YAMI4

MISRA-C:2012 readiness report

For YAMI4Industry, v.1.3.1

Table of Contents

Document scope

The purpose of this document is to describe the level of standard compatibility and system requirements of the YAMI4 package dedicated for use in MISRA-C projects.

The document applies to the 1.3.1 version of the YAMI4Industry package.
See the following web site for general information on the project:

http://www.inspirel.com/yami4/industry.html

Introduction

YAMI4, a messaging solution for distributed systems, is a set of lightweight and easy to use libraries for various programming languages that support high-level communication patterns.

The YAMI4Industry package is a dedicated library written in the C programming language and is intended for use in embedded systems that are expected to offer the highest standards of quality. In particular, those projects that aim at safety certification, external audits or other forms of rigorous validation can decide to use the acclaimed MISRA-C coding standard to facilitate code reviews and reasoning about important code properties.

The YAMI4Industry package is a third-party component that implements useful messaging abstractions and that can be easily adapted and incorporated in MISRA-oriented projects.

MISRA-C compatibility

The source code of YAMI4Industry package was written with the MISRA-C:2012 standard in mind and was verified with the static code analysis tool.

It should be recognized that some of the MISRA-C rules are explicitly flagged as “undecidable”, which in practice means that code analysis tools can differ in how precise is their analysis and how many false positives and false negatives they can report for such rules. For this reason the package is not claimed to be a final and 100% finished product, but rather should be treated as a candidate for further adaptations - our intent is to assist users in resolutions of all remaining compliance issues in the context of actual target projects and tool sets.

There are only three potential MISRA-C deviations that are explained and justified below.

Use of char type for string literals

MISRA-C discourages programmers from using standard integral types (Advisory Directive 4.6) and it applies to the `char` type as well due to the fact that neither its size nor the signedness are guaranteed by the language - instead the coding standard suggests the use of `int8_t` or `uint8_t`.

This rule is followed except for the `char` type, which is used for string literals, for example:

```
char message_tag[] = "message";
```

The reason for this deviation is that such code is not vulnerable to any of the problems that are typical to the use of integral types as long as the character values are not used for arithmetics or for comparisons. The above code is also directly compliant with the language standard, which explicitly defines string literals to have `char[]` type.

Note, however, that whenever the code value of any given character is used, the operation is performed with the `int32_t` type.

Use of pointer conversions between unrelated pointer types

MISRA-C forbids pointer conversions between unrelated pointer types (Required Rule 11.3).

YAMI4 uses POSIX sockets for transport services and as such has to comply with requirements of the POSIX standard, which curiously relies on pointer conversions to achieve some forms of “polymorphism” between structure types devoted to different types from the same family (like, for example, to handle different address types).

For example, the `bind` function, which is used to assign local address to a socket descriptor, can be used with several address types, but has only one signature that allows one “base” type for all different address types.

The example usage presented in the POSIX standard:

http://pubs.opengroup.org/onlinepubs/9699919799/functions/bind.html

shows pointer conversion between two pointer types: `struct sockaddr_un *` and `struct sockaddr *`. These types are formally unrelated, but conversion between them is the expected way of using the socket API:

```
struct sockaddr_un my_addr;
/* ... */
```

```
      bind(sfd, (struct sockaddr *) &my_addr,
          sizeof(struct sockaddr_un))
```

Such type conversions are unavoidable and there are **seven** places in the whole package that rely on this kind of conversions, all of them located in the `network_utils.c` source file and marked with the following comment written in the preceding line in the source code:

```
      /* MISRA-C deviation, expected by POSIX. */
```

Similarly, the same POSIX functions expect some of their arguments to be cleared before call (that is, all of the bytes that the given structure is composed of should be set to zero), but since the actual structure is not fully defined by the standard, the only portable way to fulfill this requirement is to overwrite the complete object with the sequence of zero bytes.

