${var_prefix}_MINOR_VERSION` environment variable containing the minor version number component
  - `${var_prefix}_RELEASE_VERSION` environment variable containing the release version number component
  - `${var_prefix}_ALPHA_VERSION` environment variable containing the alpha version number component
  - `${var_prefix}_BETA_VERSION` environment variable containing the beta version number component
  - `${var_prefix}_CVS_VERSION` environment variable containing the CVS version number component
- Example usage:

```
# Get the macro.

sinclude(../../../../config/lam_get_version.m4)
```

```
# Invoke the LAM_GET_VERSION macro. The first argument is
# constant (LAM's top−level config directory). The second
# argument is the VERSION file in the TCP SSI module directory
# (note the use of $srcdir for VPATH builds). The third
# argument is the prefix of environment variable names to
# be assigned.

LAM_GET_VERSION(../../../../config, $srcdir/VERSION,
    LAM_SSI_RPI_TCP)

# After the macro, all 6 components of the version number are
# assigned to LAM_SSI_RPI_TCP_<component>_VERSION, and
# LAM_SSI_RPI_TCP_VERSION has an overall string name of all
# components put together (eliminating 0's). These can be used
# ac arguments to AC_DEFINE, for example.

AC_DEFINE_UNQUOTED(LAM_SSI_RPI_TCP_VERSION,
    "$LAM_SSI_RPI_TCP_VERSION",
    [Overall LAM SSI RPI TCP version number])
AC_DEFINE_UNQUOTED(LAM_SSI_RPI_TCP_MAJOR_VERSION,
    $LAM_SSI_RPI_TCP_MAJOR_VERSION,
    [Major LAM SSI RPI TCP version])
AC_DEFINE_UNQUOTED(LAM_SSI_RPI_TCP_MINOR_VERSION,
    $LAM_SSI_RPI_TCP_MINOR_VERSION,
    [Minor LAM SSI RPI TCP version])
# ...etc. (other version numbers)

# The version number can also be displayed to stdout.

echo Configuring SSI rpi tcp module v$LAM_SSI_RPI_TCP_VERSION
```

Basic Setup Macro. The following macro performs several basic setup functions. It is an optional macro; it handles some of the more mundane aspects of configuration.

- Macro name: `LAM_BASIC_SETUP`
- Summary: basic configuration setup functions
- `sinclude` file: `lam_functions.m4`
- Input arguments:
  1. None
- Output:
  - Set `CLEANFILES` to eliminate `emacs` and `vim` backup files (`*~`)
  - Call `AC_CANONICAL_HOST` to figure out the host type (in the `$host` environment variable)

  - Set the prefix based on `--prefix` or where `lamclean` is found
  - Setup to handle `--enable-dist` flags (see Section 3.2.3, page 17)

- Example usage:

```
sinclude(../../../../config/lam_functions.m4)
LAM_BASIC_SETUP
```

Setting up the C compiler. Setting up the C compiler requires several steps, and may involve user-specified `CFLAGS` and/or `LDFLAGS`. Additionally, it is usually preferable to compile with some form of optimization flags (unless otherwise specified by the user). The entire process has been automated into the macro `LAM_SETUP_CC` .

- Macro name: `LAM_SETUP_CC`
- Summary: setup the C compiler and determine associated `CFLAGS` and `LDFLAGS`.
- `sinclude` file: `lam_setup_cc.m4`
- Input arguments:

  1. None

- Output:

  - `CC` environment variable is set to the C compiler.
  - `CFLAGS` environment variable has been set or augmented.
  - `LDFLAGS` environment variable has been set or augmented.

- Example usage:

```
sinclude(../../../../config/lam_setup_cc.m4)
LAM_SETUP_CC
```

[Not] Setting up the C++ compiler. Unfortunately, since GNU Libtool did not yet support portably making C++ libraries when LAM 7.0 entered its release cycle (see Section 1.2.1), using C++ in SSI modules is not yet supported. As such, there is no `LAM_SETUP_CXX` macro.

Even though Libtool can portably build C++ libraries with the GNU C++ compiler (`g++`), this does not fit the requirement of "portable". Indeed, there are symbol name mangling / linking issues when mixing `g++` with a native C compiler, as well as serious performance implications on platforms with native C/C++ compilers. Therefore, "just use `g++`" is not a compelling enough reason to supply the `LAM_SETUP_CXX` macro.

Checking for a function. The LAM Team's coding standards demands that preprocessor macros are always defined. That is, a "true" value is not inferred by a macro being defined, and a corresponding "false" value is not inferred by a macro *not* being defined – the macro should be defined to be either 1 or 0.

The difference is slight, and mainly philosophical. The main rationale is that the following C preprocessor statement should only yield a "true" result if the macro is actually defined to be a nonzero integer – it should *not* yield a "true" result of the macro is defined to be zero:

```
#if RESULT_OF_SOME_TEST
  /* Should only be compiled if RESULT_OF_SOME_TEST is a nonzero
     integer; should *not* be compiled if RESULT_OF_SOME_TEST is 0. */
#endif
```

To that end, the the Autoconf macro `LAM_CHECK_FUNC` is available. It behaves exactly like `AC_CHECK_FUNC` except that it will always define the output macro to be either 0 or 1.

