PRECISION32™ SI32LIBRARY OVERVIEW

1. Introduction

A 32-bit platform with large memory enables a big and complex firmware system on the device. This complexity can slow development, as firmware consists of more layers with interweaving tasks and threads that are more difficult to create and debug.

The si32Library is a set of flexible, reusable, and portable source modules enabling core application level functionality for Silicon Laboratories 32-bit Precision32™ MCUs. It includes facilities for debug logging, memory allocation, data collections, data transfers, and cooperative multitasking. The si32Library package provides working abstractions of the hardware layer, reduces coding effort, and provides structure to aid and speed up top-layer application development.

The si32Library is written in C99 and leverages the Silicon Laboratories si32Hal hardware access layer. This document describes the components of the si32Library. A detailed and complete description of the si32Library component functions can be found in the corresponding header files. The library is installed as part of the Precision32 Software Development Kit package available at www.silabs.com/32bit-software.

The primary architectural elements of an application built with the si32Library are:

- **Application**: the application-specific code
- **si32Library**: reusable object based software modules
- **Real Time Operating System**: 3rd party multitasking library
- **Hardware Access**: CMSIS-inspired HAL encapsulating hardware peripherals
- **Hardware**: the MCU itself, support chips, and boards

Figure 1 depicts this si32Library system.

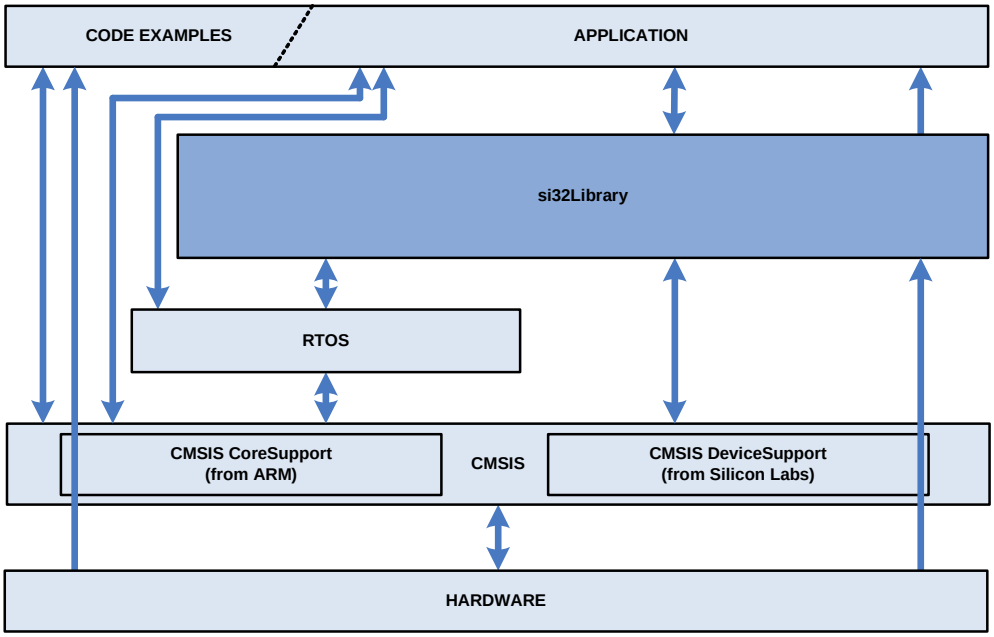


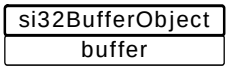
Figure 1. si32Library System

1.1. Diagram Conventions

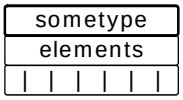
Portions of the si32Library are diagrammed using a UML-like convention, as depicted by the types of graphical elements shown in Figure 2.

A double line box is used to indicate the subject of the diagram.

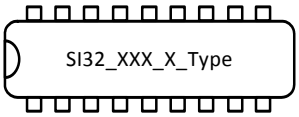
A single line box is used to indicate a base type, from which the thing being described is derived.



A named box is used to indicate an object aggregated by or composited within the subject. The name used by the subject to reference it is indicated.

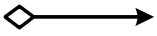


An "array" box is used to indicate a cArray of some integral or structural type; i.e., a non-object. The name that it is given by the object that owns it is indicated beneath the element type.

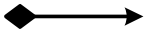


A chip box indicates a hardware peripheral.

This type of arrow means that the thing pointed to is the base type, then the thing pointing to it is the derived type.



This type of arrow means that the thing on the diamond side has a pointer, or an index, to the thing pointed to. It indicates "aggregation".



This type of arrow means that the thing on the diamond side contains the thing pointed to, typically as a struct field. It indicates "composition".

This type of arrow indicates a relationship. It is used to indicate things like "these are indices into those", or "these are pointers to those", or "this talks to that".

Figure 2. Graphical Elements

2. Relevant Documentation

Precision32 Application Notes are listed on the following website: www.silabs.com/32bit-mcu.

- AN664: Precision32™ CMSIS and HAL User's Guide
- AN673: Precision32™ Software Development Kit (SDK) Overview

3. Examples

The examples for the si32Library install with the Precision32 package to the **si32-x.y\Examples\si32Library** directory, where **x** is the major SDK version number and **y** is the minor SDK version directory. Some of these examples are simple demonstrations of how to use the components, and some are full application examples like a HID USB-to-UART bridge interface.

4. Component Architecture

The si32Library is structured as a collection of independent and cooperating components. A component is a collection of related objects with a header file that includes any other components it may require, as well as any internal objects or sub-components that it provides. Therefore, to use a component in an application or another component, it is sufficient to include that component's header file and add the component's sources and its dependencies to the project.

All components are named according to the form **somethingComponent**, where **something** is a meaningful name like USB. The only exception is **si32Library**, which is the top level and does not have the **Component** suffix.

Every component is represented in the source tree as a directory. That directory contains the following types of files:

- *** .h**: The component and object header files collectively comprise the interface. All of the documentation about the interface is here and not in the .c files.
- *** .c**: The individual source files collectively comprise the implementation for the component. There may be more than one implementation of any given interface. Documentation relevant to a specific implementation is found here.

Some component directories contain subdirectories whose names also end in **Component**. These are sub-components and follow the same structure recursively. Sub-components are used to provide logical groupings of components.

There are some specific files that each application must possess. These files contain various elements that are necessary for a build, but that are specific to each individual application.

- **myBuildOptions.h** contains all of the build options. It allows the application developer to enable or disable various compile time options, including assertions and debug logging.
- **myLinkerOptions_<chain>.<ext>** contains the definitions for application specific configuration options such as the desired stack and heap sizes. This file is included by the tool chain during compilation and is used to configure the stack and heap. The supported **<chain>.<ext>** pairs are **_p32.ld**, **_arm.sct**, **_iar.icf**.

Applications do not include the **myBuildOptions.h** file directly because the **si32BaseComponent** of the si32Library includes it. It is the very first file included because it specifies the build options for the si32Library. Changes to this file to specify different build options require recompliation of the library. This allows each individual application to configure the si32Library as desired without needing to modify any of the library code.

Figure 3 shows the dependencies of an si32Library-based application.