The same POSIX page, linked above, suggests the following approach:

```
      memset(&my_addr, '\0', sizeof(struct sockaddr_un));
```

It is not possible to perform this kind of operations without violating MISRA-C rules related to pointer conversions (note: since by design there is no dependency on the standard library, a custom replacement function is used instead of `memset`), so it is necessary to deviate in order to meet POSIX expectations.

Such operations exist in **four** places in the package, all of them in the `network_utils.c` source file and marked with similar comment.

Serialization of floating point values

Another point that is a potential compliance issue is the serialization of floating point values. YAMI4 includes floating point values in its data model and to enable data exchange with nodes written in other programming languages the YAMI4Industry module also supports floating point data type. The problem with such values is that the binary serialization algorithm can be reasonably implemented in ways that violate MISRA-C rules related to pointer conversions or aliasing (Required Rule 11.3 for pointer conversions or perhaps Advisory Rule 19.2 for the use of unions as a way to implement object overlays).

The approach taken in the YAMI4Industry package is that the code implementing the serialization of floating point values is included in the library, but is deactivated by default and requires the definition of `YAMI4_WITH_DOUBLE_FLOAT` preprocessor macro for activation.

In other words, the code does not violate MISRA-C rules by default (and those users who are concerned with deactivated code can remove appropriate conditional sections altogether), but supporting sections can be activated by users who are aware of and accept the resulting coding standard deviations.

Possible restrictions in the use of callback functions

MISRA-C forbids recursive function calls as they make it more difficult to statically analyze worst-case stack usage (Required Rule 17.2).

YAMI4 uses callback functions to report important events like arrival of message, creation of communication channel or I/O error and the library is generally designed to accept further YAMI4 calls from such callbacks. It should be noted that depending on the actual callback invocation some further calls to YAMI4 might result in subsequent callbacks, thus leading to recursive calls between user code and the YAMI4 library.

The functions that can lead to callback invocations are those that are related to I/O work and are listed below:

- `yami_clean`
- `yami_set_listener`
- `yami_open_connection`
- `yami_close_connection`
- `yami_do_some_work`

These functions should not be called from user callbacks.

The optional message broker

The optional message broker is a stand-alone program that can route messages between communicating nodes based on their intended subscription patterns. Even though such an intermediary is not a necessary component in YAMI4 systems, it can be a useful building block in some high-level designs.

The message broker was also implemented with MISRA-C in mind, but since it is entirely based on the YAMI4 library, it does not introduce any potential deviations on its own and has no particular requirements with regard to the target platform.

Note: the message broker contains a very simple logging facility that reports its actions on the standard output. This facility, implemented in `src/broker/log.c` source file, is intended for demonstration only and should be replaced with a solution that will be more adequate on the actual target platform. In particular, the whole logging facility can be replaced with empty implementations on those targets where activity reporting is not necessary or is impractical for technical reasons. This also means that even though the supplied demonstration logging mechanism is implemented in terms of `stdio.h`, it does not violate the MISRA-C standard, which explicitly forbids its use (Required Rule 21.6).

Expected system environment

YAMI4Industry package was implemented with POSIX-like systems in mind, but takes into account constraints that are typical to embedded platforms. In order to facilitate integration within the target system, the following sections describe the expected system environment:

Language Runtime Services

The YAMI4Industry package has **no dependency on the C language runtime**.

The only dependencies on the standard C library are of compile-time nature and exist only to satisfy the expectations of MISRA-C coding standard and are related to the use of several type definitions. The following header files are used by the package:

C header file	Symbols used
`stddef.h`	`NULL, size_t`
`stdint.h`	`int32_t, int64_t, uint8_t, uint16_t, uint32_t`

POSIX Services

The YAMI4 messaging system relies on network services of the underlying operating system to deliver messages between communicating agents. The high-level libraries use TCP, UDP and local sockets (so-called Unix sockets) as transport layers, but the current version of YAMI4Industry package supports only TCP and UDP transmission.

