SoftPLC C++ Toolkit

Version 5

Table of Contents

Chapter 1. Overview

1.1. Introduction

This document describes the installation, usage, and functionality of the SoftPLC C++ Toolkit for the amazing for SoftPLC runtime version 5.x. The SoftPLC runtime is a world leading programmable logic controller engine that has tremendous capacity, flexibility, and utility for automation applications. It runs on an embedded linux architecture with full access to debian trixie supplementary packages. As of this writing the linux kernel supplied by SoftPLC Corporation is currently at version 5.10.

1.2. Host System Requirements for Toolkit

Requirement	Specification
Operating System	• **Linux:** Any modern distribution running on an `x86_64` (`amd64`) architecture. • **macOS:** Intel-based Macs or Apple Silicon Macs (M-series) running Docker Desktop. • **Windows:** Windows 10/11 (Home, Pro, Enterprise) or Windows Server running Docker Desktop.
Processor Architecture	`x86_64` (`amd64`). NOTE: Apple Silicon and ARM64 hosts can run this image via built-in Docker emulation, but compilation performance will be significantly degraded.
Container Engine	**Docker Engine** (Linux) or **Docker Desktop** (macOS / Windows) installed and active.
System Memory (RAM)	**4 GB minimum** allocated to the Docker daemon. 8 GB or more is recommended for parallelized multi-core (`make -j`) GCC compilations.
Disk Space	**10 GB of free space** minimum to accommodate the SoftPLC Toolkit docker image which contains cross-compiler toolchains. Space is also required for your source code and build artifacts.

1.3. Definitions

Runtime

The powerful core realtime userspace process running the show. It can load multiple purpose subsidiary components called TOPDOC Loadable Modules, and for this reason can be extended in nearly unlimited ways. Congratulations on being in a position to contribute to the expansion. The runtime load and run a large number of TLMs concurrently.

TLM

A *TOPDOC Loadable Module* that you write in C++. One TLM can implement the following

features, either one or all:

- a driver
- one or more custom ladder logic instructions
- one or more finite state machines

  Since a TLM is executable within the SoftPLC runtime, it must be downloaded to a runtime CPU.

TLI

A *TOPDOC Loadable Instruction* is a custom ladder instruction written in C++ that resides in a TLM.

FSM

A finite state machine is a C++ software construct that runs code based on which state it is in. External events and time will cause it to change state, and as it changes state it will be in a position to run state specific alternative code. It is a wonderfully efficient design model. But more importantly, it is absolutely spectacular for troubleshooting. This is because its state is always known, and if it is not transitioning between states as intended, the problem search space as to why it did not transition out of its current state is very small and easy to understand. This toolkit provides a base class FSM which you can extend for a particular process or machine. It has very helpful timing support in it. There is no limit to how many FSMs you have in a single TLM, and because each is extremely efficient, the CPU burden is trivial.

Chapter 2. Terms of Use

Because of the variety of uses of the information described in this manual, the users of, and those responsible for applying this information must satisfy themselves as to the acceptability of each application and use of the information. In no event will SoftPLC Corporation be responsible or liable for its use, nor for any infringements of patents or other rights of third parties which may result from its use.

SOFTPLC CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE.

SoftPLC Corporation reserves the right to change product specifications at any time without notice. No part of this document may be reproduced by any means, nor translated, nor transmitted to any magnetic medium without the written consent of SoftPLC Corporation.

SoftPLC, TOPDOC, and TOPDOC NexGen are registered trademarks of SoftPLC Corporation.

First Printing: July, 2005
Latest Printing: August, 2026

SoftPLC Corporation
25603 Red Brangus Drive
Spicewood, Texas 78669

USA Telephone: 1-800-SoftPLC
URL: http://softplc.com
Email: support@softplc.com

Chapter 3. Usage

The SoftPLC C++ Toolkit builds **TLMs** (Topdoc Loadable Modules) for the SoftPLC runtime. The toolkit is a freely downloadable self-contained **Docker image** that carries the cross compilers, the toolkit headers and libraries, the CMake build system, and this documentation. You do not install a compiler toolchain on your workstation — you run the toolkit's docker image and allow it to access one or more working directories on your development computer. This keeps your development computer free from installations that you may not need to use all the time.

This is a **C++** toolkit, meaning a C++ compiler is used for compiling the main source file of the TLM. However if you have C code that you want to link into the TLM this is possible using either of two methods. 1) you may compile your C code using the C++ compiler, and this is sometimes quite easy, just rename the files to have a .cc extension or .cpp extension. 2) you may compile your C code with the included C compiler and simply link those outputs into the TLM. You will not be able to use APIs in the runtime from C however, but you can include complete C libraries that don't look upwards into the runtime. **CMake** is the supported build tool. To compile a C file, simply make sure its file extension is '.c' and CMake will automatically know to use the C compiler on that file. Again, the main file must be C++. CMake is a wonderfully powerful build system.

3.1. What you need

- **Docker** — Docker Desktop on macOS or Windows, or Docker Engine on Linux.
- Either the `standup-toolkit.sh` script (command-line workflow) **or** VS Code with the **Dev Containers** extension (GUI workflow). Either way, the toolkit image is pulled from Docker Hub automatically by Docker the first time you try to use the toolkit image.

3.1.1. Windows and macOS

The toolkit image itself is the same on every host — only the way you start it differs. There is no Windows batch file and none is needed:

- **VS Code Dev Containers** (Option B) works natively on Windows and macOS with nothing extra. `devcontainer.json` carries no host-specific settings, and Docker Desktop maps bind-mount file ownership for you. This is the recommended path on both.
- `standup-toolkit.sh` (Option A) is a bash script. On macOS it runs as-is. On Windows, run it from a **WSL2** shell — Docker Desktop's WSL2 backend puts the `docker` command inside your WSL distribution, and the script then works unmodified.

Under WSL2, keep your TLM source and build directories in the **WSL filesystem** (for example `~/tlms/mytlm`), not under `/mnt/c`. Windows drives are reached through a translation layer that is markedly slower for compile workloads and that only emulates Unix permissions.

Git Bash is not a supported way to run `standup-toolkit.sh`: it rewrites the container-side half of `-v host:/source` into a Windows path, and `docker run -it` there needs `winpty` to get a terminal. Use WSL2 instead.

3.2. Two directories: source and build

Every TLM build uses at least two directory trees, which are mounted into the container:

Host role	In container	Contents
source	`/source`	your TLM's source tree — the `.cc` / `.h` files and its `CMakeLists.txt`
build	`/build`	an initially empty directory that receives all build output; keep it separate from source and delete it any time

A common layout keeps one build directory per target architecture:

```
mytlm/                  <- source tree  (mounted at /source within the container)
  CMakeLists.txt
  mytlm.cc
  build-arm64/          <- build directory (mounted at /build within the container)
```

3.3. Starting the toolkit

3.3.1. Option A — command line (standup-toolkit.sh)

The script is not part of the toolkit package — download it once from https://softplc.com/usermanuals/SoftPLC-C++-Toolkit/docs/standup-toolkit.sh, save it somewhere on your `PATH`, and make it executable:

```
$ chmod +x standup-toolkit.sh
```

This copy is always the current one; it is published from the same file the toolkit maintainers use.

From your build directory, run the script with the path to your source tree as the only command line argument. Your current directory becomes `/build`; the argument becomes `/source`:

```
$ cd ~/tlms/mytlm/build-arm64      # ~/tlms/mytlm/build-arm64 becomes /build
$ standup-toolkit.sh ~/tlms/mytlm  # ~/tlms/mytlm becomes /source
```

You land in a shell inside the container, already in `/build`.

On Windows, run these commands from a WSL2 shell, with both directories inside the WSL filesystem. See Windows and macOS.

3.3.2. Option B — VS Code Dev Containers (friendlier)

This option runs the whole toolkit inside VS Code — you edit your source, build, and get a terminal that is already inside the container, all in one window and with no `docker` commands to type. It takes a one-time setup, after which starting the toolkit is just a click or two.

One-time setup (once per computer)

1. Install **Docker** (see What you need above) and **Visual Studio Code** from https://code.visualstudio.com.
2. Start VS Code and install Microsoft's **Dev Containers** extension: click the Extensions icon in the left tool bar (or press `Ctrl+Shift+X`, or `Cmd+Shift+X` on macOS), type `Dev Containers` in the search box, and press **Install** on the extension published by Microsoft.

Then, for each TLM project

1. In the top folder of your TLM project, create a subfolder named `.devcontainer`, and inside it a file named `devcontainer.json` containing exactly this:

```
{
  "name": "SoftPLC Toolkit 5",
  "image": "softplc/toolkit:trixie_latest",

  // Your TLM source is mounted at /source; builds go in /build (created on the
  // host first so the bind mount is not root-owned).
  "workspaceMount": "source=${localWorkspaceFolder},target=/source,type=bind",
  "workspaceFolder": "/source",
  "initializeCommand": "mkdir -p ${localWorkspaceFolder}/build-arm64",
  "mounts": [
    "source=${localWorkspaceFolder}/build-arm64,target=/build,type=bind"
  ],

  // Run as the image's non-root "builder" user so build outputs are owned by you,
  // not root.  On Linux, updateRemoteUserUID makes VS Code rewrite builder's
  // uid/gid to match your host user automatically -- nothing to hard-code.  (On
  // macOS/Windows Docker Desktop maps ownership for you, so this is a no-op
there.)
  "remoteUser": "builder",
  "updateRemoteUserUID": true,

  "customizations": {
    "vscode": { "extensions": ["ms-vscode.cpptools", "ms-vscode.cmake-tools"] }
  }
}
```

2. In VS Code, choose **File** > **Open Folder…** and open your TLM project's top folder (the one that now contains the `.devcontainer` subfolder).
3. Reopen that folder inside the container, either way:
   - Click **Reopen in Container** in the pop-up notification VS Code shows at the lower-right, **or**
   - Open the Command Palette — press `F1` (or `Ctrl+Shift+P`, or `Cmd+Shift+P` on macOS) — type `Dev Containers: Reopen in Container`, and press Enter.

   The first time, VS Code downloads the toolkit image and starts the container, which can take a few minutes; later starts are quick. When it finishes, the whole VS Code window is running

inside the toolkit.

4. Open a terminal with **Terminal** > **New Terminal**. It opens in `/source`; type `cd /build` to reach the build directory, then follow Building a TLM below.
5. **Optional — build with one keystroke.** If you would rather press a key than type `cd /build && make`, create a second subfolder named `.vscode` beside `.devcontainer`, and in it a file named `tasks.json` containing this:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Build TLM",
      "type": "shell",
      "command": "make",

      // The task runs inside the container, so this is the container's /build.
      "options": { "cwd": "/build" },

      // Makes this the default build task, so Ctrl+Shift+B runs it.
      "group": { "kind": "build", "isDefault": true },

      // Turns compiler messages into clickable entries in the Problems panel.
      "problemMatcher": "$gcc"
    }
  ]
}
```

   Now **Terminal** > **Run Build Task** — or just `Ctrl+Shift+B` (`Cmd+Shift+B` on macOS) — runs `make` in `/build`, and any compiler error becomes a clickable link that opens the offending line in the editor. You still run `makemake.sh` by hand the first time (see Building a TLM); if you build for one architecture all day, you can add it as a second task the same way.

To return to normal (non-container) VS Code, run `Dev Containers: Reopen Folder Locally` from the Command Palette, or just close the window. See Leaving the container for what becomes of the container itself.

3.4. Building a TLM

Inside the container, from `/build`, configure with `makemake.sh`, then build with `make`:

```
$ /opt/softplc/c++-toolkit-5/bin/makemake.sh ARCH /source
$ make
```

`makemake.sh` runs CMake for you. Its **first argument is the target architecture** (ARCH) which is the CPU type you are building for. Pick from here:

ARCH	Target
`arm64`	64-bit ARM
`amd64`	64-bit x86

Think of the first argument as the **architecture**, not a specific board. The rare case of two different CPU boards that share one architecture can be handled inside the TLM's own `CMakeLists.txt` (see below), not on the `makemake.sh` command line.

The second argument is always `/source`.

`make` produces your TLM as a shared object named `<name>.tlm.so` in the build directory.

3.5. Deploying to the SoftPLC

There are two ways to get your TLM onto the target, depending on where you are in development.

3.5.1. Quick dev-test loop

While iterating, copy the freshly built `<name>.tlm.so` straight into the running SoftPLC's `/SoftPLC/tlm/` directory:

```
$ scp <name>.tlm.so root@<target-ip>:/SoftPLC/tlm/
```

Fast to iterate, but it drops an unmanaged file onto the target.

3.5.2. Release install (recommended)

For the final form of a TLM, ship it as a Debian package so it is version-tracked and cleanly removable by the target's package manager. Build the package with `make package`, then install it in two commands — copy it to the target's `/tmp`, and `dpkg -i` it there over SSH:

```
$ make package        # builds tlm-<name>_<version>_<arch>.deb
$ scp tlm-<name>_<version>_<arch>.deb root@<target-ip>:/tmp/
$ ssh root@<target-ip> dpkg -i /tmp/tlm-<name>_<version>_<arch>.deb
```

The package installs the same `<name>.tlm.so` into `/SoftPLC/tlm/` and records its dependency on `softplc-runtime`.

3.6. Your TLM's CMakeLists.txt

Each TLM is an ordinary CMake project with its own `CMakeLists.txt`. The toolkit supplies the cross toolchain, compiler flags, and runtime libraries through a single `include( tlm )`; your file names the TLM and its sources. A minimal example:

```
set( TLM mytlm )
project( ${TLM} )
cmake_minimum_required( VERSION 3.10 )

include( tlm )                      # toolkit flags, libraries, toolchain

set( SOURCES mytlm.cc )

add_library( ${TLM} SHARED ${SOURCES} )
set_target_properties( ${TLM} PROPERTIES PREFIX "" SUFFIX ".tlm.so" )

target_link_libraries( ${TLM}
    ${LIBSTATE_LOGIC_LIBRARY}
    rt
    )

install( FILES ${CMAKE_CURRENT_BINARY_DIR}/${TLM}.tlm.so
         DESTINATION /SoftPLC/tlm/ )

include( CPack )                    # enables `make package` (the .deb)
```

3.6.1. Libraries you can link

Add any of these to your TLM's `target_link_libraries()` call. The first two are mandatory for every TLM; link the rest only if you use them. This list is a starting point and will grow.

Link with	Mandatory	Description
`${LIBSTATE_LOGIC_LIBRARY}`	Yes	TLM entry point and FSM state-logic support; required by every TLM.
`rt`	Yes	POSIX real-time library (timers, high-resolution clocks, shared memory).
`${LIBCOMM_LIBRARY}`	No	Serial / communications I/O support (backs the comio API).
`${LIBSEXPR_LIBRARY}`	No	S-expression parsing and serialization (used by the DSN lexer).
`${TDLIB}`	No	Datatable (td) access library.
`PocoFoundation` (or another `Poco…` component)	No	POCO C++ libraries from `libpoco-dev` — networking, JSON/XML, crypto, utilities; link the specific component(s) you use, e.g. `PocoNet`.
`boost_system` (or another `boost_…` component)	No	Boost C++ libraries from `libboost-dev`; most parts are header-only (no link needed), a few compile to a linkable component.

3.6.2. Other cases

`CMakeLists.txt` is also where you handle the less common needs:

- **Third-party C library from source** — add its sources or targets with the usual CMake commands; the C++ compiler builds well-formed C without complaint.
- **Board-specific ("machine") variation within one architecture** — define the board symbol here (for example with `target_compile_definitions`), so that architecture selection (`makemake.sh`) and board selection (`CMakeLists.txt`) stay separate concerns.

3.7. Rebuilding

Edit your sources in your usual editor — outside the container, or in the VS Code Dev Container — then, in the container's `/build`, just run `make` again. You only re-run `makemake.sh` when you change `CMakeLists.txt` or switch architecture.

3.8. Leaving the container

How you leave depends on how you started, but either way the build output in your host's build directory remains — it lives on the host, not in the container.

3.8.1. From the command line (Option A)

Type `exit`. The container stops and is removed, because `standup-toolkit.sh` starts it with `--rm`. Nothing of the container survives, so anything you installed inside it with `apt` is gone; next time you get a clean container.

3.8.2. From VS Code (Option B)

Run `Dev Containers: Reopen Folder Locally` from the Command Palette, or simply close the VS Code window. VS Code manages the container's life for you.

Unlike Option A, the container is **not** started with `--rm`: VS Code keeps it (and a small derived image it builds for your project) so that reopening the folder is fast. You can therefore find it later with:

```
$ docker ps -a
```

If you want it gone — to reclaim disk space, or to start from a clean container — stop and remove it by the name shown there:

```
$ docker rm -f <name>
```

Reopening the folder in VS Code simply builds a fresh one.

Chapter 4. C/C++ Function Categories

The code you write and the code that you use for a TLM can be categorized as below:

Purpose	Description
TLMS	How to code a TLM. These are C++ constructs that comprize or make a body of C++ into a TLM. For example, a TLM must declare a TLM class instance in a specific way.
Helper Functions	Functions that live in the SoftPLC runtime that are callable by a TLM. The TLM when calling these functions actually enters the SoftPLC runtime for services such as reading the datatable, changing operating mode, etc. They are named with an **Hlp** prefix and for that reason are called **Helper functions**. Helper functions return quickly back into the calling TLM.
Complete API Reference	The toolkit headers live in opt/softplc/c++-toolkit/h. So this is everything the toolkit headers declare. This includes functions not included in the two rows above, plus the two rows above.
Undocumented C/C++ Support	The toolkit Docker image as also includes libraries and headers that are not documented in this document, but are none-the-less very useful and are well documented on the web. These include but are not limited to libxml2, libpoco-dev, libboost-dev and several other standard linux C and C++ libraries.

Chapter 5. TLMs

A TOPDOC Loadable Module (**TLM**) is written in C++ and can be used to extend the SoftPLC runtime. TLMs can implement I/O drivers and/or custom TOPDOC Loadable Instructions (**TLIs**) which are ladder logic extensions.

The following section is for developers familiar with previous versions of the SoftPLC toolkit. If this is your first experience with the toolkit, you may skip past the next section.

5.1. Differences from Version 3.x Toolkit

Read this section only if you are familiar with writing TLMs for version 3.x. There are only a handful of changes in version 4.x from version 3.x:

- `struct request` uses an anonymous union, dropping the "a" member. This means that constructs like `req->a.cmdcode` become `req->cmdcode`.
- `LMENTRY dispatch( void* reqp, void* datap );` becomes
  `ERRCODE dispatch( request* reqp );`
  This means that `datap` is no longer used. The only place where this was previously used was in the command line arguments to the TLM. Those are now available through `reqp->init.cmdLine`
- FUNCTION_TABLE does not need to be public. Instead, you must supply your function table array to the `TLM_MAIN_ENTRY( aFunctionTable )` macro exactly one time. See `tlmExample.c`.
- Interrupts are not supported as part of the Helper Function API. You must write a linux kernel driver to support interrupts.
- There are new thread Helpers, see threads.
- There are additional dispatch command codes, but you do not need to concern yourself with these. They are only used for loading and unloading.
- There are new library load and unload Helpers, see here.
- HlpPrintf() or printf() statements should not start with a '\n' character but should now end with a '\n' character instead. **Unless you follow this your output will look odd and unconventional, and may even not print when you think it should.**
- In version 3.x, unless you defined `NOHELPERS` on the compiler command line, the `splcstdc.h` file would remap stdio functions to helpers. So previously the absence of the `NOHELPERS` define would cause remapping. Now, you must define `REMAP_STDIO` on the compiler command line to get remapping. This is done for you in the template `build.xml` file.
- Instead of using `WORD` and `float` to represent PLC datatable types, use **PLCINT** and **PLCFLOAT** respectively. PLCINT replaces WORD, but PLCINT is signed whereas WORD was not. This is now consistent with the interpretation of SoftPLC's 16 bit integer type, but could lead to sign extension problems that were not present before this change.
- Makefiles are no longer the standard build environment. We have switched to Ant. You can continue to use your own Makefile, but do not ask us to support you on this. Note that `Makefiles` on linux require true 'tab' characters for the indents, spaces will not work, so check your text editor for this capability.

5.2. TLM General Properties

A TLM is a file that contains executable code and data. It gets loaded at startup time by the SoftPLC runtime. The format of a TLM is that of a linux "dynamic shared object" file. TLMs have the file extension of `.tlm.so`. So a valid TLM filename might be `tealware.tlm.so`. There is no 'lib' filename prefix. On Linux filenames are case sensitive. TLMs on SoftPLC version 5.x always use **all lower case** filenames.

A TLM is made known to SoftPLC by a line entry in the `MODULE.LST` file. This file lists all the TLMs to be loaded when the runtime starts. Each entry in the `MODULE.LST` file has to start with either the keyword `DRIVER` or the keyword `MODULE`. The format of `MODULE.LST` is discussed in detail in the upcoming section on MODULE.LST.

A TLM should use printf() for its output, not C++ streams, and not fprintf(). This is because the toolkit swaps printf() calls using a macro to an internal function named HlpPrintf(), and that function is capable of routing the output text to either the console or to the syslog as the runtime determines. Users will see output in the syslog when the runtime runs as a daemon, and on the console when the runtime runs not as a daemon but at the console. By using printf() you will have automatic participation in syslog content when appropriate. Summary: all output intended for user eyes should use printf().

Output going through printf() should end in a newline '\n' character to play nice with the output of other TLMs. The function argument TLM___→Name() should be used to earmark output as being from your TLM. Examples:

printf("%s: started.\n", TLM___→Name());

printf("%s: exiting.\n", TLM___→Name());

Each TLM has this **TLM___** pointer pointing to **its** TLM class instance, and Name() is a member function of that class.

The Name() function will return a space padded string of exactly 8 characters in length. This ensures that all TLMs have that colon in the same column. For this reason, your TLM name should not exceed 8 characters in length. But if you just want attention and want to be a non-conformant, make your TLM name longer. The TLM name comes from a string argument that you pass to the TLM_DECLARE() macro.

TLMs must be written using a cooperative programming strategy. That means you must do your stuff and quickly return from the TLM back into the runtime so that you do not hog or block the main thread. You may however take your time in the TLM::Init() function, which is where you read and parse your configuration file if you have one. Additionally you may create your own service thread or threads in TLM::Init(). By contrast service threads should block and stay in the TLM in a loop. Likewise shut down a service thread in TLM::DeInstall() before you return from there. There is a Helper function for creating a thread. There is the class MSG_BOX<> to queue objects and pass them between threads. MSG_BOX is thread safe.

You should use helper functions for creating threads in case you ever need to port your TLM to another operating system supported by SoftPLC.

5.3. Dispatch Function

All TLMs must define a special C function with the following name and prototype:

TLM Entry Point Function
ERRCODE dispatch(request* reqp);

Function dispatch() is a TLM's main entry point which is called by SoftPLC. There is no main() function in a TLM; dispatch() acts as a main() function for TLMs. However, whereas a main() function within a normal program is called only once by the operating system when a C/C++ program is first started, the dispatch() function will be called many times by SoftPLC at runtime. In turn, dispatch() can call other functions within the TLM to carry out the various activities.

Function dispatch() gets a single argument, a pointer to a RequestPacket union. The request packet itself resides within SoftPLC. It is filled in by SoftPLC just before calling a TLM's dispatch() function. The first field in the request packet is a command code, `int cmdcode` that tells the TLM what action is to be performed. Command specific data will follow the command code; the layout of each variant is given in the RequestPacket union.

The dispatch() function returns an ERRCODE type, and should return a value of SUCCESS to indicate the TLM successfully handled the dispatch, or something other than this to indicate failure.

Each TLM must be written to return from dispatch() as quickly as possible, because the call to dispatch() is made on the same thread as the main ladder thread within SoftPLC.

When servicing the command codes, it is necessary that your TLM return to SoftPLC as quickly as it can. No TLM should hog the CPU or sit in a loop wasting time or waiting for something.

The only exception to the above rule is in the `FNC_INIT` handler. This command code is the first one sent, and it is sent only once and it is before SoftPLC is fully operational. It is allowable to wait for hardware to be initialized, or to open and read from disk files in this command code handler.

From within dispatch(), a switch() is used to branch out to the various command functions based on command code: `reqp→cmdcode`. This way each command code handler may be implemented by a separate C function. Of course this is not the only choice. You can service the various command codes in line within the dispatch() function's switch() statement.

```
ERRCODE dispatch( request* reqp )
{
    ERRCODE ec=SUCCESS;

    int cmdcode = reqp->cmdcode;

    switch( cmdcode )
    {
    case FNC_INIT:
        // add init code here
```

```
        break;
    case FNC_DEINSTALL:
        // add de-install code here
        break;
    case FNC_UNLOAD:
        // add unload code here
        break;
    case FNC_SCAN:
        // add scan code here
        break;
    case FNC_SETMODE:
        // add setmode code here
        break;
    case FNC_CHECKPOINT:
        // add checkpoint code here
        break;
    case FNC_BXFER_SUBMIT:
        // add bxfer code here
        break;
    default:
        ec = ERROR_CMDCODE_NOT_HANDLED;
    }

    return ec;
}
```

5.3.1. Command Codes

With few exceptions, whenever SoftPLC needs to send a command code to the dispatch() function of a TLM, it sends the same command code to all interested TLMs in the same order or sequence as determined by the order the TLMs appear in the `MODULE.LST` file. This sequential calling of all the TLMs is called a **dispatch loop**. Below are the command codes and their purpose.

Command Code	Purpose
FNC_INIT	FNC_INIT is dispatched only once at startup time. This dispatch loop happens just as the TLMs are loaded. At this point, the TLMs have a chance to check and initialize the I/O hardware, install interrupt vectors, decide whether FNC_CHECKPOINT handling is needed (by setting bit 0 of DrvCapability), decide whether FNC_BXFER_SUBMIT handling is needed (by setting bit 1 of DrvCapability), and report errors indicating whether the system should be started up or not. If any TLM returns other than SUCCESS, the system will not start up.

Command Code	Purpose
FNC_SCAN	FNC_SCAN is dispatched repeatedly when SoftPLC is in RUN mode. This is where SoftPLC and the TLMs execute all the control logic for the application. During this dispatch loop each TLM that is defined as a DRIVER in the MODULE.LST file is called with the FNC_SCAN command code. **Those TLMs that are defined as MODULEs rather than DRIVERs are not called with this command code. If a TLM is implementing Ladder Instructions (TLIs) and not intending to be an I/O driver, then it may be declared as a MODULE in MODULE.LST rather than as a DRIVER.**
FNC_CHECKPOINT	FNC_CHECKPOINT may be sent several times during a program scan depending on the size of the ladder program that SoftPLC is executing. During this dispatch loop, TLMs capable of servicing FNC_CHECKPOINT requests are called. A FNC_CHECKPOINT request is sent to the interested TLMs only if SoftPLC is in one of RUN, REM_RUN, TEST, or REM_TEST modes that perform ladder program execution. Generally it is like the FNC_SCAN dispatch loop, but rather than occurring only once per scan, it can occur several times per scan. It is a way for an I/O driver to get control more frequently if it needs to check hardware that often.
FNC_SETMODE	FNC_SETMODE is dispatched whenever SoftPLC detects a change in operating mode. A TLM should record the new mode and change its operating conditions accordingly. **Most importantly, OUTPUTS should be turned off in this command code handler if the new mode is not OPM_RUN or OPM_REM_RUN.**
FNC_BXFER_SUBMIT	FNC_BXFER_SUBMIT is dispatched when SoftPLC executes a Block Transfer ladder instruction. During this dispatch loop, TLMs capable of handling block transfers are called with the FNC_BXFER_SUBMIT request. Currently block transfer is only supported by the AB KTx driver.
FNC_DEINSTALL	FNC_DEINSTALL is dispatched whenever SoftPLC wishes to unload a TLM. The TLM is to release any resources it owns, such as open files, interrupt vectors, dynamically allocated memory, etc. The TLM will not be used after this dispatch loop and may be unloaded.

5.4. Helper Functions

Helper Functions reside within SoftPLC itself, and are callable from within TLMs. Each Helper Function has its function prototype in the include file `tlm.h`.

Every TLM will include the header file tlm.h.

TLMs may use the Helper Functions listed here. If you prefer, you can remap certain C functions into Helper Functions. For example the include file `splcstdc.h` has lines like this:

```
#define malloc HlpMemAlloc
```

so your source code can use malloc(), yet the compiled code will actually be using HlpMemAlloc().

The C preprocessor replaces all occurrences of malloc() with HlpMemAlloc() so you are actually calling HlpMemAlloc() instead of malloc(). This mechanism is not mandatory, but is available to you by doing this:

1. including `tlm.h`, which includes `splcstdc.h` and
2. defining REMAP_STDIO on the compiler command line, something which `build.xml` does as a default

Note that if REMAP_STDIO is not defined and your source code uses the stdio functions, then the remapping is not performed and you end up using the genuine C runtime library versions of the stdio functions.

5.4.1. printf() Redirection

The implementation of printf() within SoftPLC is special. Actually, this refers to the helper function HlpPrintf(), not printf(). You will be using HlpPrintf() only if you have explicitly coded to HlpPrintf(), or you have coded to printf() and the REMAP_STDIO feature is enabled. HlpPrintf() switches between two modes depending on whether SoftPLC is running as a daemon or from the command line. When running as a daemon, the output of HlpPrintf() is directed to the syslog. When running from the command line, SoftPLC's HlpPrintf() will show up on the console.

You can view the most recent syslog contents by running the following command from a SoftPLC console:

```
# logread
```

The genuine C runtime library implementation of printf() does not have this syslog output capability, only HlpPrintf() does.

5.5. TLIs

TLI is an acronym for TOPDOC Loadable Instruction. You can add your own instructions to the SoftPLC control program instruction set. A TLM can contain one or more TLIs and even additional code for a driver. TLIs are called by SoftPLC through a mechanism similar to the dispatch loop. Each TLI is like its own little dispatch loop. To call your TLIs, SoftPLC needs to know the names and locations of each TLI in every module.

5.5.1. FUNCDEF

As we have seen, all modules require the presence of the function `dispatch()`. Similarly, for any TLM implementing a TLI, there is a second mandatory structure that you declare with the macro **FUNCDEF**.

The FUNCDEF lists the names, descriptions, entry point addresses, number and types of arguments for each TLI in the TLM. Given this FUNCDEF, SoftPLC has everything it needs to know about TLIs and can call them directly at run time. TOPDOC also consults the FUNCDEF to display TLIs with correct function names, descriptions, number and name of arguments.

For TLIs, execution does not pass through the dispatch() function, but rather goes directly from

SoftPLC to the TLI being called from the ladder program. The ladder program contains the function name of the TLI. SoftPLC searches through the TLMs listed in the MODULE.LST file once at start up time. After that SoftPLC can call TLIs with virtually no scantime overhead.

The data type of the FUNCDEF is found in the header file `tlm.h`. It is an array of type `funcdef` and is reproduced below:

```
///  TLI definition
typedef struct funcdef {
    char        fnc_name[MAXFNCNAMELEN+1];      ///< 0   Function name
    char        fnc_desc[MAXDESCLEN+1];         ///< 16  Function description
    TLIPTR      fnc_ptr;                        ///< 48  Pointer to function
    UBYTE       fnc_numoptionalargs;            ///< 52  Number of optional args
    UBYTE       fnc_type;                       ///< 53  Output or permissive
    USHORT      fnc_numargs;                    ///< 54  Number of arguments
    PARMDEF     p[MAXPARMNUM];                  ///< 56  Array of parameters
} funcdef;                                      ///< 172 = 56 + 9 x 16
#define FUNCDEF funcdef __attribute__ ((section ("TLMDEF")))
```

fnc_name

Contains the name of the function. It has to be all upper-case with NO embedded spaces. Maximum length is MAXFNCNAMELEN characters, plus a terminating 0.

fnc_desc

Contains a description of the function. It can be mixed case. Maximum length is MAXDESCLEN characters, plus a terminating 0.

fnc_ptr

Contains the address of the TLI which will be called from SoftPLC when the ladder diagram execution triggers it.

fnc_numoptionalargs

Contains any optional args. This number indicates how many of the parameters, indicated by `fnc_numargs`, may be omitted when entering the instruction with TOPDOC. The omitted parameters may be omitted from the end of the parameter list at time of instruction entry. A value of zero means there are no optional parameters, all are mandatory.

fnc_type

Is ignored by SoftPLC. This field is used only by TOPDOC to decipher the type of instruction as TLI_OUTPUT or TLI_PERMISSIVE. This determines how to display the instruction in the ladder editor. fnc_type can take one or the other of the following values:

Value	Meaning
TLI_OUTPUT	Indicates that the TLI is an output instruction. This means that the instruction can not change the rung state. Output Energize (OTE) is an example of a built-in output instruction for SoftPLC.

Value	Meaning
TLI_PERMISSIVE	Indicates that the TLI is a permissive instruction. This means that the instruction can change the rung state. If the return value of a permissive TLI is non-zero, the instruction evaluates to TRUE condition. As a result the execution is passed to the instructions to the right of it on the rung. Examine Input Closed (XIC) is an example of a permissive instruction. If the return value is zero (FALSE), then the rung power flow is cut off at this instruction.

fnc_numargs

Contains the number of parameters that the instruction uses, 0 - 9. If `fnc_numoptionalargs` is not zero, then `fnc_numargs` represents the maximum number of parameters that the instruction may take.