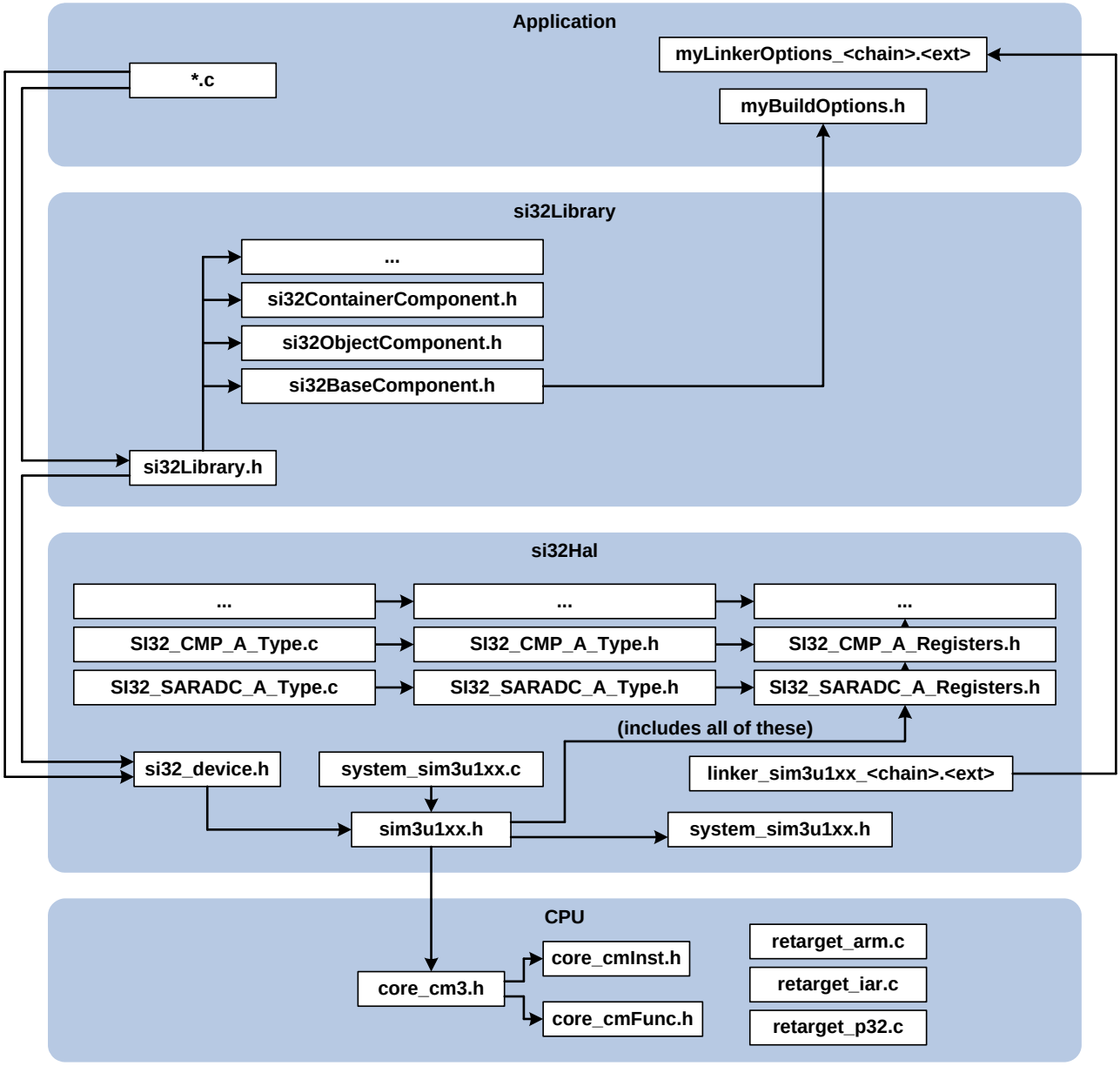


Figure 3. si32Library Application Dependencies

5. Base Component

COMPONENT: si32BaseComponent

REQUIRES: none

PROVIDES: none

This component provides facilities for logging, error handling, memory allocation, and control flow.

5.1. Build Options

si32Base.h provides benign defaults for all available build options. To change a build option for an application, enable or disable that option as appropriate in the application's **myBuildOptions.h**.

- **si32BuildOption_enable_assertions**: Enables si32Assert(). Default is off.
- **si32BuildOption_enable_logging**: Enables logging. Logging must be enabled to use any si32LogXXX() routine. Default is off.
- **si32BuildOption_log_flow**: Enables si32LogPrologue(), si32LogSelf(), si32LogArg(), si32LogAttr(), si32LogVar, si32LogEpilogue(). Default is off.
- **si32BuildOption_log_ref_counts**: Enables logging of reference counts if _log_flow is also enabled. Default is off.
- **si32BuildOption_tally_allocations**: Dynamic allocations encode size information for leak checking. Default is off.
- **si32BuildOption_retention_zone_size**: Retention allocation zone reserve (for MCUs with retention RAM). Default is 1.
- **si32BuildOption_incremental_zone_size**: Incremental allocation zone reserve. Default is 1.
- **si32BuildOption_addressable_labels**: Enable use of computed labels instead of switch abuse for local continuations. Default is off; switch/case is used.
- **si32BuildOption_logPath**: Workstation platforms can log to a file. Default is "/dev/stdout".

5.2. Logging

si32BaseLogger.h defines a number of macros for logging waypoints during program execution. These depend on the si32HAL and require use of **si32HAL/CPU/retarget_<chain>.c** to redirect printf/scanf primitives to ITM, a UART, or some other device.

- **void si32StartLogging(void)**: Configures the logging subsystem and initiates logging. For simulation builds on workstations this opens the log file.
- **void si32StopLogging(void)**: Terminates logging. For simulation builds on workstations this closes the log file.
- **void si32LogBeSilent(bool yn)**: Used to temporarily change the state of the logger without disabling it, or recompiling to turn it off. When silent the log does not send data to ITM etc.
- **bool si32LogIsSilent(void)**: Queries whether the log is silent, or verbose.
- **void si32LogBeIndented(bool yn)**: Controls whether the log is indented. When YES, si32LogPrologue() indents the log, and si32LogEpilogue() outdents the log, in addition to logging the entry/exit of a function, respectively. The default is YES, indent the log.
- **bool si32LogIsIndented(void)**: Queries whether the log is indented, or flat.
- **int si32LogPrint(char* fmt, ...)**: Prints to the log, with parameters similar to printf. Does not indent, even when preceded by si32LogBeIndented(YES). Does not append a newline. For io retargeting, it essentially just wraps printf.
- **int si32LogTrace(char* fmt, ...)**: Prints to the log, with parameters similar to printf. It indents per si32LogBeIndented(yn) and appends a newline.
- **int si32LogWarning(char* fmt, ...)**: Prints to the log, with parameters similar to printf. It indents per si32LogBeIndented(yn). Prepends "WARNING: " and appends a newline.
- **int si32LogError(char* fmt, ...)**: Prints to the log, with parameters similar to printf. It indents per

si32LogBeIndented(yn), prepends "ERROR: " and appends a newline.

- **void si32LogPrologue(void)**: Logs the name of the surrounding function preceded by ">" and indented per si32LogBeIndented(yn). Useful for marking function entry. Intended to be paired with si32LogEpilogue() particularly when preceded by si32LogBeIndented(YES).
- **void si32LogEpilogue(void)**: Logs the name of the surrounding function preceded by "<" and indented per si32LogBeIndented(yn). Useful for marking function exit. Intended to be paired with si32LogPrologue() particularly when preceded by si32LogBeIndented(YES).

5.3. Error Handling

Error handling in si32Library adopts the strategy of storing an error code, with a get/set error methodology. This style is chosen over the "error out parameter" style in order to reduce the number of parameters to functions in the general case, improving the compiler's opportunity to pass parameters via registers. The error code is a char*, not an integer code, and points to a non-localized string. Error strings are intended specifically to aid the developer in debugging the application. These entry points are all macros.

- **si32Assert(expression)**: Evaluates an expression and "halts" if that expression evaluates to false.
- **si32If (expression) { handler statements }**: If the expression is true, sets an error with the text of the expression, and executes the handler statements. It is 'if', but with internal error check, set, and log functionality. Use it just like 'if'.
- **IF (expression) { handler statements }**: Identical to si32If, but intended to be less disruptive when reading code.
- **si32BreakIf(expression)**: Conditional break statement.
- **si32ContinueIf(expression)**: Conditional continue statement.
- **si32Catch(exception)**: Control flow escape for local exception handing. Catches a local throw from within the same function.
- **si32Throw(exception)**: Control flow escape for local exception handing. Throws to the corresponding catch within the same function.
- **si32ThrowExitIf(expression)**: Conditional throw to 'exit'.
- **si32ThrowIf(expression, exception)**: Similar to si32ThrowTrapIf except instead of throwing 'trap' it throws the specified exception, and does not specify an error string.
- **si32ThrowTrapIf(expression)**: Conditional throw to 'trap'. Evaluates an expression and throws a trap exception if that expression evaluates to true.
- **si32TrapIf(expression)**: Synonymous with si32ThrowTrapIf. si32TrapIf is deprecated but still exists for compatibility with vintage code.
- **si32Halt()**: Halts, triggering a debugger breakpoint if possible, or simulates one via while(1).
- **bool si32SetError(fmt, ...)**: Logs and pushes an error. Error stack depth is implementation dependent, and varies across embedded and workstation targets. StdC implementations are 1 deep. ObjC implementations are deeper. Parameters work like printf. Current implementations always return YES.
- **char* si32GetError()**: Peeks the current error. I.e, returns the error string but does not pop the error stack.
- **si32ClearError**: For embedded StdC implementation, clears the current error. For ObjC implementations, pops the error stack.

5.4. Memory Management

The si32Library supports static/compile time memory allocation, dynamic one-time allocation, and dynamic heap-based allocation.

- **Static allocation** relies on declaring all memory usage as data structures to be reserved and initialized by the compiler when the system boots.
- **Dynamic one-time allocation** relies on a statically allocated block of memory that is used to incrementally satisfy allocation requests at runtime. Memory allocated in this manner is not recoverable via deallocation or reallocation. It is only recoverable by resetting the entire memory block.
- **Dynamic heap-based allocation** is essentially the malloc/free model supporting allocation, reallocation, and deallocation on a heap.

Static allocation is performed by writing C code that declares and initializes data structures, and is outside the scope of this discussion.

Dynamic one-time allocation is performed by overriding the default value of **si32BuildOption_incremental_zone_size**, thus creating a block of memory for servicing allocations. Calls to **si32Base_allocate(SI32_INCREMENTAL_ZONE, size, alignment)** claim memory from this block, called the incremental zone. Attempts to free this memory via **si32Base_deallocate(pointer)** will succeed, but the memory will not be reclaimed. Similarly, calls to **si32Base_reallocate(pointer)** will reallocate the pointer by claiming a new block from the incremental zone, effectively leaking the previous allocation.

Dynamic heap allocation allows allocation, reallocation, and deallocation. s**i32Base_allocate(SI32_HEAP_ZONE, size, alignment)** behaves similarly to **calloc()**. **si32Base_reallocate(pointer)** behaves similarly to **realloc()** when called on a heap pointer. **si32Base_deallocate(pointer)** behaves similarly to **free()** when called on a heap pointer.

Note: The performance of heap-based dynamic memory depends on the heap implementation provided by the tool chain. Each supported compiler provides a unique implementation.