- Macro name: `LAM_CHECK_FUNC`
- Summary: check to see if a function exists
- `sinclude` file: `lam_check_func.m4`
- Input arguments:
  1. Function name to check for
  2. Name of preprocessor macro to define
  3. Arguments to add to linker command line (optional)
- Output:
  - Calls `AC_DEFINE` on the second argument, setting it to 0 if the function does not link properly (per `AC_CHECK_FUNC`), or 1 if it does.
- Example usage:

```
sinclude(../../../../config/lam_check_func.m4)
LAM_CHECK_FUNC(printf, LAM_SSI_RPI_FOO_HAVE_PRINTF)
```

Utility Autoconf Macros. Three utility macros provide the ability to write to the `config.log` file. Writing information to this file can be immensely useful when helping debug `configure` scripts as well as providing a single file for users to submit to developers when they run into problems with esoteric configurations.

- Macro name: `LAM_LOG_MSG`
- Summary: Output a string message into the `config.log` file.
- `sinclude` file: `lam_functions.m4`
- Input arguments:

1. String message.
2. If the second argument is not empty, the output message in `config.log` will include the line number that it was invoked from in the `configure` script.

- Output:
  - None.
- Example usage:

```
sinclude(../../../../config/lam_functions.m4)
LAM_LOG_MSG([This is going to config.log])
LAM_LOG_MSG([This is going to config.log with a line number], yes)
```

The `LAM_LOG_FILE` macro is used to send output to the `config.log` logfile.

- Macro name: `LAM_LOG_FILE`
- Summary: Output a file into the `config.log` file.
- `sinclude` file: `lam_functions.m4`
- Input arguments:
  1. Filename to output.
- Output:
  - None.
- Example usage:

```
sinclude(../../../../config/lam_functions.m4)
LAM_LOG_MSG([Attempt to compile the following test:], yes)
LAM_LOG_FILE([conftest.c])
```

The `LAM_LOG_COMMAND` macro is used to run a command and send the output to the `config.log` logfile.

- Macro name: `LAM_LOG_COMMAND`
- Summary: output a command and the results of the command to the `config.log` file.
- `sinclude` file: `lam_functions.m4`
- Input arguments:
  1. Command to execute.
  2. Action(s) to execute if the command succeeds (i.e., if the executed command has a return status of zero)

3. Action(s) to execute if the command fails (i.e., if the executed command has a return status not equal to zero)

- Output:
  - None.
- Example usage:

```
sinclude(../../../../config/lam_functions.m4)
LAM_LOG_MSG([Attempt to compile the following test:], yes)
LAM_LOG_FILE([conftest.c])
LAM_LOG_COMMAND([$CC $CFLAGS conftest.c -c], [SUCCESS=1], [SUCCESS=0])
```

3.2 Running `configure` Scripts

Each module must have an executable (for maximum portability, this should probably be a script) named `configure` (regardless of whether it is generated by LAM's top-level `autogen.sh` script or not). If a module does not have an executable named `configure`, it will not be configured or built by LAM. Additionally, if there is a file named `.lam_ignore` in the module's top-level directory, its `configure` script will be skipped, and LAM will ignore that module.

Each module's `configure` executable will automatically be invoked by LAM's top-level `configure` script. All the same command line flags and environment variables that were used to invoke the top-level `configure` script will be passed down to the module's `configure` script.

If the module's `configure` executable succeeds (i.e., returns a status code of 0), it will be included in the overall LAM build. If the module's `configure` executable fails (i.e., returns a status code of anything other than 0), it will be totally skipped by the LAM build process. This mechanism is useful for modules that are dependent upon specific kinds of systems and/or system configurations (e.g., specific hardware). If the module's specific requirements are not met, it can have its `configure` executable fail, and therefore not be compiled.

3.2.1 Module-Specific Parameters

It is possible to have module-specific `--with` and/or `--enable` command line switches that enable configure-time parameters. In order for these switches to be transparently passed through other `configure` scripts, the standard prefix naming convention should be used (although the leading "lam-mpi" may be left off, since, when running LAM's `configure` script – a script whose entire purpose is to configure LAM/MPI – explicitly mentioning "`lam-mpi`" would be redundant). For example:

```
AC_ARG_WITH(ssi-rpi-tcp-short,
            AC_HELP_STRING([--with-ssi-rpi-tcp-short=BYTES],
                           [Size of shortest TCP long message]),
            [TCPSHORT=$withval])
```

Similarly, it is possible to have module-specific environment variables that are used to pass values to the module's `configure` script. These variables should also adhere to the prefix rule. For example, the tcp rpi's `configure` script could use an environment variable as follows:

```
AC_ARG_VAR([SSI_RPI_TCP_SHORT],
           [Size of shortest TCP long message])
```

3.2.2 Returning Values From `configure`

There are two forms of output from a module's `configure` script: its exit status and arbitrary strings.

Exit Status The exit status of a module's `configure` script determines whether the module will be compiled into LAM or not. A zero exit status means that the `configure` script was successful and can be built on this platform. A non-zero exit status means that the `configure` script failed and/or the module cannot be built for some reason.

LAM will only add a module to the build list if its `configure` script returns a zero value.

Arbitrary Strings (compiler flags, linker flags, etc.) Some types of modules need to return additional flags to the lop-level LAM infrastructure to enable LAM to link properly. For example, an rpi module that uses an underlying communications library must add "`-lfoo`" to all MPI-compiled programs (and potentially "`-L/path/to/libfoo`"). `-l` and `-L` arguments need to be integrated not only into LAM's overall build process, but also into the "wrapper" MPI compilers (`mpicc`, `mpic++`, and `mpif77`).

