Bern University of Applied Sciences

Exercise - 24

Ring Buffer with DMA and UART

Andreas Habegger | Adrian Steiner
BTS4230 | Version 1.0.0 of 17.03.2025

In the previous exercises you have learnt about problems and possible solutions for sending messages with UART from the super-loop and an interrupt handler. You also implemented a ring buffer in Lab-2.01: Ring Buffer Implementation with arbitrary data type handling. Since the solutions and methods in the exercises are not satisfactory, we extend these exercises and use the ring buffer to handle our messages and send them to the computer using DMA and UART. This new solution provides optimised message handling and solves the basic problem of data loss.

Objectives

- Using your own ring buffer module
- Handling strings with your ring buffer
- Preventing interrupt messages from being overwritten
- Using a non-blocking superloop

Outcomes

- Rewrite existing implementations
- Use DMA optimised string ring buffers
- Understand the difference between CPU and DMA workloads
- Using a deterministic superloop without a blocking function

Description

The goal of Lab-2.01 was to implement a ring buffer module. An additional abstraction from the ring buffer can handle c-strings. These strings are then sent using the blocking HAL UART send function. In

Ex-23: UART with DMA, a possible optimisation of the blocking send function was worked on and this is now to be combined with the ring buffer.

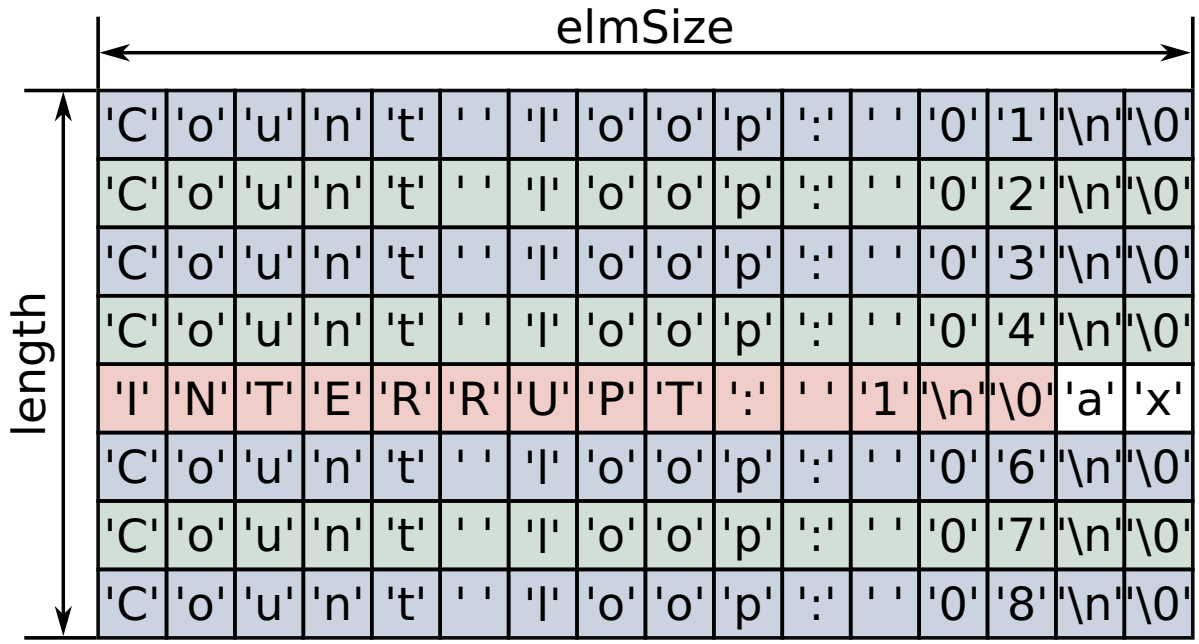
As always, optimisations may introduce new problems that need to be identified and discussed. This exercise is based on the two previous exercises and laboratory. Make sure that these exercises have been solved and understood.

Tasks

C-String abstraction module

The first task is to optimise the string abstraction module for DMA transfers. To transfer a string by DMA to the UART peripheral, the DMA process needs the start address of the memory segment and its length. The string ring buffer is used as a handler for complete strings. A string can be loaded into the buffer using the enqueue function. The dequeue function returns the start address of a correct c-character string. The length can be determined with the C standard library function `strlen()`. See the figure below for more information.

Buffer Memory Management



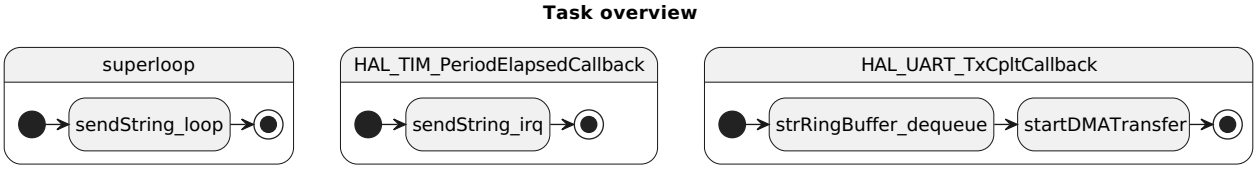
Buffer memory block example

Tip

The possible solutions in Lab-2.01 are designed for this abstraction and can be used.