6. Object Component

COMPONENT: si32ObjectComponent

REQUIRES: si32BaseComponent

PROVIDES: si32RootObject

Although it is sometimes practical to implement applications in C++ by using only a subset of the language's features, use of C++ may be precluded by the target application's requirements. The si32Library object system provides a C99 based subset of the capabilities of C++ for use in applications where C++ is inappropriate or unavailable. The si32Library is object based, providing encapsulation, inheritance, and polymorphism. The object system provides a simple class hierarchy, but is neither truly class or prototype based. It is simply a design pattern that uses structure composition and aggregation to implement a type system wherein new types can be derived from base types, inheriting their behavior and properties. Therefore, the term class is generally avoided.

In many cases the precise type of an object is known a-priori, at compile time. In such cases, early binding can be employed effectively to eliminate indirect function calls. For example:

```
_SomeType_SomeOperation(SomeObject, SomeArguments);
```

In other cases, however, late binding is necessary because the type of an object is not known until run-time. This is particularly true for objects representing optional hardware that is discovered at run-time. For example:

```
SomeObject->Operations->SomeOperation(SomeObject, SomeArguments);
```

The **si32ObjectComponent** provides both early and late binding capabilities.

Objects are implemented as C structures. These structures contain specific fields and are constructed in a manner that facilitates the implementation of inheritance by means of structure composition. Figure 4 shows the essential structure of a base object.

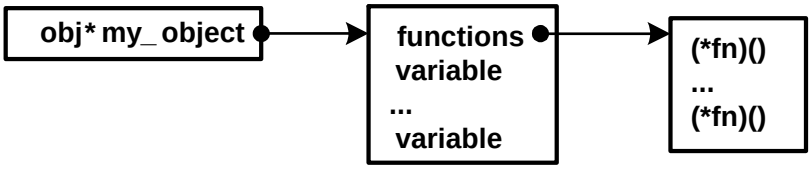


Figure 4. Object Structure

An object's variables can be accessed in this manner:

```
my_object->context.variable
```

The object's functions can be called as follows:

```
my_object->context.functions->fn(my_object, ...)
```

An object reference (or a pointer to an object) is represented by the type **obj***. This is an anonymous (void) pointer that can reference anything. Therefore, it must be assigned to a variable of the appropriate type and checked at runtime.

Object functions typically follow this pattern:

```
uint32 result = 0;


// meta:{{si32SomeObject}do_something}
si32Assert(is_si32SomeObject(self));
si32SomeObject *my = self;
result = my->someObjectContext.variable;


// meta:{{si32SomeObject}do_something}
return result;
```

The first line declares a variable for the function result and initializes it. The comment marks the end of the function entry and the start of the body. si32Assert is an assertion that halts in the debugger if the expression is false. is_si32SomeObject performs the runtime type check. Next, a variable of the appropriate type is declared and bound to **self**. Note that this is 'safe' because the runtime type check succeeded. The symbol **my** is just chosen to reinforce that referenced attributes belong to the object bound to **self**. Next, the contents of the object's variables (i.e., its attributes) are accessed through the typed pointer. The second comment marks the end of the body and starts the function exit.

Logging and assertions only have a code footprint in debug builds. They are intended for use during development and debugging, not for final production code. Therefore, build options exist to cause these constructs to compile away to nothing for release builds.

Calling the functions of an object (i.e., invoking its operations) is accomplished in a similar fashion. However, it is not necessary to employ a **type* my = self** statement in order to invoke an operation on an object. The following is sufficient:

```
uint32 result = si32SomeObject_do_something(&my_object, arg);
```

Macros are employed to access the object's function tables in order to determine the correct implementation of the operation to invoke. This is how runtime function overloading is implemented by the **si32ObjectComponent** and, therefore, all of the si32Library. For example:

```
uint32 _si32ListObject_get_capacity(obj* /*self*/);

#define si32ListObject_get_capacity(self)
         as_si32ListObject(get_capacity, (self))
```

The first line provides the function prototype for the early binding. Calling this directly invokes that specific function, bypassing the function table and thus preventing polymorphism. Early bindings are preceded by an underscore.

The second line provides the late binding, which calls through the function table, enabling polymorphism.

6.1. Derived Types

Each object contains a context, which contains a pointer to a function table, and zero or more attributes. Derived objects contain multiple contexts, each with their own function pointer and attributes. This allows derived types to have attributes with the same names as those of their base types that are independent of the same named attributes of the base type. Figure 5 shows the derived object structure.

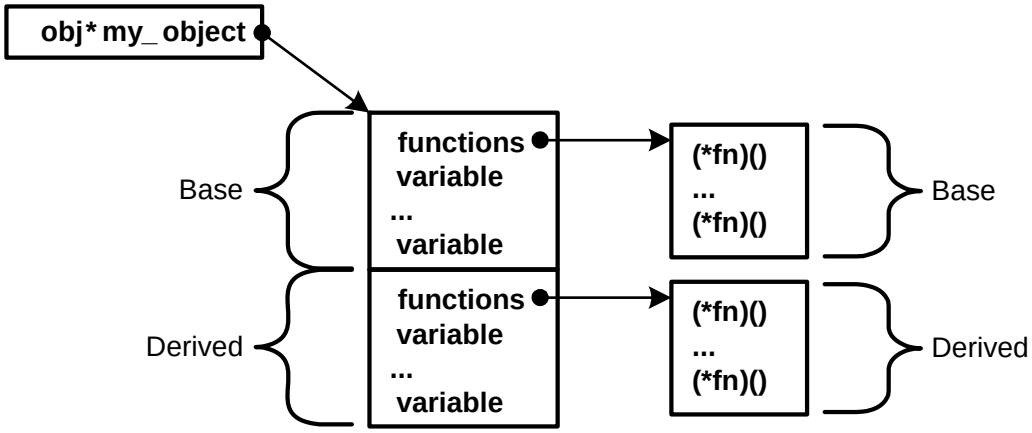


Figure 5. Derived Object Structure

Objects also have a type signature that is not represented in the diagram. This type signature allows operations to assert that the **self** parameter is an object of a specific type. Those assertions can be compiled away in release builds. The current implementation uses **cstrings** and **strcmp** to perform runtime type checking. This allows checking whether an object is derived from some other object and is legible in memory. Occasionally attempting to call **strcmp** on a non-object can result in a fault. This is arguably less friendly than an assertion, but the error is caught in the debugger.

6.2. Object Life Cycles

All objects in the library derive from the si32RootObject. This provisions all of the library objects with a small set of fundamental capabilities. All objects in the library, including those that are not dynamically allocated on the heap, employ reference counting to govern their life cycles. The operations for accomplishing this are:

- **allocate**: An object can be allocated on the stack, or it can be incrementally, or dynamically, allocated from one of several memory zones. The act of allocating an object reserves memory for that object and configures it for use as an object by preparing it's reference count, function tables, etc.
- **initialize**: All objects must be initialized prior to being used.
- **retain**: Increments the object's reference count.
- **release**: Decrements the object's reference count. When a dynamic object's reference count decrements from 1 to 0 it is deallocated.
- **deallocate**: Called when the reference count reaches 0. Only objects dynamically allocated from the heap zone are actually deallocated.

This reference counting mechanism is employed throughout the si32Library. The policy is that newly created objects have a reference count of 1, and it is the responsibility of the creator of that object to release it once it is no longer needed. When a reference to an object is provided to some other object, the receiving object must retain it in order use it beyond the current function call. Otherwise it may be released, potentially resulting in deallocation; thus, subsequent use may result in a pointer error. The policy is to retain it if it will continue to be used. Consequently, every call to retain must be paired with a corresponding call to release. This allows one object to allocate a second object, hand it over to a third object, and then release it, thus relieving itself of ownership. The object that performed the allocation is not required to be responsible for its eventual deallocation.

In general, memory is allocated at compile time or automatically on the stack at runtime. Heap-based dynamic memory allocation functions are provided to allocate, reallocate, and deallocate RAM in a controlled, monitored

manner. These functions are provided by the **si32BaseComponent** and should be used for all dynamic memory allocation because they provided tracked, aligned, memory management transparently to the library and its applications.

6.2.1. Allocation of Objects

Objects must be allocated and initialized before they can be used. They are typically allocated on the stack as automatic variables. For example:

```
si32UsartAPortalObject my_usart = si32UsartAPortalObject();
```

Objects whose lifetime must exceed that of the scope in which they are defined can be declared static, just like any other variable in C.

```
static si32UsartAPortalObject my_usart = si32UsartAPortalObject();
```

Objects can also be dynamically allocated.

```
si32UsartAPortalObject* my_usart = si32UsartAPortalObject_allocate();
```

Objects are allocated from the default allocation zone, one of the si32AllocationZoneEnumType values: SI32_RETENTION_ZONE, SI32_INCREMENTAL_ZONE, or SI32_HEAP_ZONE. The default zone is SI32_HEAP_ZONE. Applications are free to change the default allocation zone at runtime.

Both the retention zone and the incremental zone are incremental. They are simply blocks of bytes, and each allocation request just advances a pointer within the zone by the amount requested. Allocating from these zones does not incur additional RAM overhead. However, it is not possible to reclaim memory allocated from these zones except by resetting the entire zone. They are intended to support a programming model wherein all memory is preallocated at startup time and persists throughout the duration of the program's execution.