A module can return these values by creating a file in its top-level directory containing these strings. The file is named `ssi_<kind>_<name>_config.sh`, where `<kind>` corresponds to the kind of the SSI (see Table 1 on page 4), and `<name>` refers to the module's name.

The `ssi_<kind>_<name>_config.sh` file can contain Bourne shell variable assignments for the variables listed in Table 2. The Bourne shell variables can be of the form `<scope>_EXTRA_<flags>`, where `<scope>` can be any of the following values:

- `LIBLAM`: Values required to compile/link to this SSI module. This scope should only need to be used for LAM SSI kinds, and will be used throughout the LAM source tree.
- `LIBMPI`: Values required to compile/link to this SSI module. This scope should only need to be used for MPI SSI kinds, and will be used throughout the LAM source tree.
- `WRAPPER`: Values required to compile/link to this SSI module. scope should only need to be used for MPI SSI kinds, and will be used by the wrapper compilers to link user MPI programs.

`<flags>` can be any of the following values:

- `CFLAGS`: Any flags that need to be passed to the C compiler in the given scope. This may be compiler warning flags, debugging flags, optimization flags, etc. Note that this specifically does *NOT* include "`-I`" and "`-D`" arguments. Such arguments are likely to only be specific to the SSI module, and need not be propagated to the rest of the LAM source tree.
- `CXXFLAGS`: Just like `CFLAGS`, but will be passed to the C++ compiler.[4]
- `FFLAGS`: Just like `CFLAGS`, but will be passed to the Fortran compiler.
- `LDFLAGS`: Typically "`-L`" flags that need to be passed to the linker to link to this SSI module.

[4] Note that some parts of LAM are actually written in C++, so if you are setting `CFLAGS`, you should probably also be setting `CXXFLAGS`.

- `LIBS`: Typically "`-l`" flags that need to be passed to the linker to link to this SSI module.

Note that GNU Libtool automatically takes care of propagating any required `LDFLAGS` and `LIBS` arguments when building the rest of LAM. Specifically, if Libtool is used to build an SSI module, the module's `configure` script should set appropriate `LDFLAGS` and `LIBS` values that can be used to link a final executable. Libtool will then automatically propagate these flags to any executable in the LAM source tree that needs them. Hence, setting `LIB*_EXTRA_LDFLAGS` and `LIB*_EXTRA_LIBS` is frequently not necessary; the `WRAPPERS_EXTRA_*` variables are typically the only values that need to be passed upward.

It is *never* necessary for a module to pass "`-I`" and "`-D`" flags back to the upper-level LAM building/configuration environment. This is why Table 2 does not have any variables for `CPPFLAGS`. Module-specific header files, by definition, will only be needed to compile the module. They will not be needed by the rest of LAM, nor user LAM or MPI programs.

3.2.3 When the `configure` Script *Must* Succeed

When building a distribution package of LAM/MPI, it is desirable to include *all* SSI modules, regardless of whether they can configure successfully or not. In this case, modules that normally only allow themselves to be configured successfully when certain conditions are met (e.g., a module that only builds when specific third party libraries and header files are present) should bypass these checks and allow themselves to be configured pseudo-successfully. Specifically, the module's `configure` scripts must return an exit status of 0 and produce `Makefile`s that, at a bare minimum, have working "`dist`" make targets.

LAM requires this behavior when the `--enable-dist` switch is used. Specifically: if LAM (and all of its SSI modules) is configured with this switch, all SSI modules must produce a top-level `Makefile` with a working "`dist`" target. Note that no other targets are required to be functional. Hence, all conditional tests can be skipped such that valid `Makefile`s can be generated, even if the module cannot actually be built.

SSI modules that use the `LAM_BASIC_SETUP` macro from the `lam_functions.m4` file automatically include handling of the `--enable-dist` switch. If `--enable-dist` is specified, a warning is displayed that the `dist` target may be the only functional make target, and the `LAM_WANT_DIST` environment variable is set to "`yes`". Hence, conditionally-configured SSI modules can simply check the value of `$LAM_WANT_DIST` to know whether to perform conditional tests or not.

SSI modules that are not built conditionally can ignore this entire section, since they always produce fully functional `Makefile`s that include a valid `dist` target.

4 Building the Module

The rest of this document describes modules that have previously run their `configure` executables and had them succeed. Modules that failed the configuration phase will not be built or installed. The main output after building a module is a GNU Libtool file in the module's top-level directory named `liblam_ssi_<kind>_<name>.la`.

The `all` target may function any way that it wishes as long as the GNU Libtool file `liblam_ssi_<kind>_<name>.la` is produced at the end. The LAM build infrastructure will simply invoke "`make all`" and wait for the output file to be created.