```
maximum number of parameters = fnc_numargs
minimum number of parameters = fnc_numargs - fnc_numoptionalargs
```

5.5.2. PARMDEF

The nature of each parameter passed to a TLI at runtime is determined ahead of time by a corresponding **PARMDEF** structure. All the PARMDEF structures for a given TLI are provided as an array within the FUNCDEF, seen above. A single PARMDEF structure is defined below:

```
///  TLI function parameter
typedef struct parm {
    char        prm_name[MAXPARMNAMELEN+1];     ///<  0  Argument name
    USHORT      prm_argtype;                    ///<  12 Argument type
    USHORT      prm_len;                        ///<  14 Argument length
} PARMDEF;                                      ///<  16
```

prm_name

Can be up to MAXPARMNAMELEN characters in length plus a terminating 0. `prm_name` is used by TOPDOC to display a meaningful name for each parameter of the TLI.

prm_argtype

Allows you to tell SoftPLC how to pass data to the TLI and the direction of the data flow. Data for any given parameter may be passed as an integer value, a floating-point value, a timer, a counter, a string, a control, or an array (which we call block). This field also tells TOPDOC what to allow at time of instruction editing/entry. This field takes bits from the following two categories ORed together:

- Data Types Flags
- Directional Flags

Data Type Flags	Meaning
TLI_ARG_INT	Integer argument
TLI_ARG_FLT	Float argument
TLI_ARG_BLOCK	Indicates an array, and must be combined with either TLI_ARG_INT, TLI_ARG_FLOAT, or TLI_ARG_STRING. Same as TLI_ARG_FILE, which is now deprecated.
TLI_ARG_TIMER	Timer structure
TLI_ARG_COUNTER	Counter structure
TLI_ARG_CONTROL	Control structure
TLI_ARG_STRING	String element

Directional Flags	Meaning
TLI_ARG_IN	Associated parameter is for reading
TLI_ARG_OUT	Associated parameter is for writing

One or both of the Directional Flags should be OR'ed with one or more of the Data Type Flags and placed into `prm_argtype` to indicate the type of each parameter to both SoftPLC and TOPDOC. A parameter can be an input to your TLI, an output of your TLI, or bidirectional. Bidirectional means that a value can be passed from the ladder program to the TLI and the TLI returns a value in the same parameter location back to the ladder program.

For example, if you want to make one of the parameters to a TLI an input word that can take only integer values, then the `prm_argtype` should be set to

```
TLI_ARG_IN | TLI_ARG_INT        // input param that is an integer value
```

Similarly, if one of the parameters of a TLI is an output of that TLI and it is a pointer to the start of an array of floating-point locations in SoftPLC's datatable, the `prm_argtype` for that parameter should be set to

```
TLI_ARG_OUT | TLI_ARG_FLT | TLI_ARG_BLOCK    // output param of float array
```

prm_len

Is the number of elements that will be accessed within the datatable starting at the address entered with the instruction into the ladder program. **Set this to 1 except when you have a TLI_ARG_BLOCK Data Type Flag ORed into the** `prm_argtype`, in which case you can set it from 1-10000. TOPDOC uses this field to create datatable memory at time of instruction entry or modification. With TLI_ARG_BLOCK, set it to the largest, i.e. worst case size, array that you will be needing, otherwise you cannot be sure all the required datatable memory will exist at runtime.

5.5.3. TLI Function Arguments

At first glance, 9 parameters may not seem to be enough to implement certain functions. However, any of these parameters can be the start of a data block. Your software can then use a number of consecutive item locations within the data block starting at the supplied parameter. In effect, a TLI can work with hundreds of parameters.

The calling convention of TLIs is slightly different than that of the dispatch() function. It is very similar to the convention used to call the main() function in a C program, which uses the well known argc and argv arguments.

All TLIs will have an identical function prototype as shown below:

```
/// function type for defining a TLI ladder instruction
typedef BOOL (SPLCUSR * TLIPTR)( int rungState, int argc, TLIPARAM* argv );
```

rungState

Tells the TLI whether the rung logic preceding the TLI evaluated to TRUE or FALSE. You can take different action in your C function for the TLI based on the rung state.

argc

Gives the number of parameters passed, 0 - 9. This should agree with the value you placed into the funcdef's `fnc_numargs` field. It is present as a safety precaution to confirm that SoftPLC's understanding of the function agrees with its implementation.

argv

Is an array of **TLIPARAM** elements where each points to datatable items within SoftPLC's datatable which are interpreted as function parameters. SoftPLC can pass parameters to the TLI either by value or by reference. Below is the definition of TLIPARAM structure that is filled for each parameter by SoftPLC before calling a TLI.

```
struct TLIPARAM
{
    union {
        PLCINT      wd;     ///<  Word value
        float       fl;     ///<  Float value
        ULONG       hls;    ///<  LS part of handle
    } val;
    union {
        PLCINT*     wptr;   ///<  Word pointer
        float*      fptr;   ///<  Float pointer
        TIMER*      ptmr;   ///<  Timer block
        COUNTER*    pcnt;   ///<  Counter block
        CONTROL*    pctl;   ///<  Control block
        ULONG       hms;    ///<  MS part of handle
    } ptr;
    BOOL isfloat;       ///<  TRUE if the parameter is float, FALSE if PLCINT
```

```
};
```

The union named `val` is used if SoftPLC is passing a parameter to the TLI **by value**. This is useful for cases where numeric constants such as 3.14159 or single word parameters with a Directional Flag of only TLI_ARG_IN are being passed to the TLI.

The other way a parameter can be passed to a TLI is **by pointer**. When this happens the union named `ptr` is used and the corresponding pointer field within the `ptr` union will point to the parameter within the datatable. This allows the TLI to access multiple values by doing pointer arithmetic on a single pointer. A parameter is passed by pointer when either of these two criteria are met:

- If SoftPLC is passing a parameter whose `PARMDEF prm_argtype` flags include any Data Type Flag other than TLI_INT or TLI_FLOAT. That is, all but simple INT or FLOAT values are passed by pointer.
- If any parameter's Directional Flags include TLI_ARG_OUT. That is, anything that must be modified in the datatable must be passed by pointer.

5.5.4. TLI Return Value

If the TLI wants to pass logically TRUE rung power flow to the rest of the ladder rung following itself, it should return TRUE, else FALSE.

5.5.5. Example TLIs

Below is sample source fragment for 2 TLIs, named **CIRCLEAREA** and **ARRAY_AVERAGE**. The CIRCLEAREA TLI takes 1 input parameter which is the radius of a circle (as **either** an INT or FLOAT) and 1 output parameter which is set to the area of the circle as a FLOAT when the TLI is executed. The ARRAY_AVERAGE TLI takes a block of FLOAT or INT values and averages them. The length of the block is variable up to 100 elements max, although this could be extended to 10000.

```
#include <tlm.h>
#include <math.h>       // defines "PI" as M_PI

// forward function declarations
BOOL SPLCUSR CircleArea( int rungState, int argc, TLIPARAM* params );
BOOL SPLCUSR ArrayAve( int rungState, int argc, TLIPARAM* params );

FUNCDEF function_table [] = {
{   "CIRCLEAREA",               // Function name
    "Calculate area of circle", // Description
    CircleArea,                 // Pointer to function
    0,                          // no. optional params.
    TLI_OUTPUT,
    2,                          // 2 params: "Radius" and "Area"
    // Parameter array
    {   {
            "Radius:",
```

```
            TLI_ARG_IN | TLI_ARG_FLT | TLI_ARG_INT,
            1,                  // prm_len
        },
        {
            "Area:",
            TLI_ARG_OUT | TLI_ARG_FLT,
            1,
        },
    }
},

#define MAX_ARRAY_SIZE      100

{   "ARRAY_AVERAGE",
    "Average array, up to 100 words",
    ArrayAve,
    0,                          // no. optional params
    TLI_OUTPUT,
    3,
    {   {
            "Array:",
            // handle either INT or FLOAT arrays:
            TLI_ARG_IN | TLI_ARG_FILE | TLI_ARG_INT | TLI_ARG_BLOCK,
            MAX_ARRAY_SIZE,     // worst case allowed size
        },
        {
            "Count:",
            TLI_ARG_IN | TLI_ARG_INT,
            1,
        },
        {
            "Ave:",
            TLI_ARG_OUT | TLI_ARG_FLT,
            1,
        },
    }
},
{ 0 },  // *Required* end-of-table indicator

};      // FUNCDEF end


BOOL SPLCUSR CircleArea( int rungState, int argc, TLIPARAM* params )
{
    // Only calculate if Rung is true.
    if( rungState )
    {
        float radius;

        /* get float radius by value from parameter 0.  Since we allowed either
           INT or FLOAT for this parameter, we have to expect either:
```

```
        */
        if( params[0].isfloat )
            radius = params[0].val.fl;
        else
            radius = params[0].val.wd;

        // output the area of the circle in parameter 1
        *params[1].ptr.fptr = M_PI * radius * radius;
    }

    return TRUE;
}


/* Calculate the average of the array */
BOOL SPLCUSR ArrayAve( int rungState, int argc, TLIPARAM* params )
{
    // Only calculate if Rung is true.
    if( rungState )
    {
        // the CPU works best with natural (32 bit) "int" type, so grab the
        // "Count:" param into an int, not into a PLCINT:
        int elemCount = params[1].val.wd;    // assuredly an INT, see prm_argtype
above

        if( elemCount <= 0 )
        {
            *params[2].ptr.fptr = 0.0;
            return FALSE;       // stop powerflow to indicate failure
        }

        if( elemCount > MAX_ARRAY_SIZE )
             elemCount = MAX_ARRAY_SIZE;

        double total = 0.0;  // use double as intermediate, not float

        /* Since we allowed either INT or FLOAT for Array: parameter,
           we have to expect either:
        */
        if( params[0].isfloat )
        {
            int i;
            for( i=0;  i<elemCount;  ++i )
                total += params[0].ptr.fptr[i];
        }
        else
        {
            int i;
            for( i=0;  i<elemCount;  ++i )
                total += params[0].ptr.wptr[i];
        }
```

```
        // divide the total by the number of elements, save to param 2's float
        *params[2].ptr.fptr = (PLCFLOAT) (total / elemCount);
    }

    return TRUE;
}
```

5.6. Debugging

To debug your TLM add printf() statements that are conditionally compiled in. Once the TLM code is debugged, remove the extraneous printf() statements.

5.7. Building the Template TLM

In this section we go through the steps to build the template TLM. You should have first read and understood the material under Toolkit Basics.

Here are the steps to create a new project for this example:

```
1    C:>cd \splcwork
2    C:>md mytlm
3    C:>cd mytlm
4    C:>\SoftPLCToolkit\setenv
5    C:>copy \SoftPLCToolkit\templates\build.*  .
6    C:>edit build.properties
7    C:>ant tlm
```

In step 6 we set the TLM.ROOT.NAME to **mytlm**. In step 7 the template code is compiled, linked, and downloaded to the TARGET.IP SoftPLC. In a real project, you would edit the `tlm.c` before performing step 7 above.

5.8. Adding a TLM to MODULE.LST

Within your SoftPLC, you may edit `/SoftPLC/tlm/MODULE.LST` in either of two ways:

1. By using the `edit` command from a SoftPLC console.
2. By using TOPDOC NexGen's ***PLC*** | ***Modules*** editor. In order to get this option to work the first time you have to tell TOPDOC about your TLM by cloning one of the files in your TOPDOC installation on your development system found here: `\SoftPLC\plc\SAMPLE.DEF`. Copy this file to a new name with extension ".DEF", in the same directory `\SoftPLC\plc\` and then edit it according to the comments in the file. Then restart TOPDOC. Then go back into TOPDOC ***PLC*** | ***Modules*** and you should see your TLM listed, select its ***Use*** column. Then ***Send*** and then ***Save***.

Each time after downloading a revision of your TLM, you will have to restart SoftPLC. This is most easily done using PUTTY.EXE from your Windows box.

SoftPLC does a remapping of the text found in `/SoftPLC/tlm/MODULE.LST`. It does this so it can support either a version 3.x or 4.x system with the same MODULE.LST file. The remapping is as follows. If the system is a 4.x SoftPLC, then any TLM name ending in TLM and consisting of **all** upper case characters, is converted to lowercase and its extension is changed from ".tlm" to ".tlm.so".

So on version 4.x, if you have the following line:
`MODULE=/SoftPLC/tlm/SOMETHING.TLM`
then according to the remapping algorithm, this is equivalent to listing:
`MODULE=/SoftPLC/tlm/something.tlm.so`
As you make your edits to your cloned `\SoftPLC\plc\SAMPLE.DEF` file you should use the uppercase shorter synonym, not the longer **all lowercase name that is required of all 4.x TLMs** as they actually exist on disk. Eventually this remapping business will fade away as the version 3.x systems get replaced with 4.x systems.

5.9. Example TLMs

Each example below is a complete, buildable TLM of just a few files, published untarred so you can read it in your browser before taking it. Follow a link, then save each file from the directory listing into a new source directory of your own; build it exactly as described in Building a TLM.

Table 1. Example TLMs published at https://softplc.com/usermanuals/SoftPLC-C++-Toolkit/examples/

Example	What it shows	Link
`service_thread`	A TLM that starts its own service thread and communicates with the scan through a message box — the pattern to follow whenever work must proceed independently of the PLC scan.	service_thread
`canbus`	Use of the SoftPLC CANBUS API from a TLM.	canbus
`tdk`	A real device driver: data processing for the TDK EZA2500-32048 DC-DC converter. Shows how a driver-style TLM is organized across a `.cc` and a `.h`.	tdk

The starting-point files a new TLM is cloned from are published the same way, at https://softplc.com/usermanuals/SoftPLC-C++-Toolkit/templates/.

Chapter 6. SoftPLC Toolkit API Reference

6.1. The Modern TLM C++ Entrypoint Interface: Class TLM

Topdoc Loadable Modules (TLMs) must be callable from the SoftPLC runtime, and modern TLMs make this happen by declaring and implementing an instance of class TLM.

The declaration is done by using macro DECLARE_TLM() once at the top of the main source file. The implementation is done by coding a few class member functions.

Classes

TLM

Table 2. Variables

Name	Type	Description
TLM___	TLM *const	This is a global ptr to the TLM instance in each TLM. It is automatically declared as part of the macro DECLARE_TLM().

6.1.1. DECLARE_TLM()

```
#define DECLARE_TLM(aClassName, aName, aFunctionTable) aClassName M___( aName,
aFunctionTable ); TLM* const TLM___ = &M___; HLPEnv* SoftPLC___; ERRCODE DLLEXPORT
SPLCUSR mainEntry4( request* req ) {     return M___.Dispatch( req ); }
```

Macro DECLARE_TLM is used to establish the entrypoint into a C++ TLM.

It is placed once at the top of your main source file for each TLM.

Parameters

aClassName	is the class name and should be simply TLM unless you derived class based on TLM.
aName	is an upper case string less than or equal to 8 characters that will be used in console or syslog messages for this TLM.
aFunctionTable	is the FUNCDEF array or NULL.

6.1.2. TLM

```
#include <tlm.h>
```

```
class TLM
```

Class TLM presents the C++ interface of a Topdoc Loadable Module (TLM or module).

It is the doorway into the TLM that the runtime uses to call it and ask it do do things. Most of the functions are intended as entry point functions that the runtime will call as it goes through its various states. But there are a few informational functions that the TLM itself can call that are helpful in producing messages or changes in behavior. What makes it easy to use is that you do not have to implement all the functions in this interface, only the ones that you feel should operate in a non-default manor. If you do not implement a function, then a default one will come in from the ${STATELOGIC_LIBRARY}. If there is additional instance data that you want to include, then that may be a reason to derive an extended C++ class from this one, but this is often not necessary. If the runtime loads a module as MODULE= from the MODULE.LST file, it will never call the Scan() function below. If the runtime loads a module as DRIVER= from the MODULE.LST file, it will call the Scan() function below.

Public Constructors	
	`TLM(const char *, funcdef *)` Constructor TLM(const char* aName, funcdef* aFunctionTable)
Public Destructors	
	`~TLM()`
Public Methods	
void	`Init(req_init *)` Function Init() is called right after the driver is loaded.
void	`Scan()` Function Scan() is called only if the module is loaded as a result of DRIVER= in the MODULE.LST file.
void	`FirstScan()` Function FirstScan() is called once for each transition to a RUN mode from a non-RUN mode and only if the module was loaded as DRIVER= in MODULE.LST.
void	`Idle()` Function Idle() is called in lieu of Scan() when the runtime is in PROGRAM or FAULTED mode.
void	`ModeChange(int, PLCINT *)` Function ModeChange() is called from the runtime engine on every mode change.
void	`DeInstall()` Function DeInstall is called exactly once, just before the runtime process exits.

Public Constructors	
void	`CheckPoint(PLCINT *)` Function CheckPoint() is called every 400 rungs or so, and at the end of ladder program, but only if you register that this is wanted.
void	`BlockTransfer(req_bxfer *)` Function BlockTransfer() should process a BTW or BTR ladder instruction.
const char *	`Name() const` Function Name returns the name of the TLM padded up to 8 characters for printing into the SoftPLC log or on screen.
const char *	`CmdLine() const` Function CmdLine returns the command line which was passed into req_init::cmdLine.
const char *	`TlmPathAndName() const` Function TlmPathAndName returns a string like like argv[0] in a C program, it contains the full path of this *.tlm.so name.
std::string	`MakeFilename(const char *) const` Function MakeFilename returns a full path and filename which is created by combining the path that this TLM was loaded from, along with *aBaseFilename*.
int	`MaxRacks() const` Function MaxRacks returns the maximum Input or Output capacity of the host SoftPLC as determined by its copy protection device, where the maximum capacity is expressed in (max allowed I or O file size in words)/8, note that the maximum size may be greater than the current size.
int	`Mode() const` Function Mode() returns the current operating mode of this TLM.
void	`ShowRegistry() const` Function ShowRegistry dumps the DT_OBJECTs that have been instantiated in this TLM with their types and specs.
int	`ResolveAndVerify()` Function ResolveAndVerify resolves (sets the pointer to datatable memory inside of each DT_OBJECT) and verifies that that DT_OBJECT specific region of datatable memory is sufficiently sized to meet its needs.

Members

TLM

```
TLM(const char * aName,
```

```
    funcdef * aFunctionTable = NULL)
```

Constructor TLM(const char* aName, funcdef* aFunctionTable)

Parameters

aName	should be an uppercase name not exceeding 8 characters in length
aFunctionTable	is an array of TLIs or NULL or 0 (**Default value**: NULL)

~TLM

```
~TLM()
```

Init

```
void Init(req_init * req)
```

Function Init() is called right after the driver is loaded.

If you don't provide this function a default will be provided for you. This is where you should read configuration files from disk and verify that the runtime environment is ready for your usage of it such as opening any communications ports.

You can throw an ERROR exception if want to indicate an error condition. When formulating the text of the ERROR, you need not supply the name of the TLM, as that will be provided by the dispatcher, and you need not supply a terminating \n. For example:

```
throw ERROR( E_MYTLM_OPEN_COMM_PORT,
     "%s while opening port %d",
     open_result.c_str(), com_port
     );
```

The SoftPLC runtime treats ERRORs thrown from any driver's TLM::Init() as "reboot required". The normal way to recover from this state is with a reboot.

However with the "double clutching" capability of most drivers, which is a transition from OPM_REM_PROG to OPM_REM_PROG initiated by NexGen, most drivers can recover from a configuration file error by reloading a modified file. This assumes a user takes the time to edit the config file and send it before initiating the double clutching. The philosophy of each driver's implementation is potentially variable with respect to its desire to avoid the "reboot required" state. Note that a driver avoids it by never throwing ERROR from TLM::Init(). However in a situation where double clutching is not performed and we have returned success from TLM::Init() as a lie, it

is necessary to remember within each driver whether TLM::Init() actually succeeded since the driver would have reported success even if it failed. By remembering, we can then use a driver local TLM::Init() failed status to deny a transition to a run mode later in TLM::ModeChange(). If a double clutch is performed with NexGen and bugs in a newly loaded configuration file go away, the driver local StartUpError should then be cleared so as to allow a transition to a run mode later.

Parameters

`req`

Throws

ERROR — should be thrown by you when indicating the TLM is not ready to continue with startup, say for example a configuration file is not available or a comm port could not be accessed. For recoverable errors it is ok to return without throwing ERROR so long as you deny a transition to a RUN mode later, at least until double clutching is used to clear up the error.

Scan

```
void Scan()
```

Function Scan() is called only if the module is loaded as a result of DRIVER= in the MODULE.LST file.

If instead the module is loaded as a result of MODULE= in the MODULE.LST file, then neither this function nor FirstScan() will be called at all. In the former caes, the runtime will call Scan() when it is in a RUN mode (i.e. when Mode() >= OPM_TEST) for each scan of the main thread (except the first RUN mode scan where FirstScan() is called instead of Scan()). This function is not called when the runtime is in OPM_FAULTED, OPM_PROG, or OPM_REM_PROG modes.

Throws

ERROR — You may throw an ERROR in here, and if so this will cause SoftPLC to enter OPM_FAULT mode exiting OPM_REM_RUN mode and a ModeChange() will be issued during that transition.

FirstScan

```
void FirstScan()
```

Function FirstScan() is called once for each transition to a RUN mode from a non-RUN mode and only if the module was loaded as DRIVER= in MODULE.LST.

You may throw an ERROR in here, and if so this will cause SoftPLC to enter OPM_FAULT mode exiting OPM_REM_RUN mode and a ModeChange() will be issued during that transition.

If you don't provide this function a default will be provided for you.

Throws

ERROR — You may throw an ERROR in here, and if so this will cause SoftPLC to enter OPM_FAULT mode exiting OPM_REM_RUN mode and a ModeChange() will be issued during that transition.

Idle

```
void Idle()
```

Function Idle() is called in lieu of Scan() when the runtime is in PROGRAM or FAULTED mode.

All the DT_OBJECT's are not guaranteed to be Resolved at the time this function is called. Segfaults can arise if you don't resolve them before using them. It should be possible to bleed off comports in these two modes however.

Throws

ERROR — You may throw an ERROR in here, and if so this will cause SoftPLC to enter OPM_FAULT mode exiting OPM_REM_RUN mode and a ModeChange() will be issued during that transition.

ModeChange

```
void ModeChange(int aNewMode,
                PLCINT * stsFile)
```

Function ModeChange() is called from the runtime engine on every mode change.

You may deny a requested mode change to OPM_REM_RUN or OPM_RUN by throwing an ERROR.

If you do not supply this function, a default implementation is provided for you by the state_logic.a library.

Parameters

`aNewMode`

`stsFile`

Throws

ERROR — You may throw an ERROR in here to deny the transition and stay in the current mode.

DeInstall

```
void DeInstall()
```

Function DeInstall is called exactly once, just before the runtime process exits.

In this function you would close any open ports and de-install interrupt handlers.

If you do not supply this function, a default implementation will come from the state_logic.a library.

CheckPoint

```
void CheckPoint(PLCINT * stsFile)
```

Function CheckPoint() is called every 400 rungs or so, and at the end of ladder program, but only if you register that this is wanted.

That registration can be made in Init() by setting a bit in Init()'s req_init structure. If you do not register this wish, then this function will never be called within a particular TLM.

See

req_init

Parameters

`stsFile`	pointer to the datatable STATUS file, which is file 2.

BlockTransfer

```
void BlockTransfer(req_bxfer * req)
```

Function BlockTransfer() should process a BTW or BTR ladder instruction.

This is only called if you register as wanting it called in Init(). This is used by the RIO master TLM to implement BTW and BTR.

See

req_init

Parameters

`req`

Name

```
const char * Name() const
```

Function Name returns the name of the TLM padded up to 8 characters for printing into the SoftPLC log or on screen.

CmdLine

```
const char * CmdLine() const
```

Function CmdLine returns the command line which was passed into req_init::cmdLine.

TlmPathAndName

```
const char * TlmPathAndName() const
```

Function TlmPathAndName returns a string like like argv[0] in a C program, it contains the full path of this *.tlm.so name.

For example `/SoftPLC/tlm/tealware.tlm.so`

MakeFilename

```
std::string MakeFilename(const char * aBaseFilename) const
```

Function MakeFilename returns a full path and filename which is created by combining the path that this TLM was loaded from, along with *aBaseFilename*.

Parameters

`aBaseFilename`	is a filename without a path, such as HILSCHER.LST and is typically the name of an uppercase configuration file which is expected to reside in the same directory as this TLM.

Returns

std::string - the path portion from TlmPathAndName() with *aBaseFilename* on the end instead of the tlm's filename.

MaxRacks

```
int MaxRacks() const
```

Function MaxRacks returns the maximum Input or Output capacity of the host SoftPLC as determined by its copy protection device, where the maximum capacity is expressed in (max allowed I or O file size in words)/8, note that the maximum size may be greater than the current size.

Mode

```
int Mode() const
```

Function Mode() returns the current operating mode of this TLM.

Returns

int - one of the OPM_* constants given in tlm.h

ShowRegistry

```
void ShowRegistry() const
```

Function ShowRegistry dumps the DT_OBJECTs that have been instantiated in this TLM with their types and specs.

ResolveAndVerify

```
int ResolveAndVerify()
```

Function ResolveAndVerify resolves (sets the pointer to datatable memory inside of each DT_OBJECT) and verifies that that DT_OBJECT specific region of datatable memory is sufficiently sized to meet its needs.

By default, this function is called before entering a RUN mode and not before that. Therefore it is not legal typically to read or write to DT_OBJECTs until you are in a RUN mode. In a RUN mode, datatable memory may not be moved or resized. In a PROGRAM or FAULTED mode, Topdoc NexGen may be resizing or moving datatable memory. So if you must access a DT_OBJECT in a PROGRAM or FAULTED mode, then you should call this function just before accessing any DT_OBJECT. Sometimes it is useful to do this in the TLM::Init() function, which is always called with the system in a non-running mode. If you are not going to access DT_OBJECT variables in a non RUN mode, then there is no reason to ever call this function from within your TLM. Calls to this function will be

automatically done for you as needed (except for the case where you must access a DT_OBJECT in PROGRAM mode.)

Returns

int - the number of errors found due to missing tags or undersized datatable. If this is not zero, then it is unsafe to proceed, and the runtime will have logged all the errors to the console or syslog.

6.2. Datatable Types

These are some of the types found in SoftPLC datatable files.

Classes

COUNTER

TIMER

CONTROL

STRING

6.2.1. enum DT_T

Enum DT_T is the set of datatable file types, and is used in helper HlpDtReadFileTypeSize()'s section argument.

Value	Init	Description
SFT_OUTPUT	``	0
SFT_INPUT	``	1
SFT_STATUS	``	2
SFT_BIT	``	3
SFT_TIMER	``	4
SFT_COUNTER	``	5
SFT_CONTROL	``	6
SFT_INTEGER	``	7
SFT_FLOAT	``	8
SFT_ASCII	``	9
SFT_BCD	``	10
SFT_STRING	``	11
SFT_BLOCKXFER	``	12
SFT_PID	``	13

Value	Init	Description
SFT_MESSAGE	``	14
SFT_SFC_STATUS	``	15
SFT_FUNCTIONX	= 19	
SFT_PROGRAM	= 20	
SFT_FUNCTION	= 22	
SFT_UNTYPED	= 24	
SFT_IOCFG	= 25	

Table 3. Typedefs

Name	Definition	Description
PLCINT	typedef int16_t PLCINT	16 bit signed integer
PLCFLOAT	typedef float PLCFLOAT	32 bit ieee float
WP	typedef PLCINT* WP	pointer to PLCINT

Table 4. Constants

Name	Value	Description
STS_MODE_RUN	(1<<1)	SoftPLC mode indicator bits in status file (S2), word 1. run or remote run mode
STS_MODE_TEST	(1<<2)	test or remote test mode
STS_MODE_PGM	(1<<3)	program or remote program mode
STS_BACKUP	(1<<4)	operating mode is a BACKUP version of REM_RUN or REM_TEST
STS_MODE_REM	(1<<7)	remote keyswitch position
STS_FIRST_SCAN	(1<<15)	true only during first scan after entering run mode
STS_MODE_MASK	(~(	

6.2.2. COUNTER

```
#include <tlm.h>

struct COUNTER
```

struct COUNTER is the data storage format for a SoftPLC counter instruction

Use with type TLI_ARG_COUNTER

Public Variables	
PLCINT	`ctl` control word
PLCINT	`pre` preset word of COUNTER
PLCINT	`acc` accumulated word of COUNTER

6.2.3. TIMER

```
#include <tlm.h>

struct TIMER
```

struct TIMER is the data storage format for a SoftPLC timer instruction

Use with type TLI_ARG_TIMER

Public Variables	
PLCINT	`ctl` control word
PLCINT	`pre` preset word of TIMER
PLCINT	`acc` accumulated word of TIMER

6.2.4. CONTROL

```
#include <tlm.h>

struct CONTROL
```

struct CONTROL is the data storage format for a SoftPLC control object

Use with type TLI_ARG_CONTROL

Public Variables	
PLCINT	`ctl` control word
PLCINT	`len` length word of CONTROL
PLCINT	`pos` position word of CONTROL

6.2.5. STRING

```
#include <tlm.h>

struct STRING
```

Use with type TLI_ARG_STRING.

Public Variables	
PLCINT	`length` no. unicode characters, 0 - DT_MAX_STRING_LEN
uint16_t	`string` unicode string

6.3. SoftPLC Operating Modes

These defines represent each operating mode that SoftPLC can be in at any given moment.

When SoftPLC changes mode, due to a request by TOPDOC NexGen or by a TLM, then SoftPLC will inform each TLM of the requested mode change by calling its entrypoint and passing in the requested new mode.

Table 5. Constants

Name	Value	Description
`OPM_FAULTED`	`(-1)`	fault mode
`OPM_REM_PROG`	`0`	remote program mode, keyswitch in REMOTE
`OPM_PROG`	`1`	program mode, keyswitch not in REMOTE
`OPM_TEST`	`2`	test mode, keyswitch not in REMOTE
`OPM_REM_TEST`	`3`	remote test mode, keyswitch in REMOTE
`OPM_RUN`	`4`	run mode, keyswitch not in REMOTE

Name	Value	Description
OPM_REM_RUN	5	remote run mode, keyswitch in REMOTE
OPM_BACKUP_TEST	6	remote test mode, except informed TLMs do not read inputs
OPM_BACKUP_RUN	7	remote run mode, except informed TLMs do not control I/O

6.4. TLI Support

Some structures used in implementing TLIs (TOPDOC Loadable Instructions), which are customer ladder instructions for SoftPLC.

Classes

PARMDEF

funcdef

TLIPARAM

Table 6. Typedefs

Name	Definition	Description
TLIPTR	`typedef bool( * TLIPTR) (int rungState, int argc, TLIPARAM *argv)`	pointer to function type for defining a TLI ladder instruction

Table 7. Constants

Name	Value	Description
MAXFNCNAMELEN	15	Max function name length.
MAXPARMNAMELEN	11	Max param name length.
MAXPARMNUM	9	Max number of params.
MAXDESCLEN	31	Max description length.
TLI_ARG_IN	0x0001	INput parameter: value not changed.
TLI_ARG_OUT	0x0002	OUTput parameter: value can change.
TLI_ARG_INT	0x0004	INTeger argument.
TLI_ARG_FLT	0x0008	FLoaTing point argument.
TLI_ARG_BLOCK	0x0010	BLOCK address.
TLI_ARG_FILE	0x0010	deprecated use TLI_ARG_BLOCK instead
TLI_ARG_TIMER	0x0020	arg may be TIMER
TLI_ARG_COUNTER	0x0040	arg may be COUNTER
TLI_ARG_CONTROL	0x0080	arg may be CONTROL

Name	Value	Description
TLI_ARG_STRING	0x0100	arg may be STRING
TLI_OUTPUT	0	TLI is an output instruction.
TLI_PERMISSIVE	1	TLI is a permissive instruction.
TLI_JAVA	2	TLI is in Java ⇒ use handles where pointers would otherwise be used. Internal use only.
FUNCDEF	``	prefix for a funcdef so that the whole structure gets put into a special elf section.

6.4.1. PARMDEF

```
#include <tlm.h>

struct PARMDEF
```

Struct PARMDEF is a TLI function parameter.

There will be several of these per TLI.

See

funcdef::p

Public Variables	
char	prm_name Argument name.
uint16_t	prm_argtype Argument type.
uint16_t	prm_len Argument length.

6.4.2. funcdef

```
#include <tlm.h>

struct funcdef
```

Struct funcdef provides the definition and expected arguments to a Topdoc Loadable Instruction, a.k.a.

TLI. An array of these can be given to DECLARE_TLM or TLM_MAIN_ENTRY macros. Use the

FUNCDEF macro when declaring the array so that the array is put into the proper section of the link image. Make sure the array is zero terminated with a trailing NULL sentinel.

Public Variables	
char	`fnc_name` Function name.
char	`fnc_desc` Function description.
TLIPTR	`fnc_ptr` Pointer to function.
uint8_t	`fnc_numoptionalargs` Number of optional args.
uint8_t	`fnc_type` Output or permissive.
uint16_t	`fnc_numargs` Number of arguments.
PARMDEF	`p` Array of parameters.

6.4.3. TLIPARAM

```
#include <tlm.h>

struct TLIPARAM
```

A TLI parameter, an array of these are passed to a TLI each time it is invoked.

Public Variables	
PLCINT	`wd` Word value.
float	`fl` Float value.
uint32_t	`hls` LS part of handle.

Public Variables	
union TLIPARAM	`val` used when TLI is passed an argument by value
PLCINT *	`wptr` Word pointer.
float *	`fptr` Float pointer.
TIMER *	`ptmr` Timer block.
COUNTER *	`pcnt` Counter block.
CONTROL *	`pctl` Control block.
STRING *	`pstr` string element
uint32_t	`hms` MS part of handle.
union TLIPARAM	`ptr` used when TLI is passed an argument by value
bool	`isfloat` true if the parameter is float, false if PLCINT

6.5. Old School dispatch() Function Support

The dispatch() function is the main point of entry into the TLM from SoftPLC (except for ladder instructions, each of which defines its own point of entry).

Modern TLMs should use the C++ TLM class instead of this interface which was originally crafted in C.

Modules

Dispatch Command Codes	Function codes used within the request packets for the dispatch() function.

Request Packets	There is one unique type of request packet for each command code.

Functions

dispatch()	Function dispatch should be declared and implemented in your TLM.

6.5.1. dispatch()