Network services are used via the POSIX socket API. The following header files, functions and definitions are used:

POSIX header file	Symbols used
`arpa/inet.h`	`htonl, htons, ntohs`
`errno.h`	`EINPROGRESS, EINTR, errno`
`fcntl.h`	`fcntl, F_GETFL, F_SETFL`

netinet/in.h	INADDR_ANY, IPPROTO_TCP, sockaddr_in
netinet/tcp.h	TCP_NODELAY
sys/select.h	FD_ISSET, FD_SET, FD_ZERO, select, timeval
sys/socket.h	accept, AF_INET, bind, connect, getsockopt, listen, MSG_NOSIGNAL, recvfrom, send, sendto, setsockopt, sockaddr, socket, socklen_t, SO_ERROR, SO_KEEPALIVE, SO_REUSEADDR, SOCK_DGRAM, SOCK_STREAM, SOL_SOCKET
unistd.h	close, O_NONBLOCK, read

The user should ensure that the above symbols and their semantics are properly supported on the target platform.

Note that the YAMI4Industry package does not rely on POSIX threads support and as such is also not intended for liberal use in multithreading environments. It is safe to use separate YAMI4 agents in separate threads, but a single agent should not be accessed by more than one thread unless the application ensures proper mutual exclusion by its own means. Note that this restriction is a property of the YAMI4Industry package only and is not present in other YAMI4 libraries.

Memory requirements

According to MISRA-C rules, the code does not rely on dynamic memory. All data structures and buffers are sized statically and are part of the main `struct yami_agent` that the user code should instantiate. It is also expected that agent initialization performed by means of `yami_init_with_user_array` or `yami_init_with_options_ua` will be done with a static array of channels allocated outside of the agent object. Following this restriction is at the discretion of the library user.

All sizes are defined in the `limits.h` header file and special care should be taken to tune the values as required for the target system. It should be noted that input and output buffers for all channels together will typically contribute significantly to the memory requirement of the YAMI4 agent (`sizeof(struct yami_agent)`) and for this reason the total size of the agent should be taken into account at the code design stage. Even though the test programs allocate the agent object on the stack for simplicity, it will not be a proper solution for target embedded systems, where stack space might be very restricted.

Additional consideration should be given to the natural limitation of the UDP packet size - there is no point in allocating message buffers bigger than this limit if only UDP communication is used.

In the case of optional message broker, memory requirements depend on the size of subscription table, which is defined in the `broker_limits.h` file and on the size of the contained agent itself. It should be noted that the subscription table can have size that is independent on the maximum number of channels in the agent and normally it is the agent that is the most significant contributor to the memory requirements of the whole broker.

YAMI4 protocol restrictions

In order to simplify the internal implementation and to make it easier to comply with MISRA-C rules, the following tradeoffs were made:

1. The data model and the serializer offer only limited support for nested parameters objects. This restriction is related to the fact that a reasonable full implementation of nesting would have to rely on recursive calls, which are not allowed in MISRA-C compliant code.

2. Note that it is still possible to create and parse message payloads containing nested values by hand, but the library does not offer any assistance in skipping over unknown (not recognized by the application) nested fields.

3. The channels are dimensioned statically and as such do not allow arbitrarily long messages. Moreover, since having an upper bound on the message size allows to define a message frame size that will be sufficient for every possible message, the wire protocol is further simplified to allow only single-frame messages. This restriction can introduce wire-compatibility issues with other distributed components built in terms of other YAMI4 libraries, if they send messages that are bigger than their configured frame sizes (as this leads to multiple-framed messages, which are not understood by the YAMI4Industry package).

4. The DNS services are not used (they are most likely restricted in the embedded environment anyway) and therefore no host name resolution is attempted - all addresses have to be provided in their numerical form, for example:

   ```
   "tcp://127.0.0.1:12345"
   ```

The user should make sure that other components in the distributed system are aware of these restrictions as well, so that unknown nested objects or multiple-framed messages are not sent to those nodes that are implemented in terms of YAMI4 Industry package.

Revision history

Revision	Comment
1	Initial revision, reformatted from previous document with no changes. Refers to the 1.3.0 version of the YAMI4Industry source package.
2	Reviewed and updated for the 1.3.1 library version.