MemoryRange

# MemoryRange

Embedded applications regularly handle blocks of memory. Such memory may either be typed (for example, consisting of a number of characters) or untyped. Traditionally, blocks of memory are handled by having a pointer to its first element and a separate size. Since passing around two separate but related parameters is cumbersome at best and error-prone at its worst, we have improved on dealing with blocks of memory by introducing the class template MemoryRange. A MemoryRange is an object which defines a block of memory with a start and an end, and which has a number of functions that makes dealing with those blocks as a whole easier. This approach reduces the potential for buffer overflows, since while it is easy to shrink a block or to split it, it is much more difficult to create a larger block, thereby overflowing the original block of memory.

The MemoryRanges that are most often used are MemoryRange<uint8\_t>, which has an alias ByteRange, and its const counterpart ConstByteRange which aliases to MemoryRange<const uint8\_t>. Everywhere where untyped memory is used ByteRange or ConstByteRange is used to define where the data resides in memory. For example: the SPI interface SendData method takes a ConstByteRange parameter specifying the data which should be sent.

A MemoryRange does not include the storage it points to. Modifying a MemoryRange does not modify the storage. If storage is needed, std::array is a good candidate. After creating a std::array and filling it with data, a MemoryRange can be constructed from it by using MakeRange, after which the MemoryRange can be used for passing around the data.

# MemoryRange Accessors

These are the most used accessors of MemoryRange:

|  |  |
| --- | --- |
| begin() | Returns a pointer to the first element of the data |
| end() | Returns a pointer to one position beyond the last element of the data |
| empty() | Returns true if and only if size() == 0 |
| size() | Returns the number of elements pointed to: end() - begin() |
| pop\_front(num) | Remove num elements from the front of the range |
| pop\_back(num) | Remove num elements from the back of the range |
| shrink\_from\_front\_to(newSize) | If size() > newSize, pop elements from the front until size() == newSize |
| shrink\_from\_back\_to(newSize) | If size() > newSize, pop elements from the back until size() == newSize |

# Helper Functions

In addition to the members of MemoryRange, a number of helper functions exist:

|  |  |
| --- | --- |
| MakeRange(...) | Convert the parameter(s) to a MemoryRange. Works with std::array, infra::BoundedVector, etc. |
| Head(range, size) | Returns range if its size is less than size, otherwise a new range consisting of the first size elements of range |
| Tail(range, size) | Returns range if its size is less than size, otherwise a new range consisting of the last size elements of range |
| DiscardHead(range, size) | Returns an empty range if size is greater than range.size(), otherwise a new range consisting of range except for the first size elements |
| DiscardTail(range, size) | Returns an empty range if size is greater than range.size(), otherwise a new range consisting of range except for the last size elements |

# Example

This example function shows how to use the MemoryRange functions to cut up a block of data into smaller pieces:

void SendInBlocks(infra::ConstByteRange sendData, std::size\_t blockSize)

{

while (!sendData.empty())

{

Send(infra::Head(sendData, blockSize));

sendData = infra::DiscardHead(sendData, blockSize);

}

}

void Example()

{

std::array<uint8\_t, 256> data = { ... };

SendInBlocks(data, 16);

}