```
ERRCODE dispatch(request *reqp)
```

Function dispatch should be declared and implemented in your TLM.

It is called repeatedly by SoftPLC at various points in execution for various purposes. The specific meaning of each call is given by a pointer to a Request Packets union *reqp*, which contains a command code (*cmdcode*) and the corresponding parameters.

Parameters

`reqp`	a pointer to a Request Packets union which holds the command code and all the parameters for that command code.

Returns

ERRCODE - SUCCESS if successful

Table 8. Constants

Name	Value	Description
`FAIL_FNC_NOT_SUPPORTED`	`100`	returned by dispatch when dispatched command code is not suppored

6.5.2. TLM_MAIN_ENTRY()

```
#define TLM_MAIN_ENTRY(aFunctionTable) HLPEnv* SoftPLC___; ERRCODE DLLEXPORT SPLCUSR
mainEntry4( request* req ) {     if( req->cmdcode < FNC_INIT )     {         if( req-
>cmdcode == FNC_LOAD )             SoftPLC___ = req->load.callback;         else
req->pre_init.functionTable = aFunctionTable;     }     return dispatch( req ); }
```

establishes the main entry point to the TLM and sets up the call back mechanism to the helpers and registers the FUNCTION TABLE.

Use it only once and only in your main source file.

6.5.3. Dispatch Command Codes

Function codes used within the request packets for the dispatch() function.

Table 9. Constants

Name	Value	Description
FNC_LOAD	(-2)	see req_load
FNC_PRE_INIT	(-1)	see req_pre_init
FNC_INIT	0	see req_init
FNC_DEINSTALL	1	see req_deinstall
FNC_UNLOAD	2	see req_unload
FNC_SCAN	3	see req_scan
FNC_SETMODE	4	see req_setmode
FNC_CHECKPOINT	5	see req_checkpoint
FNC_BXFER_SUBMIT	6	see req_bxfer

6.5.4. Request Packets

There is one unique type of request packet for each command code.

SoftPLC fills a request packet with its desired _ command code _ and corresponding parameters and then calls the dispatch(request* reqp) function in your TLM.

Classes

req_load

req_pre_init

req_init

req_scan

req_deinstall

req_unload

req_setmode

req_checkpoint

req_bxfer

request

Table 10. Constants

Name	Value	Description
DENY_HOTBACKUP_MODE_CHANGE	(-4)	returned from FNC_SETMODE by backup aware TLM.

Name	Value	Description
SETMODE_RECURSION_CALLER	(-14)	returned by all but the lowest recursion level in the case of recursive setmode calls

req_load

```
#include <tlm.h>

struct req_load
```

Request Packet req_load is the very first dispatch() made to a TLM.

It happens only once, immediately after the TLM is loaded. It establishes the helper function availability.

Public Variables	
int	cmdcode FNC_LOAD.
HLPEnv *	callback Helper Table address.

req_pre_init

```
#include <tlm.h>

struct req_pre_init
```

Request Packet req_pre_init is the 2nd dispatch() made to a TLM.

It happens only once, immediately after the req_load dispatch. It allows the TLM to register TLIs merely by using the TLM_MAIN_ENTRY() macro.

Public Variables	
int	cmdcode FNC_PRE_INIT.
funcdef *	functionTable TLM's must set this during FNC_PRE_INIT to their FUNCTION_TABLE[], but this is done automatically by macro TLM_MAIN_ENTRY()

req_init

```
#include <tlm.h>

struct req_init
```

Request Packet req_init is dispatch()ed exactly once, after the req_pre_init.

Load any configuration files that you may have in here. Initialize and verify your I/O hardware. SoftPLC is not scanning yet. (Program mode is still in effect.) In the event of some hardware or configuration error, you may return non-SUCCESS to prevent SoftPLC from entering RUN mode.

Public Variables	
int	`cmdcode` FNC_INIT.
int	`splc_revision` revision of SoftPLC, as = version*10+revision
int	`splc_caps` capability Info, all set by SoftPLC not TLM's
int	`modNdx` index into an internal TLM table that holds this TLM
int	`DrvCapability` DrvCapability Info.
int	`DrvUsage` This WORD is READ-ONLY for TLMS!
int	`maxRacks` This is the maximum Input or Output capacity of the host SoftPLC as determined by its copy protection device, where the maximum capacity is expressed in (max allowed I or O file size in words)/8, note that the maximum size may be greater than the current size.
const char *	`tlmPathAndName` like argv[0] in a C program, contains full path of *.tlm.so name.
PLCINT *	`stsFile` S2 file in datatable.
const char *	`cmdLine` Contains the command line options string.

Members

```
int splc_caps
```

capability Info, all set by SoftPLC not TLM's

- Bit 0 - SET means SoftPLC is passing in the tlmPathAndName
- Bit 2 - stsFile is present in req_init

```
int DrvCapability
```

DrvCapability Info.

- Bit 0 - SET this if TLM wants req_checkpoint
- Bit 1 - SET this if TLM supports A-B Block Xfer

```
int DrvUsage
```

This WORD is READ-ONLY for TLMS!

- Bit 0 - SET if TLM is being loaded into TOPDOC, RESET if into SoftPLC
- Bit 1 - Internal to SoftPLC

```
const char * tlmPathAndName
```

like argv[0] in a C program, contains full path of *.tlm.so name.

For example `/SoftPLC/tlm/tealware.tlm.so`

```
const char * cmdLine
```

Contains the command line options string.

For example, if MODULE.LST had a line like `DRIVER=/SoftPLC/tlm/tealware.tlm.so WDOG=8` then this string would be "WDOG=8"

---

req_scan

```
#include <tlm.h>

struct req_scan
```

Request Packet req_scan is dispatch()ed repeatedly in any of the SoftPLC Operating Modes.

Any TLM must keep track of the operating mode and act accordingly when being called with this request packet req_scan.

See

req_setmode

Public Variables	
int	`cmdcode` FNC_SCAN.
PLCINT *	`stsFile` pointer to S2 file in datatable
int	`numracks` momentary size of I or O datatable file in words/8

req_deinstall

```
#include <tlm.h>

struct req_deinstall
```

Request Packet req_deinstall is dispatch()ed only once just as SoftPLC is exiting to the operating system, and just before req_unload.

Use this opportunity to deactivate your I/O hardware and possibly save any retentative information to disk.

Public Variables	
int	`cmdcode` FNC_DEINSTALL.
int	`modNdx` index into an internal table of this TLM

req_unload

```
#include <tlm.h>

struct req_unload
```

Request Packet req_unload is dispatch()ed only once just before the TLM is unloaded.

Public Variables	
int	cmdcode FNC_UNLOAD.

req_setmode

```
#include <tlm.h>

struct req_setmode
```

Request Packet req_setmode is dispatch()ed any time there is a change in operating mode.

On power up, the very first mode is OPM_REM_PROG. If the STARTUP.LST file calls for a different operating mode on startup, then just after req_init, there will be a single req_setmode sent with a *mode* value corresponding to the STARTUP.LST value. Thereafter, you should expect this dispatch() whenever TOPDOC or the keyswitch (if any) initiates a mode change, or if the SoftPLC faults out into OPM_FAULT mode. Save the *mode* value into a global so you can act accordingly in req_scan.

See

SoftPLC Operating Modes

Public Variables	
int	cmdcode FNC_SETMODE.
PLCINT *	stsFile S2 file in datatable.
int	mode one of: OPM_PROG, OPM_RUN, OPM_REM_RUN, etc.

req_checkpoint

```
#include <tlm.h>
```

```
struct req_checkpoint
```

Request Packet req_checkpoint is optionally dispatch()ed to a TLM in RUN mode several times per scan.

This dispatch() is only performed if it was requested in req_init by setting bit 0 of DvrCapability. You can do polling of I/O in here.

Public Variables	
int	`cmdcode` FNC_CHECKPOINT.
PLCINT *	`stsFile` S2 file in datatable.

req_bxfer

```
#include <tlm.h>

struct req_bxfer
```

Request Packet req_bxfer is used only by the Allen-Bradley driver to implement block transfer.

used with FNC_BXFER_SUBMIT command code

Public Variables	
int	`cmdcode` FNC_BXFER_SUBMIT.
PLCINT *	`stsFile` S2 file in datatable.
void *	`pDatatab` BTR control block in the datatable.
int	`rungstate` the "true-ness" of the rung power flow
bool	`isOldPlatform` false if BT dt section, else true if N dt section

request

```
#include <tlm.h>
```

```
struct request
```

A generic request packet which is a union of all the command code specific request packets.

Public Variables	
int	`cmdcode`
req_load	`load`
req_pre_init	`pre_init`
req_init	`init`
req_scan	`scan`
req_deinstall	`deinstall`
req_setmode	`setmode`
req_checkpoint	`checkpoint`
req_bxfer	`bxfer`
req_unload	`unload`
union request	`request`

6.6. Helper Functions for C++

Helper functions are characterized by the fact that they are implemented within the runtime and there is a callback from the TLM into the runtime and upon return control returns to the calling TLM.

The Helpers are grouped into the following categories:

Datatable Access Helpers

Memory Management Helpers

File (stdio like) Helpers

Timing Helpers

Debugging Helpers

PCI Bus Configuration Helpers

Thread Helpers

Shared Library Loading and Unloading Helpers

Descriptor Helpers

Property Helpers

6.7. Datatable Access Helpers

Functions that access the datatable.

Classes

BINADDR	

Functions

IsFloat()	Function IsFloat.
HlpAddressParse()	Function HlpAddressParse takes a pointer to a pointer to a nul terminated byte string and parses that string as a SoftPLC datatable address, and then advances the caller's pointer to the string past the address.
HlpAddressFormat()	Function HlpAddressFormat formats a BINADDR into an ASCII datatable address string.
HlpAddressNormalize()	Function HlpAddressNormalize takes a C string datatable address and re-formats it with spelling consistent with the addresses used in the descriptor table.
HlpDtElemSize()	Function HlpDtElemSize returns the size of a full datatable element of a given fileType.
HlpInputsPut()	Function HlpInputsPut puts a number of process data words into the input image table portion of the datatable (at datatable file number 1).
HlpOutputsGet()	Function HlpOutputsGet gets a number of output image table datatable words from datatable file 0.
HlpDtWordPtr()	Function HlpDtWordPtr returns a pointer to a datatable word or NULL if the requested word does not exist.
HlpDtReadFileTypeSize()	Function HlpDtReadFileTypeSize gets information about the size and type of a datatable file.
HlpWordsGet()	Function HlpWordsGet retrieves a block of words from the datatable.
HlpWordsPut()	Function HlpWordsPut copies a block of words into the datatable.

6.7.1. enum DTTYPE

Enum DTTYPE is the set of all types returned in BINADDR given to HlpAddressParse().

These values will be in BINADDR::data_type field.

Value	Init	Description
DTTYPE_BIT	= (1<<0)	a bit datatable type

Value	Init	Description
DTTYPE_WORD	= (1<<1)	a PLCINT or PLCFLOAT datatable type (scaler)
DTTYPE_BLOCK	= (1<<2)	a contiguous sequence of elements
DTTYPE_TIMER	= (1<<3)	a TIMER struct
DTTYPE_COUNTER	= (1<<4)	a COUNTER struct
DTTYPE_CONTROL	= (1<<5)	a CONTROL struct
DTTYPE_PID	= (1<<6)	a PID struct
DTTYPE_MSG	= (1<<7)	a MSG struct
DTTYPE_STRING	= (1<<8)	a STRING struct
DTTYPE_BXFER	= (1<<9)	a BT struct
DTTYPE_SFC	= (1<<10)	an SFC struct

6.7.2. IsFloat()

```
bool IsFloat(const BINADDR *a)
```

Function IsFloat.

Returns

true iff the BINADDR *a* is of type PLCFLOAT

6.7.3. HlpAddressParse()

```
int HlpAddressParse(const char **strAddr, BINADDR *binAddr)
```

Function HlpAddressParse takes a pointer to a pointer to a nul terminated byte string and parses that string as a SoftPLC datatable address, and then advances the caller's pointer to the string past the address.

Parameters

strAddr	the address of a pointer to an address string. After parsing, the caller's pointer is advanced past the address portion of the string, which must be the first part of the string.
binAddr	where to put the binary representation of the address

Returns

int - a minus 1 (-1) means success. A number > -1 is the character index into the address string

where the first error occurred. In the event of an error, the *binAddr* information is not valid, and the caller's string pointer is not advanced.

6.7.4. HlpAddressFormat()

```
std::string HlpAddressFormat(const BINADDR &binAddr)
```

Function HlpAddressFormat formats a BINADDR into an ASCII datatable address string.

Parameters

`binAddr`	is a binary representation of the PLC datatable address that is to be converted to ASCII.

Returns

std:string - the resultant datatable address string.

6.7.5. HlpAddressNormalize()

```
int HlpAddressNormalize(char *output, const char *input)
```

Function HlpAddressNormalize takes a C string datatable address and re-formats it with spelling consistent with the addresses used in the descriptor table.

This will typically have a full set of leading zeros and all fields will be a standard width.

Parameters

`output`	A buffer at least 128 bytes long which is to receive a C string containing the formatted address.
`input`	A C string containing the datatable address to re-format.

Returns

int - a minus 1 (-1) means success. A number > -1 is the character index into the address string where the first error occurred.

6.7.6. HlpDtElemSize()

```
int HlpDtElemSize(enum DT_T aFileType)
```

Function HlpDtElemSize returns the size of a full datatable element of a given fileType.

Parameters

`aFileType`	is one of the DT_T enumerated datatable file type values.

Returns

int - The size of the element in PLCINT's, or 1 if an invalid input argument is provided.

6.7.7. HlpInputsPut()

```
int HlpInputsPut(int startWord, int num, const PLCINT *myWords)
```

Function HlpInputsPut puts a number of process data words into the input image table portion of the datatable (at datatable file number 1).

Input *forcing* may affect the transfer. TOPDOC NexGen controls whether input forcing is in effect.

Parameters

`startWord`	the starting element number within the input image table to start putting into.
`num`	the count of words to put into the input datatable file 1.
`myWords`	a pointer to a block of words to put into the input datatable file at file 1, and offset *startWord*. Normally each bit within each word of this block is a separate digital input obtained by reading input hardware modules.

Returns

int - the number of words successfully put, so anything less than *num* is an error.

6.7.8. HlpOutputsGet()

```
int HlpOutputsGet(int startWord, int num, PLCINT *myWords)
```

Function HlpOutputsGet gets a number of output image table datatable words from datatable file 0.

Output *forcing* may affect the transfer. TOPDOC NexGen controls whether output forcing is in effect.

Parameters

`startWord`	the starting element number within the output image table to start getting from.
`num`	the count of words to get from the output datatable file number 0.

`myWords`	a pointer to a place to copy the (potentially forced) output image table words which will originate at offset *startWord* within the output image table (datatable file 0). Normally each bit within each word of this block is a separate digital output obtained by ladder program logic.

Returns

int - the number of words successfully read, so anything less than *num* is an error.

6.7.9. HlpDtWordPtr()

```
void * HlpDtWordPtr(int file, int elem, int word)
```

Function HlpDtWordPtr returns a pointer to a datatable word or NULL if the requested word does not exist.

You normally cast the return value to the type of datatable object that you are refering to, either a PLCINT*, PLCFLOAT*, COUNTER*, TIMER*, CONTROL*, etc. Because datatable files can be moved to new memory locations while in PROGRAM mode, the returned pointer is valid only until the next transition from OPM_[REM_]PROGRAM to OPM_[REM_]RUN mode. It should be requested again each time you see a FNC_SET_MODE with a mode of OPM_[REM_]RUN or OPM_[REM_]TEST.

Parameters

`file`	the requested datatable file number
`elem`	the requested datatable element number
`word`	the requested datatable word within the element, often zero.

Returns

void* - a pointer to the requested datatable word or NULL if that word does not exist.

6.7.10. HlpDtReadFileTypeSize()

```
ERRCODE HlpDtReadFileTypeSize(int file, int *nwordsp, int *nelemsp, enum DT_T
*sectionp)
```

Function HlpDtReadFileTypeSize gets information about the size and type of a datatable file.

Parameters

`file`	the datatable file number to query, 0-9999.
`nwordsp`	where to put the size of the file, measured in PLCINTs. This is the number of bytes divided by 2.

`nelemsp`	where to put the number of datatable elements in the file. For example a timer element has 3 PLCINTs in it, so the obtained value here would be **nwordsp/3*.
`sectionp`	where to put the type of the file

Returns

ERRCODE - SUCCESS if the file exists, else a non-zero (non-SUCCESS) error code

---

6.7.11. HlpWordsGet()

```
int HlpWordsGet(int file, int elem, int word, int num, PLCINT *myWords)
```

Function HlpWordsGet retrieves a block of words from the datatable.

No I/O forcing is used, and if the requested block resides within a FLOAT file, then all the PLCFLOAT values are converted to PLCINTs as they are fetched. Otherwise the requested words are copied as PLCINTs without conversion.

Parameters

`file`	the requested datatable file number 0-9999.
`elem`	the starting requested datatable element number 0-9999.
`word`	the starting word within the starting element, normally 0.
`num`	howmany PLCINTs to retrieve.
`myWords`	where to put the PLCINTs.

Returns

int - howmany were retrieved. If the file does not exist or if the request is too close to the end of the file for the requested *num* words, then the return value will be less than *num* and possibly zero. If only some of the words can be read, then that will be done.

6.7.12. HlpWordsPut()

```
int HlpWordsPut(int file, int elem, int word, int num, const PLCINT *myWords)
```

Function HlpWordsPut copies a block of words into the datatable.

No I/O forcing is used, and if the target block resides within a FLOAT file, then all the supplied PLCINT values are converted to PLCFLOATs as they are copied. Otherwise the requested words are copied as PLCINTs without conversion.

Parameters

`file`	the target datatable file number 0-9999.
`elem`	the starting target datatable element number 0-9999.
`word`	the starting word within the starting element, normally 0.
`num`	howmany PLCINTs to put.
`myWords`	the source of the PLCINTs.

Returns

int - howmany were copied. If the file does not exist or if the request is too close to the end of the file for the requested *num* words, then the return value will be less than *num* and possibly zero. If only some of the words can be put without running off the end of the file, then that will be done.

6.7.13. BINADDR

```
#include <tlm.h>

struct BINADDR
```

Type for HlpAddressParse()

Public Variables	
int	`data_type` DTTYPE_BIT, DTTYPE_WORD, DTTYPE_TIMER, DTTYPE_PID, etc.
enum DT_T	`section` datatable file type
int	`file` datatable file number
int	`element` datatable element number
int	`word` word within element, 0 normally
int	`bit` bit within word

6.8. Memory Management Helpers

Functions used to manage memory.

Functions

HlpMemAlloc()	Function HlpMemAlloc allocates a number of bytes of RAM in application space.
HlpMemReAlloc()	Function HlpMemReAlloc changes the size of the memory block pointed to by ptr to size bytes.
HlpMemFree()	Function HlpMemFree frees the memory block pointed to by ptr, which must have been returned by a previous call to HlpMemAlloc() or HlpMemReAlloc().
HlpMapPhysical()	Function HlpMapPhysical is used to map a physical memory address that may be associated with a piece of hardware into the process address space.

6.8.1. HlpMemAlloc()

```
void * HlpMemAlloc(unsigned nbytes)
```

Function HlpMemAlloc allocates a number of bytes of RAM in application space.

Return NULL if no more memory is available.

Parameters

`nbytes`	How many bytes to allocate

Returns

void* - a pointer to the allocated memory or NULL if none.

6.8.2. HlpMemReAlloc()

```
void * HlpMemReAlloc(void *ptr, unsigned size)
```

Function HlpMemReAlloc changes the size of the memory block pointed to by ptr to size bytes.

The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If ptr is NULL, the call is equivalent to HlpMemAlloc(size); if size is equal to zero, the call is equivalent to HlpMemFree(ptr). Unless ptr is NULL, it must have been returned by an earlier call to HlpMemAlloc(), HlpMemReAlloc(). Returns NULL if no more memory is available and preserve the original block.

Parameters

`ptr`	the old memory block or NULL if none.
`size`	How many bytes to allocate

Returns

void* - a pointer to the allocated memory or NULL if none.

6.8.3. HlpMemFree()

```
void HlpMemFree(void *ptr)
```

Function HlpMemFree frees the memory block pointed to by ptr, which must have been returned by a previous call to HlpMemAlloc() or HlpMemReAlloc().

Otherwise, or if HlpMemFree(ptr) has already been called before, undefined behavour occurs. If ptr is NULL, no operation is performed.

Parameters

`ptr`	The memory block to free

6.8.4. HlpMapPhysical()

```
char * HlpMapPhysical(uintptr_t physical, unsigned length)
```

Function HlpMapPhysical is used to map a physical memory address that may be associated with a piece of hardware into the process address space.

The returned pointer may then be assumed to point to the memory space on the hardware. You should call this only once per physical block of hardware for the entire life of the TLM. Save the return value in a global.

Parameters

`physical`	the address of a piece of memory mapped hardware, such as 0xD0000, or 0xD8000
`length`	the number of bytes starting at *physical* that should be mapped into the process address space.

Returns

char* - a pointer that can be used to address the hardware directly. You can cast this to a struct*

of a particular type. The return value may be NULL and that indicates that the request failed for some reason.

6.9. File (stdio like) Helpers

Functions that manipulate files.

They are like their stdio.h equivalents.

Functions

HlpPrintf()	Function HlpPrintf is used like printf(), but its output may be directed to the syslogger, depending on how SoftPLC is started.
HlpFopen()	
HlpFread()	
HlpFgetc()	
HlpFputc()	
HlpFwrite()	
HlpFclose()	
HlpFgets()	
HlpFputs()	
HlpFtell()	
HlpFseek()	
HlpFprintf()	
HlpSprintf()	

6.9.1. HlpPrintf()

```
void HlpPrintf(const char *fmt,...)
```

Function HlpPrintf is used like printf(), but its output may be directed to the syslogger, depending on how SoftPLC is started.

Parameters

`fmt`	The format string or string to print
`...`	

Returns

int - The number of characters that were output

6.9.2. HlpFopen()

```
FILE * HlpFopen(const char *name, const char *mode)
```

6.9.3. HlpFread()

```
int HlpFread(void *buf, unsigned elemz, unsigned nelems, FILE *fp)
```

6.9.4. HlpFgetc()

```
int HlpFgetc(FILE *fp)
```

6.9.5. HlpFputc()

```
int HlpFputc(int cc, FILE *fp)
```

6.9.6. HlpFwrite()

```
int HlpFwrite(void *buf, unsigned elemz, unsigned nelems, FILE *fp)
```

6.9.7. HlpFclose()

```
int HlpFclose(FILE *fp)
```

6.9.8. HlpFgets()

```
char * HlpFgets(char *buf, int len, FILE *fp)
```

6.9.9. HlpFputs()

```
int HlpFputs(char *buf, FILE *fp)
```

6.9.10. HlpFtell()

```
long HlpFtell(FILE *fp)
```

6.9.11. HlpFseek()

```
int HlpFseek(FILE *fp, long spot, int mode)
```

6.9.12. HlpFprintf()

```
int HlpFprintf(FILE *fp, const char *fmt,...)
```

6.9.13. HlpSprintf()

```
int HlpSprintf(char *out, const char *fmt,...)
```

6.10. Timing Helpers

Functions used for timing.

Functions

HlpTimerInstall()	Function HlpTimerInstall will install a *func* that will be called at regular time based intervals.
HlpTimerDeinstall()	Function HlpTimerDeinstall will de-install an interval function.
HlpPtimerSet()	Function HlpPtimerSet is used in concert with HlpPtimerPoll() to perform low overhead timing.
HlpPtimerPoll()	Function HlpPtimerPoll is used in concert with HlpPtimerSet() to perform low overhead timing.

Table 11. Typedefs

Name	Definition	Description
`TIMERFUNC`	`typedef void( * TIMERFUNC) (void)`	A function pointer argument used by HlpTimerInstall(). The supplied function will be called at a specified interval on a thread separate from the main ladder thread. Be careful about what is called from this function as it needs to be thread safe.

6.10.1. HlpTimerInstall()

```
ERRCODE HlpTimerInstall(TIMERFUNC func, int modNdx, unsigned milliSecsDelta)
```

Function HlpTimerInstall will install a *func* that will be called at regular time based intervals.

Never cast the *func* argument to fit the requirements of this call, instead always define a function whose prototype matches TIMERFUNC exactly. Make sure your *func* returns as quicly as possible. It is called in PROGRAM, TEST, RUN, or FAULT modes, so keep track of the mode in req_setmode.

Parameters

`func`	the TIMERFUNC that you want to be called periodically.
`modNdx`	what you got from from req_init.
`milliSecsDelta`	the interval at which to call your *func*

Returns

ERRCODE - SUCCESS or error code

See also

HlpTimerDeinstall

6.10.2. HlpTimerDeinstall()

```
ERRCODE HlpTimerDeinstall(TIMERFUNC func)
```

Function HlpTimerDeinstall will de-install an interval function.

After calling this, the *func* will no longer be called at regular intervals.

Parameters

`func`	the TIMERFUNC to de-install

Returns

ERRCODE - SUCCESS or error code

See also

HlpTimerInstall

6.10.3. HlpPtimerSet()

```
uint32_t HlpPtimerSet(uint32_t msecsDelay)
```

Function HlpPtimerSet is used in concert with HlpPtimerPoll() to perform low overhead timing.

First call this function with some maximum delay value in msecs (*msecsDelay*). This function always returns immediately with a future time value, which is computed as the sum of the given *msecsDelay* plus an internal relative monotonically ascending clock value. After returning, start watching for your hardware device or activity to complete. While you are waiting, you repeatedly call HlpPtimerPoll() with the value you got back from HlpPtimerSet(). When your timing interval is expired, HlpPtimerPoll() returns a value ⇐ 0, otherwise it returns a positive number.

Parameters

`msecsDelay`	the maximum number of milliseconds that you will be polling

Returns

uint32_t - the relative time of some point in the future at which time the desired timing interval will be complete.

See also

HlpPtimerPoll()

6.10.4. HlpPtimerPoll()

```
int32_t HlpPtimerPoll(uint32_t futureTime)
```

Function HlpPtimerPoll is used in concert with HlpPtimerSet() to perform low overhead timing.

The strategy is to first call HlpPtimerSet() with some maximum delay value. You immediately get control back and you start watching for your device or activity to complete. While you are waiting, you repeatedly call HlpPtimerPoll() with the value you got back from HlpPtimerSet(). When your timing interval is expired, HlpPtimerPoll() returns a value ⇐ 0, otherwise it returns a positive number.

Parameters

`futureTime`	is a value that you obtained from HlpPtimerSet()

Attention

You should not hog the CPU in any dispatch() call other than req_init. Otherwise, you should perform the HlpPtimerPoll test over a number of separate dispatch() calls.

Returns

int32_t - ⇐0 means timed out, >0 means still timing.

See also

HlpPtimerSet()

6.11. Debugging Helpers

Functions used for debugging.

Functions

dprint()	Function dprint provides a printf() like function that accepts the same format string followed by arguments.

6.11.1. dprint()

```
unsigned dprint(const char *fmt,...)
```

Function dprint provides a printf() like function that accepts the same format string followed by arguments.

It is intended for single-line debug print statements, and automatically includes a newline in the printed string. Additionally, it prepends time information to the front of the string in two six-digit numbers (both in microseconds). The second tracks the time delta between dprint() calls. The first tracks the total relative time, or the sum of the deltas.

Parameters

`fmt`	the format string, similar to printf()
`...`	

Attention

The relative microsecond timer will roll over every second.

6.12. PCI Bus Configuration Helpers

A few functions that make it possible to query PCI card configuration data from user space.

Functions

HlpPciBiosFindDevice()	Function HlpPciBiosFindDevice searches the PCI bus(es) for a card with the given *vendor* and *device_id* and returns PCIBIOS_SUCCESSFUL if found.
HlpPciBiosFindClass()	Function HlpPciBiosFindClass searches the PCI bus(es) for a card with the given *class_code* and returns PCIBIOS_SUCCESSFUL if found.
HlpPciBiosReadConfigByte()	Function HlpPciBiosReadConfigByte reads a *val* from the configuration space of a card located at the *bus* and *device_fn* (obtained from HlpPciBiosFindDevice() or HlpPciBiosFindClass()).
HlpPciBiosReadConfigWord()	Function HlpPciBiosReadConfigWord reads a *val* from the configuration space of a card located at the *bus* and *device_fn* (obtained from HlpPciBiosFindDevice() or HlpPciBiosFindClass()).
HlpPciBiosReadConfigDword()	Function HlpPciBiosReadConfigDword reads a *val* from the configuration space of a card located at the *bus* and *device_fn* (obtained from HlpPciBiosFindDevice() or HlpPciBiosFindClass()).
HlpPciBiosWriteConfigByte()	Function HlpPciBiosWriteConfigByte writes a *val* to the configuration space of a card located at the *bus* and *device_fn* (obtained from HlpPciBiosFindDevice() or HlpPciBiosFindClass()).
HlpPciBiosWriteConfigWord()	Function HlpPciBiosWriteConfigWord writes a *val* to the configuration space of a card located at the *bus* and *device_fn* (obtained from HlpPciBiosFindDevice() or HlpPciBiosFindClass()).
HlpPciBiosWriteConfigDword()	Function HlpPciBiosWriteConfigDword writes a *val* to the configuration space of a card located at the *bus* and *device_fn* (obtained from HlpPciBiosFindDevice() or HlpPciBiosFindClass()).

6.12.1. HlpPciBiosFindDevice()

```
ERRCODE HlpPciBiosFindDevice(int vendor, int device_id, int index, uint8_t *bus,
uint8_t *device_fn)
```

Function HlpPciBiosFindDevice searches the PCI bus(es) for a card with the given *vendor* and *device_id* and returns PCIBIOS_SUCCESSFUL if found.

The *index* argument is used to find a specific instance number of a type of card. For the first instance always use *index* = 0, for the second card use *index* = 1, etc.

Parameters

`vendor`	the 16 bit unique vendor identifier which comes from your card vendor.

`device_id`	the 16 bit unique device identifier which comes from your card vendor to identify a specific card type from the given *vendor*
`index`	the instance number of the requested card on the PCI bus(es).
`bus`	where to put the pci bus number of the found card.
`device_fn`	where to put the slot and function number combo.

Returns

ERRCODE - PCIBIOS_SUCCESSFUL if found, PCIBIOS_DEVICE_NOT_FOUND if not.

6.12.2. HlpPciBiosFindClass()

```
ERRCODE HlpPciBiosFindClass(int class_code, int index, uint8_t *bus, uint8_t
*device_fn)
```

Function HlpPciBiosFindClass searches the PCI bus(es) for a card with the given *class_code* and returns PCIBIOS_SUCCESSFUL if found.

The *index* argument is used to find a specific instance number of a type of card. For the first instance always use *index* = 0, for the second card use *index* = 1, etc.

Parameters

`class_code`	the unique class or "category of card" identifier which comes from your card vendor.
`index`	the instance number of the requested card on the PCI bus(es).
`bus`	where to put the pci bus number of the found card.
`device_fn`	where to put the slot and function number combo.

Returns

ERRCODE - PCIBIOS_SUCCESSFUL if found, PCIBIOS_DEVICE_NOT_FOUND if not.

6.12.3. HlpPciBiosReadConfigByte()

```
ERRCODE HlpPciBiosReadConfigByte(uint8_t bus, uint8_t device_fn, uint8_t where,
uint8_t *val)
```

Function HlpPciBiosReadConfigByte reads a *val* from the configuration space of a card located at the *bus* and *device_fn* (obtained from HlpPciBiosFindDevice() or HlpPciBiosFindClass()).

Parameters

`bus`

`device_fn`

`where` the byte index into the configuration space for the card at which to perform the read

`val` to put the obtained value

Returns

ERRCODE - PCIBIOS_SUCCESSFUL if successful.

6.12.4. HlpPciBiosReadConfigWord()

```
ERRCODE HlpPciBiosReadConfigWord(uint8_t bus, uint8_t device_fn, uint8_t where,
uint16_t *val)
```

Function HlpPciBiosReadConfigWord reads a *val* from the configuration space of a card located at the *bus* and *device_fn* (obtained from HlpPciBiosFindDevice() or HlpPciBiosFindClass()).

Parameters

`bus`

`device_fn`

`where` the byte index into the configuration space for the card at which to perform the read

`val` to put the obtained value

Returns

ERRCODE - PCIBIOS_SUCCESSFUL if successful.

6.12.5. HlpPciBiosReadConfigDword()

```
ERRCODE HlpPciBiosReadConfigDword(uint8_t bus, uint8_t device_fn, uint8_t where,
uint32_t *val)
```

Function HlpPciBiosReadConfigDword reads a *val* from the configuration space of a card located at the *bus* and *device_fn* (obtained from HlpPciBiosFindDevice() or HlpPciBiosFindClass()).

Parameters

`bus`

`device_fn`

`where`	the byte index into the configuration space for the card at which to perform the read
`val`	to put the obtained value

Returns

ERRCODE - PCIBIOS_SUCCESSFUL if successful.

6.12.6. HlpPciBiosWriteConfigByte()

```
ERRCODE HlpPciBiosWriteConfigByte(uint8_t bus, uint8_t device_fn, uint8_t where,
uint8_t val)
```

Function HlpPciBiosWriteConfigByte writes a *val* to the configuration space of a card located at the *bus* and *device_fn* (obtained from HlpPciBiosFindDevice() or HlpPciBiosFindClass()).

Parameters

`bus`	
`device_fn`	
`where`	the byte index into the configuration space for the card at which to perform the write
`val`	the value to write

Returns

ERRCODE - PCIBIOS_SUCCESSFUL if successful.

6.12.7. HlpPciBiosWriteConfigWord()