The heap zone is implemented using the allocator provided by the tool chain. It allows dynamic deallocation and reallocation, at the cost of potential memory fragmentation and subsequent compaction.

To determine the default allocation zone use:

```
si32AllocationZoneEnumType si32Base_get_default_allocation_zone()
```

To change the default allocation use:

```
si32Base_set_default_allocation_zone(si32AllocationZoneEnumType zone)
```

6.2.2. Rules for Initializing Objects

Regardless of how an object is allocated, it should not be used until it has been initialized. Initialization sets up the object's attributes and configures it for operation.

```
si32UsartAPortalObject_initialize(&my_usart,
   SI32_PORTAL_OBJECT_CAPABILITY_FULL_DUPLEX
   |SI32_PORTAL_OBJECT_CAPABILITY_TRANSMIT
   |SI32_PORTAL_OBJECT_CAPABILITY_RECEIVE,
   SI32_USART_0);
```

Attempting to invoke an object function on an uninitialized object results in an assertion (for debug builds) or (occasionally) a fault.

6.2.3. Usage

Once an object has been allocated and initialized, it can be used.

```
si32UsartAPortalObject_set_configuration(&my_usart,
   SI32_PORTAL_OBJECT_CONFIGURATION_FULL_DUPLEX
   |SI32_PORTAL_OBJECT_CONFIGURATION_TRANSMIT
   |SI32_PORTAL_OBJECT_CONFIGURATION_RECEIVE);
actual_rd = si32UsartAPortalObject_read(&my_usart, buffer, N);
si32UsartAPortalObject_release(&my_usart);
```

Another example:

```
// allocate an object on the stack.
si32PseudoThreadObject my_thread = si32PseudoThreadObject();
// initialize it.
si32PseudoThreadObject_initialize(&my_thread, &my_run_loop);
// use it.
si32PseudoThreadObject_run(&thread, my_thread_fn);
// eventually release it, once it is no longer needed.
si32PseudoThreadObject_release(&thread);
```

6.2.4. Rules for Retaining and Releasing Objects

Regardless of how an object is allocated, it should be retained and released in order to correctly manage its reference count. In some cases, a reference to an object may be passed to a function, and that object may or may not be dynamically allocated. If the reference to the object must remain valid even after the function returns (i.e., it was stored somewhere for later use), then it must be retained. All calls to retain must be matched with a call to release, once the retainer of the object no longer needs a reference to it. Reference counting allows dynamic objects that are no longer needed to become deallocated, and allows static objects that are no longer needed to release any aggregated objects whose references they hold.

For example:

```
// retain it.
si32BufferObject_retain(&my_tx_buffer);
// use it...
// release it.
si32BufferObject_release(&my_tx_buffer);
```

All objects are initialized with a reference count of one. This means that statically allocated objects created on the stack (but not those defined at file scope) must be released prior to return from the function wherein they are allocated. The reason is that those objects may contain references to other objects. These other objects must be released when the statically allocated stack object goes out of scope; otherwise, their reference counts can never decrement to zero.

For example:

```
void f()
{
   // allocate an object on the stack and zero it.
   si32BufferObject my_buffer = si32BufferObject();
   // initialize it.
   si32BufferObject_initialize(&my_buffer, ...);
   // use it...
   // release it.
   si32BufferObject_release(&my_tx_buffer);
}
```

6.3. si32RootObject

si32RootObject is the root of the object hierarchy.

7. Container Component

COMPONENT: si32ContainerComponent

REQUIRES: si32BaseComponent, si32ObjectComponent

PROVIDES: si32BufferObject, si32BufferObjectV2, si32PoolObject, si32QueueObject, si32QueueObjectV2, si32ValueObject

si32ContainerComponent provides a variety of objects that can hold and manage blocks of memory, and other objects.

7.1. si32BufferObject

An si32BufferObject is a one dimensional collection of homogenous elements whose type is indeterminate. Its buffer associates a block of memory with its capacity and provides simple accessors that allow other objects to access its contents in a uniform way. Elements of a buffer are accessible by index and can be addressed in any order. Operations on buffers read and write entire elements, not pointers to them. Buffers are mutable and inelastic; the elements are modifiable but the capacity is constant. Once initialized, the buffer size cannot be changed.

Figure 6 shows the si32BufferObject diagram.

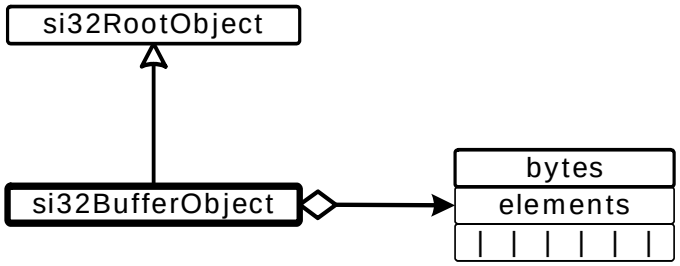


Figure 6. si32BufferObject Diagram

7.2. si32BufferObjectV2

An si32BufferObjectV2 is a one-dimensional collection of homogenous elements whose type is indeterminate. Its buffer associates a block of memory with its capacity and provides simple accessors that allow other objects to access its contents in a uniform way. Elements of a buffer are accessible by index and can be addressed in any order. Operations on buffers read and write entire elements, not pointers to them. BufferV2 is mutable and elastic; memory for the elements of the buffer is allocated from the default allocation zone whenever the capacity is changed using **_set_Capacity()**.

Figure 7 depicts the si32BufferObjectV2 diagram.

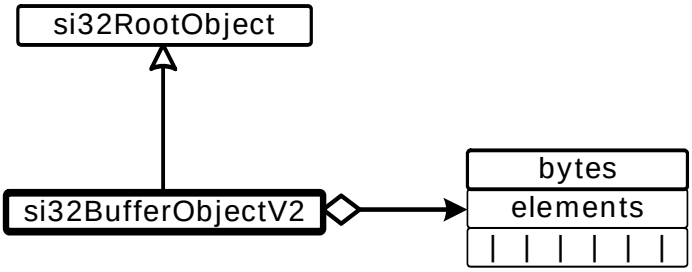


Figure 7. si32BufferObjectV2 Diagram

7.3. si32PoolObject

An si32PoolObject is a collection of homogenous elements whose type is indeterminate. Pools are mutable and inelastic; the elements are modifiable but the capacity is constant. A pool employs a queue to manage a buffer of elements, and the elements can only be accessed in first-in-first-out order. Pools read and write only pointers to elements, not the elements themselves.

Figure 8 depicts the si32PoolObject diagram.

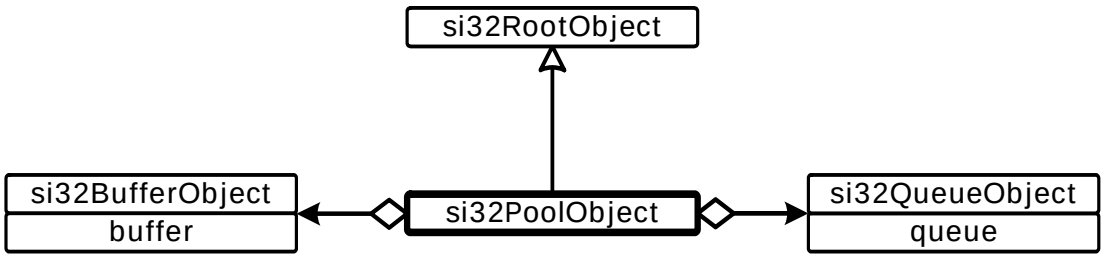


Figure 8. si32PoolObject Diagram

7.4. si32QueueObject

An si32QueueObject is a collection of homogenous elements whose type is indeterminate. Queues are mutable and inelastic; the elements are modifiable but the capacity is constant. Queues are circular, and elements can only be accessed in first-in-first-out order. Queues read and write entire elements, not pointers to them. When initialized, a queue can use memory provided by the client or it can allocate memory from the default allocation zone.

Figure 9 shows the si32QueueObject diagram.

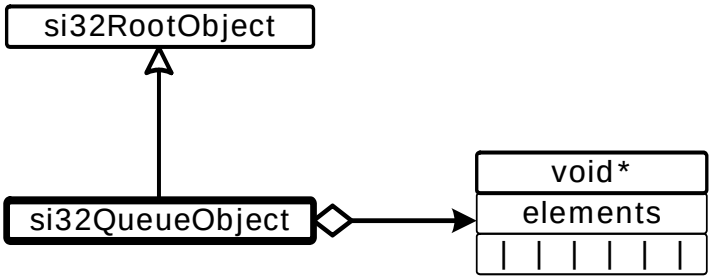


Figure 9. si32QueueObject Diagram

7.5. si32QueueObjectV2

An si32QueueObjectV2 is a collection of objects. Elements are retained when written and released when read. The memory to store the object pointers is allocated from the platform.

Figure 10 depicts the si32QueueObjectV2 diagram.