5 Installing the Module

If a module is built statically, "`make install`" should be a no-op because the entire contents of the module will be folded into the upper-level LAM or MPI library. For example, if using GNU Automake and

Name	Description
LIBLAM_EXTRA_CFLAGS	Extra flags that need to be passed to the C compiler for LAM SSI kinds.
LIBLAM_EXTRA_CXXFLAGS	Extra flags that need to be passed to the C++ compiler for LAM SSI kinds.
LIBLAM_EXTRA_FFLAGS	Extra flags that need to be passed to the Fortran compiler for LAM SSI kinds.
LIBLAM_EXTRA_LDFLAGS	Extra "-L" arguments required for building executables that link to liblam.* for LAM SSI kinds.
LIBLAM_EXTRA_LIBS	Extra "-l" arguments required for building executables that link to liblam.* for LAM SSI kinds.
LIBMPI_EXTRA_CFLAGS	Extra flags that need to be passed to the C compiler for MPI SSI kinds.
LIBMPI_EXTRA_CXXFLAGS	Extra flags that need to be passed to the C++ compiler for MPI SSI kinds.
LIBMPI_EXTRA_FFLAGS	Extra flags that need to be passed to the Fortran compiler for MPI SSI kinds.
LIBMPI_EXTRA_LDFLAGS	Extra "-L" arguments required for building executables that link to liblam.* for MPI SSI kinds.
LIBMPI_EXTRA_LIBS	Extra "-l" arguments required for building executables that link to liblam.* for MPI SSI kinds.
WRAPPER_EXTRA_CFLAGS	Extra flags that need to be passed through the wrapper compilers to the C compiler to compile user MPI programs.
WRAPPER_EXTRA_CXXFLAGS	Extra flags that need to be passed through the wrapper compilers to the C++ compiler to compile user MPI programs.
WRAPPER_EXTRA_FFLAGS	Extra flags that need to be passed through the wrapper compilers to the Fortran compiler to compile user MPI programs.
WRAPPER_EXTRA_LDFLAGS	Extra "-L" arguments that need to be passed through the wrapper compilers to the linker when linking user MPI programs.
WRAPPER_EXTRA_LIBS	Extra "-l" arguments that need to be passed through the wrapper compilers to the linker when linking user MPI programs.

Table 2: Table of available return variables from module configure scripts.

Libtool to build the module, the `Makefile.am` macro "`noinst_LTLIBRARIES`" can be used to specify the module library to be built.

If a module is built dynamically, "`make install`" may install the resulting library to LAM's `$pkglibdir` directory. This is typically defined as `$exec_prefix/lib/lam`.[5] if using GNU Automake and Libtool to build the module, the `Makefile.am` macro "`pkglib_LTLIBRARIES`" can be used to specify the module library to be built.

It should be noted that there is no harm in installing a statically-built module – it will just never be referenced or used.[6]

Finally, a module may install its own custom helpfile – a text file containing templates for detailed help messages that can be displayed at run time. These files may be named arbitrarily, but should follow the prefix rule, and should be installed into LAM's `$sysconfdir`. For example, if using GNU Automake, the following line can be used to properly install a help file (and include it in a tarball distribution):

```
sysconf_DATA = lam-ssi-rpi-tcp-helpfile
```

See Section 7.2 (page 26) for more details on help files.

6 Selecting the Module at Run Time

For some component types, there must be at least one available module at run-time. For example, when invoking `lamboot`, there must be at least one available boot SSI module. If not, an error occurs, and `lamboot` will abort. Other component types do not necessarily need a module available at run-time. For example, it is *not* an error is there is no checkpoint-restart modules available when running an MPI application; the application will simply run without checkpoint-restart support.

However, there may be cases where there are more than one available module for a given kind, and LAM must choose which one to use at run time. A simplistic system is used that is based on levels of priority, but can be overridden by user-supplied selections.

6.1 Terminology

An *available* module is one that:

- Was successfully configured, compiled, and linked into an executable
- Notifies the SSI framework at run time that it can be successfully executed

A module is *selected* when the SSI framework determines that it will be used at run-time.

The *scope* of the selected module is the set of conditions in which the selected module will be used. For example, some types will only have one module selected for the life of the entire process, while other types may select a different module for each MPI communicator. As such, the scope is defined by each SSI kind.

[5] Note that LAM does not currently support loading dynamic modules. Dynamically-loadable modules are mainly mentioned for future expansion. See Footnote 1.2.1 on page 4 for more details.

[6] Indeed, it may be required for future support of dynamic modules that all SSI modules *are* installed to LAM's `$pkglib` directory, even if static linking is used. This is trivial to change in relevant `Makefile.ams`, however.

6.2 General Scheme

A user may select a specific module at run-time via the mechanisms described in Section 8. Specifically, the environment variable `LAM_MPI_SSI_<kind>` or command line parameters "`-ssi <kind> <name>`" are used to select a particular module. In this case, LAM will search for a module of the specified kind with the desired name. If a module by that name is not found, an error will occur. If a matching module is found, the module will be queried to see if it is able to be run in the desired scope. If it is, that module will be used. If the module indicates that it cannot be run, an error will occur.

If the user does not specify which module to use, LAM will attempt to select the "best" module from the set of available modules. Each module will be queried to see if it is able to run, and if so, what its priority is (priorities are *only* observed when the user has not selected which module to use). Priorities are integers in the range of $[0, 100]$, with 100 being the highest priority. The module with the highest priority will typically be selected as the winner (although other factors may enter the selection criteria for MPI programs – see Section 6.2.1, below); the losing module(s) will be ignored in the selection scope. If there is a tie, LAM is free to pick any of the modules with the highest priority to be the winner.