```
ERRCODE HlpPciBiosWriteConfigWord(uint8_t bus, uint8_t device_fn, uint8_t where,
uint16_t val)
```

Function HlpPciBiosWriteConfigWord writes a *val* to the configuration space of a card located at the *bus* and *device_fn* (obtained from HlpPciBiosFindDevice() or HlpPciBiosFindClass()).

Parameters

`bus`	
`device_fn`	
`where`	the byte index into the configuration space for the card at which to perform the write
`val`	the value to write

Returns

ERRCODE - PCIBIOS_SUCCESSFUL if successful.

6.12.8. HlpPciBiosWriteConfigDword()

```
ERRCODE HlpPciBiosWriteConfigDword(uint8_t bus, uint8_t device_fn, uint8_t where,
uint32_t val)
```

Function HlpPciBiosWriteConfigDword writes a *val* to the configuration space of a card located at the *bus* and *device_fn* (obtained from HlpPciBiosFindDevice() or HlpPciBiosFindClass()).

Parameters

`bus`	
`device_fn`	
`where`	the byte index into the configuration space for the card at which to perform the write
`val`	the value to write

Returns

ERRCODE - PCIBIOS_SUCCESSFUL if successful.

Table 12. Constants

Name	Value	Description
`PCIBIOS_SUCCESSFUL`	`0x00`	worked ok
`PCIBIOS_FUNC_NOT_SUPPORTED`	`0x81`	function is not supported
`PCIBIOS_BAD_VENDOR_ID`	`0x83`	vendor id is bad
`PCIBIOS_DEVICE_NOT_FOUND`	`0x86`	device was not found
`PCIBIOS_BAD_REGISTER_NUMBER`	`0x87`	out of range index
`PCIBIOS_SET_FAILED`	`0x88`	unable to write
`PCIBIOS_BUFFER_TOO_SMALL`	`0x89`	buffer to small

6.12.9. PCI_DEVFN()

```
#define PCI_DEVFN(slot, func) ((((slot) & 0x1f) << 3) | ((func) & 0x07))
```

Make a "device_fn" combo slot and function value.

6.12.10. PCI_SLOT()

```
#define PCI_SLOT(devfn) (((devfn) >> 3) & 0x1f)
```

Isolate a "slot" from a "device_fn".

6.12.11. PCI_FUNC()

```
#define PCI_FUNC(devfn) ((devfn) & 0x07)
```

Isolate a "function" from a "device_fn".

6.13. Thread Helpers

Functions and enums that make it possible to create threads and control their execution priorities.

Functions

HlpThreadStart()	Function HlpThreadStart creates a new thread.
HlpThreadCurrentHandle()	Function HlpThreadCurrentHandle returns the THREADID for the calling thread.
HlpThreadJoin()	Function HlpThreadJoin will block until the thread identified by *threadHandle* terminates by returning from its *threadBegin* function.
HlpThreadSetPriority()	Function HlpThreadSetPriority sets the priority of any thread for which you have a THREADID *threadHandle*.
HlpThreadGetPriority()	Function HlpThreadGetPriority returns the THREAD_PRIORITY for the thread given by *threadHandle*.
HlpThreadSleep()	Function HlpThreadSleep will suspend the execution of the calling thread for a number of microseconds given by *microSecs*.
HlpThreadName()	Function HlpThreadName returns the name of the thread given by *threadHandle*.
HlpThreadKill()	Function HlpThreadKill sends a signal to a given thread.

6.13.1. enum THREAD_PRIORITY

Allowable thread priorities for HlpThreadStart() and HlpThreadSetPriority()

Value	Init	Description
PRIORITY_LOWLOW	= 0	The lowest SoftPLC priority.
PRIORITY_LOW	= 1	One priority lower than PRIORITY_MIDDLE.
PRIORITY_MIDDLE	= 2	SoftPLC's main control thread runs at this priority.
PRIORITY_HIGH	= 3	ONE's transmit and receive threads run at this priority.
PRIORITY_HIGHHIGH	= 4	One priority higher than PRIORITY_HIGH.

Table 13. Typedefs

Name	Definition	Description
THREADID	typedef void* THREADID	handle of a SoftPLC thread
THREADFUNC	typedef void(* THREADFUNC)(void *startData)	Function argument used by HlpThreadStart()

6.13.2. HlpThreadStart()

```
THREADID HlpThreadStart(THREADFUNC threadBegin, const char *threadName, void *data,
enum THREAD_PRIORITY priority, unsigned stackSize)
```

Function HlpThreadStart creates a new thread.

There is some overhead to this, so normally it is best to create threads while processing req_init where it safe to hog the CPU for some time. A thread runs until it returns from its *threadBegin* function.

Parameters

threadBegin	where to start executing the thread, never apply a cast to this argument.
threadName	the name of the thread, used in page fault handling text messages.
data	a pointer to any data that you want to pass to *threadBegin*, may be NULL.
priority	must be one of enum THREAD_PRIORITY. If this is too high you can starve the main ladder thread.
stackSize	the size of the threads stack to used, typically 60K is about right.

Returns

THREADID which can be used in other thread functions.

6.13.3. HlpThreadCurrentHandle()

```
THREADID HlpThreadCurrentHandle()
```

Function HlpThreadCurrentHandle returns the THREADID for the calling thread.

6.13.4. HlpThreadJoin()

```
ERRCODE HlpThreadJoin(THREADID threadHandle)
```

Function HlpThreadJoin will block until the thread identified by *threadHandle* terminates by returning from its *threadBegin* function.

Parameters

`threadHandle`	which thread to wait for

6.13.5. HlpThreadSetPriority()

```
ERRCODE HlpThreadSetPriority(THREADID threadHandle, enum THREAD_PRIORITY priority)
```

Function HlpThreadSetPriority sets the priority of any thread for which you have a THREADID *threadHandle*.

Parameters

`threadHandle`	which thread to change the priority on
`priority`	must be one of enum THREAD_PRIORITY

Returns

ERRCODE - SUCCESS or error code

6.13.6. HlpThreadGetPriority()

```
enum THREAD_PRIORITY HlpThreadGetPriority(THREADID threadHandle)
```

Function HlpThreadGetPriority returns the THREAD_PRIORITY for the thread given by *threadHandle*.

6.13.7. HlpThreadSleep()

```
void HlpThreadSleep(unsigned microSecs)
```

Function HlpThreadSleep will suspend the execution of the calling thread for a number of microseconds given by *microSecs*.

One thousand microseconds equals one millisecond.

Parameters

`microSecs`	how long to sleep

6.13.8. HlpThreadName()

```
const char * HlpThreadName(THREADID threadHandle)
```

Function HlpThreadName returns the name of the thread given by *threadHandle*.

Parameters

`threadHandle`	which thread to query

Returns

const char* - the name of the thread used in HlpThreadStart()

6.13.9. HlpThreadKill()

```
ERRCODE HlpThreadKill(THREADID threadHandle, int signo)
```

Function HlpThreadKill sends a signal to a given thread.

Parameters

`threadHandle`	which thread to send signal to
`signo`	is the signal number to send, normally SIGUSR1

6.14. Shared Library Loading and Unloading Helpers

Functions that allow loading and unloading of shared libraries and/or TLMs, and one to get the

address of symbols within a shared library.

Functions

HlpLibraryLoad()	Function HlpLibraryLoad loads a shared library that may be specially built to use Helper Functions.
HlpLibraryUnload()	Function HlpUnLibraryLoad unloads a library that was previously loaded with HlpLibraryLoad() and is identified by the handle returned by that function.
HlpSymbolAddress()	Function HlpSymbolAddress returns the address of a symbol (function or data variable) within a shared library.

Table 14. Typedefs

Name	Definition	Description
HLIBRARY	typedef void* HLIBRARY	handle of a loadable module

6.14.1. HlpLibraryLoad()

```
ERRCODE HlpLibraryLoad(const char *path, bool dispatch_LOAD, HLIBRARY *pHandle, char
errMsg[ERRMSGLEN])
```

Function HlpLibraryLoad loads a shared library that may be specially built to use Helper Functions.

Parameters

path	the full path of the shared library to load.
dispatch_LOAD	is a bool that determines whether the library is built to use SoftPLC Helper functions, which means it has a dispatch() and a TLM_MAIN_ENTRY(aFunctionTable)
pHandle	where to put a uint32_t that is a library handle that can be used with HlpLibraryUnload().
errMsg	where to put a nul terminated string explaining why it failed if it did.

Returns

ERRCODE - SUCCESS or error code

6.14.2. HlpLibraryUnload()

```
void HlpLibraryUnload(HLIBRARY handle, bool dispatch_UNLOAD)
```

Function HlpUnLibraryLoad unloads a library that was previously loaded with HlpLibraryLoad() and is identified by the handle returned by that function.

Calling dispatch() is optional and is controlled by the *dispatch_UNLOAD* argument.

Parameters

`handle`	the library handle returned by HlpLibraryLoad()
`dispatch_UNLOAD`	is true if this library should be sent the FNC_UNLOAD command thru its dispatch method.

6.14.3. HlpSymbolAddress()

```
void * HlpSymbolAddress(HLIBRARY libraryHandle, const char *symbolName)
```

Function HlpSymbolAddress returns the address of a symbol (function or data variable) within a shared library.

Parameters

`libraryHandle`	pertains to a shared library and is obtained from HlpLibraryLoad().
`symbolName`	is the name of the symbol to find within the shared library.

Returns

void* - The address or 0 if not found.

6.15. Descriptor Helpers

Functions that operate on the Descriptor Table.

Within each SoftPLC application program there is a descriptor table consiting of 3 columns: tag, address, and description.

Functions

HlpDescFindByAddr()	Function HlpDescFindByAddr finds a descriptor from a given address.
HlpDescFindNextByAddr()	Function HlpDescFindNextByAddress finds a descriptor from a given address, then advances one address alphabetically and returns that descriptor.
HlpDescFindByTag()	Function HlpDescFindByTag finds a descriptor from a given tag.
HlpDescFindNextByTag()	Function HlpDescFindNextByTag finds a descriptor from a given tag, then advances one tag alphabetically and returns that descriptor.
HlpDescDeleteByAddress()	Function HlpDescDeleteByAddress deletes the descriptor associated with the given address.

HlpDescDeleteByTag()	Function HlpDescDeleteByTag deletes the descriptor associated with the given tag.
HlpDescInsert()	Function HlpDescInsert inserts a desciptor into the descriptor table.

6.15.1. HlpDescFindByAddr()

```
bool HlpDescFindByAddr(const char *aFindAddress, char *aAddress, int aAddressSize,
char *aTag, int aTagSize, char *aDescription, int aDescriptionSize)
```

Function HlpDescFindByAddr finds a descriptor from a given address.

Parameters

aFindAddress	A C string containing the PLC address of the descriptor that should be found.
aAddress	Where to put the nul terminated address string. This buffer should be at least 256 bytes long. This string can be parsed with HlpAddressParse().
aAddressSize	You tell the API how long aAddress buffer is here.
aTag	the tag if it exists is put here, as a C string, else null string. This buffer should be at least 256 bytes long.
aTagSize	you tell the API how long aTag buffer is here.
aDescription	Where to put the description part of the descriptor. This buffer should be at least 256 bytes long.
aDescriptionSize	You tell the API how long aDescription buffer is here.

Returns

bool - true if the descriptor was found, else false. If not found, all receive buffers are set to the empty string.

6.15.2. HlpDescFindNextByAddr()

```
bool HlpDescFindNextByAddr(const char *aFindAddress, char *aAddress, int aAddressSize,
char *aTag, int aTagSize, char *aDescription, int aDescriptionSize)
```

Function HlpDescFindNextByAddress finds a descriptor from a given address, then advances one address alphabetically and returns that descriptor.

You can use this function to walk through all descriptors if you start with an address of "", and upon each call use *aAddress* returned as *aFindAddr* in your next call.

Parameters

aFindAddress	A C string containing the PLC address of the descriptor that should be found.

aAddress	Where to put the nul terminated address string. This buffer should be at least 256 bytes long. This string can be parsed with HlpAddressParse().
aAddressSize	You tell the API how long aAddress buffer is here.
aTag	The tag if it exists is put here, as a C string, else null string. This receive buffer should be at least 256 bytes long.
aTagSize	you tell the API how long aTag buffer is here.
aDescription	Where to put the description part of the descriptor. This buffer should be at least 256 bytes long.
aDescriptionSize	You tell the API how long aDescription buffer is here.

Returns

bool - true if the descriptor was found, else false. If not found, all receive buffers are set to the empty string.

6.15.3. HlpDescFindByTag()

```
bool HlpDescFindByTag(const char *aFindTag, char *aAddress, int aAddressSize, char
*aTag, int aTagSize, char *aDescription, int aDescriptionSize)
```

Function HlpDescFindByTag finds a descriptor from a given tag.

Parameters

aFindTag	A C string containing the tag of the descriptor that should be found.
aAddress	Where to put the nul terminated address string. This buffer should be at least 256 bytes long. This string can be parsed with HlpAddressParse().
aAddressSize	You tell the API how long aAddress buffer is here.
aTag	Where to put the nul terminated tag string if it exists, else null string. This buffer should be at least 256 bytes long.
aTagSize	you tell the API how long aTag buffer is here.
aDescription	Where to put the description part of the descriptor. This buffer should be at least 256 bytes long.
aDescriptionSize	You tell the API how long aDescription buffer is here.

Returns

bool - true if the descriptor was found, else false. If not found, all receive buffers are set to the empty string.

6.15.4. HlpDescFindNextByTag()

```
bool HlpDescFindNextByTag(const char *aFindTag, char *aAddress, int aAddressSize, char
*aTag, int aTagSize, char *aDescription, int aDescriptionSize)
```

Function HlpDescFindNextByTag finds a descriptor from a given tag, then advances one tag alphabetically and returns that descriptor.

You can use this function to walk through all descriptors that have tags, if you start with a tag of "", and upon each call use *aTag* returned, in your next call to this function as the *aFindTag* parameter.

Parameters

`aFindTag`	A C string containing the tag of the descriptor that should be found.
`aAddress`	Where to put the nul terminated address string. This buffer should be at least 256 bytes long. This string can be parsed with HlpAddressParse().
`aAddressSize`	You tell the API how long aAddress buffer is here.
`aTag`	Where to put the nul terminated tag string if it exists, else null string. This buffer should be at least 256 bytes long.
`aTagSize`	you tell the API how long aTag buffer is here.
`aDescription`	Where to put the description part of the descriptor. This buffer should be at least 256 bytes long.
`aDescriptionSize`	You tell the API how long aDescription buffer is here.

Returns

bool - true if the descriptor was found, else false. If not found, all receive buffers are set to the empty string.

6.15.5. HlpDescDeleteByAddress()

```
bool HlpDescDeleteByAddress(const char *aAddressToDelete)
```

Function HlpDescDeleteByAddress deletes the descriptor associated with the given address.

Parameters

`aAddressToDelete`	The address identifying the descriptor. This address must be a normalized spelling of the address. Therefore you will likely have to use HlpAddressNormalize() on the address text before passing it into this function to make certain the address can be found within the descriptor table.

Returns

bool - true if the address was found and the descriptor could be deleted.

6.15.6. HlpDescDeleteByTag()

```
bool HlpDescDeleteByTag(const char *aTagToDelete)
```

Function HlpDescDeleteByTag deletes the descriptor associated with the given tag.

Parameters

aTagToDelete	The tag identifying the descriptor.

Returns

bool - true if the tag was found and the descriptor could be deleted.

6.15.7. HlpDescInsert()

```
ERRCODE HlpDescInsert(char *aAddress, const char *aTag, const char *aDescription, bool
overwriteExisting)
```

Function HlpDescInsert inserts a desciptor into the descriptor table.

Address formatting (spelling) can vary with regard to the number of leading zeros on file, element or word components. However, since NexGen assumes a specific type of spelling for each address that it needs to display, (called normalized form) it is mandatory that any address that finds its way into the descriptor table be formatted in exactly this normalized form. For this reason, the normalized formatted address will be * returned * to the caller.

Parameters

aAddress	A C string SoftPLC memory address of the new (or replacement) descriptor. This field must be present and valid. The string buffer containing this text must be at least 128 characters long, because the runtime will reformat the address and return it back to you in this same place, and it may be longer than the string you passed in.
aTag	A C string tagname of the new (or replacement) descriptor. This tag must be unique amoung all descriptor tags, and consist of only legal tag characters. The tag can be encoded in international characters using UTF8. It may be NULL if no tag is desired for this descriptor.

`aDescription`	The description part of the new (or replacement) descriptor. There is a maximum size to this text. It can be encoded in international characters using a \0 terminated UTF8 string.
`overwriteExisting`	A bool which if true, means it is OK to overwrite any existing descriptor with a matching *aAddress*. If false, then return an error if the new descriptor would be a duplicate.

Returns

ERRCODE - one of: * SUCCESS

- E_DESC_TAGTOOBIG
- E_DESC_RECORDTOOBIG
- E_DESC_TAGEXISTS
- E_DESC_ADDRESSEXISTS
- E_DESC_ILLEGALTAG
- E_DESC_ILLEGALADDRESS

Table 15. Constants

Name	Value	Description
`E_DESC_TAGTOOBIG`	1	the tag exceeded a maximum length of 20
`E_DESC_RECORDTOOBIG`	2	the combination of address, plus tag, plus description exceed a maximum of 248
`E_DESC_TAGEXISTS`	3	the tag already exists and your *overwriteExisting* was false
`E_DESC_ADDRESSEXISTS`	4	the address already exists and your *overwriteExisting* was false
`E_DESC_ILLEGALTAG`	5	there was one or more illegal characters in your aTag
`E_DESC_ILLEGALADDRESS`	6	there is something wrong with your address

6.16. Property Helpers

Functions that operate on the Property Tables.

Within each SoftPLC application program there are property tables, each consiting of a two column table with columns for name and value.

Functions

HlpPropFindByName()	Function HlpPropFindByName finds a property pair from a given property name.

HlpPropFindNextByName()	Function HlpPropFindNextByName finds a property pair from a given property name, then advances one name alphabetically and returns that property name/value pair.
HlpPropDelete()	Function HlpPropDelete deletes a name/value pair, i.e.
HlpPropInsert()	Function HlpPropInsert inserts a new property into a property table, or can be used to replace an existing property with the same name key as *aName*.

6.16.1. HlpPropFindByName()

```
bool HlpPropFindByName(int aFileNum, const char *aFindName, char *aName, int
aNameSize, char *aValue, int aValueSize)
```

Function HlpPropFindByName finds a property pair from a given property name.

Parameters

aFileNum	The property file number to search (0-999).
aFindName	The C string containing the property name to find.
aName	Where to put the property name. This receive buffer should be about 256 bytes long.
aNameSize	You tell the API how long the *aName* buffer is here.
aValue	Where to put the property value. This receive buffer should be about 256 bytes long.
aValueSize	You tell the API how long *AValue* is.

Returns

bool - true if the property was found, else false. If not found, the receive buffers are set to the empty string.

6.16.2. HlpPropFindNextByName()

```
bool HlpPropFindNextByName(int aFileNum, const char *aFindName, char *aName, int
aNameSize, char *aValue, int aValueSize)
```

Function HlpPropFindNextByName finds a property pair from a given property name, then advances one name alphabetically and returns that property name/value pair.

You can use this function to walk through all properties if you start with a name of "", and upon each call use *aName* returned in your next call to this function. as aFindAddr in your next call.

Parameters

`aFileNum`	The property file number to search (0-999).
`aFindName`	The C string containing the property name to find.
`aName`	Where to put the property name. This receive buffer should be about 256 bytes long.
`aNameSize`	You tell the API how long the *aName* buffer is here.
`aValue`	Where to put the property value. This receive buffer should be about 256 bytes long.
`aValueSize`	You tell the API how long *AValue* is.

Returns

bool - true if the property was found, else false. If not found, the receive buffers are set to the empty string.

6.16.3. HlpPropDelete()

```
ERRCODE HlpPropDelete(int aFileNum, const char *aName)
```

Function HlpPropDelete deletes a name/value pair, i.e.

a property given by *aName*, from a property table given by *aFileNum*.

Parameters

`aFileNum`	The property file number to delete from.
`aName`	The name part of the property (which is the database key).

Returns

ERRCODE - one of: * SUCCESS

- E_PROP_FILEDOESNOTEXIST
- E_PROP_NAMENOTFOUND

6.16.4. HlpPropInsert()

```
ERRCODE HlpPropInsert(int aFileNum, const char *aName, const char *aValue, bool
overwriteExisting)
```

Function HlpPropInsert inserts a new property into a property table, or can be used to replace an existing property with the same name key as *aName*.

Parameters

`aFileNum`	The property table/file number to modify.
`aName`	The name part of the name/value pair constituting a property.
`aValue`	The value part of the property.
`overwriteExisting`	A bool which if true, means it is OK to overwrite any existing property with a matching *aName*. If false, then return an error when the aName already exists within the property file and do not modify the table.

Returns

ERRCODE - one of: * SUCCESS

- E_PROP_RECORDTOOBIG
- E_PROP_PROPEXISTS
- E_PROP_FILEDOESNOTEXIST

Table 16. Constants

Name	Value	Description
`E_PROP_FILEDOESNOTEXIST`	1	the property file *aFileNum* does not exist, use HlpFileCreate() first.
`E_PROP_NAMENOTFOUND`	2	*aName* is not in property file *aFileNum*
`E_PROP_RECORDTOOBIG`	3	the combination of name plus value exceeded a maximum of 248.
`E_PROP_PROPEXISTS`	4	the property name already exists and your *overwriteExisting* was false.

6.17. Miscellaneous Helpers

Helper Functions that don't fit another category.

Functions

HlpFileCreate()	Function HlpFileCreate creates a file within the SoftPLC runtime APP image.
HlpFileDelete()	Function HlpFileDelete deletes a file within the SoftPLC runtime APP image.
HlpCRC16()	Function HlpCRC16 calculates a CRC16 checksum from a running total and a new byte.
HlpGetAPIVersion()	Function HlpGetAPIVersion returns the version of the Helper API.
HlpSetMode()	Function HlpSetMode will change the operating mode of SoftPLC to one of the OPM_* defines.

HlpShowMode()	Function HlpShowMode will convert a mode integer to string and return it.
HlpGetAuthor()	Function HlpGetAuthor returns authorization status of the given @ auth code.

6.17.1. HlpFileCreate()

```
ERRCODE HlpFileCreate(int aArea, int aFile, enum DT_T aType)
```

Function HlpFileCreate creates a file within the SoftPLC runtime APP image.

The file may be a datatable file or a property file. The runtime must be in a program mode to create or delete files.

0-datatable

4-properties

Parameters

aArea	The area of memory in which to add the file:
aFile	The file number to add.
aType	The type of file to add, one of the SFT_?? from enum DT_T. This parameter is ignored when creating a property file.

Returns

ERRCODE - SUCCESS or some non-zero error code.

6.17.2. HlpFileDelete()

```
ERRCODE HlpFileDelete(int aArea, int aFile)
```

Function HlpFileDelete deletes a file within the SoftPLC runtime APP image.

The file may be a datatable file or a property file. The runtime must be in a program mode to create or delete files.

0-datatable

4-properties

Parameters

aArea	The area of memory from which to delete the file:

`aFile`	The file number to delete.

Returns

ERRCODE - SUCCESS or some non-zero error code.

6.17.3. HlpCRC16()

```
uint16_t HlpCRC16(uint16_t crc, uint8_t cc)
```

Function HlpCRC16 calculates a CRC16 checksum from a running total and a new byte.

Call this in a loop starting with an initial value for *crc* and for each byte in a datastream. On each call except for the first, pass the value returned from this function as the new *crc* argument. The final call yields the CRC16 for all bytes in the loop.

Parameters

`crc`	the running total, set to zero initially.
`cc`	the next byte to sum into the CRC16.

Returns

uint16_t - the CRC16 for a string of bytes.

6.17.4. HlpGetAPIVersion()

```
int HlpGetAPIVersion()
```

Function HlpGetAPIVersion returns the version of the Helper API.

This can be used to verify that helpers your TLM needs are actually present.

Returns

int - what ever is current.

6.17.5. HlpSetMode()

```
ERRCODE HlpSetMode(int mode)
```

Function HlpSetMode will change the operating mode of SoftPLC to one of the OPM_* defines.

If neither SoftPLC itself nor any TLMs deny this mode change, then the change will occur. TLMs can deny the mode change by returning other than SUCCESS from dispatch() on the FNC_SETMODE command code. Note that calling this helper will result in a recursive call to the calling TLM on the dispatch() function using the FNC_SETMODE command code.

Parameters

`mode`	the new mode to enter, must be one of the OperatingModes

See also

SoftPLC Operating Modes

Returns

ERRCODE - SUCCESS if mode change occurred.

6.17.6. HlpShowMode()

```
const char * HlpShowMode(int aMode)
```

Function HlpShowMode will convert a mode integer to string and return it.

Only present in GetAPIVersion() >= 6.

Parameters

`aMode`	is the OPM_* value to translate.

See also

SoftPLC Operating Modes

Returns

const char* - textual mode name.

6.17.7. HlpGetAuthor()

```
int HlpGetAuthor(int auth, int mVers)
```

Function HlpGetAuthor returns authorization status of the given @ auth code.

Parameters

`auth`	is the code to look for.
`mVers`	also looks for a runtime version, use -1 for now.

Returns

int - 0 if not found, 1 if found, 2 if demo.

Table 17. Constants

Name	Value	Description
SUCCESS	0	Macro SUCCESS is used as a return value from helpers indicating successful operation.

6.18. Byte and Bit Functions

These are functions which can be used to assemble byte packets or to read byte packets in a machine independent way.

There are bit vector functions here too.

Classes

ByteBuf

BufWriter

BufReader

Functions

U16PutLE()	Function U16PutLE writes a 16 bit word to a byte array in little endian (lsb then msb, that is, "L"ittle "E"ndian) byte order.
U16GetLE()	Function U16GetLE gets a 16 bit word from a byte array in little endian (lsb then msb, that is, "L"ittle "E"ndian) byte order.
U16PutBE()	Function U16PutBE writes a 16 bit word in big endian fashion.
U16GetBE()	Function U16GetBE gets a 16 bit word from a byte array in big endian byte order.
U32GetLE()	Function U32GetLE gets a 32 bit word from a byte array in little endian (lsb then msb, that is, "L"ittle "E"ndian) byte order.
U32PutLE()	Function U32PutLE writes a 32 bit word to a byte array in little endian (lsb then msb, that is, "L"ittle "E"ndian) byte order.
U32GetBE()	Function U32GetBE gets a 32 bit word from a byte array in big endian (msb then lsb, that is, "B"ig "E"ndian) byte order.
U32PutBE()	Function U32PutLE writes a 32 bit word to a byte array in big endian (msb then lsb, that is, "B"ig "E"ndian) byte order.
FloatGetLE()	Function FloatGetLE de-serializes a 32 bit IEEE floating point value from a little endian format in a byte array.

FloatPutLE()	Function FloatPut serializes a 32 bit IEEE floating point value in little endian format into a byte array.
BitTest()	Function BitTest tests a bit in a uint8_t array.
BitSet()	Function BitSet sets or clears a bit in a uint8_t array.

6.18.1. U16PutLE()

```
uint8_t * U16PutLE(uint8_t *array, uint16_t val)
```

Function U16PutLE writes a 16 bit word to a byte array in little endian (lsb then msb, that is, "L"ittle "E"ndian) byte order.

Parameters

`array`	Where to put the 2 bytes
`val`	The 16 bit value to write

Returns

uint8_t* - the location just after the last placed byte.

6.18.2. U16GetLE()

```
uint16_t U16GetLE(const uint8_t *array)
```

Function U16GetLE gets a 16 bit word from a byte array in little endian (lsb then msb, that is, "L"ittle "E"ndian) byte order.

Parameters

`array`	Where to get the 2 bytes from

Returns

uint16_t - The 16 bit word

6.18.3. U16PutBE()

```
uint8_t * U16PutBE(uint8_t *array, uint16_t val)
```

Function U16PutBE writes a 16 bit word in big endian fashion.

Parameters

`array`	Where to put the 2 bytes
`val`	The 16 bit value to write

Returns

uint8_t* - the location just after the last placed byte.

6.18.4. U16GetBE()

```
uint16_t U16GetBE(const uint8_t *array)
```

Function U16GetBE gets a 16 bit word from a byte array in big endian byte order.

Parameters

`array`	Where to get the 2 bytes from

Returns

uint16_t - The 16 bit word

6.18.5. U32GetLE()

```
uint32_t U32GetLE(const uint8_t *array)
```

Function U32GetLE gets a 32 bit word from a byte array in little endian (lsb then msb, that is, "L"ittle "E"ndian) byte order.

Parameters

`array`	Where to get the 4 bytes from

Returns

uint32_t - The 32 bit word

6.18.6. U32PutLE()

```
uint8_t * U32PutLE(uint8_t *array, uint32_t val)
```

Function U32PutLE writes a 32 bit word to a byte array in little endian (lsb then msb, that is, "L"ittle "E"ndian) byte order.

Parameters

`array`	Where to put the 4 bytes
`val`	The 32 bit value to write

Returns

uint8_t* - the location just after the last placed byte.

6.18.7. U32GetBE()

```
uint32_t U32GetBE(const uint8_t *array)
```

Function U32GetBE gets a 32 bit word from a byte array in big endian (msb then lsb, that is, "B"ig "E"ndian) byte order.

Parameters

`array`	Where to get the 4 bytes from

Returns

uint32_t - The 32 bit word

6.18.8. U32PutBE()

```
uint8_t * U32PutBE(uint8_t *array, uint32_t val)
```

Function U32PutLE writes a 32 bit word to a byte array in big endian (msb then lsb, that is, "B"ig "E"ndian) byte order.

Parameters

`array`	Where to put the 4 bytes
`val`	The 32 bit value to write

Returns

uint8_t* - the location just after the last placed byte.

6.18.9. FloatGetLE()

```
PLCFLOAT FloatGetLE(const uint8_t *array)
```

Function FloatGetLE de-serializes a 32 bit IEEE floating point value from a little endian format in a byte array.

6.18.10. FloatPutLE()

```
uint8_t * FloatPutLE(uint8_t *array, PLCFLOAT val)
```

Function FloatPut serializes a 32 bit IEEE floating point value in little endian format into a byte array.

Returns

uint8_t* - the location just after the last placed byte.

6.18.11. BitTest()

```
bool BitTest(const uint8_t *array, int bitNum)
```

Function BitTest tests a bit in a uint8_t array.

Parameters

`array`	the uint8_t array
`bitNum`	the bit index into the array, 0-7 ⇒ first byte, 8-15⇒second byte, etc.

Returns

bool - true if bit is set, else false.

6.18.12. BitSet()

```
void BitSet(uint8_t *array, int bitNum, bool value)
```

Function BitSet sets or clears a bit in a uint8_t array.

Parameters

`array`	the uint8_t array
`bitNum`	the bit index into the array, 0-7 ⇒ first byte, 8-15⇒second byte, etc.

`value`	non-zero ⇒ set or zero ⇒ clear

6.18.13. ByteBuf

```
#include <byte_bufs.h>

class ByteBuf
```

Class ByteBuf delimits the starting point, ending point, and size of a byte array.

It does not take ownership of such memory, merely points to it. There are no setters among the accessors because it is simple enough to use the assignment operator and overwrite this object with a newly constructed one.

Public Constructors	
	`ByteBuf(uint8_t *, size_t)`
	`ByteBuf(const BufWriter &)`
	`ByteBuf(const BufReader &)`
Public Methods	
uint8_t *	`data() const`
uint8_t *	`end() const`
ssize_t	`size() const`
Protected Variables	
uint8_t *	`start`
uint8_t *	`limit`

Members

ByteBuf

```
ByteBuf(uint8_t * aStart,
        size_t aSize)
```

Parameters

`aStart`

`aSize`

ByteBuf

```
ByteBuf(const BufWriter & aWriter)
```

Parameters

aWriter

ByteBuf

```
ByteBuf(const BufReader & aReader)
```

Parameters

aReader

data

```
uint8_t * data() const
```

end

```
uint8_t * end() const
```

size

```
ssize_t size() const
```

6.18.14. BufWriter

```
#include <byte_bufs.h>

class BufWriter
```

Class BufWriter outlines a writable byte buffer with little endian putters.

It protects from buffer overruns by throwing std::overflow_error. It is useful when serializing data into a byte memory buffer.

Public Constructors	
	`BufWriter(uint8_t *, size_t)` Constructor BufWriter(uint8_t* aStart, size_t aCount)
	`BufWriter(const ByteBuf &)` Constructor BufWriter(const ByteBuf& aBuf)
	`BufWriter()`
Public Operators	
BufWriter &	`operator+=(size_t)` Advance the start of the buffer by the specified number of bytes and trim the capacity().
BufWriter	`operator+(size_t)` Construct a new BufWriter from this one but advance its start by n bytes.
uint8_t &	`operator*()`
BufWriter &	`operator()++` prefix ++
BufWriter	`operator(int)++` postfix ++
BufWriter &	`operator=(const ByteBuf &)`
Public Methods	
uint8_t *	`data() const`
uint8_t *	`end() const`
ssize_t	`capacity() const` Return the unused size of the buffer, the remaining capacity which is empty. A negative value would indicate an overrun, but that also indicates a bug in in this class because protections are everywhere to prevent overruns.
BufWriter &	`put8(uint8_t)` Write a byte to the buffer.
BufWriter &	`put16(uint16_t)` Write a 16 bit integer little endian.
BufWriter &	`put32(uint32_t)` Write a 32 bit integer little endian.

Public Constructors	
BufWriter &	`put64(uint64_t)` Write a 64 bit integer little endian.
BufWriter &	`put_float(float)` Write a 32 bit float little endian.
BufWriter &	`put_double(double)` Write a 64 bit double little endian.
BufWriter &	`put_SHORT_STRING(const std::string &, bool)` Serialize a CIP SHORT_STRING.
BufWriter &	`put_STRING(const std::string &, bool)` Serialize a CIP STRING.
BufWriter &	`put_STRING2(const std::string &)` Serialize a CIP STRING2.
BufWriter &	`put16BE(uint16_t)` Write a 16 bit integer Big Endian.
BufWriter &	`put32BE(uint32_t)` Write a 32 bit integer Big Endian.
BufWriter &	`append(const uint8_t *, size_t)` Write a byte array.
BufWriter &	`append(const BufReader &)` Write the entire contents of a BufReader.
BufWriter &	`fill(size_t, uint8_t)` Write aValue byte to the buffer aCount times.
Protected Variables	
uint8_t *	`start`
uint8_t *	`limit`
Protected Methods	
void	`overrun() const`

Members

BufWriter