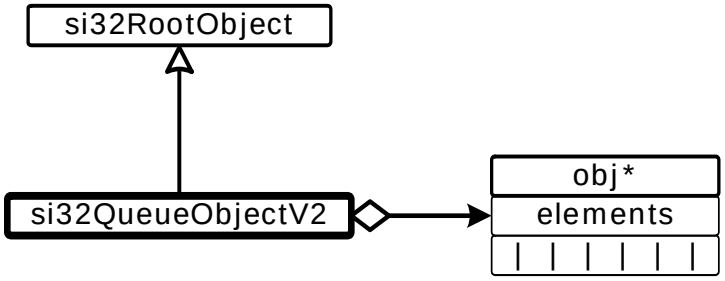


Figure 10. si32QueueObjectV2 Diagram

7.6. si32ValueObject

An si32ValueObject is a box intended to allow non-object entities, such as integers, pointers, and structs, to be managed by containers that operate on objects. This implementation limits the size of wrapped structures to 255 bytes. Value objects can wrap integers, unsigned integers, floats, pointers, and structs. Additional memory is allocated from the platform to wrap structures, but not for integral types. That memory is deallocated when the value object is deallocated. In this implementation a value object configured to wrap a structure cannot be reused to wrap some other type or the structure memory will leak.

Figure 11 depicts the si32ValueObject diagram.

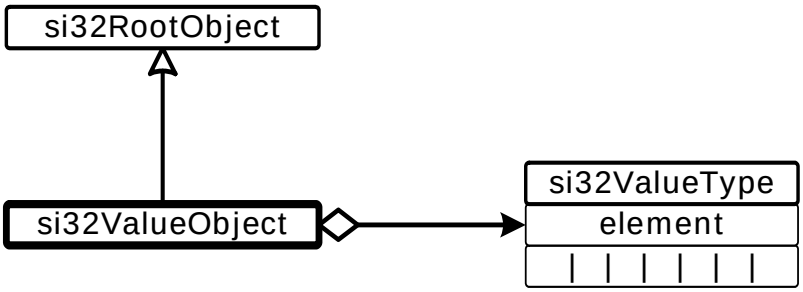


Figure 11. si32ValueObject Diagram

8. I/O Component

COMPONENT: si32IoComponent

REQUIRES: si32BaseComponent, si32ObjectComponent

PROVIDES: si32IoObject, si32PortalObject, si32TimerObject

The **si32IoComponent** contains objects that abstract the behavior of hardware. Derived objects in **si32McuComponent** implement hardware specific concrete instances of these objects.

8.1. si32PortalObject

In general, there are several types of streaming I/O operations possible:

- **Non-blocking Synchronous**: Reads return immediately with an actual count of 0 or more, up to the desired count, depending on how much data is available (i.e., hardware receive FIFO). Writes return immediately with an actual count of 0 or more, up to the desired count, depending on how much space is available (i.e., hardware transmit FIFO).
- **Non-blocking Asynchronous**: Reads/writes take a callback and return immediately. The callback may be invoked before the read/write call returns. The callback always indicates completion, and the actual count is always the desired count unless there was an error or the request was aborted.
- **Blocking Synchronous**: Reads/writes do not return until the full desired count has been moved or an error occurs.
- **Blocking Asynchronous**: Essentially the same as non-blocking asynchronous, except that it is possible for reads/writes to block before returning, even after the callback has been invoked.

The si32Library portal objects implement un-buffered, non-blocking, synchronous and asynchronous operations. They are designed to be as simple as possible in that they do not require other library objects or components. In other words, si32UsartAPortalObject is derived from si32PortalObject but does not depend on other objects such as timers, queues, buffers, pseudo-threads, or run-loops. This means that portals do not implement timeouts, they do not implement transmit/receive queues, they do not support multiple sessions, and they do not offer blocking operations. Additional entities are required in order to support those functionalities and are outside of the scope of portals. Note that portals do implement operations to abort reads/writes in progress, enabling timeout/error handlers to regain control of the underlying hardware.

Asynchronous operations on portal objects typically invoke the callback from interrupt dispatch level. The callback function must restrict its behavior to only perform interrupt- (and possibly thread-) safe operations. Ensuring correct behavior of callbacks is outside of the scope of the portal objects.

Figure 12 shows the si32PortalObject diagram.

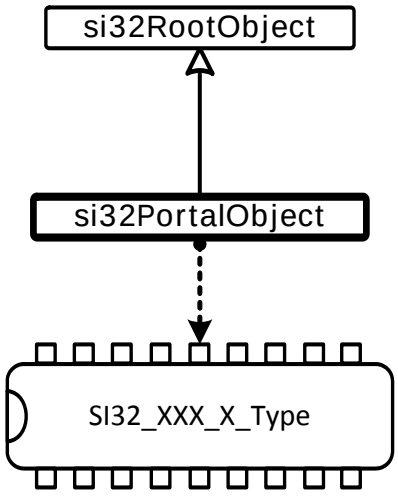


Figure 12. si32PortalObject Diagram

8.2. si32IoObject

Low level streaming I/O is handled by hardware specific objects directly wrapping the HAL. These objects are called portals and implement the simple API for moving data. The si32Library provides portals for USART, I2C, SPI, I2S, and USB. However, portals do not implement timeouts, and they do not implement transmit/receive queues; the si32IoObject adds those capabilities. Io objects may invoke their callbacks from the interrupt or foreground levels. Callback functions must restrict their behavior to only perform interrupt- (and possibly thread-) safe operations. Ensuring correct behavior of callbacks is outside of the scope of the Io objects.

Figure 13 depicts the si32IoObject diagram.

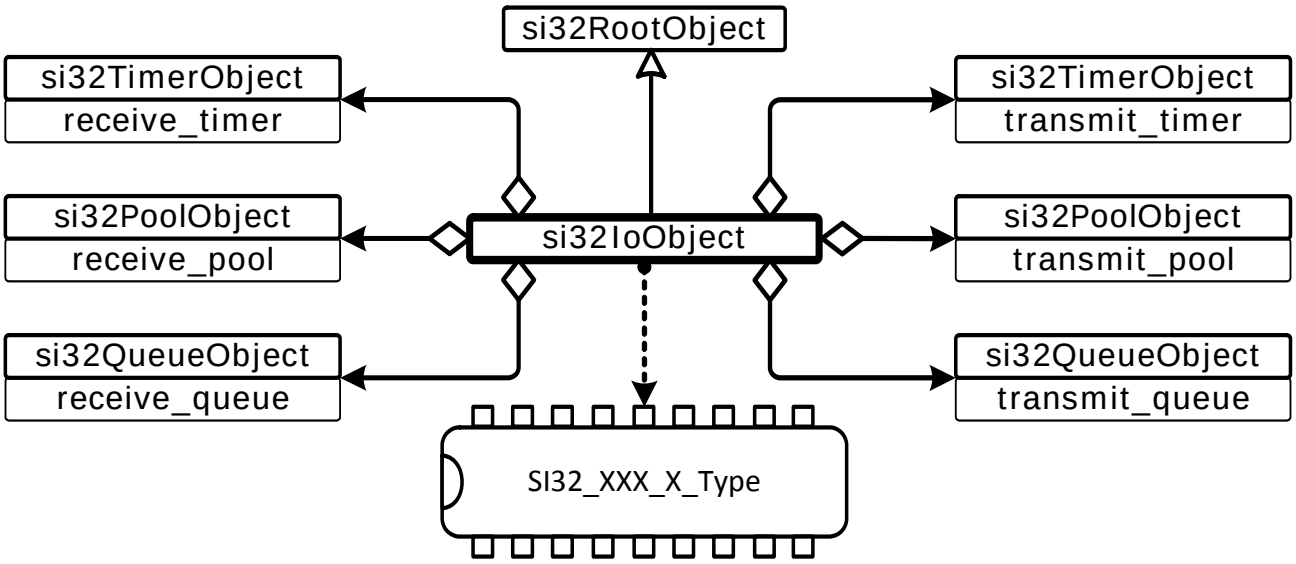


Figure 13. si32IoObject Diagram

8.3. si32TimerObject

si32TimerObject represents an abstract timer that is independent of specific timer hardware. Timer objects manage groups of si32TimerType structures, one per interval. In other words, a timer object can time multiple time intervals concurrently, each with its own duration and callback. Hardware specific subtypes of si32TimerObject access the MCU's hardware timers within **si32McuComponent**. For example, si32SystickTimerObject operates off of the CMx SysTick timer hardware.

Timeouts are handled by timer objects. Timer objects are hardware specific objects directly wrapping the HAL. A timer can be started by calling _**start()** with a time interval and a callback. A time interval can be stopped prematurely by calling _**stop()**. Timers monitor the time remaining on each of their active intervals and invoke the callback when the interval expires. Timers are designed to be as simple as possible in that they do not require other si32Library objects or components. For example, si32SystickTimerObject is derived from si32TimerObject and does not depend on other objects such as queues, buffers, pseudo-threads, or run-loops.

Depending on the circumstance leading to the rescheduling operation that terminates a timer interval, timer objects can invoke their callbacks at either foreground or interrupt levels. Therefore timer callback functions must restrict their behavior to only perform interrupt- (and possibly thread-) safe operations. Ensuring correct behavior of callbacks is outside of the scope of the timer objects.

Figure 14 illustrates the si32TimerObject diagram.