Although the exact meanings of priorities are arbitrary, the following guidelines are provided for module authors in assigning priority values:

- A priority of 0 should be interpreted as "if nothing else is available, this one will work." It is typically reserved for a last-choice type of module.
- A priority of at least 50 should be given when a module detects that it is running in its "native" environment. For example, if `lamboot` is executed in a PBS batch queuing environment, the `tm` boot SSI module will return a priority of at least because it knows that it can run.
- A priority of at least 75 should be given when a module detects that it is running in its "native" environment, and a compile-time switch was enabled to make that module the default for its type.
- A priority of 100 should be reserved for modules that *must* be selected.

It is also recommended that module authors provide mechanisms for users to override the returned priority levels. This will allow arbitrary control of run-time selection by users (and/or system administrators).

6.2.1 MPI Selection Algorithm

Since MPI applications must share at least some common set of MPI SSI modules, the decision of which MPI modules to use has to be a global choice, not a local choice.

In the current implementation, LAM has a fairly simplistic selection mechanism. Each MPI process will independently choose which RPI and CR modules it will use (based on priority and the MPI thread level) and sends them all to `mpirun`. If `mpirun` detects that all processes chose the same module set, it will kill all the MPI processes and print an appropriate error message. Hence, the module selection choice is not really global – it is just an error check to ensure that the selection choice was the same among all processes.

The selection algorithm in each MPI process is also fairly simplistic, and is determined by at least two variables: priority and the requested MPI thread level. Each MPI SSI kind conveys the range of MPI thread levels that it supports (denoted $[t_l, t_u]$, for the lower and upper bounds of MPI thread level support that it can provide, where $t_l \leq t_u$). The user application indicates the desired level of thread support via MPI_INIT_THREAD (MPI_THREAD_SINGLE is assumed if MPI_INIT is used, unless it is overridden by the `LAM_MPI_THREAD_LEVEL` environment variable). The MPI selection algorithm operates as follows:

1. The requested thread level support is denoted as t.

2. Find the highest priority rpi where $t \leq t_u$. If no rpi modules are found matching this criteria, abort.
3. If the rpi module's t_l is higher than the requested thread level, increment the requested thread level to t_l.
4. Find the highest priority cr module within the rpi module's supported thread level range. If no cr module is found matching this criteria, the process will simply not run with cr support.
5. Eliminate all coll modules that do not fall within the rpi module's supported thread level range.

This mechanism is far from perfect – it is easy to construct examples where MPI processes have an overlapping set of MPI modules yet still fail because the processes independently chose different modules. Future implementations will contain a more intelligent algorithm, such as performing a global intersection of the available MPI modules between all MPI processes and making a global choice (vs. ensuring that everyone independently selected the same modules).

6.3 Specific Selection Mechanisms

Each kind defines its own scoping rules and specific selection mechanism. However, all kinds share one part of the mechanism: the module's open function. See Section 7.1.1.

It is important to node that module selection mechanisms typically run in a distributed fashion. As such, it is critical that all instances in the distributed job come to a mutually agreeable consensus on which modules should be selected for a given scope. In many cases, it is sufficient for each module instance in the distributed job to independently examine the run time environment and come to its own conclusions, relying on the assumption that all the other module instances will come to the same conclusions. However, module authors will need to ensure that this condition can be guaranteed by selecting environment characteristics that will be common across the distributed job. In particular, note that only some types of environments will automatically propagate environment variables (the boot SSI is of notable relevance here).

If distributed consensus cannot be independently achieved, communication between module instances may be required. Each kind's SSI documentation describes when the modules are initialized and examined for availability and selection – this determines what level of communication is available to the modules.

7 Module Source Code

The rest of this section (and document) describes modules that have previously run their `configure` executables and had them succeed. Modules that failed the configuration phase will not reach this phase. Each module must provide:

- A C `struct` is exported of type `lam_ssi_<kind>_t` named `lam_ssi_<kind>_<name>_module`. Each SSI kind is responsible for defining (and documenting) its `lam_ssi_<kind>_t`.
- The exported `struct` is specific to each SSI kind, but will likely contain function pointers to API calls. The function pointers must be invokable from C and C++.
- Using C++ in the internals of the module is strongly discouraged as building C++ libraries is not [yet] portable. When GNU Libtool supports making C++ libraries portably, this recommendation will be removed. See Section 1.2.1 (page 4).
- All public symbols (function names, global variables, etc.) in the library must follow the prefix rule; they must have the prefix `lam_ssi_<kind>_<name>_`.

The top-level SSI include file is `share/ssi/include/lam-ssi.h`. This file declares some top-level SSI functions and the C type `lam_ssi_t` (a `struct`). See Section 7.1 for more details. If using GNU Automake and the macros suggested in Section 3.1.3 (page 8), this file can be included by adding an appropriate `-I` flag to Automake's `AM_CPPFLAGS` macro. Additionally, the preprocessor symbol `LAM_BUILDING` *must* be defined and set to the value of 1. For example:

```
AM_CPPFLAGS = -DLAM_BUILDING=1 -I$(top_lam_srcdir)/share/ssi/include
```

Each SSI kind will likely have its own top-level include file, probably located in the same directory as `lam-ssi.h`. Consult the documentation for each SSI kind for the specific location of that kind's header file. Each SSI kind's header file will define the datatype `lam_ssi_<kind>_t` which will be used in all modules of that kind.