```
BufWriter(uint8_t * aStart,
          size_t aCount)
```

Constructor BufWriter(uint8_t* aStart, size_t aCount)

Parameters

aStart	is the start of the memory buffer
aCount	is the number of bytes in the buffer

BufWriter

```
BufWriter(const ByteBuf & aBuf)
```

Constructor BufWriter(const ByteBuf& aBuf)

Parameters

aBuf	is a ByteBuf outlining the start and length of a memory buffer.

BufWriter

```
BufWriter()
```

---

operator+=

```
BufWriter & operator+=(size_t advance)
```

Advance the start of the buffer by the specified number of bytes and trim the capacity().

Parameters

advance

---

operator+

```
BufWriter operator+(size_t n)
```

Construct a new BufWriter from this one but advance its start by n bytes.

Parameters

n

operator*

```
uint8_t & operator*()
```

operator++

```
BufWriter & operator++()
```

prefix ++

operator++

```
BufWriter operator++(int)
```

postfix ++

Parameters

``

operator=

```
BufWriter & operator=(const ByteBuf & aRange)
```

Parameters

aRange

data

```
uint8_t * data() const
```

end

```
uint8_t * end() const
```

capacity

```
ssize_t capacity() const
```

Return the unused size of the buffer, the remaining capacity which is empty. A negative value would indicate an overrun, but that also indicates a bug in in this class because protections are everywhere to prevent overruns.

put8

```
BufWriter & put8(uint8_t aValue)
```

Write a byte to the buffer.

Parameters

`aValue`

put16

```
BufWriter & put16(uint16_t aValue)
```

Write a 16 bit integer little endian.

Parameters

`aValue`

put32

```
BufWriter & put32(uint32_t aValue)
```

Write a 32 bit integer little endian.

Parameters

aValue

---

put64

```
BufWriter & put64(uint64_t aValue)
```

Write a 64 bit integer little endian.

Parameters

aValue

---

put_float

```
BufWriter & put_float(float aValue)
```

Write a 32 bit float little endian.

Parameters

aValue

---

put_double

```
BufWriter & put_double(double aValue)
```

Write a 64 bit double little endian.

Parameters

aValue

---

put_SHORT_STRING

```
BufWriter & put_SHORT_STRING(const std::string & aString,
                             bool doEvenByteCountPadding)
```

Serialize a CIP SHORT_STRING.

Parameters

aString

doEvenByteCountPad
ding

---

put_STRING

```
BufWriter & put_STRING(const std::string & aString,
                       bool doEvenByteCountPadding)
```

Serialize a CIP STRING.

Parameters

aString

doEvenByteCountPad
ding

---

put_STRING2

```
BufWriter & put_STRING2(const std::string & aString)
```

Serialize a CIP STRING2.

Parameters

aString

---

put16BE

```
BufWriter & put16BE(uint16_t aValue)
```

Write a 16 bit integer Big Endian.

Parameters

aValue

---

put32BE

```
BufWriter & put32BE(uint32_t aValue)
```

Write a 32 bit integer Big Endian.

Parameters

`aValue`

---

append

```
BufWriter & append(const uint8_t * aStart,
                   size_t aCount)
```

Write a byte array.

Parameters

`aStart`

`aCount`

---

append

```
BufWriter & append(const BufReader & aReader)
```

Write the entire contents of a BufReader.

Parameters

`aReader`

---

fill

```
BufWriter & fill(size_t aCount,
                 uint8_t aValue = 0)
```

Write aValue byte to the buffer aCount times.

Parameters

`aCount`

`aValue` **(Default value**: `0`**)**

overrun

```
void overrun() const
```

6.18.15. BufReader

```
#include <byte_bufs.h>

class BufReader
```

Class BufReader outlines a read only byte buffer with little endian getters.

It protects from buffer overruns by throwing std::range_error.

Public Constructors	
	`BufReader()`
	`BufReader(const uint8_t *, size_t)`
	`BufReader(const BufWriter &)`
	`BufReader(const ByteBuf &)`
Public Operators	
BufReader &	`operator+=(size_t)` Advance the start of the buffer by the specified number of bytes and trim the size().
BufReader	`operator+(size_t)` Construct a new BufReader from this one but advance its start by n bytes.
BufReader &	`operator()++` prefix ++
BufReader	`operator(int)++` postfix ++
uint8_t	`operator*()`
uint8_t	`operator[](int)`
BufReader &	`operator=(const ByteBuf &)`
Public Methods	
const uint8_t *	`data() const`
const uint8_t *	`end() const`

Public Constructors	
ssize_t	`size() const` Return the un-consumed size of the buffer, the count of bytes remaining in the buffer. A negative value would indicate an overrun, but that also indicates a bug in in this class because protections are everywhere to prevent overruns.
uint8_t	`get8()` Read a byte and return it as unsigned.
uint16_t	`get16()` Read a 16 bit integer little endian first and return it as unsigned.
uint32_t	`get32()` Read a 32 bit integer little endian first and return it as unsigned.
uint64_t	`get64()` Read a 64 bit integer little endian first and return it as unsigned.
float	`get_float()` Read a 32 bit float little endian first.
double	`get_double()` Read a 64 bit double little endian first.
std::string	`get_SHORT_STRING(bool)` Deserialize a CIP SHORT_STRING.
std::string	`get_STRING(bool)` Deserialize a CIP STRING.
std::string	`get_STRING2()` Deserialize a CIP STRING2 and encode the result as UTF8 within a std::string.
uint16_t	`get16BE()` Get a 16 bit integer as Big Endian.
uint32_t	`get32BE()` Get a 32 bit integer as Big Endian.
Protected Variables	
const uint8_t *	`start`
const uint8_t *	`limit`
Protected Methods	
void	`overrun() const`

Members

BufReader

```
BufReader()
```

BufReader

```
BufReader(const uint8_t * aStart,
          size_t aCount)
```

Parameters

aStart

aCount

BufReader

```
BufReader(const BufWriter & aWriter)
```

Parameters

aWriter

BufReader

```
BufReader(const ByteBuf & aBuf)
```

Parameters

aBuf

---

operator+=

```
BufReader & operator+=(size_t advance)
```

Advance the start of the buffer by the specified number of bytes and trim the size().

Parameters

`advance`

---

operator+

```
BufReader operator+(size_t n)
```

Construct a new BufReader from this one but advance its start by n bytes.

Parameters

`n`

---

operator++

```
BufReader & operator++()
```

prefix ++

---

operator++

```
BufReader operator++(int)
```

postfix ++

Parameters

``

---

operator*

```
uint8_t operator*() const
```

---

operator[]

```
uint8_t operator[](int aIndex) const
```

Parameters

`aIndex`

---

operator=

```
BufReader & operator=(const ByteBuf & aRange)
```

Parameters

`aRange`

---

data

```
const uint8_t * data() const
```

---

end

```
const uint8_t * end() const
```

---

size

```
ssize_t size() const
```

Return the un-consumed size of the buffer, the count of bytes remaining in the buffer. A negative value would indicate an overrun, but that also indicates a bug in in this class because protections are everywhere to prevent overruns.

---

get8

```
uint8_t get8()
```

Read a byte and return it as unsigned.

---

get16

```
uint16_t get16()
```

Read a 16 bit integer little endian first and return it as unsigned.

---

get32

```
uint32_t get32()
```

Read a 32 bit integer little endian first and return it as unsigned.

---

get64

```
uint64_t get64()
```

Read a 64 bit integer little endian first and return it as unsigned.

---

get_float

```
float get_float()
```

Read a 32 bit float little endian first.

---

get_double

```
double get_double()
```

Read a 64 bit double little endian first.

---

get_SHORT_STRING

```
std::string get_SHORT_STRING(bool ExpectPossiblePaddingToEvenByteCount)
```

Deserialize a CIP SHORT_STRING.

Parameters

ExpectPossiblePadd
ingToEvenByteCount

---

get_STRING

```
std::string get_STRING(bool ExpectPossiblePaddingToEvenByteCount)
```

Deserialize a CIP STRING.

Parameters

ExpectPossiblePadd
ingToEvenByteCount

---

get_STRING2

```
std::string get_STRING2()
```

Deserialize a CIP STRING2 and encode the result as UTF8 within a std::string.

---

get16BE

```
uint16_t get16BE()
```

Get a 16 bit integer as Big Endian.

---

get32BE

```
uint32_t get32BE()
```

Get a 32 bit integer as Big Endian.

---

overrun

```
void overrun() const
```

6.19. Miscellaneous Functions

Non-Helper Functions that don't fit another category.

Functions

MicroSecsNow()	Function MicroSecsNow returns current relative time in microseconds.
MilliSecsNow()	Function MicroSecsNow returns current relative time in milliseconds.
NanoSecsNow()	Function NanoSecsNow returns current relative time in nanoseconds.
StrPrintf()	Function StrPrintf is like sprintf() but the output is appended to a std::string instead of to a character array.
StrPrintf()	Function StrPrintf is like sprintf() but the output is returned in a std::string instead of being sent to a character array.
STcpy()	Function STcpy copies into a aDatatableString from aSourceString assuming 1 byte = 1 character.
fromST()	Function fromST copies from aDatatableString into a std::string and returns that.
LookupTagAndAddress()	Function LookupTagAndAddress looks up *aTagOrAddress* and outputs 3 results.

Table 18. Typedefs

Name	Definition	Description
USECS	typedef unsigned USECS	Typedef USECS is an unsigned integer earmarked to hold microseconds.
MSECS	typedef unsigned MSECS	Typedef MSECS is an unsigned integer earmarked to hold milliseconds.

6.19.1. MicroSecsNow()

```
USECS MicroSecsNow()
```

Function MicroSecsNow returns current relative time in microseconds.

6.19.2. MilliSecsNow()

```
MSECS MilliSecsNow()
```

Function MicroSecsNow returns current relative time in milliseconds.

6.19.3. NanoSecsNow()

```
uint64_t NanoSecsNow()
```

Function NanoSecsNow returns current relative time in nanoseconds.

6.19.4. StrPrintf()

```
int StrPrintf(std::string *aResult, const char *aFormat,...)
```

Function StrPrintf is like sprintf() but the output is appended to a std::string instead of to a character array.

Parameters

`aResult`	is the string to append to, previous text is not clear()ed.
`aFormat`	is a printf() style format string.
`...`	

Returns

int - the count of bytes appended to the result string, no terminating nul is included.

6.19.5. StrPrintf()

```
std::string StrPrintf(const char *format,...)
```

Function StrPrintf is like sprintf() but the output is returned in a std::string instead of being sent to a character array.

Parameters

`format`	is a printf() style format string.
`...`	

Returns

std::string - the result of the sprintf().

6.19.6. STcpy()

```
void STcpy(STRING *aDatatableString, const char *aSourceString)
```

Function STcpy copies into a aDatatableString from aSourceString assuming 1 byte = 1 character.

Parameters

`aDatatableString`	is the destination STRING, its length will also be set.
`aSourceString`	is a C string holding the 8 bit source string.

6.19.7. fromST()

```
std::string fromST(const STRING *aDatatableString)
```

Function fromST copies from aDatatableString into a std::string and returns that.

6.19.8. LookupTagAndAddress()

```
void LookupTagAndAddress(const char *aTagOrAddress, std::string *aTag, std::string
*aAddress, BINADDR *aBinAddr)
```

Function LookupTagAndAddress looks up *aTagOrAddress* and outputs 3 results.

Parameters

`aTagOrAddress`	- can be either a tag or address that should be looked up.
`aTag`	- where to put the tag for aTagOrAddress
`aAddress`	- where to put the beatified address for aTagOrAddress
`aBinAddr`	- where to put the binary form of the address

Throws

ERROR — if no sense can be made of *aTagOrAddress* because it cannot be parsed as an address and cannot be found as a tag.

6.20. Datatable Objects for C++

These are classes and defines that:

1. make C++ access to the datatable extremely easy.

2. make writing a TLM in C++ extremely easy.

Classes

DT_OBJECT

DT_INT16

DT_BIT

DT_FLOAT32

DT_ARRAY

DT_INT16_ARRAY

DT_FLOAT32_ARRAY

DT_STRUCT

TypeName

DT_PTR

DT_OBJECTS

6.20.1. ENABLE_DT_PTR()

```
#define ENABLE_DT_PTR(A) template<> struct TypeName<A>         { static const char
*Name() { return #A; } };
```

Use this once for each DT_PTR type. It provides a TypeName specialization for each kind of DT_PTR you intend to use.

6.20.2. DT_OBJECT

```
#include <dt_objects.h>

class DT_OBJECT
```

Class DT_OBJECT is an abstract base class which defines some interface functions for all derived datatable access objects.

It provides foundational support for high speed direct memory access to datatable objects in RUN or TEST modes only (not PROGRAM mode).

Public Enclosed Types	
enum	FMT_CTL Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.

Public Enclosed Types	
Public Constructors	
	`DT_OBJECT(const char *, TLM *)` Constructor accepts *aTagOrAddress* and registers this object with the TLM's DT_OBJECT registry.
Public Destructors	
	`~DT_OBJECT()`
Public Static Methods	
static DT_OBJECT *	`Make(const char *, TLM *)` Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.
Public Methods	
const char *	`ClassName() const` Function ClassName returns the name of this class.
const std::string &	`Address() const` Function Address returns the plc memory address of this object.
const std::string &	`Tag() const` Function Tag returns the tag of this object.
const std::string &	`Spec() const` Function Spec returns the specification given as *aTagOrAddress* to the constructor.
const std::string &	`Name() const` Function Name returns the tag if known, else the address if known, else the spec.
void	`ResolveAndVerify()` Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.
std::string	`Format(int) const` Function Format prints information characterizing this object into a std::string and returns that string.
Protected Constructors	
	`DT_OBJECT(const DT_OBJECT &)`
Protected Operators	
DT_OBJECT &	`operator=(const DT_OBJECT &)`
Protected Variables	

Public Enclosed Types	
TLM &	`tlm` Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.
void *	`pointer` to implementation or datatable memory
std::string	`spec` a copy of aTagOrAddress from constructor.
std::string	`address` the SoftPLC memory address of this word.
std::string	`tag` the SoftPLC tag of this word, if any.
BINADDR	`binAddr` the SoftPLC address in binary.

FMT_CTL

```
#include <opt/softplc/c++-toolkit-5/h/dt_objects.h>

enum DT_OBJECT::FMT_CTL
```

Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.

TYPE = (1<<0)	show classname
SPEC = (1<<1)	show constructor's aTagOrAddress
TAG = (1<<2)	show tag
ADDR = (1<<3)	show address

Members

DT_OBJECT

```
DT_OBJECT(const char * aTagOrAddress,
          TLM * aTLM)
```

Constructor accepts *aTagOrAddress* and registers this object with the TLM's DT_OBJECT registry.

Parameters

aTagOrAddress	is a tag from the descriptor table or a datatable memory address.
aTLM	is a copy of global TLM___ and is provided by the compiler, so don't provide it.

~DT_OBJECT

```
~DT_OBJECT()
```

Make

```
static DT_OBJECT * Make(const char * aTagOrAddress,
                        TLM * aTLM)
```

Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.

Parameters

aTagOrAddress	can be either a tag or a PLC address.
aTLM	is the calling TLM, and can normally be omitted so that the default value is used.

Returns

DT_OBJECT* - this is a pointer to an instance of a class derived from DT_OBJECT, according to the datatable type of the tag or address. Caller owns the DT_OBJECT derivative upon return.

ClassName

```
const char * ClassName() const
```

Function ClassName returns the name of this class.

Address

```
const std::string & Address() const
```

Function Address returns the plc memory address of this object.

Tag

```
const std::string & Tag() const
```

Function Tag returns the tag of this object.

Spec

```
const std::string & Spec() const
```

Function Spec returns the specification given as *aTagOrAddress* to the constructor.

Name

```
const std::string & Name() const
```

Function Name returns the tag if known, else the address if known, else the spec.

ResolveAndVerify

```
void ResolveAndVerify()
```

Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.

Throws

ERROR — if the datatable memory does not exist or cannot be tested because of a missing Tag.

Format

```
std::string Format(int aCtl = FMT_DEFAULT) const
```

Function Format prints information characterizing this object into a std::string and returns that string.

Parameters

aCtl	is a bitmapped int assembled by OR-ing together values from Enum FMT_CTL. (**Default value**: FMT_DEFAULT)

DT_OBJECT

```
DT_OBJECT(const DT_OBJECT &)
```

Parameters

``

operator=

```
DT_OBJECT & operator=(const DT_OBJECT &)
```

Parameters

``

```
TLM & tlm
```

Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.

It is used by the logger TLM. which TLM have I been assigned to?

6.20.3. DT_INT16

```
#include <dt_objects.h>

class DT_INT16
```

Class DT_INT16 allows testing and setting of a specific datatable PLCINT.

Usage Example

```
#include "dt_objects.h"

// DT_INT16's are used simplest as globals.
// Here one is constructed by tag, the other by address.
DT_INT16  FlowRate( "FlowRate" );    // tag must exist in descriptor table
DT_INT16  Pressure( "N77:44" );      // fixed address
```

```
int Example()
{
     // Access the DT_INT16's like they were simple PLCINT's.  Understand
     // that this is C++ magic accessing live and current datatable values.
     return FlowRate * Pressure;
}
```

Public Enclosed Types	
enum	`FMT_CTL` Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.
Public Constructors	
	`DT_INT16(const char *, TLM *)` Constructor that creates a DT_INT16 from *aTagOrAddress*.
Public Operators	
PLCINT &	`operator PLCINT &()` Function operator PLCINT& () const provides a casting operator so that this object can be used directly in C++ expressions where a PLCINT could be used.
PLCINT	`operator=(int)` Function operator=(int aValue) overrides the assignment operator allowing normal C++ syntax be used to assign to a PLCINT in the datatable.
Public Static Methods	
static DT_OBJECT *	`Make(const char *, TLM *)` Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.
Public Methods	
const char *	`ClassName() const` Function ClassName returns the name of this class.
void	`ResolveAndVerify()` Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.
const std::string &	`Address() const` Function Address returns the plc memory address of this object.
const std::string &	`Tag() const` Function Tag returns the tag of this object.

Public Enclosed Types	
const std::string &	`Spec() const` Function Spec returns the specification given as *aTagOrAddress* to the constructor.
const std::string &	`Name() const` Function Name returns the tag if known, else the address if known, else the spec.
std::string	`Format(int) const` Function Format prints information characterizing this object into a std::string and returns that string.
Protected Variables	
TLM &	`tlm` Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.
void *	`pointer` to implementation or datatable memory
std::string	`spec` a copy of aTagOrAddress from constructor.
std::string	`address` the SoftPLC memory address of this word.
std::string	`tag` the SoftPLC tag of this word, if any.
BINADDR	`binAddr` the SoftPLC address in binary.

FMT_CTL

```
#include <opt/softplc/c++-toolkit-5/h/dt_objects.h>

enum DT_INT16::FMT_CTL
```

Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.

TYPE = (1<<0)	show classname
SPEC = (1<<1)	show constructor's aTagOrAddress
TAG = (1<<2)	show tag

ADDR = (1<<3)	show address

Members

DT_INT16

```
DT_INT16(const char * aTagOrAddress,
         TLM * aTLM)
```

Constructor that creates a DT_INT16 from *aTagOrAddress*.

Parameters

`aTagOrAddress`	is a tag string as it exists in the descriptor table, or valid datatable address string.
`aTLM`	

---

operator PLCINT &

```
PLCINT & operator PLCINT &() const
```

Function operator PLCINT& () const provides a casting operator so that this object can be used directly in C++ expressions where a PLCINT could be used.

Note that any such C++ expression should only be executed while SoftPLC is in a RUN mode, because before entering a RUN mode, the ResolveAndVerify() function will have been called to update the pointer.

---

operator=

```
PLCINT operator=(int aValue)
```

Function operator=(int aValue) overrides the assignment operator allowing normal C++ syntax be used to assign to a PLCINT in the datatable.

Parameters

`aValue`	is a PLCINT or an int that you want to assign to the datatable. If its an int, there will be a truncation down to 16 bits.

Returns

PLCINT - the same as aValue, but down-casted to 16 bits.

Make

```
static DT_OBJECT * Make(const char * aTagOrAddress,
                        TLM * aTLM)
```

Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.

Parameters

aTagOrAddress	can be either a tag or a PLC address.
aTLM	is the calling TLM, and can normally be omitted so that the default value is used.

Returns

DT_OBJECT* - this is a pointer to an instance of a class derived from DT_OBJECT, according to the datatable type of the tag or address. Caller owns the DT_OBJECT derivative upon return.

ClassName

```
const char * ClassName() const
```

Function ClassName returns the name of this class.

ResolveAndVerify

```
void ResolveAndVerify()
```

Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.

Throws

ERROR — if the datatable memory does not exist or cannot be tested because of a missing Tag.

Address

```
const std::string & Address() const
```

Function Address returns the plc memory address of this object.

Tag

```
const std::string & Tag() const
```

Function Tag returns the tag of this object.

Spec

```
const std::string & Spec() const
```

Function Spec returns the specification given as *aTagOrAddress* to the constructor.

Name

```
const std::string & Name() const
```

Function Name returns the tag if known, else the address if known, else the spec.

Format

```
std::string Format(int aCtl = FMT_DEFAULT) const
```

Function Format prints information characterizing this object into a std::string and returns that string.

Parameters

`aCtl`	is a bitmapped int assembled by OR-ing together values from Enum FMT_CTL. (**Default value**: `FMT_DEFAULT`)

```
TLM & tlm
```

Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.

It is used by the logger TLM. which TLM have I been assigned to?

6.20.4. DT_BIT

```
#include <dt_objects.h>

class DT_BIT
```

Class DT_BIT allows testing and setting of a specific datatable bit.

Public Enclosed Types	
enum	FMT_CTL Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.
Public Constructors	
	DT_BIT(const char *, TLM *) Constructor DT_BIT(const char* aTagOrAddress) creates a DT_BIT and registers it.
Public Operators	
bool	operator bool() Function operator bool provides a cast operator so that this object can be used directly in C++ expressions where a bool could be used.
DT_BIT &	operator=(bool) Operator = (DT_BIT) assignment overload assigns a value to the datatable memory referenced by this object.
Public Static Methods	
static DT_OBJECT *	Make(const char *, TLM *) Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.
Public Methods	
const char *	ClassName() const Function ClassName returns the name of this class.
void	ResolveAndVerify() Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.
const std::string &	Address() const Function Address returns the plc memory address of this object.
const std::string &	Tag() const Function Tag returns the tag of this object.

Public Enclosed Types	
const std::string &	`Spec() const` Function Spec returns the specification given as *aTagOrAddress* to the constructor.
const std::string &	`Name() const` Function Name returns the tag if known, else the address if known, else the spec.
std::string	`Format(int) const` Function Format prints information characterizing this object into a std::string and returns that string.
Protected Variables	
PLCINT	`mask` a bit test mask with a single bit turned on
TLM &	`tlm` Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.
void *	`pointer` to implementation or datatable memory
std::string	`spec` a copy of aTagOrAddress from constructor.
std::string	`address` the SoftPLC memory address of this word.
std::string	`tag` the SoftPLC tag of this word, if any.
BINADDR	`binAddr` the SoftPLC address in binary.

FMT_CTL

```
#include <opt/softplc/c++-toolkit-5/h/dt_objects.h>

enum DT_BIT::FMT_CTL
```

Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.

TYPE = (1<<0)	show classname

SPEC = (1<<1)	show constructor's aTagOrAddress
TAG = (1<<2)	show tag
ADDR = (1<<3)	show address

Members

DT_BIT

```
DT_BIT(const char * aTagOrAddress,
       TLM * aTLM)
```

Constructor DT_BIT(const char* aTagOrAddress) creates a DT_BIT and registers it.

Parameters

aTagOrAddress	The TAG as expected in the descriptor table of the SOFTPLC.APP, or valid PLC memory address string.
aTLM	

---

operator bool

```
bool operator bool() const
```

Function operator bool provides a cast operator so that this object can be used directly in C++ expressions where a bool could be used.

Note that any such C++ expression should only be executed while SoftPLC is in a RUN mode, because before entering a RUN mode, the ResolveAndVerify() function will have been called to update the pointer.

---

operator=

```
DT_BIT & operator=(bool rval)
```

Operator = (DT_BIT) assignment overload assigns a value to the datatable memory referenced by this object.

Parameters

rval	a bool expression to assign to this memory bit.

Returns

DT_BIT& - this object

Make

```
static DT_OBJECT * Make(const char * aTagOrAddress,
                        TLM * aTLM)
```

Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.

Parameters

aTagOrAddress	can be either a tag or a PLC address.
aTLM	is the calling TLM, and can normally be omitted so that the default value is used.

Returns

DT_OBJECT* - this is a pointer to an instance of a class derived from DT_OBJECT, according to the datatable type of the tag or address. Caller owns the DT_OBJECT derivative upon return.

ClassName

```
const char * ClassName() const
```

Function ClassName returns the name of this class.

ResolveAndVerify

```
void ResolveAndVerify()
```

Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.

Throws

ERROR — if the datatable memory does not exist or cannot be tested because of a missing Tag.

Address

```
const std::string & Address() const
```

Function Address returns the plc memory address of this object.

Tag

```
const std::string & Tag() const
```

Function Tag returns the tag of this object.

Spec

```
const std::string & Spec() const
```

Function Spec returns the specification given as *aTagOrAddress* to the constructor.

Name

```
const std::string & Name() const
```

Function Name returns the tag if known, else the address if known, else the spec.

Format

```
std::string Format(int aCtl = FMT_DEFAULT) const
```

Function Format prints information characterizing this object into a std::string and returns that string.

Parameters

`aCtl`	is a bitmapped int assembled by OR-ing together values from Enum FMT_CTL. (**Default value**: `FMT_DEFAULT`)

```
TLM & tlm
```

Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.

It is used by the logger TLM. which TLM have I been assigned to?

6.20.5. DT_FLOAT32

```
#include <dt_objects.h>

class DT_FLOAT32
```

Class DT_FLOAT32 allows testing and setting of a specific datatable float within SoftPLC.

Public Enclosed Types	
enum	FMT_CTL Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.
Public Constructors	
	DT_FLOAT32(const char *, TLM *) Constructor DT_FLOAT32(const char* aAddress) creates a DT_FLOAT32, initializes the binAddr, and notifies the master object repository for this TLM about this object's existence.
Public Operators	
PLCFLOAT &	operator PLCFLOAT &() Function operator PLCFLOAT provides a cast operator so that this object can be used directly in C++ expressions where a PLCFLOAT could be used.
PLCFLOAT	operator=(double) Function operator=(double aValue) is an assignment override that enables normal C++ language syntax to set the value of a PLCFLOAT residing in the datatable.
Public Static Methods	
static DT_OBJECT *	Make(const char *, TLM *) Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.
Public Methods	
const char *	ClassName() const Function ClassName returns the name of this class.
void	ResolveAndVerify() Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.

Public Enclosed Types	
const std::string &	`Address() const` Function Address returns the plc memory address of this object.
const std::string &	`Tag() const` Function Tag returns the tag of this object.
const std::string &	`Spec() const` Function Spec returns the specification given as *aTagOrAddress* to the constructor.
const std::string &	`Name() const` Function Name returns the tag if known, else the address if known, else the spec.
std::string	`Format(int) const` Function Format prints information characterizing this object into a std::string and returns that string.
Protected Variables	
TLM &	`tlm` Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.
void *	`pointer` to implementation or datatable memory
std::string	`spec` a copy of aTagOrAddress from constructor.
std::string	`address` the SoftPLC memory address of this word.
std::string	`tag` the SoftPLC tag of this word, if any.
BINADDR	`binAddr` the SoftPLC address in binary.

FMT_CTL

```
#include <opt/softplc/c++-toolkit-5/h/dt_objects.h>

enum DT_FLOAT32::FMT_CTL
```

Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that

function's output.

TYPE = (1<<0)	show classname
SPEC = (1<<1)	show constructor's aTagOrAddress
TAG = (1<<2)	show tag
ADDR = (1<<3)	show address

Members

DT_FLOAT32

```
DT_FLOAT32(const char * aTagOrAddress,
           TLM * aTLM)
```

Constructor DT_FLOAT32(const char* aAddress) creates a DT_FLOAT32, initializes the binAddr, and notifies the master object repository for this TLM about this object's existence.

Parameters

aTagOrAddress	The TAG as expected in the descriptor table of the SOFTPLC.APP, or valid PLC memory address string.
aTLM	

---

operator PLCFLOAT &

```
PLCFLOAT & operator PLCFLOAT &() const
```

Function operator PLCFLOAT provides a cast operator so that this object can be used directly in C++ expressions where a PLCFLOAT could be used.

Note that any such C++ expression should only be executed while SoftPLC is in a RUN mode, because before entering a RUN mode, the ResolveAndVerify() function will have been called to update the pointer.

---

operator=

```
PLCFLOAT operator=(double aValue)
```

Function operator=(double aValue) is an assignment override that enables normal C++ language syntax to set the value of a PLCFLOAT residing in the datatable.

Parameters

`aValue`	is either PLCFLOAT, a float, or a double.

Returns

PLCFLOAT - potentially down-casted from double.

Make

```
static DT_OBJECT * Make(const char * aTagOrAddress,
                        TLM * aTLM)
```

Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.

Parameters

`aTagOrAddress`	can be either a tag or a PLC address.
`aTLM`	is the calling TLM, and can normally be omitted so that the default value is used.

Returns

DT_OBJECT* - this is a pointer to an instance of a class derived from DT_OBJECT, according to the datatable type of the tag or address. Caller owns the DT_OBJECT derivative upon return.

ClassName

```
const char * ClassName() const
```

Function ClassName returns the name of this class.

ResolveAndVerify

```
void ResolveAndVerify()
```

Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.

Throws

ERROR — if the datatable memory does not exist or cannot be tested because of a missing Tag.

Address

```
const std::string & Address() const
```

Function Address returns the plc memory address of this object.

Tag

```
const std::string & Tag() const
```

Function Tag returns the tag of this object.

Spec

```
const std::string & Spec() const
```

Function Spec returns the specification given as *aTagOrAddress* to the constructor.

Name

```
const std::string & Name() const
```

Function Name returns the tag if known, else the address if known, else the spec.

Format

```
std::string Format(int aCtl = FMT_DEFAULT) const
```

Function Format prints information characterizing this object into a std::string and returns that string.

Parameters

`aCtl`	is a bitmapped int assembled by OR-ing together values from Enum FMT_CTL. (**Default value**: `FMT_DEFAULT`)

```
TLM & tlm
```

Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.

It is used by the logger TLM. which TLM have I been assigned to?

6.20.6. DT_ARRAY

```
#include <dt_objects.h>

class DT_ARRAY
```

Class DT_ARRAY is base class that provides a window into the datatable for a block of PLCINTs or PLCFLOATs.

Public Enclosed Types	
enum	ELEM_TYPE
enum	FMT_CTL Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.
Public Constructors	
	DT_ARRAY(const char *, unsigned, TLM *)
Public Static Methods	
static DT_OBJECT *	Make(const char *, TLM *) Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.
Public Methods	
const char *	ClassName() const Function ClassName returns the name of this class.
unsigned	Length() const Function Length returns the count of elements in this array.
void *	Data() const
ELEM_TYPE	Type() const
unsigned	ByteCount() const
const std::string &	Address() const Function Address returns the plc memory address of this object.

Public Enclosed Types	
const std::string &	`Tag() const` Function Tag returns the tag of this object.
const std::string &	`Spec() const` Function Spec returns the specification given as *aTagOrAddress* to the constructor.
const std::string &	`Name() const` Function Name returns the tag if known, else the address if known, else the spec.
void	`ResolveAndVerify()` Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.
std::string	`Format(int) const` Function Format prints information characterizing this object into a std::string and returns that string.
Protected Variables	
unsigned	`count`
TLM &	`tlm` Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.
void *	`pointer` to implementation or datatable memory
std::string	`spec` a copy of aTagOrAddress from constructor.
std::string	`address` the SoftPLC memory address of this word.
std::string	`tag` the SoftPLC tag of this word, if any.
BINADDR	`binAddr` the SoftPLC address in binary.

ELEM_TYPE

```
#include <opt/softplc/c++-toolkit-5/h/dt_objects.h>

enum DT_ARRAY::ELEM_TYPE
```

ELEM_INT16	
ELEM_FLOAT32	

FMT_CTL

```
#include <opt/softplc/c++-toolkit-5/h/dt_objects.h>

enum DT_ARRAY::FMT_CTL
```

Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.

TYPE = (1<<0)	show classname
SPEC = (1<<1)	show constructor's aTagOrAddress
TAG = (1<<2)	show tag
ADDR = (1<<3)	show address

Members

DT_ARRAY

```
DT_ARRAY(const char * aTagOrAddress,
         unsigned aElementCount,
         TLM * aTLM)
```

Parameters

`aTagOrAddress`

`aElementCount`

`aTLM`

Make