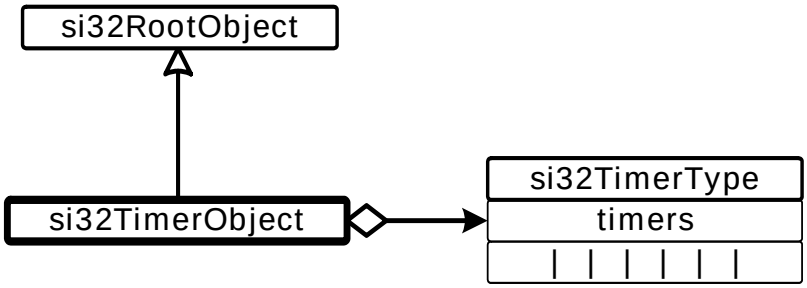


Figure 14. si32TimerObject Diagram

9. PseudOS Component

COMPONENT: si32PseudOsComponent

REQUIRES: si32BaseComponent, si32ObjectComponent, si32ContainerComponent

PROVIDES: si32PseudoThreadObject, si32RunLoopObject

The **si32PseudOsComponent** provides simplistic cooperative multitasking capabilities. The si32RunLoopObject is a cyclic executive that dispatches event handlers called work requests. Work requests are submitted to a run loop for processing via **_post_work_request()** and can be marked as high or low priority. Applications built on si32Library typically have only one run loop object that dispatches events throughout the system.

In addition to the run loop object, this component also provides si32PseudoThreadObject, which represents a simplistic cooperative thread or task. Each pseudo-thread maintains a pointer to a block of application defined thread local storage. Pseudo-threads have an application defined entry point established via **_run()**. To cause a pseudo-thread to execute it is necessary to call **_resume()** on that thread. This posts a work request on the current run loop, resulting in subsequent invocation of the thread's entry point. The thread then executes until it yields.

9.1. si32PseudoThreadObject

The si32PseudoThreadObject is not intended as an RTOS replacement. It is intended for use in very simple applications where multitasking is convenient, but the added complexity of an RTOS is not warranted.

The si32PseudoThreadObject represents a green thread form of light-weight thread similar to a protothread. This implementation is inspired by the work of:

- Tom Duff: http://en.wikipedia.org/wiki/Duff's_device
- Simon Tatham: http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html
- Adam Dunkels: http://www.sics.se/~adam/dunkels06protothreads.pdf
- Larry Ruane: http://code.google.com/p/protothread/
- green threads: http://en.wikipedia.org/wiki/green_threads

A pseudo-thread is an object that maintains an entry point, a pointer to private storage, and a push-down stack for spawning sub-threads. A thread's entry point is established when the thread is activated via **_run()**, which installs the entry point and resumes the thread by posting a work request on the current run-loop. Each time the thread resumes, its entry point is called.

The "pseudo-thread" concept is inspired and heavily influenced by protothreads. Pseudo-threads fundamentally behave like protothreads, even to the point of offering build options to choose whether the implementation employs switch abuse or computed labels. Pseudo-threads differ, however, in that they are si32 objects requiring a run-loop to drive them, memory management, and a push-down stack of execution contexts allowing thread entry points to be invoked in a manner similar to subroutines.

Figure 15 shows the si32PseudoThreadObject diagram.

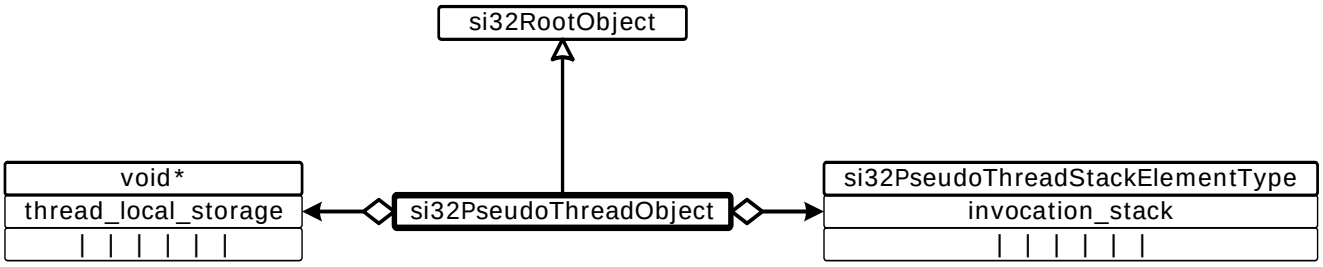


Figure 15. si32PseudoThreadObject Diagram

9.2. si32RunLoopObject

The si32RunLoopObject is a cyclic executive that watches two queues for events and dispatches them in priority order. The current implementation provides a fixed priority bias wherein the high priority work request queue is processed three times more often than the low priority work request queue. Runtime adjustable biasing is planned. The capacities of the work request queues must be configured when the run loop is created.

Figure 16 depicts the si32RunLoopObject diagram.

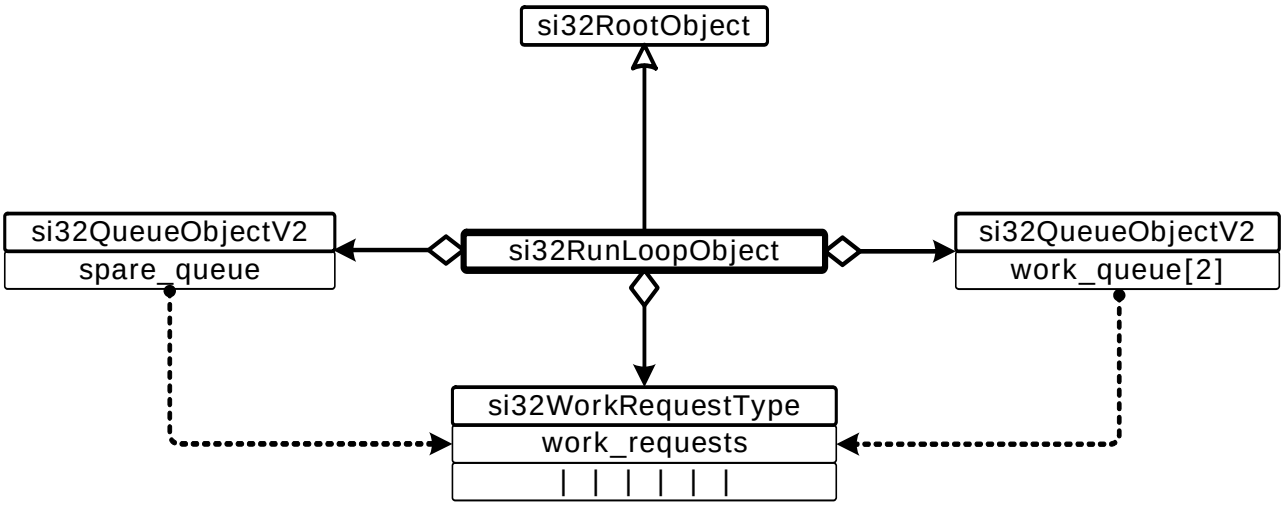


Figure 16. si32RunLoopObject Diagram

10. USB Component

COMPONENT: si32UsbComponent

REQUIRES: none

PROVIDES: si32UsbAudioComponent, si32UsbCdcComponent, si32UsbCoreComponent, si32UsbDfuComponent, si32UsbHidComponent, si32UsbMscComponent

The USB component is a collection of sub-components that add specific USB functionality:

- **si32UsbAudioComponent** provides the USB Audio Device Class functionality and definitions.
- **si32UsbMscComponent** adds USB Mass Storage Class functionality and definitions.
- **si32UsbHidComponent** adds USB HID Class functionality.
- **si32UsbDfuComponent** adds USB DFU Device Class functionality.

All class components depend on the **si32USBCoreComponent** to provide the basic USB definitions required by the core USB Specification and are optional; they only need to be compiled if the functionality is needed.

The **si32UsbCoreComponent** provides the basic USB Device model and device controller functions.

10.1. si32UsbCoreComponent

COMPONENT: si32UsbCoreComponent

REQUIRES: none

PROVIDES: si32UsbConfigurationObject, si32UsbDefaultEndpointObject, si32UsbDeviceObject, si32UsbEndpointObject, si32UsbInterfaceObject

The **si32UsbCoreComponent** provides support for the standard USB Device Model and the standard USB Requests required for USB Enumeration.

This component includes objects that provide the basic functionality that collectively can be used to act as a stand-alone USB device. The objects, however, are intended to serve as the base objects that are extended by the USB Device Class components or custom applications.

10.1.1. si32UsbDeviceObject

This is the root object of any USB Device. An application can define many si32UsbDeviceObjects, but only one can be associated with the USB device controller at any given time. An si32UsbDeviceObject also has a reference to an si32UsbStringTable, which is used to look up strings by index and language ID, and a collection of one or more si32UsbConfigurationObjects.

All si32UsbDeviceObjects contain a default endpoint object that can be shared among all interfaces contained in the device's configurations. Control requests for standard Chapter 9 requests for enumeration, as well as any device class or vendor specific requests, will all be received by the default endpoint and dispatched by the si32UsbDeviceObject.

An si32UsbDeviceObject receives notification of a new control request and is a control request recipient, so it may dispatch control requests that specify a Device recipient to itself.

10.1.2. si32UsbConfigurationObject

This object contains the specific power and function configuration provided by the USB device. An si32UsbDeviceObject contains a collection of one or more configuration objects. When a configuration object has been set through a USB SET_CONFIG standard request, the device is considered to be functional and enumerated.

An si32UsbConfigurationObject contains a collection of si32UsbInterfaceObjects that provide the functionality supported by the configuration.