7.1 Base SSI API

Every SSI module – regardless of kind – must export basic description information.

7.1.1 Module Open Function

A module may have an *open* function. The open function has two main purposes:

- Make the first determination whether it is possible for the module to run or not.[7]
- Allocate any one-time, module-specific (but action-agnostic) resources.

The open function is described by a pointer type defined as follows:

```
typedef int (*lam_ssi_open_module_fn_t)(OPT *args);
```

A typical module open function may be prototyped as follows:

```
int lam_ssi_kind_name_open_module(OPT *args);
```

Although `opt` is passed in (which is a conglomeration of the command line parameters – see LAM's man page `all_opt(3)` for details on how to use it), it is expected that most parameters and selection criteria will be passed through the environment (see Section 8).

Relevant environment variables will usually adhere to the prefix rule – they will be named in the form of `LAM_MPI_SSI_<kind>_<name>`,[8] where `<kind>` and `<name>` will usually be lower case. See the man pages for `mpirun(1)`, `lamssi(7)`, and `lamssi_rpi(7)`, as well as Sections 6 and 8 for more information about passing parameters to SSI modules through environment variables.

Determine whether the module can run. The open function may check the current run-time environment and determine if it is able to run. This may entail allocating resources and/or checking environmental or other external factors. If the module determines that it is able to be run, it should eventually return 1. If not, it should return 0, and the SSI framework will ignore the module for the remainder of the execution of that process.

If an open function is not provided by the module, it is implied that the open function returned 1, and becomes eligible for consideration.

[7] See Section 6 for more details on how a given module is selected (or not) at run time.

[8] Note that all variables in `mpirun`'s environment that begin with the prefix `LAM_MPI_` are automatically copied to the environment of each MPI process that is started by `mpirun`. In this way, setting the environment variable `LAM_MPI_SSI_<kind>_<name>_parameter` will automatically be propagated out to all MPI processes in the parallel job.

Allocate one-time resources. Depending on the SSI kind, some modules may be invoked multiple times in different scopes. In such situations, it may be desirable to have one-time initialization and/or resource allocations that apply to all scopes.

7.1.2 Module Close Function

Each module may also have a close function. The main purpose of the close function is to free any resources allocated by the module.

This functions is described by a function pointer type defined as follows:

```
typedef int (*lam_ssi_close_module_fn_t)(void);
```

A typical module close function is prototyped as follows:

```
int lam_ssi_kind_name_close_module(void);
```

The close function will be invoked exactly once for each module whose open function returned 1. The timing of when the close function is invoked varies depending on the kind. For example, the boot SSI has only one scope – so unselected modules will be closed during the initialization of the process. But the coll SSI potentially has many scopes; so available coll modules will not be closed until `MPI_FINALIZE`.

7.1.3 The Base SSI Datatype: `lam_ssi_t`

Every module must export a public symbol named `lam_ssi_<kind>_<name>_module` of type `lam_ssi_<kind>_t`. While each SSI kind is responsible for declaring exactly what that type is, it *must* contain a `lam_ssi_t` as its first element. The `lam_ssi_t` type is used to store basic information about a module instance (including pointers to its open and close functions, as described in Sections 7.1.1 and 7.1.2).

The `lam_ssi_t` type is defined as follows:

```
typedef struct lam_ssi_1_0_0 {

  /* Integer version numbers indicating which SSI API version
     this module conforms to. */

  int ssi_major_version;
  int ssi_minor_version;
  int ssi_release_version;

  /* Information about the kind and the version of the kind's API
     that it conforms to*/

  char ssi_kind_name[LAM_MPI_SSI_BASE_MAX_KIND_NAME_LEN];
  int ssi_kind_major_version;
  int ssi_kind_minor_version;
  int ssi_kind_release_version;

  /* Information about the module itself */

  char ssi_module_name[LAM_MPI_SSI_BASE_MAX_MODULE_NAME_LEN];
  int ssi_module_major_version;
```

```
  int ssi_module_minor_version;
  int ssi_module_release_version;

  /* Functions for opening and closing the module */

  lam_ssi_open_module_fn_t ssi_open_module;
  lam_ssi_close_module_fn_t ssi_close_module;
} lam_ssi_1_0_0_t;

/* Set the default type to use version 1.0.0 of the SSI struct */

typedef lam_ssi_1_0_0_t lam_ssi_t;
```

The members of this `struct` are:

- The first group of three members (`ssi_*_version`) refers to the version of SSI that the module conforms to. In this case, they should be hard-coded to 1, 0, and 0, respectively. This is for version control purposes, and is explained in greater detail below.
- The second group of three members (`ssi_kind_*_version`) refers to the API version of the kind. This is also for version control purposes, and is explained below.
- The third group of three members (`ssi_module_*_version`) refers to the version of the module instance itself. The contents of these members are left up to the module.
- The open and close function pointers point to functions as described in Sections 7.1.1 and 7.1.2, respectively. NULL values may be supplied for these pointers if the module does not have open and close functions. Note, however, that a NULL value for the open function implies that the module is eligible for availability consideration (i.e., it is as if the open function *did* exist, and returned 1 when invoked).