```
static DT_OBJECT * Make(const char * aTagOrAddress,
                        TLM * aTLM)
```

Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.

Parameters

`aTagOrAddress` can be either a tag or a PLC address.

`aTLM` is the calling TLM, and can normally be omitted so that the default value is used.

Returns

DT_OBJECT* - this is a pointer to an instance of a class derived from DT_OBJECT, according to the datatable type of the tag or address. Caller owns the DT_OBJECT derivative upon return.

ClassName

```
const char * ClassName() const
```

Function ClassName returns the name of this class.

Length

```
unsigned Length() const
```

Function Length returns the count of elements in this array.

This will always be whatever *aElementCount* was in the constructor.

Data

```
void * Data() const
```

Type

```
ELEM_TYPE Type() const
```

ByteCount

```
unsigned ByteCount() const
```

Address

```
const std::string & Address() const
```

Function Address returns the plc memory address of this object.

Tag

```
const std::string & Tag() const
```

Function Tag returns the tag of this object.

Spec

```
const std::string & Spec() const
```

Function Spec returns the specification given as *aTagOrAddress* to the constructor.

Name

```
const std::string & Name() const
```

Function Name returns the tag if known, else the address if known, else the spec.

ResolveAndVerify

```
void ResolveAndVerify()
```

Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.

Throws

ERROR — if the datatable memory does not exist or cannot be tested because of a missing Tag.

Format

```
std::string Format(int aCtl = FMT_DEFAULT) const
```

Function Format prints information characterizing this object into a std::string and returns that string.

Parameters

aCtl	is a bitmapped int assembled by OR-ing together values from Enum FMT_CTL. (**Default value**: FMT_DEFAULT)

```
TLM & tlm
```

Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.

It is used by the logger TLM. which TLM have I been assigned to?

6.20.7. DT_INT16_ARRAY

```
#include <dt_objects.h>

class DT_INT16_ARRAY
```

Class DT_INT16_ARRAY is used to reference a block of consecutive PLCINTSs in a single datatable file.

Public Enclosed Types	
enum	ELEM_TYPE
enum	FMT_CTL Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.
Public Constructors	
	DT_INT16_ARRAY(const char *, unsigned, TLM *) Constructor.
Public Operators	
const PLCINT &	operator[](unsigned) Function operator[](unsigned aIndex) returns a reference to a PLCINT datatable word iff aIndex is within range.
PLCINT &	operator[](unsigned)

Public Enclosed Types	
Public Static Methods	
static DT_OBJECT *	`Make(const char *, TLM *)` Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.
Public Methods	
const char *	`ClassName() const` Function ClassName returns the name of this class.
std::string	`Format(int) const` Function Format prints information characterizing this object into a std::string and returns that string.
void	`ResolveAndVerify()` Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.
ELEM_TYPE	`Type() const`
unsigned	`ByteCount() const`
unsigned	`Length() const` Function Length returns the count of elements in this array.
void *	`Data() const`
const std::string &	`Address() const` Function Address returns the plc memory address of this object.
const std::string &	`Tag() const` Function Tag returns the tag of this object.
const std::string &	`Spec() const` Function Spec returns the specification given as *aTagOrAddress* to the constructor.
const std::string &	`Name() const` Function Name returns the tag if known, else the address if known, else the spec.
Protected Variables	
unsigned	`count`
TLM &	`tlm` Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.

Public Enclosed Types	
void *	`pointer` to implementation or datatable memory
std::string	`spec` a copy of aTagOrAddress from constructor.
std::string	`address` the SoftPLC memory address of this word.
std::string	`tag` the SoftPLC tag of this word, if any.
BINADDR	`binAddr` the SoftPLC address in binary.

ELEM_TYPE

```
#include <opt/softplc/c++-toolkit-5/h/dt_objects.h>

enum DT_INT16_ARRAY::ELEM_TYPE
```

ELEM_INT16	
ELEM_FLOAT32	

FMT_CTL

```
#include <opt/softplc/c++-toolkit-5/h/dt_objects.h>

enum DT_INT16_ARRAY::FMT_CTL
```

Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.

TYPE = (1<<0)	show classname
SPEC = (1<<1)	show constructor's aTagOrAddress
TAG = (1<<2)	show tag
ADDR = (1<<3)	show address

Members

DT_INT16_ARRAY

```
DT_INT16_ARRAY(const char * aTagOrAddress,
               unsigned aElementCount,
               TLM * aTLM)
```

Constructor.

Parameters

aTagOrAddress	is a tag from the descriptor table that refers to an integer datatable word, and must coincide with the first element of the array.
aElementCount	is the number of elements in this array, may not exceed 10,000 which is the maximum size of an integer datatable file.
aTLM	

---

operator[]

```
const PLCINT & operator[](unsigned aIndex) const
```

Function operator[](unsigned aIndex) returns a reference to a PLCINT datatable word iff aIndex is within range.

Parameters

aIndex

Throws

ERROR — if aIndex is out of range and this will fault SoftPLC.

---

operator[]

```
PLCINT & operator[](unsigned aIndex)
```

Parameters

aIndex

Make

```
static DT_OBJECT * Make(const char * aTagOrAddress,
                        TLM * aTLM)
```

Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.

Parameters

`aTagOrAddress`	can be either a tag or a PLC address.
`aTLM`	is the calling TLM, and can normally be omitted so that the default value is used.

Returns

DT_OBJECT* - this is a pointer to an instance of a class derived from DT_OBJECT, according to the datatable type of the tag or address. Caller owns the DT_OBJECT derivative upon return.

ClassName

```
const char * ClassName() const
```

Function ClassName returns the name of this class.

Format

```
std::string Format(int aCtl = FMT_DEFAULT) const
```

Function Format prints information characterizing this object into a std::string and returns that string.

Parameters

`aCtl`	is a bitmapped int assembled by OR-ing together values from Enum FMT_CTL. (**Default value**: `FMT_DEFAULT`)

ResolveAndVerify

```
void ResolveAndVerify()
```

Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.

Throws

`ERROR` — if the datatable memory does not exist or cannot be tested because of a missing Tag.

Type

```
ELEM_TYPE Type() const
```

ByteCount

```
unsigned ByteCount() const
```

Length

```
unsigned Length() const
```

Function Length returns the count of elements in this array.

This will always be whatever *aElementCount* was in the constructor.

Data

```
void * Data() const
```

Address

```
const std::string & Address() const
```

Function Address returns the plc memory address of this object.

Tag

```
const std::string & Tag() const
```

Function Tag returns the tag of this object.

Spec

```
const std::string & Spec() const
```

Function Spec returns the specification given as *aTagOrAddress* to the constructor.

Name

```
const std::string & Name() const
```

Function Name returns the tag if known, else the address if known, else the spec.

```
TLM & tlm
```

Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.

It is used by the logger TLM. which TLM have I been assigned to?

6.20.8. DT_FLOAT32_ARRAY

```
#include <dt_objects.h>

class DT_FLOAT32_ARRAY
```

Class DT_FLOAT32_ARRAY is used to reference a block of consecutive PLCFLOATSs in a single datatable file.

Public Enclosed Types	
enum	ELEM_TYPE
enum	FMT_CTL Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.
Public Constructors	
	DT_FLOAT32_ARRAY(const char *, unsigned, TLM *) Constructor.
Public Operators	

Public Enclosed Types	
const PLCFLOAT &	`operator[](unsigned)` Function operator[](unsigned aIndex) returns a reference to a PLCFLOAT datatable word iff aIndex is within range.
PLCFLOAT &	`operator[](unsigned)`
Public Static Methods	
static DT_OBJECT *	`Make(const char *, TLM *)` Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.
Public Methods	
const char *	`ClassName() const` Function ClassName returns the name of this class.
std::string	`Format(int) const` Function Format prints information characterizing this object into a std::string and returns that string.
void	`ResolveAndVerify()` Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.
ELEM_TYPE	`Type() const`
unsigned	`ByteCount() const`
unsigned	`Length() const` Function Length returns the count of elements in this array.
void *	`Data() const`
const std::string &	`Address() const` Function Address returns the plc memory address of this object.
const std::string &	`Tag() const` Function Tag returns the tag of this object.
const std::string &	`Spec() const` Function Spec returns the specification given as *aTagOrAddress* to the constructor.
const std::string &	`Name() const` Function Name returns the tag if known, else the address if known, else the spec.
Protected Variables	
unsigned	`count`

Public Enclosed Types	
TLM &	`tlm` Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.
void *	`pointer` to implementation or datatable memory
std::string	`spec` a copy of aTagOrAddress from constructor.
std::string	`address` the SoftPLC memory address of this word.
std::string	`tag` the SoftPLC tag of this word, if any.
BINADDR	`binAddr` the SoftPLC address in binary.

ELEM_TYPE

```
#include <opt/softplc/c++-toolkit-5/h/dt_objects.h>

enum DT_FLOAT32_ARRAY::ELEM_TYPE
```

ELEM_INT16	
ELEM_FLOAT32	

FMT_CTL

```
#include <opt/softplc/c++-toolkit-5/h/dt_objects.h>

enum DT_FLOAT32_ARRAY::FMT_CTL
```

Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.

TYPE = (1<<0)	show classname
SPEC = (1<<1)	show constructor's aTagOrAddress
TAG = (1<<2)	show tag
ADDR = (1<<3)	show address

Members

DT_FLOAT32_ARRAY

```
DT_FLOAT32_ARRAY(const char * aTagOrAddress,
                 unsigned aElementCount,
                 TLM * aTLM)
```

Constructor.

Parameters

aTagOrAddress	is a tag from the descriptor table that refers to a float datatable word, and must coincide with the first element of the array.
aElementCount	is the number of elements in this array, may not exceed 10,000 which is the maximum size of an integer datatable file.
aTLM	

---

operator[]

```
const PLCFLOAT & operator[](unsigned aIndex) const
```

Function operator[](unsigned aIndex) returns a reference to a PLCFLOAT datatable word iff aIndex is within range.

Parameters

aIndex

Throws

ERROR — if aIndex is out of range and this will fault SoftPLC.

---

operator[]

```
PLCFLOAT & operator[](unsigned aIndex)
```

Parameters

aIndex

Make

```
static DT_OBJECT * Make(const char * aTagOrAddress,
                        TLM * aTLM)
```

Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.

Parameters

`aTagOrAddress`	can be either a tag or a PLC address.
`aTLM`	is the calling TLM, and can normally be omitted so that the default value is used.

Returns

DT_OBJECT* - this is a pointer to an instance of a class derived from DT_OBJECT, according to the datatable type of the tag or address. Caller owns the DT_OBJECT derivative upon return.

ClassName

```
const char * ClassName() const
```

Function ClassName returns the name of this class.

Format

```
std::string Format(int aCtl = FMT_DEFAULT) const
```

Function Format prints information characterizing this object into a std::string and returns that string.

Parameters

`aCtl`	is a bitmapped int assembled by OR-ing together values from Enum FMT_CTL. (**Default value**: `FMT_DEFAULT`)

ResolveAndVerify

```
void ResolveAndVerify()
```

Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.

Throws

`ERROR` — if the datatable memory does not exist or cannot be tested because of a missing Tag.

Type

```
ELEM_TYPE Type() const
```

ByteCount

```
unsigned ByteCount() const
```

Length

```
unsigned Length() const
```

Function Length returns the count of elements in this array.

This will always be whatever *aElementCount* was in the constructor.

Data

```
void * Data() const
```

Address

```
const std::string & Address() const
```

Function Address returns the plc memory address of this object.

Tag

```
const std::string & Tag() const
```

Function Tag returns the tag of this object.

Spec

```
const std::string & Spec() const
```

Function Spec returns the specification given as *aTagOrAddress* to the constructor.

Name

```
const std::string & Name() const
```

Function Name returns the tag if known, else the address if known, else the spec.

```
TLM & tlm
```

Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.

It is used by the logger TLM. which TLM have I been assigned to?

6.20.9. DT_STRUCT

```
#include <dt_objects.h>

class DT_STRUCT
```

Class DT_STRUCT is a base class used by the template class DT_PTR in order to map any kind of structure onto a SoftPLC datatable file.

Public Enclosed Types	
enum	FMT_CTL Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.
Public Constructors	
	DT_STRUCT(const char *, TLM *) Constructor used by template DT_PTR to make an arbitrary structure on a range of datatable words.
Public Static Methods	

Public Enclosed Types	
static DT_OBJECT *	`Make(const char *, TLM *)` Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.
Public Methods	
const char *	`ClassName() const` Function ClassName returns the name of this class.
void	`ResolveAndVerify()` Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.
const std::string &	`Address() const` Function Address returns the plc memory address of this object.
const std::string &	`Tag() const` Function Tag returns the tag of this object.
const std::string &	`Spec() const` Function Spec returns the specification given as *aTagOrAddress* to the constructor.
const std::string &	`Name() const` Function Name returns the tag if known, else the address if known, else the spec.
std::string	`Format(int) const` Function Format prints information characterizing this object into a std::string and returns that string.
Protected Variables	
TLM &	`tlm` Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.
void *	`pointer` to implementation or datatable memory
std::string	`spec` a copy of aTagOrAddress from constructor.
std::string	`address` the SoftPLC memory address of this word.
std::string	`tag` the SoftPLC tag of this word, if any.

Public Enclosed Types	
BINADDR	binAddr the SoftPLC address in binary.
Protected Methods	
size_t	size_of() const

FMT_CTL

```
#include <opt/softplc/c++-toolkit-5/h/dt_objects.h>

enum DT_STRUCT::FMT_CTL
```

Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.

TYPE = (1<<0)	show classname
SPEC = (1<<1)	show constructor's aTagOrAddress
TAG = (1<<2)	show tag
ADDR = (1<<3)	show address

Members

DT_STRUCT

```
DT_STRUCT(const char * aTagOrAddress,
          TLM * aTLM)
```

Constructor used by template DT_PTR to make an arbitrary structure on a range of datatable words.

Parameters

aTagOrAddress

aTLM

Make

```
static DT_OBJECT * Make(const char * aTagOrAddress,
                        TLM * aTLM)
```

Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.

Parameters

aTagOrAddress	can be either a tag or a PLC address.
aTLM	is the calling TLM, and can normally be omitted so that the default value is used.

Returns

DT_OBJECT* - this is a pointer to an instance of a class derived from DT_OBJECT, according to the datatable type of the tag or address. Caller owns the DT_OBJECT derivative upon return.

ClassName

```
const char * ClassName() const
```

Function ClassName returns the name of this class.

ResolveAndVerify

```
void ResolveAndVerify()
```

Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.

Throws

ERROR — if the datatable memory does not exist or cannot be tested because of a missing Tag.

Address

```
const std::string & Address() const
```

Function Address returns the plc memory address of this object.

Tag

```
const std::string & Tag() const
```

Function Tag returns the tag of this object.

Spec

```
const std::string & Spec() const
```

Function Spec returns the specification given as *aTagOrAddress* to the constructor.

Name

```
const std::string & Name() const
```

Function Name returns the tag if known, else the address if known, else the spec.

Format

```
std::string Format(int aCtl = FMT_DEFAULT) const
```

Function Format prints information characterizing this object into a std::string and returns that string.

Parameters

`aCtl`	is a bitmapped int assembled by OR-ing together values from Enum FMT_CTL. (**Default value**: `FMT_DEFAULT`)

```
TLM & tlm
```

Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.

It is used by the logger TLM. which TLM have I been assigned to?

---

size_of

```
size_t size_of() const
```

6.20.10. TypeName

```
#include <dt_objects.h>

template<typename T>
struct TypeName
```

TypeName is for ENABLE_DT_PTR macro only.

Template Parameters	typename T
Public Static Methods	
static const char *	Name()

Members

Name

```
static const char * Name()
```

6.20.11. DT_PTR

```
#include <dt_objects.h>

template<typename T>
class DT_PTR
```

Template DT_PTR allows construction of any kind of datatable structure pointer, including user defined structs.

You could make use of a block of consecutive integers within a datatable N file and reinterpret them as any arbitrary struct.

Usage Example

```
#include "dt_objects.h"

// DT_PTR's are used simplest as globals.

struct DT_STRING {
    PLCINT  length;
    PLCINT  s[82];
};

// put this somewhere between struct and DT_PTR usage
```

```
ENABLE_DT_PTR( DT_STRING );

DT_PTR<DT_STRING>   st1( "StatusString" );
```

Template Parameters	`typename T`
Public Enclosed Types	
enum	`FMT_CTL` Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.
Public Constructors	
	`DT_PTR(const char *, TLM *)` Constructor DT_PTR.
Public Operators	
T *	`operator->()`
T *	`operator*()`
Public Static Methods	
static DT_OBJECT *	`Make(const char *, TLM *)` Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.
Public Methods	
const char *	`ClassName() const` Function ClassName returns the name of this class.
void	`ResolveAndVerify()` Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.
const std::string &	`Address() const` Function Address returns the plc memory address of this object.
const std::string &	`Tag() const` Function Tag returns the tag of this object.
const std::string &	`Spec() const` Function Spec returns the specification given as *aTagOrAddress* to the constructor.
const std::string &	`Name() const` Function Name returns the tag if known, else the address if known, else the spec.

std::string	`Format(int) const` Function Format prints information characterizing this object into a std::string and returns that string.
Protected Variables	
TLM &	`tlm` Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.
void *	`pointer` to implementation or datatable memory
std::string	`spec` a copy of aTagOrAddress from constructor.
std::string	`address` the SoftPLC memory address of this word.
std::string	`tag` the SoftPLC tag of this word, if any.
BINADDR	`binAddr` the SoftPLC address in binary.
Protected Methods	
size_t	`size_of() const`

FMT_CTL

```
#include <opt/softplc/c++-toolkit-5/h/dt_objects.h>

enum DT_PTR::FMT_CTL
```

Enum FMT_CTL is a set of bits that can be OR-ed together and passed to Format() to control that function's output.

TYPE = (1<<0)	show classname
SPEC = (1<<1)	show constructor's aTagOrAddress
TAG = (1<<2)	show tag
ADDR = (1<<3)	show address

Members

DT_PTR

```
DT_PTR(const char * aTagOrAddress,
       TLM * aTLM)
```

Constructor DT_PTR.

Parameters

aTagOrAddress	is a tag from the descriptor table or a datatable memory address the identifies the starting point of this structure.
aTLM	

---

operator→

```
T * operator->() const
```

---

operator*

```
T * operator*() const
```

Make

```
static DT_OBJECT * Make(const char * aTagOrAddress,
                        TLM * aTLM)
```

Function Make is a factory method that creates the proper derived class based on *aTagOrAddress*.

Parameters

aTagOrAddress	can be either a tag or a PLC address.
aTLM	is the calling TLM, and can normally be omitted so that the default value is used.

Returns

DT_OBJECT* - this is a pointer to an instance of a class derived from DT_OBJECT, according to the datatable type of the tag or address. Caller owns the DT_OBJECT derivative upon return.

ClassName

```
const char * ClassName() const
```

Function ClassName returns the name of this class.

ResolveAndVerify

```
void ResolveAndVerify()
```

Function ResolveAndVerify checks the existence of the datatable memory for this object and updates the *pointer* field.

Throws

ERROR — if the datatable memory does not exist or cannot be tested because of a missing Tag.

Address

```
const std::string & Address() const
```

Function Address returns the plc memory address of this object.

Tag

```
const std::string & Tag() const
```

Function Tag returns the tag of this object.

Spec

```
const std::string & Spec() const
```

Function Spec returns the specification given as *aTagOrAddress* to the constructor.

Name

```
const std::string & Name() const
```

Function Name returns the tag if known, else the address if known, else the spec.

Format

```
std::string Format(int aCtl = FMT_DEFAULT) const
```

Function Format prints information characterizing this object into a std::string and returns that string.

Parameters

`aCtl`	is a bitmapped int assembled by OR-ing together values from Enum FMT_CTL. (**Default value**: `FMT_DEFAULT`)

```
TLM & tlm
```

Function GetFloat returns a double, its a polymorphic way to get a value out of a DT_OBJECT.

It is used by the logger TLM. which TLM have I been assigned to?

size_of

```
size_t size_of() const
```

6.20.12. DT_OBJECTS

```
#include <dt_objects.h>

class DT_OBJECTS
```

Class DT_OBJECTS is a container class for a list of DT_OBJECTs.

It can be used with DT_OBJECT::Make() for dynamic construction of DT_OBJECTS.

Public Constructors	
	DT_OBJECTS()
	DT_OBJECTS(const DT_OBJECTS &)
Public Destructors	
	~DT_OBJECTS()
Public Operators	
DT_OBJECTS &	operator=(const DT_OBJECTS &)

Members

DT_OBJECTS

```
DT_OBJECTS()
```

DT_OBJECTS

```
DT_OBJECTS(const DT_OBJECTS & r)
```

Parameters

r

~DT_OBJECTS

```
~DT_OBJECTS()
```

---

operator=

```
DT_OBJECTS & operator=(const DT_OBJECTS & r)
```

Parameters

r

6.21. Finite State Machines for C++

The FSM class facilitates writing finite state machines (FSM) in C++ for SoftPLC.

Classes

FSM

Table 19. Typedefs

Name	Definition	Description
SHOW_STATE_FUNC	typedef const char * SHOW_STATE_FUNC(int aState)	Typedef SHOW_STATE_FUNC refers to a FSM function which translates a state number to its state name C string. This mapping tends to be unique across multiple state machines and is very FSM specific because each FSM will have its own set of unique states, both in terms of name and in quantity.

6.21.1. FSM

```
#include <softplc_toolkit.h>

class FSM
```

Class FSM.

is for coding finite state machines. It manages a current state (kept in a datatable word (PLCINT), and provides a state dwell timer feature which tells how long an instance has been in a particular state. You may derive from this class to inherit baseline functionality into your state machines.

This class holds a DT_INT16 so it can save the current state in the datatable in this integer. Since it is in the datatable, it is visible from the realtime datatable viewing tools like TopDoc NexGen or an HMI.

In addition to having the state number in the datatable, there is support for state transition messages. The state transition messages may be turned on and off without recompiling. The transition message enablement can be controlled by a configuration variable that you test somewhere. It could come from a TLM command line option, a configuration file setting, or a datatable value. The constructor can take and initial value, but more often you will change this value by calling FSM::SetTracking().

In the constructor you supply a SHOW_STATE_FUNC function that takes an integer indicating a state number and return a fixed text string naming the state for that value.

Public Constructors	
	`FSM(const char *, SHOW_STATE_FUNC *, bool, TLM *)` Constructor.
Public Methods	
const char *	`ShowState(int)` Function ShowState returns the current state in a textual representation, so long as this FSM was constructed with *aShowFunc*.
int	`State() const` Function State returns the current state number.
void	`SetState(int)` Function SetState changes the current state number, resets the dwell timer to 0 and optionally HlpPrintf()'s the state change if state tracking is on.
void	`SetTracking(bool)` Function SetTracking controls whether state tracking is on or off.
bool	`Dwell(USECS, USECS)` Function Dwell returns true if this FSM has been in its current state for *aTimeout* usecs or longer.
USECS	`DwellTime(USECS)` Function DwellTime returns the number of usecs that this FSM has been in the current state.

Members

FSM

```
FSM(const char * aStateTagOrAddress,
    SHOW_STATE_FUNC * aShowFunc = NULL,
    bool doStateTracking = false,
    TLM * aTLM)
```

Constructor.

Parameters

`aStateTagOrAddress`	The tag or address of an integer 'N' word which will be used to store the current state number.
`aShowFunc`	is a function which translates a state number to a C string. (**Default value**: `NULL`)

`doStateTracking` is a bool which controls whether to HlpPrintf() every state transition, helpful when debugging to set this on, later turn off. (**Default value**: `false`)

`aTLM`

ShowState

```
const char * ShowState(int aState)
```

Function ShowState returns the current state in a textual representation, so long as this FSM was constructed with *aShowFunc*.

Parameters

`aState`

State

```
int State() const
```

Function State returns the current state number.

SetState

```
void SetState(int aState)
```

Function SetState changes the current state number, resets the dwell timer to 0 and optionally HlpPrintf()'s the state change if state tracking is on.

Parameters

`aState`

SetTracking

```
void SetTracking(bool doTracking)
```

Function SetTracking controls whether state tracking is on or off.

Parameters

`doTracking`	when true turns state transition messaging on and when false turns messaging off.

Dwell

```
bool Dwell(USECS aTimeout,
           USECS now)
```

Function Dwell returns true if this FSM has been in its current state for *aTimeout* usecs or longer.

Parameters

`aTimeout`	is the number of usecs to test against, consider this a trigger point.
`now`	is the current monotonic usecs value obtained from a recent call to MicroSecsNow().

DwellTime

```
USECS DwellTime(USECS now)
```

Function DwellTime returns the number of usecs that this FSM has been in the current state.

Parameters

`now`	is the current monotonic usecs value obtained from a recent call to MicroSecsNow().

Returns

USECS - how long in the current state.

6.22. Serial I/O Support

These are functions and defines that are used to talk to the 36 serial ports.

Modules

Serial I/O Error Codes	These are #defines that are used by the Serial I/O Support functions.

Functions

COMsetup()	Function COMsetup will open a given serial port and configure it with the given parameters.
COMrst()	Function COMrst restarts the given comport, without a close and reopen, to operate at the given configuration parameters.
COMrst_x()	Function COMrst_x restarts the given comport, without a close and reopen, to operate at the given configuration parameters.
COMdisable()	Function COMdisable closes the com port.
COMrcvclear()	Function COMrcvclear will empty the receive buffer for the given comport, dropping any received and unread bytes to that point.
COMxmitclear()	Function COMxmitclear will empty the transmit buffer, dropping any written yet unsent bytes to that point.
COMgetc()	Function COMgetc gets a byte from the comm buffer iff available, else returns -1 immediately.
COMin()	Function COMin gets a byte from the comm buffer when available, and blocks until one is available.
COMin_timeout()	Function COMin_timeout gets a byte from the comm buffer when available, and blocks until one is available or the usecsTimeout occurs.
COMread()	Function COMread reads a number of bytes from the comm buffer.
COMrcvqueue()	Function COMrcvqueue returns the number of bytes in the receive buffer for the given comport.
COMxmitqueue()	Function COMxmitqueue returns the number of bytes in the transmit buffer for the given comport.
COMwrite()	Function COMwrite writes a number of bytes out the serial port given by comport.
COMbreak()	Function COMbreak changes the line break output status to TRUE or FALSE.
COMputc()	Function COMputc outputs a single byte to the serial transmit buffer for the given comport.
COMgetoverflow()	Function COMgetoverflow gets the overflow status of the receive ring buffer.
COMgetlinebreak()	Function COMgetlinebreak gets an internal bool that indicates whether a line break has been seen since it was last reset.
COMgeterrflg()	Function COMgeterrflg gets an internal bit field that has bits indicating the historical status pertaining to:
COMgetdcd()	Function COMgetdcd gets the current status of the **Data Carrier Detect (DCD)** line for the given serial port.
COMgetcts()	Function COMgetcts gets the current status of the **Clear To Send (CTS)** line for the given serial port.
COMgetring()	Function COMgetring gets the current status of the **Ring Indicator (RI)** line for the given serial port.

COMgetdsr()	Function COMgetdsr gets the current status of the **Data Set Ready (DSR)** line for the given serial port.
COMgettxempty()	Function COMgettxempty gets the current transmit buffer empty and transmit shift register empty status of the given serial port.
COMsetrts()	Function COMsetrts sets the current status of the **Request To Send (RTS)** line for the given serial port.
COMsetdtr()	Function COMsetdtr sets the current status of the **Data Terminal Request (DTR)** line for the given serial port.
COMsetout1()	Function COMsetout1 sets the Modem Control Register's OUT1 line, if it exists, for the given serial port.
COMsetout2()	Function COMsetout2 sets the Modem Control Register's OUT2 line, if it exists, for the given serial port.
COMsetloopback()	Function COMsetloopback sets the Modem Control Register's LOOPBACK state, if it exists, for the given serial port.

6.22.1. COMsetup()

```
ERRCODE COMsetup(int comport, int rcvSize, char *rcvBuf, int xmitSize, char *xmitBuf,
long baudrate, int parity, int stopbits, int databits)
```

Function COMsetup will open a given serial port and configure it with the given parameters.

If this calls succeeds, after using the given serial port, then call COMdisable()

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed.
`rcvSize`	Unused in the Linux version of this library, use 0.
`rcvBuf`	Unused in the Linux version of this library, use 0.
`xmitSize`	Unused in the Linux version of this library, use 0.
`xmitBuf`	Unused in the Linux version of this library, use 0.
`baudrate`	May be one of 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, or 460800. Baudrates other than this will not work.
`parity`	A character value of one of 'N', 'O', 'E', 'S', 'M' meaning none, odd, even, space or mark respectively.
`stopbits`	The number of stop bits, 1 or 2
`databits`	The number of databits, 5, 6, 7, or 8

Returns

ERRCODE - SUCCESS or one of: * E_COMM_BAD_PORT

- E_COMM_ALREADY_OPEN
- E_COMM_BAD_BAUDRATE
- E_COMM_BAD_PARITY
- E_COMM_BAD_DATABITS
- E_COMM_BAD_STOPBITS

6.22.2. COMrst()

```
ERRCODE COMrst(int comport, long baudrate, int parity, int stopbits, int databits)
```

Function COMrst restarts the given comport, without a close and reopen, to operate at the given configuration parameters.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed.
`baudrate`	May be one of 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, or 460800.
`parity`	A character value of one of 'N', 'O', 'E', 'S', 'M' meaning none, odd, even, space or mark respectively.
`stopbits`	The number of stop bits, 1 or 2
`databits`	The number of databits, 5, 6, 7, or 8

Returns

ERRCODE - SUCCESS or one of: * E_COMM_BAD_PORT

- E_COMM_NOT_OPEN
- E_COMM_BAD_BAUDRATE
- E_COMM_BAD_PARITY
- E_COMM_BAD_DATABITS
- E_COMM_BAD_STOPBITS

6.22.3. COMrst_x()

```
ERRCODE COMrst_x(int comport, long baudrate, int parity, int stopbits, int databits,
int rcvFifoIntThreshold)
```

Function COMrst_x restarts the given comport, without a close and reopen, to operate at the given configuration parameters.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed.
`baudrate`	May be one of 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, or 460800.
`parity`	A character value of one of 'N', 'O', 'E', 'S', 'M' meaning none, odd, even, space or mark respectively.
`stopbits`	The number of stop bits, 1 or 2
`databits`	The number of databits, 5, 6, 7, or 8
`rcvFifoIntThreshold`	is not supported on Linux at this time, but normally is one of 1, 4, 8, or 14.

Returns

ERRCODE - SUCCESS or one of: * E_COMM_BAD_PORT

- E_COMM_NOT_OPEN
- E_COMM_BAD_BAUDRATE
- E_COMM_BAD_PARITY
- E_COMM_BAD_DATABITS
- E_COMM_BAD_STOPBITS

6.22.4. COMdisable()

```
ERRCODE COMdisable(int comport)
```

Function COMdisable closes the com port.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()

6.22.5. COMrcvclear()

```
void COMrcvclear(int comport)
```

Function COMrcvclear will empty the receive buffer for the given comport, dropping any received and unread bytes to that point.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()

6.22.6. COMxmitclear()

```
void COMxmitclear(int comport)
```

Function COMxmitclear will empty the transmit buffer, dropping any written yet unsent bytes to that point.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()

6.22.7. COMgetc()

```
int COMgetc(int comport)
```

Function COMgetc gets a byte from the comm buffer iff available, else returns -1 immediately.

Does NOT wait until a byte is available.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()

Returns

int - the byte in lower 8 bits or -1 if none there.

6.22.8. COMin()

```
int COMin(int comport)
```

Function COMin gets a byte from the comm buffer when available, and blocks until one is available.

Waits indefinitely until a byte is available.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()

Returns

int - the byte in lower 8 bits.

6.22.9. COMin_timeout()

```
int COMin_timeout(int comport, int usecsTimeout)
```

Function COMin_timeout gets a byte from the comm buffer when available, and blocks until one is available or the usecsTimeout occurs.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`usecsTimeout`	howmany microseconds to wait maximum, use -1 to mean forever.

Returns

int - the byte in lower 8 bits, or -1 to indicate a timeout.

6.22.10. COMread()

```
int COMread(int comport, char *buf, int howmany)
```

Function COMread reads a number of bytes from the comm buffer.

Reads at least one byte but this may be less than the number requested.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`buf`	Where to put the requested bytes, a buffer large enough to hold *howmany* bytes.
`howmany`	The number of bytes to read into *buf*, and should be >= 1.

Returns

int - the number actually returned, at least one 1 but up to the number requested, inclusive. The quantity read may be less than the number of available bytes in the receive buffers, so you will have to watch this return value closely.

6.22.11. COMrcvqueue()

```
int COMrcvqueue(int comport)
```

Function COMrcvqueue returns the number of bytes in the receive buffer for the given comport.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()

Returns

int - The number of received but unread bytes in the receive buffer for the given comport.

6.22.12. COMxmitqueue()

```
int COMxmitqueue(int comport)
```

Function COMxmitqueue returns the number of bytes in the transmit buffer for the given comport.

The transmit buffers can hold up to COM_MAX_XMITQUEUE bytes at any time. Bytes written beyond that number with either COMputc() or COMwrite() run the risk of blocking the calling thread.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()

Returns

int - The number of written but un-transmitted bytes in the transmit buffer for the given comport, or -1 if error.

6.22.13. COMwrite()

```
int COMwrite(int comport, const char *buf, int num)
```

Function COMwrite writes a number of bytes out the serial port given by comport.

The function will block until all the requested bytes have been transferred to the transmit buffer, then it will return. This may be before the bytes are actually output through the serial interface.

To avoid blocking the calling thread, always make sure there is room in the transmit buffer by checking the following: if(COM_MAX_XMITQUEUE > numToWrite + COMxmitqueue(port)) COMwrite(.. numToWrite...)

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`buf`	The start of the bytes to send, extending for *num* bytes.
`num`	The number of bytes to write to the transmit buffer, and should be >= 1.

Returns

int - the number actually written, which will generally be either *num*, or it can be -1 if the comport is invalid or not COMsetup() yet.

6.22.14. COMbreak()