An si32UsbConfigurationObject is not a control request recipient; therefore, it will not receive notification of control requests that may be targeted at an interface or endpoint contained in its configuration. However, the collection of available interfaces and endpoints is dependent on the active configuration, so a device object queries a configuration for its interfaces or endpoints when attempting to deliver a control request.

10.1.3. si32UsbEndpointObject

This object represents the device-side of a USB endpoint. An endpoint is bound to a specific interface object and can only be used when that interface has been set. All USB I/O occurs using an si32UsbEndpointObject or an object derived from an si32UsbEndpointObject.

An si32UsbEndpointObject is a control request recipient, so it will receive control requests that specify an endpoint recipient through its RequestHandler operation.

10.1.4. si32UsbDefaultEndpointObject

This object derives from the si32EndpointObject and provides the support expected from a USB default control endpoint.

10.1.5. si32UsbInterfaceObject

This object represents a single USB Interface that can be enabled within a configuration. The interface may contain or refer to 0 or more endpoint objects through which the firmware may communicate with the host when the interface is enabled.

The si32UsbInterfaceObject is a control request recipient, so it will receive control requests that specify an interface recipient through its RequestHandler operation.

An si32UsbInterfaceObject contains a collection of si32UsbEndpointObjects that it uses to communicate with the host.

10.2. si32UsbAudioComponent

COMPONENT: si32UsbAudioComponent

REQUIRES: none

PROVIDES: si32UsbAudioControlInterfaceObject, si32UsbAudioEndpointObject, si32UsbAudioInterfaceObject, si32UsbAudioObject, si32UsbAudioStreamingEndpointObject, si32UsbAudioStreamingInterfaceObject

The **si32UsbAudioComponent** encapsulates the set of objects that add support for the USB Audio Device Class. The objects in this component extend the standard USB device objects provided by the **si32UsbCoreComponent**.

10.3. si32UsbDfuComponent

COMPONENT: si32UsbDfuComponent

REQUIRES: none

PROVIDES: si32UsbDfuDeviceObject, si32UsbDfuInterfaceObject, si32UsbDfuObject

The **si32UsbDfuComponent** contains a set of objects that adds support for the USB Firmware Download specification, allowing applications to easily add DFU capability to their embedded applications.

10.4. si32UsbHidComponent

COMPONENT: si32UsbHidComponent

REQUIRES: none

PROVIDES: si32UsbHidInterfaceObject

The **si32UsbHidComponent** contains objects that add support for the USB Human Interface Device (HID) class specification to the **si32UsbComponent**.

The component currently consists of the minimum objects necessary to allow an application to intercept USB HID class specific messages for the application-specific HID interface and provide the application with endpoints that can be used to send or receive hid reports.

10.5. si32UsbMscComponent

COMPONENT: si32UsbMscComponent

REQUIRES: none

PROVIDES: si32ScsiDeviceObject, si32ScsiMediaObject, si32StorageObject, si32UsbMscObject

The si32UsbMscComponent contains the objects necessary to add support for the USB Mass Storage Class specification.

The component defines and implements Bulk-only transport and provides default implementations for the common SCSI Block Commands and an abstract SCSI Media Interface. An application can quickly add a new media type using this interface by providing the media capacity, block size, a read function, and a write function.

11. MCU Component

COMPONENT: si32McuComponent

REQUIRES: none

PROVIDES: si32IicAPortalObject, si32IisAPortalObject, si32SpiAPortalObject, si32SystickTimerObject, si32UsartAPortalObject, si32UsartIoObject, si32UsbEndpointIoObject, si32UsbIoObject, si32UsbPortalObject

The **si32McuComponent** provides a variety of objects that interact directly with hardware, via the si32 HAL.

11.1. si32IicAPortalObject

si32IicAPortalObject implements the si32PortalObject interface on top of SI32_I2C_A_Type modules. It does not support non-blocking synchronous operation, only asynchronous, due to the inherently state-machine nature of I^2C. This implementation is half duplex single master only.

Figure 17 shows the si32IicAPortalObject diagram.

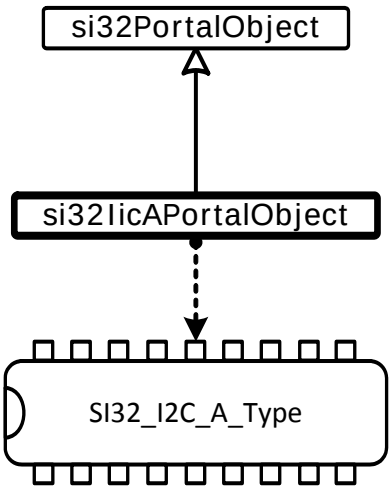


Figure 17. si32IicAPortalObject Diagram

11.2. si32IisAPortalObject

si32IisAPortalObject implements the si32PortalObject interface on top of SI32_I2S_A_Type modules. This implementation is full duplex master mode.

Figure 18 shows the si32IisAPortalObject diagram.

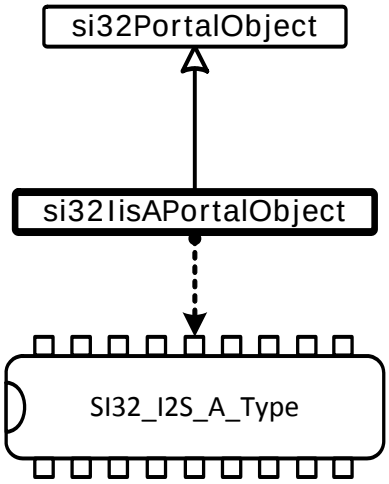


Figure 18. si32IisAPortalObject Diagram

11.3. si32SpiAPortalObject

si32SpiAPortalObject implements the si32PortalObject interface on top of SI32_SPI_A_Type modules. This implementation is full duplex master mode.

Figure 19 illustrates the si32SpiAPortalObject diagram.

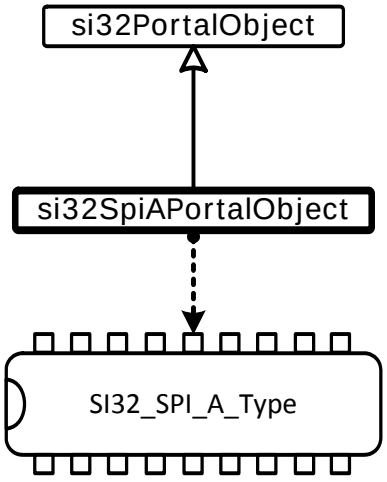


Figure 19. si32SpiAPortalObject Diagram

11.4. si32SystickTimerObject

si32SystickTimerObject is a special purpose si32TimerObject that uses the CMx's SysTick timer hardware to schedule callbacks. It is intended as a singleton; there can be only one per CPU.

Figure 20 shows the si32SystickTimerObject diagram.

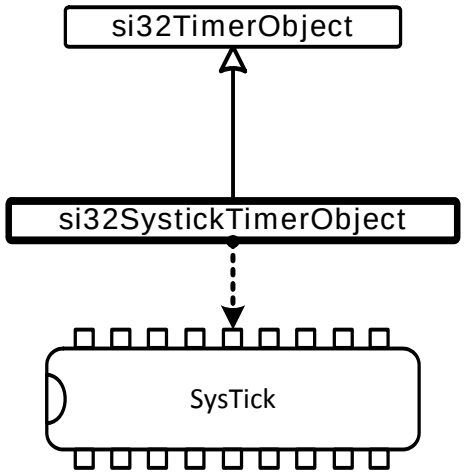


Figure 20. si32SystickTimerObject Diagram

11.5. si32UsartAPortalObject

si32UsartAPortalObject implements the si32PortalObject interface on top of SI32_USART_A_Type modules.

Figure 21 shows the si32UsartAPortalObject diagram.

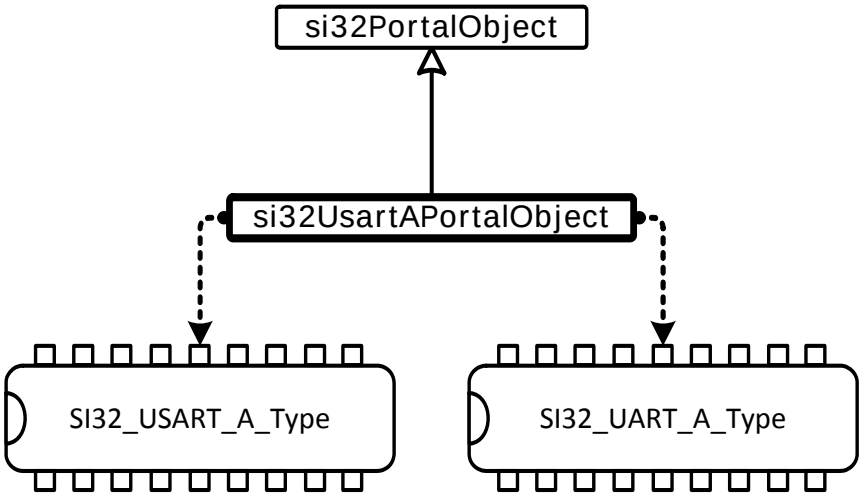


Figure 21. si32UsartAPortalObject Diagram

11.6. si32UsartIoObject

This io object implements full duplex transfers on SI32_USART_A_Type, or an SI32_UART_A_Type masked as an SI32_USART_A_Type.