The base SSI API is currently at version 1.0.0. However, future versions may alter this type, which is why the first three `int` values in the struct will always be the SSI API version number. For example, if the layout of the type changes in future SSI API versions, the LAM SSI framework will be able to apply the right type to a given module's `lam_ssi_t` by examining the first three `int` values. Hence, backward compatibility may be preserved for modules that do not keep up with the newest versions of the SSI API.[9]

Note that while the rest of this document refers to `lam_ssi_t` for simplicity, it is strongly recommended for modules to use specific versions of types (e.g., `lam_ssi_1_0_0_t` and `lam_ssi_rpi_1_0_0_t`) so that if new versions of APIs become available and the base types change, the module will still be able to compile successfully.

7.1.4 Example Usage: the rpi SSI Kind

While the fields of `lam_ssi_t` are rather self-explanatory, here is an example from the rpi SSI kind. The rpi defines a type named `lam_ssi_rpi_<version>_t`. Its first element will always be some version of `lam_ssi_t`:

[9] Note that this is not guaranteed, however. The LAM Team reserves the right to define "backwards compatibility" however it wants, such as: "outputting an error message saying, 'Sorry, this module conforms to SSI API version a.b.c, but the current version is d.e.f.' and then ignoring that module."

```
typedef struct lam_ssi_rpi_1_0_0 {
  lam_ssi_1_0_0_t lsr_meta_info;

  /* ...various other members specific to the RPI SSI... */
} lam_ssi_rpi_1_0_0_t;

typedef lam_ssi_rpi_1_0_0_t lam_ssi_rpi_t;
```

Note that the same version technique is used with the rpi SSI kind as with the base SSI API – the rpi API show here is also version 1.0.0. Future versions may change the layout of `lam_ssi_rpi_t`, but since the first member of it is guaranteed to be some version of `lam_ssi_t`, both the SSI API version and rpi API version numbers can be guaranteed to be successfully extracted.

To continue the example, the `tcp` rpi module exports a global symbol named `lam_ssi_rpi_tcp_module`. Here's an except from its code:

```
const lam_ssi_rpi_1_0_0_t lam_ssi_rpi_tcp_module = {

  /* First, the lam_ssi_1_0_0_t struct containing meta
     infomration about the module itself */

  {
    /* SSI API version */

    1, 0, 0,

    /* Module kind name and version */

    "rpi",
    1, 0, 0,

    /* Module name and version -- obtained and AC_DEFINE'ed by
       the tcp RPI configure script */

    "tcp",
    LAM_SSI_RPI_TCP_MAJOR_VERSION,
    LAM_SSI_RPI_TCP_MINOR_VERSION,
    LAM_SSI_RPI_TCP_RELEASE_VERSION,

    /* Module open and close function pointers */

    NULL,
    NULL
  },

  /* ...various other members specific to the RPI SSI... */
};
```

Note that the open and close functions are NULL. This is because the `tcp` rpi does not require any additional per-module initialization or finalization. All of its initialization and finalization code is contained within the rpi API calls. Hence, it will always be considered available for selection.

7.2 Help Messages

LAM has an internal function to help printing lengthy help messages on `stdout`: `show_help_file()`. This functions allows SSI modules to provide custom text files containing templated help messages that can be sent to `stdout` at run-time. This allows modules to print detailed, thorough help messages at run time, but alleviating the need for long sequences of `printf()` statements in code. The prototype for this function is:

```
#include <etc_misc.h>

void show_help_file(const char *filename, const char *program, const char *topic, ..., NULL);
```

The arguments are as follows:

- `filename`: The name of the file to look for the templated message. This should typically follow the prefix rule, and be a file installed in the LAM `$sysconfdir` (typically `$prefix/etc`). If NULL is provided for this argument, LAM's main help file will be used (which is probably not what you want).

  The format of the file is a simple text-based layout; LAM's main help file provides a lot of examples of templated help messages – see `$sysconfdir/lam-helpfile`). Note that multiple help message can be contained in a single file.

- `program`: The first key in finding the specific help topic in the help file. A NULL value matches the key "`ALL`".

- `topic`: The second key in finding the specific help topic. A NULL value matches the key "`ALL`".

- Additional arguments may follow (varargs-style) that will be substituted into the templated help message. The list of arguments (even if it is empty) *must* be terminated with NULL.

7.3 Verbosity

The global variable `lam_ssi_verbose` will be set to a non-negative integer if the user has asked for general SSI verbosity (otherwise it will be set to -1). See the `mpirun(1)` and `lamssi(7)` man pages for more details about SSI verbosity.

If this variable is 0, modules are encouraged to write minimal diagnostics using the LAM debug system. Larger values should cause more module-specific diagnostic information to be printed. The exact meanings of larger values are determined by the modules themselves.

The LAM debug system uses `printf`-style varags parameter passing. The function `lam_debug_output()` is used to output verbosity information to the output stream(s) that the user has chosen. For example:

```
if (lam_ssi_verbose >= 10)
  lam_debug(lam_ssi_did, "Hello, world.  I am %d of %d",
            rank, size);
```

Note the use of another global variable – `lam_ssi_did` (debug ID). It is the first parameter to the `lam_debug_output()` function call and should not be modified. Also note that "`\n`" is not necessary at the end of the string. Since the debug output stream may be sent to multiple different kinds of output (including, for example, the syslog), the LAM debug system will take care of adding a "`\n`" if it is necessary.

Also note that each SSI kind may have its own verbosity global variable and debug ID. If so, modules should use those instead of the overall SSI values.

The LAM full debugging output interface is described in [1].