```
void COMbreak(int comport, bool sts)
```

Function COMbreak changes the line break output status to TRUE or FALSE.

This function will have to be called twice with a small delay in between, passing in TRUE on the first call and FALSE on the second call.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`sts`	A bool which if TRUE means start line break, and if FALSE to end the line break condition.

6.22.15. COMputc()

```
bool COMputc(int comport, char byteValue)
```

Function COMputc outputs a single byte to the serial transmit buffer for the given comport.

If you have a buffer of **several** bytes to output, COMwrite() is more efficient than this function.

To avoid blocking the calling thread, always make sure there is room in the transmit buffer by checking the following: if(COM_MAX_XMITQUEUE > COMxmitqueue(port)) COMputc(...)

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`byteValue`	is the byte to send

Returns

bool - TRUE if a byte was output, else FALSE which can happen if the serial port has not been COMsetup() or the comport parameter is out of range.

6.22.16. COMgetoverflow()

```
bool COMgetoverflow(int comport, bool reset)
```

Function COMgetoverflow gets the overflow status of the receive ring buffer.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`reset`	TRUE ⇒ reset the internal bool after reading it, FALSE ⇒ no reset

Returns

bool - TRUE if the ring buffer has overflowed since last *reset*, else FALSE.

6.22.17. COMgetlinebreak()

```
bool COMgetlinebreak(int comport, bool reset)
```

Function COMgetlinebreak gets an internal bool that indicates whether a line break has been seen since it was last reset.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`reset`	TRUE ⇒ reset the internal bool after reading it, FALSE ⇒ no reset

Returns

bool - TRUE if the line break flag is set since last *reset*, else FALSE.

6.22.18. COMgeterrflg()

```
int COMgeterrflg(int comport, bool reset)
```

Function COMgeterrflg gets an internal bit field that has bits indicating the historical status pertaining to:

- COMMERR_PARITY
- COMMERR_FRAMING
- COMMERR_OVERRUN

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`reset`	TRUE ⇒ reset all the bits of the internal bitfield after reading it, FALSE ⇒ no reset

Returns

int - which is a bitfield with the above COMMERR_* bits potentially OR'ed in.

6.22.19. COMgetdcd()

```
bool COMgetdcd(int comport)
```

Function COMgetdcd gets the current status of the **Data Carrier Detect (DCD)** line for the given serial port.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()

Returns

bool - TRUE if the DCD line is TRUE, else FALSE if not or if *comport* is bad or not COMsetup().

6.22.20. COMgetcts()

```
bool COMgetcts(int comport)
```

Function COMgetcts gets the current status of the **Clear To Send (CTS)** line for the given serial port.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()

Returns

bool - TRUE if the CTS line is TRUE, else FALSE if not or if *comport* is bad or not COMsetup().

6.22.21. COMgetring()

```
bool COMgetring(int comport)
```

Function COMgetring gets the current status of the **Ring Indicator (RI)** line for the given serial port.

Parameters

`comport` The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()

Returns

bool - TRUE if the RI line is TRUE, else FALSE if not or if *comport* is bad or not COMsetup().

6.22.22. COMgetdsr()

```
bool COMgetdsr(int comport)
```

Function COMgetdsr gets the current status of the **Data Set Ready (DSR)** line for the given serial port.

Parameters

`comport` The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()

Returns

bool - TRUE if the DSR line is TRUE, else FALSE if not or if *comport* is bad or not COMsetup().

6.22.23. COMgettxempty()

```
bool COMgettxempty(int comport)
```

Function COMgettxempty gets the current transmit buffer empty and transmit shift register empty status of the given serial port.

Parameters

`comport` The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()

Returns

bool - TRUE if the port's transmit shift registers are empty, else FALSE.

6.22.24. COMsetrts()

```
void COMsetrts(int comport, bool state)
```

Function COMsetrts sets the current status of the **Request To Send (RTS)** line for the given serial port.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`state`	TRUE if the RTS line should be enabled, else FALSE

6.22.25. COMsetdtr()

```
void COMsetdtr(int comport, bool state)
```

Function COMsetdtr sets the current status of the **Data Terminal Request (DTR)** line for the given serial port.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`state`	TRUE if the DTR line should be enabled, else FALSE

6.22.26. COMsetout1()

```
void COMsetout1(int comport, bool state)
```

Function COMsetout1 sets the Modem Control Register's OUT1 line, if it exists, for the given serial port.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`state`	TRUE if the OUT1 line should be enabled, else FALSE

6.22.27. COMsetout2()

```
void COMsetout2(int comport, bool state)
```

Function COMsetout2 sets the Modem Control Register's OUT2 line, if it exists, for the given serial port.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`state`	TRUE if the OUT2 line should be enabled, else FALSE

6.22.28. COMsetloopback()

```
void COMsetloopback(int comport, bool state)
```

Function COMsetloopback sets the Modem Control Register's LOOPBACK state, if it exists, for the given serial port.

Parameters

`comport`	The number of the COM port, starting at 0 and extending possibly up to NUM_COMPORTS - 1, depending on how many physical serial ports you have installed. This comport should have been successfully COMsetup()
`state`	TRUE if the LOOPBACK state should be enabled, else FALSE

6.22.29. Serial I/O Error Codes

These are #defines that are used by the Serial I/O Support functions.

Table 20. Constants

Name	Value	Description
`NUM_COMPORTS`	`36`	the maximum number of comports
`COM_MAX_XMITQUEUE`	`1024`	the maximum number of bytes that may be queued in the transmit buffers under Linux as of this writing.
`COMMERR_PARITY`	`(1<<0)`	A bit within a bitfield returned by COMgeterrflg() that shows if a PARITY error occurred.

Name	Value	Description
COMMERR_FRAMING	(1<<1)	A bit within a bitfield returned by COMgeterrflg() that shows if a FRAMING ERROR occurred.
COMMERR_OVERRUN	(1<<2)	A bit within a bitfield returned by COMgeterrflg() that shows if a chip level FIFO overrun occurred.
E_COMM_BAD_PORT	``	The given comport is out of range.
E_COMM_ALREADY_OPEN	``	Returned by COMsetup() when the given comport is already open.
E_COMM_BAD_BAUDRATE	``	The given baudrate is unsupported.
E_COMM_BAD_PARITY	``	The given parity is not one of 'N', 'O', 'E', 'S', or 'M'.
E_COMM_NOT_OPEN	``	Returned by any COM function called before COMsetup() was made.
E_COMM_BAD_DATABITS	``	The given databits is not one of 5,6,7 or 8.
E_COMM_BAD_STOPBITS	``	The given databits is not one of 5,6,7 or 8.
E_COMM_NO_LOW_LATENCY	``	Unable to set low latency flag during COMsetup()

E_COMM()

```
#define E_COMM(x) ((unsigned)(x)+5000u)
```

A macro used internally to generate the other E_COMM_* #defines it is equal to ((unsigned)(x)+5000u)

6.23. S-Expression Parsing Support

Class DSNLEXER along with a supplied CMake script make it fairly simple to implement a recursive descent s-expression parser for your TLM's configuration file reader, should you decide to use s-expression format for your configuration file(s).

Classes

DSNLEXER

6.23.1. enum DSN_SYNTAX_T

Enum DSN_SYNTAX_T lists all the DSN lexer's tokens that are supported in lexing.

It is up to the parser if it wants also to support them.

Value	Init	Description
DSN_NONE	= -11	
DSN_COMMENT	= -10	
DSN_STRING_QUOTE	= -9	
DSN_QUOTE_DEF	= -8	
DSN_DASH	= -7	
DSN_SYMBOL	= -6	
DSN_NUMBER	= -5	
DSN_RIGHT	= -4	
DSN_LEFT	= -3	
DSN_STRING	= -2	
DSN_EOF	= -1	

6.23.2. DSNLEXER

```
#include <dsnlexer.h>

class DSNLEXER
```

Class DSNLEXER implements a lexical analyzer for the S-EXPRESSION file format.

It reads lexical tokens from the current LINE_READER through the NextTok() function.

Public Constructors	
	DSNLEXER(const KEYWORD *, unsigned, FILE *, const std::string &) Constructor (FILE*, const std::string&) intializes a DSN lexer and prepares to read from aFile which is already open and has aFilename.
	DSNLEXER(const KEYWORD *, unsigned, const std::string &, const std::string &) Constructor (const KEYWORD*, unsigned, const std::string&, const std::string&) intializes a DSN lexer and prepares to read from *aSExpression*.
	DSNLEXER(const std::string &, const std::string &) Constructor (const std::string&, const std::string&) intializes a DSN lexer and prepares to read from *aSExpression*.

Public Constructors	
	`DSNLEXER(const KEYWORD *, unsigned, LINE_READER *)` Constructor (LINE_READER*) intializes a DSN lexer and prepares to read from *aLineReader* which is already open, and may be in use by other DSNLEXERs also.
Public Destructors	
	`~DSNLEXER()`
Public Static Methods	
static bool	`IsSymbol(int)` Function IsSymbol tests a token to see if it is a symbol.
static const char *	`Syntax(int)`
Public Methods	
bool	`SyncLineReaderWith(DSNLEXER &)` Useable only for DSN lexers which share the same LINE_READER Synchronizes the pointers handling the data read by the LINE_READER Allows 2 DNSLEXER to share the same current line, when switching from a DNSLEXER to another DNSLEXER.
void	`PushReader(LINE_READER *)` Function PushReader manages a stack of LINE_READERs in order to handle nested file inclusion.
LINE_READER *	`PopReader()` Function PopReader deletes the top most LINE_READER from an internal stack of LINE_READERs and in the case of FILE_LINE_READER this means the associated FILE is closed.
int	`NextTok()` Function NextTok returns the next token found in the input file or DSN_EOF when reaching the end of file.
int	`NeedSYMBOL()` Function NeedSYMBOL calls NextTok() and then verifies that the token read in satisfies bool IsSymbol().
int	`NeedSYMBOLorNUMBER()` Function NeedSYMBOLorNUMBER calls NextTok() and then verifies that the token read in satisfies bool IsSymbol() or tok==DSN_NUMBER.
int	`NeedNUMBER(const char *)` Function NeedNUMBER calls NextTok() and then verifies that the token read is type DSN_NUMBER.

Public Constructors	
int	`CurTok()` Function CurTok returns whatever NextTok() returned the last time it was called.
int	`PrevTok()` Function PrevTok returns whatever NextTok() returned the 2nd to last time it was called.
char	`SetStringDelimiter(char)` Function SetStringDelimiter changes the string delimiter from the default " to some other character and returns the old value.
bool	`SetSpaceInQuotedTokens(bool)` Function SetSpaceInQuotedTokens changes the setting controlling whether a space in a quoted string is a terminator.
bool	`SetCommentsAreTokens(bool)` Function SetCommentsAreTokens changes the handling of comments.
std::vector<std::string>	`ReadCommentLines()` Function ReadCommentLines checks the next sequence of tokens and reads them into a std::vector if they are comments.
void	`Expecting(int)` Function Expecting throws an IO_ERROR exception with an input file specific error message.
void	`Expecting(const char *)` Function Expecting throws an IO_ERROR exception with an input file specific error message.
void	`Unexpected(int)` Function Unexpected throws an IO_ERROR exception with an input file specific error message.
void	`Unexpected(const char *)` Function Unexpected throws an IO_ERROR exception with an input file specific error message.
void	`Duplicate(int)` Function Duplicate throws an IO_ERROR exception with a message saying specifically that aTok is a duplicate of one already seen in current context.
void	`NeedLEFT()` Function NeedLEFT calls NextTok() and then verifies that the token read in is a DSN_LEFT.

Public Constructors	
void	NeedRIGHT() Function NeedRIGHT calls NextTok() and then verifies that the token read in is a DSN_RIGHT.
const char *	GetTokenText(int) Function GetTokenText returns the C string representation of a DSN_T value.
std::string	GetTokenString(int) Function GetTokenString returns a quote wrapped string representation of a token value.
const char *	CurText() Function CurText returns a pointer to the current token's text.
const std::string &	CurStr() Function CurStr returns a reference to current token in std::string form.
int	CurLineNumber() Function CurLineNumber returns the current line number within my LINE_READER.
const char *	CurLine() Function CurLine returns the current line of text, from which the CurText() would return its token.
const std::string &	CurSource() Function CurFilename returns the current LINE_READER source.
int	CurOffset() Function CurOffset returns the byte offset within the current line, using a 1 based index.
Protected Enclosed Types	
typedef	READER_STACK
Protected Variables	
bool	iOwnReaders on readerStack, should I delete them?
const char *	start
const char *	next
const char *	limit
char	dummy when there is no reader.

Public Constructors	
READER_STACK	`readerStack` all the LINE_READERs by pointer.
LINE_READER *	`reader` no ownership. ownership is via readerStack, maybe, if iOwnReaders
bool	`specctraMode` if true, then: 1) stringDelimiter can be changed 2) Kicad quoting protocol is not in effect 3) space_in_quoted_tokens is functional else not.
char	`stringDelimiter`
bool	`space_in_quoted_tokens` blank spaces within quoted strings
bool	`commentsAreTokens` true if should return comments as tokens
int	`prevTok` curTok from previous NextTok() call.
int	`curOffset` offset within current line of the current token
int	`curTok` the current token obtained on last NextTok()
std::string	`curText` the text of the current token
const KEYWORD *	`keywords` table sorted by CMake for bsearch()
unsigned	`keywordCount` count of keywords table
KEYWORD_MAP	`keyword_hash` fast, specialized "C string" hashtable
Protected Methods	
void	`init()`
int	`readLine()`

Public Constructors	
int	`findToken(const std::string &)` Function findToken takes aToken string and looks up the string in the keywords table.
bool	`isStringTerminator(char)`

READER_STACK

```
typedef std::vector<LINE_READER *> READER_STACK
```

Members

DSNLEXER

```
DSNLEXER(const KEYWORD * aKeywordTable,
         unsigned aKeywordCount,
         FILE * aFile,
         const std::string & aFileName)
```

Constructor (FILE*, const std::string&) intializes a DSN lexer and prepares to read from aFile which is already open and has aFilename.

Parameters

aKeywordTable	is an array of KEYWORDS holding *aKeywordCount*. This token table need not contain the lexer separators such as '(' ')', etc.
aKeywordCount	is the count of tokens in aKeywordTable.
aFile	is an open file, which will be closed when this is destructed.
aFileName	is the name of the file

DSNLEXER

```
DSNLEXER(const KEYWORD * aKeywordTable,
         unsigned aKeywordCount,
         const std::string & aSExpression,
         const std::string & aSource = "")
```

Constructor (const KEYWORD*, unsigned, const std::string&, const std::string&) intializes a DSN lexer and prepares to read from *aSExpression*.

Parameters

aKeywordTable	is an array of KEYWORDS holding *aKeywordCount*. This token table need not contain the lexer separators such as '(' ')', etc.
aKeywordCount	is the count of tokens in aKeywordTable.
aSExpression	is text to feed through a STRING_LINE_READER
aSource	is a description of aSExpression, used for error reporting. (**Default value**: "")

DSNLEXER

```
DSNLEXER(const std::string & aSExpression,
         const std::string & aSource = "")
```

Constructor (const std::string&, const std::string&) intializes a DSN lexer and prepares to read from *aSExpression*.

Use this one without a keyword table with the DOM parser in ptree.h.

Parameters

aSExpression	is text to feed through a STRING_LINE_READER
aSource	is a description of aSExpression, used for error reporting. (**Default value**: "")

DSNLEXER

```
DSNLEXER(const KEYWORD * aKeywordTable,
         unsigned aKeywordCount,
         LINE_READER * aLineReader = NULL)
```

Constructor (LINE_READER*) intializes a DSN lexer and prepares to read from *aLineReader* which is already open, and may be in use by other DSNLEXERs also.

No ownership is taken of *aLineReader*. This enables it to be used by other DSNLEXERs also.

Parameters

aKeywordTable	is an array of KEYWORDS holding *aKeywordCount*. This token table need not contain the lexer separators such as '(' ')', etc.
aKeywordCount	is the count of tokens in aKeywordTable.
aLineReader	is any subclassed instance of LINE_READER, such as STRING_LINE_READER or FILE_LINE_READER. No ownership is taken. (**Default value**: NULL)

~DSNLEXER

```
~DSNLEXER()
```

IsSymbol

```
static bool IsSymbol(int aTok)
```

Function IsSymbol tests a token to see if it is a symbol.

This means it cannot be a special delimiter character such as DSN_LEFT, DSN_RIGHT, DSN_QUOTE, etc. It may however, coincidentally match a keyword and still be a symbol.

Parameters

`aTok`

Syntax

```
static const char * Syntax(int aTok)
```

Parameters

`aTok`

SyncLineReaderWith

```
bool SyncLineReaderWith(DSNLEXER & aLexer)
```

Useable only for DSN lexers which share the same LINE_READER Synchronizes the pointers handling the data read by the LINE_READER Allows 2 DNSLEXER to share the same current line, when switching from a DNSLEXER to another DNSLEXER.

Parameters

`aLexer` = the model

Returns

true if the sync can be made (at least the same line reader)

PushReader

```
void PushReader(LINE_READER * aLineReader)
```

Function PushReader manages a stack of LINE_READERs in order to handle nested file inclusion.

This function pushes aLineReader onto the top of a stack of LINE_READERs and makes it the current LINE_READER with its own GetSource(), line number and line text. A grammar must be designed such that the "include" token (whatever its various names), and any of its parameters are not followed by anything on that same line, because PopReader always starts reading from a new line upon returning to the original LINE_READER.

Parameters

`aLineReader`

PopReader

```
LINE_READER * PopReader()
```

Function PopReader deletes the top most LINE_READER from an internal stack of LINE_READERs and in the case of FILE_LINE_READER this means the associated FILE is closed.

The most recently used former LINE_READER on the stack becomes the current LINE_READER and its previous position in its input stream and the its latest line number should pertain. PopReader always starts reading from a new line upon returning to the previous LINE_READER. A pop is only possible if there are at least 2 LINE_READERs on the stack, since popping the last one is not supported.

Returns

LINE_READER* - is the one that was in use before the pop, or NULL if there was not at least two readers on the stack and therefore the pop failed.

NextTok

```
int NextTok()
```

Function NextTok returns the next token found in the input file or DSN_EOF when reaching the end of file.

Users should wrap this function to return an enum to aid in grammar debugging while running under a debugger, but leave this lower level function returning an int (so the enum does not collide with another usage).

Returns

int - the type of token found next.

Throws

IO_ERROR — - only if the LINE_READER throws it.

NeedSYMBOL

```
int NeedSYMBOL()
```

Function NeedSYMBOL calls NextTok() and then verifies that the token read in satisfies bool IsSymbol().

If not, an IO_ERROR is thrown.

Returns

int - the actual token read in.

Throws

IO_ERROR — the next token does not satisfy IsSymbol()

NeedSYMBOLorNUMBER

```
int NeedSYMBOLorNUMBER()
```

Function NeedSYMBOLorNUMBER calls NextTok() and then verifies that the token read in satisfies bool IsSymbol() or tok==DSN_NUMBER.

If not, an IO_ERROR is thrown.

Returns

int - the actual token read in.

Throws

IO_ERROR — the next token does not satisfy the above test

NeedNUMBER

```
int NeedNUMBER(const char * aExpectation)
```

Function NeedNUMBER calls NextTok() and then verifies that the token read is type DSN_NUMBER.

If not, and IO_ERROR is thrown using text from aExpectation.

Parameters

`aExpectation`

Returns

int - the actual token read in.

Throws

`IO_ERROR` — the next token does not satisfy the above test

CurTok

```
int CurTok()
```

Function CurTok returns whatever NextTok() returned the last time it was called.

PrevTok

```
int PrevTok()
```

Function PrevTok returns whatever NextTok() returned the 2nd to last time it was called.

SetStringDelimiter

```
char SetStringDelimiter(char aStringDelimiter)
```

Function SetStringDelimiter changes the string delimiter from the default " to some other character and returns the old value.

Parameters

`aStringDelimiter`	The character in lowest 8 bits.

Returns

int - The old delimiter in the lowest 8 bits.

SetSpaceInQuotedTokens

```
bool SetSpaceInQuotedTokens(bool val)
```

Function SetSpaceInQuotedTokens changes the setting controlling whether a space in a quoted string is a terminator.

Parameters

`val`	If true, means

SetCommentsAreTokens

```
bool SetCommentsAreTokens(bool val)
```

Function SetCommentsAreTokens changes the handling of comments.

If set true, comments are returns as single line strings with a terminating newline, else they are consumed by the lexer and not returned.

Parameters

`val`	

ReadCommentLines

```
std::vector<std::string> ReadCommentLines()
```

Function ReadCommentLines checks the next sequence of tokens and reads them into a std::vector if they are comments.

Reading continues until a non-comment token is encountered, and such last read token remains as CurTok() and as CurText(). No push back or "un get" mechanism is used for this support. Upon return you simply avoid calling NextTok() for the next token, but rather CurTok().

Returns

std::vector<std::string> - a block of comments, which may be empty if none were seen.

Expecting

```
void Expecting(int aTok)
```

Function Expecting throws an IO_ERROR exception with an input file specific error message.

Parameters

aTok	is the token/keyword type which was expected at the current input location.

Throws

IO_ERROR — with the location within the input file of the problem.

Expecting

```
void Expecting(const char * aTokenList)
```

Function Expecting throws an IO_ERROR exception with an input file specific error message.

Parameters

aTokenList	is the token/keyword type which was expected at the current input location, e.g. "pin|graphic|property"

Throws

IO_ERROR — with the location within the input file of the problem.

Unexpected

```
void Unexpected(int aTok)
```

Function Unexpected throws an IO_ERROR exception with an input file specific error message.

Parameters

aTok	is the token/keyword type which was not expected at the current input location.

Throws

IO_ERROR — with the location within the input file of the problem.

Unexpected

```
void Unexpected(const char * aToken)
```

Function Unexpected throws an IO_ERROR exception with an input file specific error message.

Parameters

`aToken`	is the token which was not expected at the current input location.

Throws

`IO_ERROR` — with the location within the input file of the problem.

Duplicate

```
void Duplicate(int aTok)
```

Function Duplicate throws an IO_ERROR exception with a message saying specifically that aTok is a duplicate of one already seen in current context.

Parameters

`aTok`	is the token/keyword type which was not expected at the current input location.

Throws

`IO_ERROR` — with the location within the input file of the problem.

NeedLEFT

```
void NeedLEFT()
```

Function NeedLEFT calls NextTok() and then verifies that the token read in is a DSN_LEFT.

If it is not, an IO_ERROR is thrown.

Throws

`IO_ERROR` — the next token is not a DSN_LEFT

NeedRIGHT

```
void NeedRIGHT()
```

Function NeedRIGHT calls NextTok() and then verifies that the token read in is a DSN_RIGHT.

If it is not, an IO_ERROR is thrown.

Throws

`IO_ERROR` — the next token is not a DSN_RIGHT

GetTokenText

```
const char * GetTokenText(int aTok)
```

Function GetTokenText returns the C string representation of a DSN_T value.

Parameters

`aTok`

GetTokenString

```
std::string GetTokenString(int aTok)
```

Function GetTokenString returns a quote wrapped string representation of a token value.

Parameters

`aTok`

CurText

```
const char * CurText()
```

Function CurText returns a pointer to the current token's text.

CurStr

```
const std::string & CurStr()
```

Function CurStr returns a reference to current token in std::string form.

CurLineNumber

```
int CurLineNumber()
```

Function CurLineNumber returns the current line number within my LINE_READER.

CurLine

```
const char * CurLine()
```

Function CurLine returns the current line of text, from which the CurText() would return its token.

CurSource

```
const std::string & CurSource()
```

Function CurFilename returns the current LINE_READER source.

Returns

const std::string& - the source of the lines of text, e.g. a filename or "clipboard".

CurOffset

```
int CurOffset()
```

Function CurOffset returns the byte offset within the current line, using a 1 based index.

Returns

int - a one based index into the current line.

---

init

```
void init()
```

---

readLine

```
int readLine()
```

---

findToken

```
int findToken(const std::string & aToken)
```

Function findToken takes aToken string and looks up the string in the keywords table.

Parameters

`aToken`	is a string to lookup in the keywords table.

Returns

int - with a value from the enum DSN_T matching the keyword text, or DSN_SYMBOL if *aToken* is not in the kewords table.

---

isStringTerminator

```
bool isStringTerminator(char cc)
```

Parameters

`cc`

6.24. Reader and Formatter Classes

These classes can be used to output text to various destination types and read text lines from various sources.

Class DSNLEXER uses a LINE_READER to read input lines. But that is an abstract class whose contract is fulfilled by FILE_LINE_READER in actual practice. A LINE_READER is capable of telling you the current line number and byte offset of the input stream.

Classes

LINE_READER

FILE_LINE_READER

STRING_LINE_READER

OUTPUTFORMATTER

STRING_FORMATTER

FILE_OUTPUTFORMATTER

Table 21. Constants

Name	Value	Description
OUTPUTFMTBUFZ	500	default buffer size for any OUTPUT_FORMATTER

6.24.1. LINE_READER

```
#include <richio.h>

class LINE_READER
```

Class LINE_READER is an abstract class from which implementation specific LINE_READERs may be derived to read single lines of text and manage a line number counter.

Public Constructors	
	LINE_READER(unsigned) Constructor LINE_READER builds a line reader and fixes the length of the maximum supported line length to *aMaxLineLength*.
Public Destructors	
	~LINE_READER()
Public Operators	
char *	operator char *() Operator char* is a casting operator that returns a char* pointer to the start of the line buffer.
Public Methods	
char *	ReadLine() Function ReadLine reads a line of text into the buffer and increments the line number counter.
const std::string &	GetSource() const Function GetSource returns the name of the source of the lines in an abstract sense.
char *	Line() const Function Line returns a pointer to the last line that was read in.
unsigned	LineNumber() const Function Line Number returns the line number of the last line read from this LINE_READER.
unsigned	Length() const Function Length returns the number of bytes in the last line read from this LINE_READER.
Protected Variables	

Public Constructors	
unsigned	`m_length` no. bytes in line before trailing nul.
unsigned	`m_lineNum`
char *	`m_line` the read line of UTF8 text
unsigned	`m_capacity` no. bytes allocated for line.
unsigned	`m_maxLineLength` maximum allowed capacity using resizing.
std::string	`m_source` origin of text lines, e.g. filename or "clipboard"
Protected Methods	
void	`expandCapacity(unsigned)` Function expandCapacity will expand the capacity of *line* up to maxLineLength but not greater, so be careful about making assumptions of *capacity* after calling this.

Members

LINE_READER

```
LINE_READER(unsigned aMaxLineLength = LINE_READER_LINE_DEFAULT_MAX)
```

Constructor LINE_READER builds a line reader and fixes the length of the maximum supported line length to *aMaxLineLength*.

Parameters

`aMaxLineLength`	(**Default value**: `LINE_READER_LINE_DEFAULT_MAX`)

~LINE_READER

```
~LINE_READER()
```

operator char *

```
char * operator char *() const
```

Operator char* is a casting operator that returns a char* pointer to the start of the line buffer.

ReadLine

```
char * ReadLine()
```

Function ReadLine reads a line of text into the buffer and increments the line number counter.

If the line is larger than aMaxLineLength passed to the constructor, then an exception is thrown. The line is nul terminated.

Returns

char* - The beginning of the read line, or NULL if EOF.

Throws

`IO_ERROR` — when a line is too long.

GetSource

```
const std::string & GetSource() const
```

Function GetSource returns the name of the source of the lines in an abstract sense.

This may be a file or it may be the clipboard or any other source of lines of text. The returned string is useful for reporting error messages.

Line

```
char * Line() const
```

Function Line returns a pointer to the last line that was read in.

LineNumber

```
unsigned LineNumber() const
```

Function Line Number returns the line number of the last line read from this LINE_READER.

Lines start from 1.

Length

```
unsigned Length() const
```

Function Length returns the number of bytes in the last line read from this LINE_READER.

---

expandCapacity

```
void expandCapacity(unsigned aNewsize)
```

Function expandCapacity will expand the capacity of *line* up to maxLineLength but not greater, so be careful about making assumptions of *capacity* after calling this.

Parameters

`aNewsize`

6.24.2. FILE_LINE_READER

```
#include <richio.h>

class FILE_LINE_READER
```

Class FILE_LINE_READER is a LINE_READER that reads from an open file.

File must be already open so that this class can exist without any UI policy.

Public Constructors	
	`FILE_LINE_READER(const std::string &, unsigned, unsigned)` Constructor FILE_LINE_READER takes *aFileName* and the size of the desired line buffer and opens the file and assumes the obligation to close it.
	`FILE_LINE_READER(FILE *, const std::string &, bool, unsigned, unsigned)` Constructor FILE_LINE_READER takes an open FILE and the size of the desired line buffer and takes ownership of the open file, i.e.
Public Destructors	

Public Constructors	
	`~FILE_LINE_READER()` Destructor may or may not close the open file, depending on *doOwn* in constructor.
Public Operators	
char *	`operator char *()` Operator char* is a casting operator that returns a char* pointer to the start of the line buffer.
Public Methods	
char *	`ReadLine()` Function ReadLine reads a line of text into the buffer and increments the line number counter.
void	`Rewind()` Function Rewind rewinds the file and resets the line number back to zero.
const std::string &	`GetSource() const` Function GetSource returns the name of the source of the lines in an abstract sense.
char *	`Line() const` Function Line returns a pointer to the last line that was read in.
unsigned	`LineNumber() const` Function Line Number returns the line number of the last line read from this LINE_READER.
unsigned	`Length() const` Function Length returns the number of bytes in the last line read from this LINE_READER.
Protected Variables	
bool	`m_iOwn` if I own the file, I'll promise to close it, else not.
FILE *	`m_fp` I may own this file, but might not.
unsigned	`m_length` no. bytes in line before trailing nul.
unsigned	`m_lineNum`

Public Constructors	
char *	`m_line` the read line of UTF8 text
unsigned	`m_capacity` no. bytes allocated for line.
unsigned	`m_maxLineLength` maximum allowed capacity using resizing.
std::string	`m_source` origin of text lines, e.g. filename or "clipboard"
Protected Methods	
void	`expandCapacity(unsigned)` Function expandCapacity will expand the capacity of *line* up to maxLineLength but not greater, so be careful about making assumptions of *capacity* after calling this.

Members

FILE_LINE_READER

```
FILE_LINE_READER(const std::string & aFileName,
                 unsigned aStartingLineNumber = 0,
                 unsigned aMaxLineLength = LINE_READER_LINE_DEFAULT_MAX)
```

Constructor FILE_LINE_READER takes *aFileName* and the size of the desired line buffer and opens the file and assumes the obligation to close it.

Parameters

`aFileName`	is the name of the file to open and to use for error reporting purposes.
`aStartingLineNumber`	is the initial line number to report on error, and is accessible here for the case where multiple DSNLEXERs are reading from the same file in sequence, all from the same open file (with *doOwn* = false). Internally it is incremented by one after each ReadLine(), so the first reported line number will always be one greater than what is provided here. (**Default value**: `0`)
`aMaxLineLength`	is the number of bytes to use in the line buffer. (**Default value**: `LINE_READER_LINE_DEFAULT_MAX`)

Throws

`IO_ERROR` — if *aFileName* cannot be opened.

FILE_LINE_READER

```
FILE_LINE_READER(FILE * aFile,
                 const std::string & aFileName,
                 bool doOwn = true,
                 unsigned aStartingLineNumber = 0,
                 unsigned aMaxLineLength = LINE_READER_LINE_DEFAULT_MAX)
```

Constructor FILE_LINE_READER takes an open FILE and the size of the desired line buffer and takes ownership of the open file, i.e.

assumes the obligation to close it.

Parameters

`aFile`	is an open file.
`aFileName`	is the name of the file for error reporting purposes.
`doOwn`	if true, means I should close the open file, else not. (**Default value**: `true`)
`aStartingLineNumber`	is the initial line number to report on error, and is accessible here for the case where multiple DSNLEXERs are reading from the same file in sequence, all from the same open file (with *doOwn* = false). Internally it is incremented by one after each ReadLine(), so the first reported line number will always be one greater than what is provided here. (**Default value**: `0`)
`aMaxLineLength`	is the number of bytes to use in the line buffer. (**Default value**: `LINE_READER_LINE_DEFAULT_MAX`)

~FILE_LINE_READER

```
~FILE_LINE_READER()
```

Destructor may or may not close the open file, depending on *doOwn* in constructor.

---

operator char *

```
char * operator char *() const
```

Operator char* is a casting operator that returns a char* pointer to the start of the line buffer.

ReadLine

```
char * ReadLine()
```

Function ReadLine reads a line of text into the buffer and increments the line number counter.

If the line is larger than aMaxLineLength passed to the constructor, then an exception is thrown. The line is nul terminated.

Returns

char* - The beginning of the read line, or NULL if EOF.

Throws

`IO_ERROR` — when a line is too long.

Rewind

```
void Rewind()
```

Function Rewind rewinds the file and resets the line number back to zero.

Line number will go to 1 on first ReadLine().

GetSource

```
const std::string & GetSource() const
```

Function GetSource returns the name of the source of the lines in an abstract sense.

This may be a file or it may be the clipboard or any other source of lines of text. The returned string is useful for reporting error messages.

Line

```
char * Line() const
```

Function Line returns a pointer to the last line that was read in.

LineNumber

```
unsigned LineNumber() const
```

Function Line Number returns the line number of the last line read from this LINE_READER.

Lines start from 1.

Length

```
unsigned Length() const
```

Function Length returns the number of bytes in the last line read from this LINE_READER.

---

expandCapacity

```
void expandCapacity(unsigned aNewsize)
```

Function expandCapacity will expand the capacity of *line* up to maxLineLength but not greater, so be careful about making assumptions of *capacity* after calling this.

Parameters

`aNewsize`

6.24.3. STRING_LINE_READER