Figure 22 shows the si32UsartIoObject diagram.

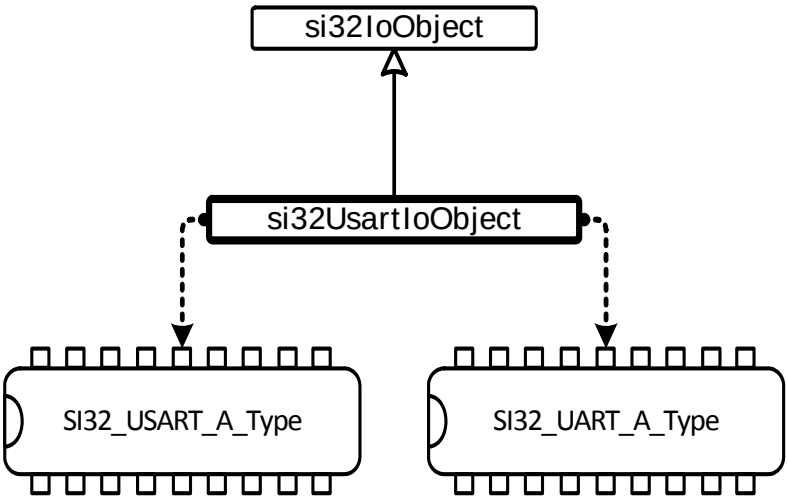


Figure 22. si32UsartIoObject Diagram

11.7. si32UsbIoObject, si32UsbEndpointIoObject, and si32UsbDefaultEndpointIoObject

The si32UsbIoObject extends the si32IoObject by adding definitions and initialization of resources required specifically to support USB I/O requests, including independent IoRequestPools, IoBufferPools, and IoRequestQueues.

The si32UsbDefaultEndpointIoObject extends the si32UsbIoObject by implementing I/O operations specifically using the USB control pipe protocol over the default endpoint (EP0). The si32UsbDefaultEndpointIoObject treats transmit and receive operations as interdependent operations and enables support for SETUP packet detection and notification. The si32UsbDefaultEndpointIoObject does not have any dependencies on the si32UsbEndpointIoObject so that USB applications that require only the default endpoint can exclude the code and memory overhead required to support unused endpoints.

si32UsbEndpointIoObject extends the si32UsbIoObject by adding definitions and the initialization of resources required to support non-EP0 USB I/O Requests, including Bulk, Interrupt and Isochronous transfers. The si32UsbEndpointIoObject requires use of both the SI32_USB_A_Type and SI32_USBEP_A_Type hardware instances. Unlike the si32UsbDefaultEndpointIoObject, each direction of an si32UsbIoObject is designed to operate completely independent of the other direction and for configuring the SI32_USBEP_A hardware on demand, dynamically reconfiguring the endpoint control to reflect the run-time configuration.

Figure 23 shows the USB objects diagram.

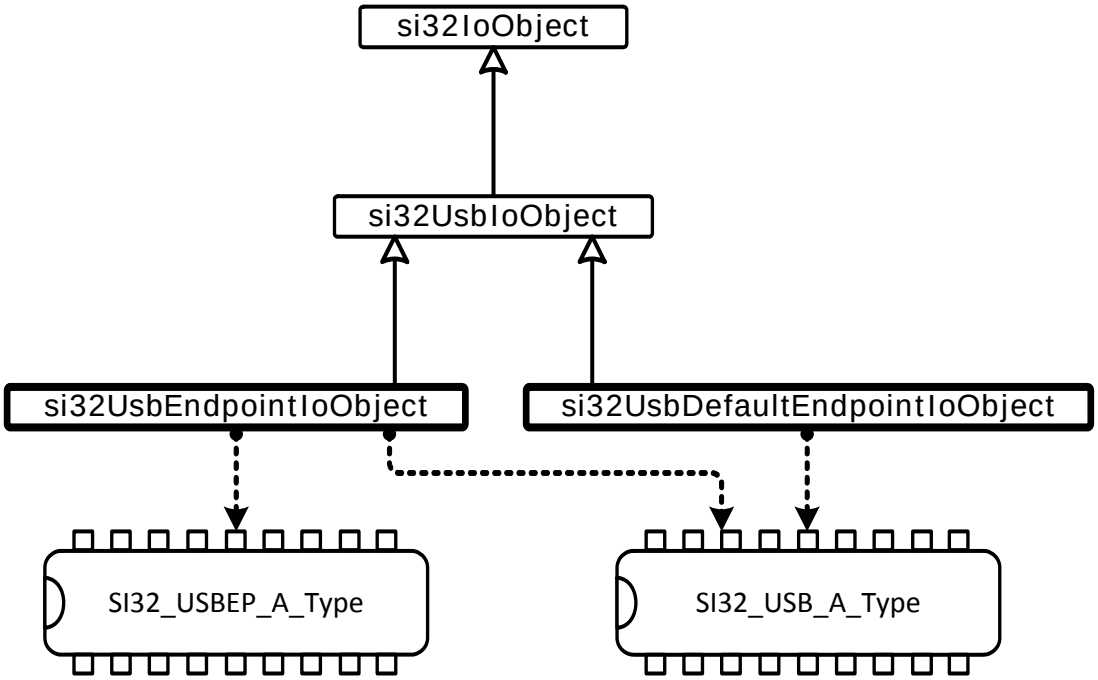
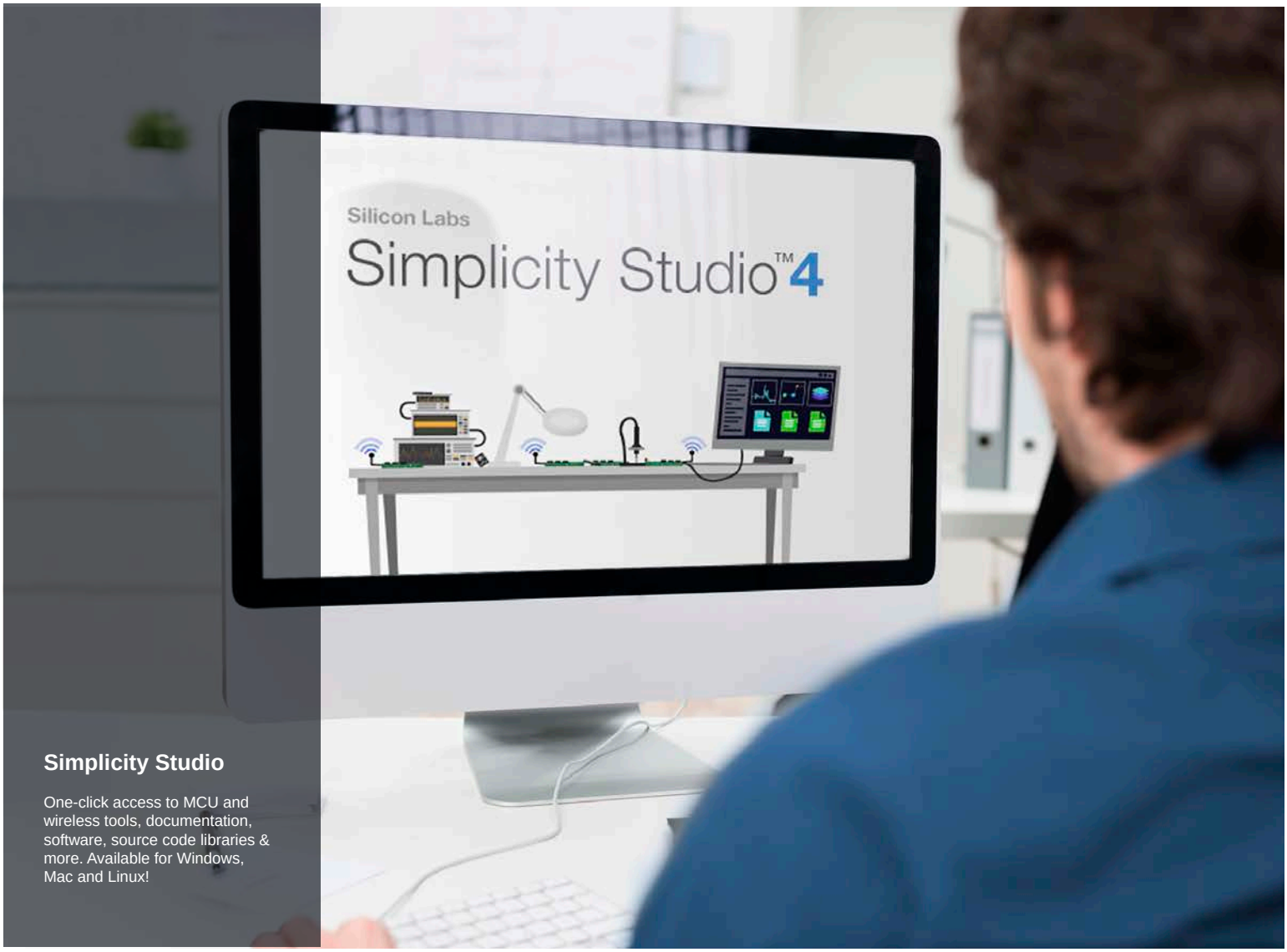


Figure 23. si32UsbIoObject, si32UsbEndpointIoOject, and si32UsbDefaultEndpointIoObject Diagram



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com