8 Passing Run-Time Parameters to the Module

Parameters can be passed to SSI modules at run-time by the command line or through the environment. Although `argc` and `argv` are passed to all module open functions, it is preferably to leave the command line in the user domain and pass all SSI module parameters through the environment. This is also somewhat influenced by the fact that there is no portable way to obtain `argc`/`argv` from Fortran programs; relying on the environment is the safest method.

The `mpirun(1)` man page goes into detail about how its `-ssi` command line switch can be used to set environment variables that will automatically be propagated to all MPI processes that are part of the parallel job. The names of these environment variables follows the prefix rule – they will be prefixed with `LAM_MPI_SSI_<kind>_<name>`. For example, the tcp RPI module can take an optional parameter "`rpi_tcp_short=size`" to specify the shortest long message size. This could be passed in one of two ways:

1. Explicitly set an environment variable:

```
$ export LAM_MPI_SSI_rpi_tcp_short=131072
$ mpirun C myprogram
```

2. Use the `-ssi` command line switch to `mpirun`:

```
$ mpirun –ssi rpi_tcp_short 131072 C myprogram
```

Both methods will yield exactly the same result – the variable `LAM_SSI_rpi_tcp_short` will be in the environment of each MPI process, and have a string value of "`131072`".

Since parameters are specific to a module, each module should document what parameters it will accept. A good place to put such documentation would be manual pages that get installed to the standard man page directory (`${prefix}/man`) during `make install`. As with almost everything else, the prefix rule would apply to manual pages; they should be named in the form of "`lam_ssi_<kind>_<name>.7`", although shortening to "`lamssi_<kind>_<name>.7`" is also acceptable.

9 Unresolved Issues

The following issues are un-resolved or otherwise less-than-optimal:

- "`./configure --help`" doesn't show all the options from the individual SSI `configure` scripts.

  Suggestion: some other top-level LAM script (e.g., "`./configure_help`"?) that does the Right Thing to print out all the available options...?

10 Acknowledgements

This work was supported by a grant from the Lily Endowment and by National Science Foundation grant 0116050.

References

[1] Brian Barrett, Jeff Squyres, and Andrew Lumsdaine. *LAM/MPI Design Document*. Open Systems Laboratory, Pervasive Technology Labs, Indiana University, Bloomington, IN. See `http://www.lam-mpi.org/`.

[2] Richard Stallman et al. *GNU Coding Standards*. Free Software Foundation, October 2001. Included in the GNU `autoconf` distribution; see `ftp://ftp.gnu.org/gnu/autoconf/`.

[3] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. Checkpoint-restart support system services interface (SSI) modules for LAM/MPI. Technical Report TR578, Indiana University, Computer Science Department, 2003.

[4] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. Boot system services interface (SSI) modules for LAM/MPI. Technical Report TR576, Indiana University, Computer Science Department, 2003.

[5] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. MPI collective operations system services interface (SSI) modules for LAM/MPI. Technical Report TR577, Indiana University, Computer Science Department, 2003.

[6] Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. Request progression interface (RPI) system services interface (SSI) modules for LAM/MPI. Technical Report TR579, Indiana University, Computer Science Department, 2003.

Index

.lam_ignore file, 8, 15
.lam_no_gnu file, 8

AC_CANONICAL_HOST M4 macro, 11
autogen.sh command, 7, 8

commands
    autogen.sh, 7, 8
    configure, 7
    get_lam_version, 10
    lamclean, 12
configure command, 7
configure flags
    --enable-dist, 12, 17
    --prefix, 12
configure.ac file, 8
configure.in file, 8

--enable-dist configure flag, 12, 17
environment variables
    LAM_MPI_THREAD_LEVEL, 20

files
    .lam_ignore, 8, 15
    .lam_no_gnu, 8
    configure.ac, 8
    configure.in, 8
    lam_check_func.m4, 13
    lam_setup_cc.m4, 12
    lam_functions.m4, 9, 11, 13, 14
    lam_get_version.m4, 10

get_lam_version command, 10

LAM_BASIC_SETUP M4 macro, 11
LAM_CHECK_FUNC M4 macro, 13
lam_check_func.m4 file, 13
LAM_CONFIGURE_SETUP M4 macro, 9
lam_debug_output(), 26
LAM_GET_VERSION M4 macro, 10
LAM_LOG_COMMAND M4 macro, 14
LAM_LOG_FILE M4 macro, 14
LAM_LOG_MSG M4 macro, 13
LAM_MPI_THREAD_LEVEL environment variable, 20
LAM_SETUP_CC M4 macro, 12
lam_setup_cc.m4 file, 12
lam_functions.m4 file, 9, 11, 13, 14
lam_get_version.m4 file, 10
lamclean command, 12

M4 macros
    AC_CANONICAL_HOST, 11
    LAM_BASIC_SETUP, 11
    LAM_CHECK_FUNC, 13
    LAM_CONFIGURE_SETUP, 9
    LAM_GET_VERSION, 10
    LAM_LOG_COMMAND, 14
    LAM_LOG_FILE, 14
    LAM_LOG_MSG, 13
    LAM_SETUP_CC, 12
MPI functions
    MPI_INIT, 20
    MPI_INIT_THREAD, 20
MPI_INIT, 20
MPI_INIT_THREAD, 20

--prefix configure flag, 12

show_help_file(), 26