```
#include <richio.h>

class STRING_LINE_READER
```

Class STRING_LINE_READER is a LINE_READER that reads from a multiline 8 bit wide std::string.

Public Constructors	
	`STRING_LINE_READER(const std::string &, const std::string &)` Constructor STRING_LINE_READER(const std::string&, const std::string&)
	`STRING_LINE_READER(const STRING_LINE_READER &)` Constructor STRING_LINE_READER(const STRING_LINE_READER&) allows for a continuation of the reading of a stream started by another STRING_LINE_READER.
Public Operators	

Public Constructors	
char *	`operator char *()` Operator char* is a casting operator that returns a char* pointer to the start of the line buffer.
Public Methods	
char *	`ReadLine()` Function ReadLine reads a line of text into the buffer and increments the line number counter.
const std::string &	`GetSource() const` Function GetSource returns the name of the source of the lines in an abstract sense.
char *	`Line() const` Function Line returns a pointer to the last line that was read in.
unsigned	`LineNumber() const` Function Line Number returns the line number of the last line read from this LINE_READER.
unsigned	`Length() const` Function Length returns the number of bytes in the last line read from this LINE_READER.
Protected Variables	
std::string	`m_lines`
size_t	`m_ndx`
unsigned	`m_length` no. bytes in line before trailing nul.
unsigned	`m_lineNum`
char *	`m_line` the read line of UTF8 text
unsigned	`m_capacity` no. bytes allocated for line.
unsigned	`m_maxLineLength` maximum allowed capacity using resizing.
std::string	`m_source` origin of text lines, e.g. filename or "clipboard"
Protected Methods	

Public Constructors	
void	`expandCapacity(unsigned)` Function expandCapacity will expand the capacity of *line* up to maxLineLength but not greater, so be careful about making assumptions of *capacity* after calling this.

Members

STRING_LINE_READER

```
STRING_LINE_READER(const std::string & aString,
                   const std::string & aSource)
```

Constructor STRING_LINE_READER(const std::string&, const std::string&)

Parameters

`aString`	is a source string consisting of one or more lines of text, where multiple lines are separated with a ' ' character. The last line does not necessarily need a trailing ' '.
`aSource`	describes the source of aString for error reporting purposes can be anything meaninful, such as wxT("clipboard").

STRING_LINE_READER

```
STRING_LINE_READER(const STRING_LINE_READER & aStartingPoint)
```

Constructor STRING_LINE_READER(const STRING_LINE_READER&) allows for a continuation of the reading of a stream started by another STRING_LINE_READER.

Any stream offset and source name are used from *aStartingPoint*.

Parameters

`aStartingPoint`	

---

operator char *

```
char * operator char *() const
```

Operator char* is a casting operator that returns a char* pointer to the start of the line buffer.

ReadLine

```
char * ReadLine()
```

Function ReadLine reads a line of text into the buffer and increments the line number counter.

If the line is larger than aMaxLineLength passed to the constructor, then an exception is thrown. The line is nul terminated.

Returns

char* - The beginning of the read line, or NULL if EOF.

Throws

IO_ERROR — when a line is too long.

GetSource

```
const std::string & GetSource() const
```

Function GetSource returns the name of the source of the lines in an abstract sense.

This may be a file or it may be the clipboard or any other source of lines of text. The returned string is useful for reporting error messages.

Line

```
char * Line() const
```

Function Line returns a pointer to the last line that was read in.

LineNumber

```
unsigned LineNumber() const
```

Function Line Number returns the line number of the last line read from this LINE_READER.

Lines start from 1.

Length

```
unsigned Length() const
```

Function Length returns the number of bytes in the last line read from this LINE_READER.

---

expandCapacity

```
void expandCapacity(unsigned aNewsize)
```

Function expandCapacity will expand the capacity of *line* up to maxLineLength but not greater, so be careful about making assumptions of *capacity* after calling this.

Parameters

`aNewsize`

6.24.4. OUTPUTFORMATTER

```
#include <richio.h>

class OUTPUTFORMATTER
```

Class OUTPUTFORMATTER is an important interface (abstract class) used to output 8 bit text in a convenient way.

The primary interface is "printf() - like" but with support for indentation control. The destination of the 8 bit wide text is up to the implementer.

The implementer only has to implement the write() function, but can also optionally re-implement GetQuoteChar().

If you want to output a std::string, then use TO_UTF8() on it before passing it as an argument to Print().

Since this is an abstract interface, only classes derived from this one may actually be used.

Public Methods	
int PRINTF_FUNC	`Print(int, const char *, ...)` Function Print formats and writes text to the output stream.
const char *	`GetQuoteChar(const char *)` Function GetQuoteChar performs quote character need determination.

Public Methods	
std::string	`Quotes(const std::string &)` Function Quotes checks *aWrapee* input string for a need to be quoted (e.g.
std::string	`Quotew(const std::string &)`
Protected Constructors	
	`OUTPUTFORMATTER(int, char)`
Protected Destructors	
	`~OUTPUTFORMATTER()`
Protected Static Methods	
static const char *	`GetQuoteChar(const char *, const char *)` Function GetQuoteChar performs quote character need determination according to the Specctra DSN specification.
Protected Methods	
void	`write(const char *, int)` Function write should be coded in the interface implementation (derived) classes.

Members

Print

```
int PRINTF_FUNC Print(int nestLevel,
                      const char * fmt,
                      ...)
```

Function Print formats and writes text to the output stream.

Parameters

`nestLevel`	The multiple of spaces to precede the output with.
`fmt`	A printf() style format string.

``

Returns

int - the number of characters output.

Throws

`IO_ERROR` — there is a problem outputting, such as a full disk.

GetQuoteChar

```
const char * GetQuoteChar(const char * wrapee)
```

Function GetQuoteChar performs quote character need determination.

It returns the quote character as a single character string for a given input wrapee string. If the wrappee does not need to be quoted, the return value is "" (the null string), such as when there are no delimiters in the input wrapee string. If you want the quote_char to be assuredly not "", then pass in "(" as the wrappee.

Implementations are free to override the default behavior, which is to call the static function of the same name.

Parameters

`wrapee`	A string that might need wrapping on each end.

Returns

const char* - the quote_char as a single character string, or "" if the wrapee does not need to be wrapped.

Quotes

```
std::string Quotes(const std::string & aWrapee)
```

Function Quotes checks *aWrapee* input string for a need to be quoted (e.g.

contains a ')' character or a space), and for " double quotes within the string that need to be escaped such that the DSNLEXER will correctly parse the string from a file later.

Parameters

`aWrapee`	is a string that might need wraping in double quotes, and it might need to have its internal content escaped, or not.

Returns

std::string - whose c_str() function can be called for passing to printf() style functions that output UTF8 encoded s-expression streams.

Throws

`IO_ERROR` — there is any kind of problem with the input string.

Quotew

```
std::string Quotew(const std::string & aWrapee)
```

Parameters

`aWrapee`

OUTPUTFORMATTER

```
OUTPUTFORMATTER(int aReserve,
                char aQuoteChar = '"')
```

Parameters

`aReserve`

`aQuoteChar` (**Default value**: `'"'`)

~OUTPUTFORMATTER

```
~OUTPUTFORMATTER()
```

GetQuoteChar

```
static const char * GetQuoteChar(const char * wrapee,
                                 const char * quote_char)
```

Function GetQuoteChar performs quote character need determination according to the Specctra DSN specification.

Parameters

`wrapee` A string that might need wrapping on each end.

`quote_char` A single character C string which provides the current quote character, should it be needed by the wrapee.

Returns

const char* - the quote_char as a single character string, or "" if the wrapee does not need to be wrapped.

---

write

```
void write(const char * aOutBuf,
           int aCount)
```

Function write should be coded in the interface implementation (derived) classes.

Parameters

aOutBuf	is the start of a byte buffer to write.
aCount	tells how many bytes to write.

Throws

IO_ERROR — there is a problem outputting, such as a full disk.

6.24.5. STRING_FORMATTER

```
#include <richio.h>

class STRING_FORMATTER
```

Class STRING_FORMATTER implements OUTPUTFORMATTER to a memory buffer.

After Print()ing the string is available through GetString()

Public Constructors	
	STRING_FORMATTER(int, char) Constructor STRING_FORMATTER reserves space in the buffer.
Public Methods	
void	Clear() Function Clear clears the buffer and empties the internal string.
void	StripUseless() Function StripUseless removes whitespace, '(', and ')' from the mystring.
const std::string &	GetString()
const char *	GetQuoteChar(const char *) Function GetQuoteChar performs quote character need determination.
int PRINTF_FUNC	Print(int, const char *, ...) Function Print formats and writes text to the output stream.

Public Constructors	
std::string	`Quotes(const std::string &)` Function Quotes checks *aWrapee* input string for a need to be quoted (e.g.
std::string	`Quotew(const std::string &)`
Protected Static Methods	
static const char *	`GetQuoteChar(const char *, const char *)` Function GetQuoteChar performs quote character need determination according to the Specctra DSN specification.
Protected Methods	
void	`write(const char *, int)` Function write should be coded in the interface implementation (derived) classes.

Members

STRING_FORMATTER

```
STRING_FORMATTER(int aReserve,
                 char aQuoteChar = '"')
```

Constructor STRING_FORMATTER reserves space in the buffer.

Parameters

`aReserve`	
`aQuoteChar`	(**Default value**: `'"'`)

Clear

```
void Clear()
```

Function Clear clears the buffer and empties the internal string.

StripUseless

```
void StripUseless()
```

Function StripUseless removes whitespace, '(', and ')' from the mystring.

GetString

```
const std::string & GetString()
```

GetQuoteChar

```
const char * GetQuoteChar(const char * wrapee)
```

Function GetQuoteChar performs quote character need determination.

It returns the quote character as a single character string for a given input wrapee string. If the wrappee does not need to be quoted, the return value is "" (the null string), such as when there are no delimiters in the input wrapee string. If you want the quote_char to be assuredly not "", then pass in "(" as the wrappee.

Implementations are free to override the default behavior, which is to call the static function of the same name.

Parameters

`wrapee`	A string that might need wrapping on each end.

Returns

const char* - the quote_char as a single character string, or "" if the wrapee does not need to be wrapped.

Print

```
int PRINTF_FUNC Print(int nestLevel,
                      const char * fmt,
                      ...)
```

Function Print formats and writes text to the output stream.

Parameters

`nestLevel`	The multiple of spaces to precede the output with.
`fmt`	A printf() style format string.

``

Returns

int - the number of characters output.

Throws

`IO_ERROR` — there is a problem outputting, such as a full disk.

Quotes

```
std::string Quotes(const std::string & aWrapee)
```

Function Quotes checks *aWrapee* input string for a need to be quoted (e.g.

contains a ')' character or a space), and for " double quotes within the string that need to be escaped such that the DSNLEXER will correctly parse the string from a file later.

Parameters

`aWrapee`	is a string that might need wraping in double quotes, and it might need to have its internal content escaped, or not.

Returns

std::string - whose c_str() function can be called for passing to printf() style functions that output UTF8 encoded s-expression streams.

Throws

`IO_ERROR` — there is any kind of problem with the input string.

Quotew

```
std::string Quotew(const std::string & aWrapee)
```

Parameters

`aWrapee`	

GetQuoteChar

```
static const char * GetQuoteChar(const char * wrapee,
                                 const char * quote_char)
```

Function GetQuoteChar performs quote character need determination according to the Specctra DSN specification.

Parameters

wrapee	A string that might need wrapping on each end.
quote_char	A single character C string which provides the current quote character, should it be needed by the wrapee.

Returns

const char* - the quote_char as a single character string, or "" if the wrapee does not need to be wrapped.

---

write

```
void write(const char * aOutBuf,
           int aCount)
```

Function write should be coded in the interface implementation (derived) classes.

Parameters

aOutBuf	is the start of a byte buffer to write.
aCount	tells how many bytes to write.

Throws

IO_ERROR — there is a problem outputting, such as a full disk.

6.24.6. FILE_OUTPUTFORMATTER

```
#include <richio.h>

class FILE_OUTPUTFORMATTER
```

Class FILE_OUTPUTFORMATTER may be used for text file output.

It is about 8 times faster than STREAM_OUTPUTFORMATTER for file streams.

Public Constructors	
	FILE_OUTPUTFORMATTER(const std::string &, const char *, char) Constructor.
Public Destructors	
	~FILE_OUTPUTFORMATTER()
Public Methods	

Public Constructors	
const char *	`GetQuoteChar(const char *)` Function GetQuoteChar performs quote character need determination.
int PRINTF_FUNC	`Print(int, const char *, ...)` Function Print formats and writes text to the output stream.
std::string	`Quotes(const std::string &)` Function Quotes checks *aWrapee* input string for a need to be quoted (e.g.
std::string	`Quotew(const std::string &)`
Protected Variables	
FILE *	`m_fp` takes ownership
std::string	`m_filename`
Protected Static Methods	
static const char *	`GetQuoteChar(const char *, const char *)` Function GetQuoteChar performs quote character need determination according to the Specctra DSN specification.
Protected Methods	
void	`write(const char *, int)` Function write should be coded in the interface implementation (derived) classes.

Members

FILE_OUTPUTFORMATTER

```
FILE_OUTPUTFORMATTER(const std::string & aFileName,
                     const char * aMode = "wt",
                     char aQuoteChar = '"')
```

Constructor.

Parameters

`aFileName`	is the full filename to open and save to as a text file.
`aMode`	is what you would pass to wxFopen()'s mode, defaults to wxT("wt") for text files that are to be created here and now. (**Default value**: `"wt"`)
`aQuoteChar`	is a char used for quoting problematic strings (with whitespace or special characters in them). (**Default value**: `'"'`)

Throws

`IO_ERROR` — if the file cannot be opened.

~FILE_OUTPUTFORMATTER

```
~FILE_OUTPUTFORMATTER()
```

GetQuoteChar

```
const char * GetQuoteChar(const char * wrapee)
```

Function GetQuoteChar performs quote character need determination.

It returns the quote character as a single character string for a given input wrapee string. If the wrappee does not need to be quoted, the return value is "" (the null string), such as when there are no delimiters in the input wrapee string. If you want the quote_char to be assuredly not "", then pass in "(" as the wrappee.

Implementations are free to override the default behavior, which is to call the static function of the same name.

Parameters

`wrapee`	A string that might need wrapping on each end.

Returns

const char* - the quote_char as a single character string, or "" if the wrapee does not need to be wrapped.

Print

```
int PRINTF_FUNC Print(int nestLevel,
                      const char * fmt,
                      ...)
```

Function Print formats and writes text to the output stream.

Parameters

`nestLevel`	The multiple of spaces to precede the output with.
`fmt`	A printf() style format string.

``

Returns

int - the number of characters output.

Throws

`IO_ERROR` — there is a problem outputting, such as a full disk.

Quotes

```
std::string Quotes(const std::string & aWrapee)
```

Function Quotes checks *aWrapee* input string for a need to be quoted (e.g.

contains a ')' character or a space), and for " double quotes within the string that need to be escaped such that the DSNLEXER will correctly parse the string from a file later.

Parameters

`aWrapee`	is a string that might need wraping in double quotes, and it might need to have its internal content escaped, or not.

Returns

std::string - whose c_str() function can be called for passing to printf() style functions that output UTF8 encoded s-expression streams.

Throws

`IO_ERROR` — there is any kind of problem with the input string.

Quotew

```
std::string Quotew(const std::string & aWrapee)
```

Parameters

`aWrapee`

GetQuoteChar

```
static const char * GetQuoteChar(const char * wrapee,
                                 const char * quote_char)
```

Function GetQuoteChar performs quote character need determination according to the Specctra DSN specification.

Parameters

`wrapee`	A string that might need wrapping on each end.
`quote_char`	A single character C string which provides the current quote character, should it be needed by the wrapee.

Returns

const char* - the quote_char as a single character string, or "" if the wrapee does not need to be wrapped.

---

write

```
void write(const char * aOutBuf,
           int aCount)
```

Function write should be coded in the interface implementation (derived) classes.

Parameters

`aOutBuf`	is the start of a byte buffer to write.
`aCount`	tells how many bytes to write.

Throws

`IO_ERROR` — there is a problem outputting, such as a full disk.

6.25. SoftPLC Exceptions

These are classes thrown as exceptions.

Classes

ERROR

RUNTIME_ERROR

IO_ERROR

PARSE_ERROR

6.25.1. THROW_IO_ERROR()

```
#define THROW_IO_ERROR(msg) throw IO_ERROR( msg, __FILE__, __FUNCTION__, __LINE__ )
```

macro which captures the "call site" values of **FILE_**, **__FUNCTION** & **LINE**

6.25.2. THROW_PARSE_ERROR()

```
#define THROW_PARSE_ERROR(aProblem, aSource, aInputLine, aLineNumber, aByteIndex)
throw PARSE_ERROR( aProblem, __FILE__, __FUNCTION__, __LINE__, aSource, aInputLine,
aLineNumber, aByteIndex )
```

6.25.3. ERROR

```
#include <exceptions.h>

class ERROR
```

Class Error is an exception that is thrown to indicate problems in the SoftPLC C++ API usage.

It holds a string that can be obtained with the what() function.

Public Constructors	
	`ERROR(int, const char *, ...)` Constructor takes a printf() style format string with a variable number of matching arguments to formulate the text of the exception.
	`ERROR(int)`
Public Destructors	
	`~ERROR()`
Public Methods	
const char *	`what() const` Function what intends to behave similar to class std::exception's what() function by returning a textual summary of the nature of the exception.
int	`ErrorCode() const` Function ErrorCode returns the int value passed as *aErrorCode* to the constructor.
ERROR &	`SetErrorCode(int)`
ERROR &	`SetText(const char *)`
Protected Variables	

Public Constructors	
std::string	`text`
int	`error_code`

Members

ERROR

```
ERROR(int aErrorCode,
      const char * aMessage,
      ...)
```

Constructor takes a printf() style format string with a variable number of matching arguments to formulate the text of the exception.

Parameters

`aErrorCode`

`aMessage`

``

ERROR

```
ERROR(int aErrorCode)
```

Parameters

`aErrorCode`

~ERROR

```
~ERROR()
```

---

what

```
const char * what() const
```

Function what intends to behave similar to class std::exception's what() function by returning a textual summary of the nature of the exception.

ErrorCode

```
int ErrorCode() const
```

Function ErrorCode returns the int value passed as *aErrorCode* to the constructor.

SetErrorCode

```
ERROR & SetErrorCode(int aErrorCode)
```

Parameters

`aErrorCode`

SetText

```
ERROR & SetText(const char * aText)
```

Parameters

`aText`

6.25.4. RUNTIME_ERROR

```
#include <exceptions.h>

class RUNTIME_ERROR
```

Class RUNTIME_ERROR is used in the SoftPLC runtime for its ability to capture execution context or datatable address context of the ladder logic engine.

It can hold an extended address consisting of file, element_or_rung, and word_or_index.

Public Constructors	
	`RUNTIME_ERROR(int, const char *, ...)` Constructor RUNTIME_ERROR(int aErrorCode, const char* aFormatString, ...) takes a printf() style format string with a variable number of matching arguments to formulate the text of the exception.

Public Constructors	
	`RUNTIME_ERROR(const char *, int)`
	`RUNTIME_ERROR(const char *, int, va_list)`
Public Methods	
std::string	`Problem() const`
RUNTIME_ERROR &	`SetFile(int)`
RUNTIME_ERROR &	`SetRung(int)`
RUNTIME_ERROR &	`SetInstrIndex(int)`
int	`File() const`
int	`Rung() const`
int	`InstrIndex() const`
bool	`IsCompileError() const`
const char *	`what() const` Function what intends to behave similar to class std::exception's what() function by returning a textual summary of the nature of the exception.
int	`ErrorCode() const` Function ErrorCode returns the int value passed as *aErrorCode* to the constructor.
ERROR &	`SetErrorCode(int)`
ERROR &	`SetText(const char *)`
Protected Variables	
int	`file`
int	`element_or_rung`
int	`word_or_index`
std::string	`text`
int	`error_code`

Members

RUNTIME_ERROR

```
RUNTIME_ERROR(int aErrorCode,
              const char * aFormatString,
              ...)
```

Constructor RUNTIME_ERROR(int aErrorCode, const char* aFormatString, ...) takes a printf() style format string with a variable number of matching arguments to formulate the text of the exception.

Parameters

aErrorCode	is an error number or perhaps a value from an enum
aFormatString	is a C printf() style format string used to format the successive parameters.

``

RUNTIME_ERROR

```
RUNTIME_ERROR(const char * aMessage,
              int aErrorCode)
```

Parameters

aMessage
aErrorCode

RUNTIME_ERROR

```
RUNTIME_ERROR(const char * aMessage,
              int aErrorCode,
              va_list ap)
```

Parameters

aMessage
aErrorCode
ap

Problem

```
std::string Problem() const
```

SetFile

```
RUNTIME_ERROR & SetFile(int aFile)
```

Parameters

aFile

SetRung

```
RUNTIME_ERROR & SetRung(int aRung)
```

Parameters

aRung

SetInstrIndex

```
RUNTIME_ERROR & SetInstrIndex(int aIndex)
```

Parameters

aIndex

File

```
int File() const
```

Rung

```
int Rung() const
```

InstrIndex

```
int InstrIndex() const
```

IsCompileError

```
bool IsCompileError() const
```

what

```
const char * what() const
```

Function what intends to behave similar to class std::exception's what() function by returning a textual summary of the nature of the exception.

ErrorCode

```
int ErrorCode() const
```

Function ErrorCode returns the int value passed as *aErrorCode* to the constructor.

SetErrorCode

```
ERROR & SetErrorCode(int aErrorCode)
```

Parameters

`aErrorCode`

SetText

```
ERROR & SetText(const char * aText)
```

Parameters

`aText`

6.25.5. IO_ERROR

```
#include <exceptions.h>
```

```
class IO_ERROR
```

Class IO_ERROR is a class used to hold an error message and may be used when throwing exceptions containing meaningful error messages.

Public Constructors	
	`IO_ERROR(const std::string &, const char *, const char *, int)` Constructor is typically not used directly, instead use macro THROW_IO_ERROR() to wrap a call to this constructor at the call site.
	`IO_ERROR()`
Public Destructors	
	`~IO_ERROR()`
Public Methods	
void	`init(const std::string &, const char *, const char *, int)`
const std::string	`Problem() const` what was the problem?
const std::string	`Where() const` where did the Problem() occur?
const std::string	`What() const` A composite of Problem() and Where()
const char *	`what() const` Function what intends to behave similar to class std::exception's what() function by returning a textual summary of the nature of the exception.
int	`ErrorCode() const` Function ErrorCode returns the int value passed as *aErrorCode* to the constructor.
ERROR &	`SetErrorCode(int)`
ERROR &	`SetText(const char *)`
Protected Variables	
std::string	`problem`
std::string	`where`
std::string	`text`
int	`error_code`

Members

IO_ERROR

```
IO_ERROR(const std::string & aProblem,
         const char * aThrowersFile,
         const char * aThrowersFunction,
         int aThrowersLineNumber)
```

Constructor is typically not used directly, instead use macro THROW_IO_ERROR() to wrap a call to this constructor at the call site.

Parameters

`aProblem`	is Problem() text.
`aThrowersFile`	is the **FILE** preprocessor macro but generated at the source file of thrower.
`aThrowersFunction`	is the function name at the throw site.
`aThrowersLineNumber`	is the source code line number of the throw.

IO_ERROR

```
IO_ERROR()
```

~IO_ERROR

```
~IO_ERROR()
```

---

init

```
void init(const std::string & aProblem,
          const char * aThrowersFile,
          const char * aThrowersFunction,
          int aThrowersLineNumber)
```

Parameters

`aProblem`

`aThrowersFile`

`aThrowersFunction`

`aThrowersLineNumber`

Problem

```
const std::string Problem() const
```

what was the problem?

Where

```
const std::string Where() const
```

where did the Problem() occur?

What

```
const std::string What() const
```

A composite of Problem() and Where()

what

```
const char * what() const
```

Function what intends to behave similar to class std::exception's what() function by returning a textual summary of the nature of the exception.

ErrorCode

```
int ErrorCode() const
```

Function ErrorCode returns the int value passed as *aErrorCode* to the constructor.

SetErrorCode

```
ERROR & SetErrorCode(int aErrorCode)
```

Parameters

`aErrorCode`

SetText

```
ERROR & SetText(const char * aText)
```

Parameters

`aText`

6.25.6. PARSE_ERROR

```
#include <exceptions.h>

struct PARSE_ERROR
```

Struct PARSE_ERROR contains a filename or source description, a problem input line, a line number, a byte offset, and an error message which contains the the caller's report and his call site information: CPP source file, function, and line number.

Public Constructors	
	`PARSE_ERROR(const std::string &, const char *, const char *, int, const std::string &, const char *, int, int)` Constructor which is normally called via the macro THROW_PARSE_ERROR so that **FILE** and **FUNCTION** and **LINE** can be captured from the call site.
Public Destructors	
	`~PARSE_ERROR()`
Public Variables	
int	`lineNumber` at which line number, 1 based index.
int	`byteIndex` at which byte offset within the line, 1 based index

Public Constructors	
std::string	`inputLine` problem line of input [say, from a LINE_READER]. this is brought up in original byte format rather than std::string form, incase there was a problem with the encoding, in which case converting to std::string is not reliable in this context.
Public Methods	
void	`init(const std::string &, const char *, const char *, int, const std::string &, const char *, int, int)`
void	`init(const std::string &, const char *, const char *, int)`
const std::string	`Problem() const` what was the problem?
const std::string	`Where() const` where did the Problem() occur?
const std::string	`What() const` A composite of Problem() and Where()
const char *	`what() const` Function what intends to behave similar to class std::exception's what() function by returning a textual summary of the nature of the exception.
int	`ErrorCode() const` Function ErrorCode returns the int value passed as *aErrorCode* to the constructor.
ERROR &	`SetErrorCode(int)`
ERROR &	`SetText(const char *)`
Protected Constructors	
	`PARSE_ERROR()`
Protected Variables	
std::string	`problem`
std::string	`where`
std::string	`text`
int	`error_code`

Members

PARSE_ERROR

```
PARSE_ERROR(const std::string & aProblem,
            const char * aThrowersFile,
```

```
            const char * aThrowersFunction,
            int aThrowersLineNumber,
            const std::string & aSource,
            const char * aInputLine,
            int aLineNumber,
            int aByteIndex)
```

Constructor which is normally called via the macro THROW_PARSE_ERROR so that **FILE** and **FUNCTION** and **LINE** can be captured from the call site.

Parameters

aProblem

aThrowersFile

aThrowersFunction

aThrowersLineNumber

aSource

aInputLine

aLineNumber

aByteIndex

~PARSE_ERROR

```
~PARSE_ERROR()
```

---

init

```
void init(const std::string & aProblem,
          const char * aThrowersFile,
          const char * aThrowersFunction,
          int aThrowersLineNumber,
          const std::string & aSource,
          const char * aInputLine,
          int aLineNumber,
          int aByteIndex)
```

Parameters

aProblem

aThrowersFile

aThrowersFunction

aThrowersLineNumber

aSource

aInputLine

aLineNumber

aByteIndex

---

init

```
void init(const std::string & aProblem,
          const char * aThrowersFile,
          const char * aThrowersFunction,
          int aThrowersLineNumber)
```

Parameters

aProblem

aThrowersFile

aThrowersFunction

aThrowersLineNumber

Problem

```
const std::string Problem() const
```

what was the problem?

Where

```
const std::string Where() const
```

where did the Problem() occur?

What

```
const std::string What() const
```

A composite of Problem() and Where()

what

```
const char * what() const
```

Function what intends to behave similar to class std::exception's what() function by returning a textual summary of the nature of the exception.

ErrorCode

```
int ErrorCode() const
```

Function ErrorCode returns the int value passed as *aErrorCode* to the constructor.

SetErrorCode

```
ERROR & SetErrorCode(int aErrorCode)
```

Parameters

`aErrorCode`

SetText

```
ERROR & SetText(const char * aText)
```

Parameters

`aText`

PARSE_ERROR

```
PARSE_ERROR()
```

6.26. Service Thread Support

These two class templates are what a TLM uses to do work on its own thread instead of on the PLC scan.

They are designed to be used together, and neither one allocates from the heap after construction, so a service thread built from them adds no unbounded latency to a running controller. MYPOOL hands out fixed-size blocks of pre-reserved memory and constructs a message in place; MSG_BOX carries that message across the thread boundary in FIFO order and blocks the reader until one arrives; the reader returns the block to the pool when it is finished. Ownership passes with the pointer at each step.

The service_thread example is a complete TLM written this way and is the place to start.

Classes

MSG_BOX

MYPOOL

6.26.1. MSG_BOX

```
#include <msg_box.h>

template<class MSG, std::size_t>
class MSG_BOX
```

CLASS MSG_BOX is a thread-safe FIFO message box implemented as a fixed-size circular ring buffer.

This class uses a standard std::mutex instead of a recursive mutex for maximum efficiency in real-time execution loops. It guarantees ZERO dynamic memory allocations or deallocations after initialization.

Template Parameters	`class MSG`
	The payload message class type. Typically features a virtual destructor.
	
	`std::size_t Capacity`

Public Constructors	
	`MSG_BOX()` Constructs the fixed-size message box.
Public Destructors	
	`~MSG_BOX()` Destructor deletes any remaining unread messages left in the ring.
Public Methods	

bool	`Test() const` Checks if there is an entry in the mailbox (Non-blocking lock-free peek).
void	`Write(const MSG *)` Posts an entry into the mailbox and assumes memory ownership.
MSG *	`Read()` Blocks indefinitely until an entry arrives, then removes and returns it.
MSG *	`Read(int)` Blocks up to a microsecond timeout waiting for a message.
MSG *	`CondRead()` Non-blocking conditional read.
MSG *	`Front() const` Returns a tracking pointer to the front item without removing it.
unsigned	`Size() const` Returns the total number of messages currently in the box.
Protected Variables	
std::mutex	`lock` Guards the ring buffer and its three indices against concurrent access.
std::counting_semaphore<Capacity>	`sem` Counts unread messages, so a reader can block until one is posted.
MSG *	`buffer` The ring of message pointers; owns every entry it holds.
std::size_t	`head` Index of the oldest unread message, where the next Read() takes from.
std::size_t	`tail` Index of the next free slot, where the next Write() puts its message.
std::size_t	`count` How many messages are currently in the ring, never more than Capacity.

Members

MSG_BOX

```
MSG_BOX()
```

Constructs the fixed-size message box.

~MSG_BOX

```
~MSG_BOX()
```

Destructor deletes any remaining unread messages left in the ring.

Test

```
bool Test() const
```

Checks if there is an entry in the mailbox (Non-blocking lock-free peek).

Write

```
void Write(const MSG * aMessage)
```

Posts an entry into the mailbox and assumes memory ownership.

If the buffer is completely full, this call drops the data to protect real-time execution loops from blocking or heap thrashing.

Parameters

`aMessage`	Pointer to the message. Can be NULL.

Read

```
MSG * Read()
```

Blocks indefinitely until an entry arrives, then removes and returns it.

Read

```
MSG * Read(int uSecsTimeout)
```

Blocks up to a microsecond timeout waiting for a message.

Parameters

`uSecsTimeout`	Max wait in microseconds. Pass -1 to wait forever.

Returns

MSG* The message pointer, or NULL if the timeout expires.

CondRead

```
MSG * CondRead()
```

Non-blocking conditional read.

Returns

MSG* The front entry if one exists, otherwise NULL immediately.

Front

```
MSG * Front() const
```

Returns a tracking pointer to the front item without removing it.

This is a lock-free peek function.

Size

```
unsigned Size() const
```

Returns the total number of messages currently in the box.

6.26.2. MYPOOL

```
#include <pool.h>

template<class RET, std::size_t, bool>
class MYPOOL
```

Class MYPOOL is a thread-safe, fixed-size block memory pool.

This class provides deterministic, high-speed allocations out of a static internal buffer. It utilizes the C++ Standard Library's polymorphic memory structures to guarantee thread safety and strict alignment without requiring custom overrides inside the allocated classes.

Template Parameters	`class RET`
	The type of object managed and allocated by the pool.
	
	`std::size_t InstanceCount`
	`bool ThreadSafe`
	Default value: `true`

Public Constructors	
	`MYPOOL()`
Public Methods	
void	`init()` Re-threads the pool into perfectly aligned, distinct memory segments.
RET *	`Alloc(Args &&...)` Function Alloc allocates an entry out of the perfectly aligned free-list and calls RET's constructor of the matching type based on arguments to this function.
void	`Free(RET *)` Function Free explicitly runs the destructor and puts the block back onto the free-list.
void *	`AllocRaw()` Function AllocRaw grabs a raw memory block without running any constructors.
void	`FreeRaw(void *)` Function FreeRaw re-pools a raw memory block without running any destructors.
int	`FreeCount() const` Function FreeCount returns the number of blocks still available, i.e.

Members

MYPOOL

```
MYPOOL()
```

init

```
void init()
```

Re-threads the pool into perfectly aligned, distinct memory segments.

Alloc

```
template<typename...>
RET * Alloc(Args &&... args)
```

Function Alloc allocates an entry out of the perfectly aligned free-list and calls RET's constructor of the matching type based on arguments to this function.

Parameters

`args`

Template Parameters

`Args`

Free

```
void Free(RET * p)
```

Function Free explicitly runs the destructor and puts the block back onto the free-list.

Parameters

`p`

AllocRaw

```
void * AllocRaw()
```

Function AllocRaw grabs a raw memory block without running any constructors.

FreeRaw

```
void FreeRaw(void * p)
```

Function FreeRaw re-pools a raw memory block without running any destructors.

Parameters

`p`

FreeCount

```
int FreeCount() const
```

Function FreeCount returns the number of blocks still available, i.e.

the number of further Alloc() calls that can succeed before the pool is exhausted.