Functions needed

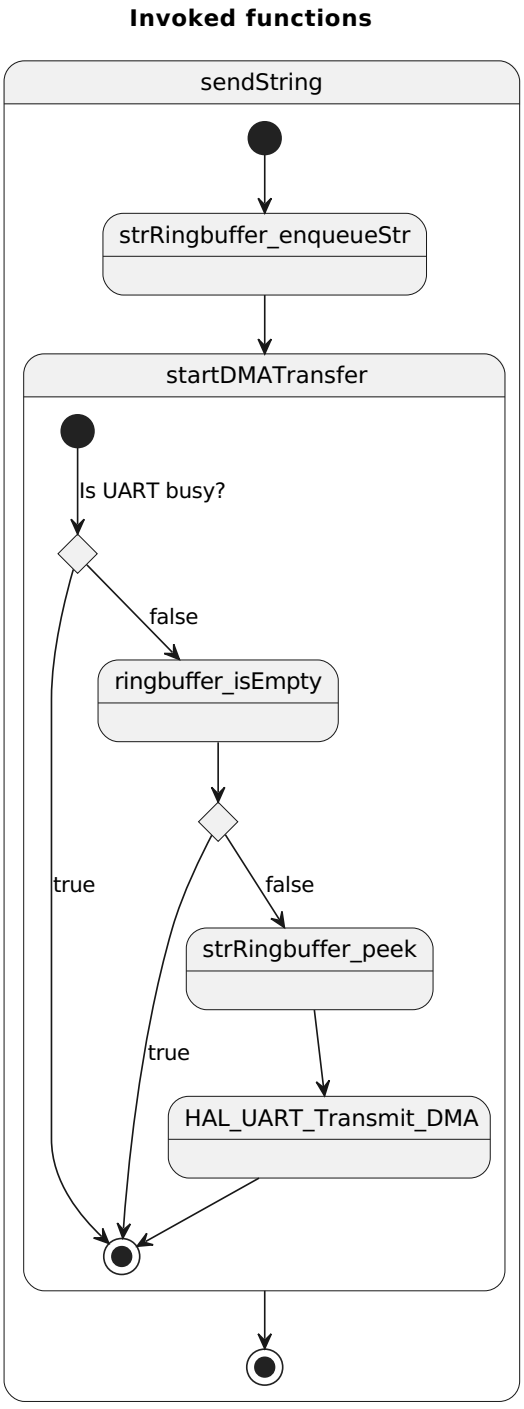
Task overview



The tasks at a glance

There are three different functionalities – encapsulated in different tasks/functions, that are used for the basic application of this exercise. The superloop and timer interrupt callback function is used to enqueue data and trigger a DMA transfer if necessary. The `HAL_UART_TxCpltCallback()` function dequeues a string to free the current buffer segment and triggers a new DMA transfer if the ring buffer is not empty. The task uses two new functions as shown in the flowchart on the right side.

The purpose of the `startDMATransfer()` function is to trigger a new DMA transfer if data is available in the string ring buffer. This function is called in all tasks to always attempt to empty the string ring buffer. The peek function is used to pass the data, but not to release the element. This ensures that the data cannot be overwritten. As mentioned above, the `HAL_UART_TxCpltCallback()` function dequeues the string ring buffer to free the data.



The flowchart of invoked functions

Implementation

To start this exercise, create a new project or use a project from previous exercises as a template. The first implementation is to combine the two base projects.

Project setup from previous exercises

1. Create a new project or use as base the project from the previous exercises.

2. Enable DMA and interrupt for UART2 as configured in Ex-23: UART with DMA.
3. Configure and enable a timer based IRQ that fires periodically (frequency $10\,\mathrm{Hz}$).
4. Add your ring buffer module and string abstraction from Lab-2.01: Ring Buffer Implementation.
5. If necessary, redesign your string ring buffer abstraction as explained in the C-String abstraction module section.

After configuring the project and merging the previous exercises, the goal is to implement the new job functions. As all tasks use `startDMATransfer()`, this is the first function that needs to be implemented.

startDMATransfer()

6. Implement the `startDMATransfer()` function:
   a. Check if the UART is busy
   b. Check that the ring buffer is not empty
   c. Get the current data using the peek function
   d. Send the stored c string using the `HAL_UART_Transmit_DMA()` function

Possible Function Header

```c
/**
 * @brief Starts a new DMA process to send the next message
 *        if huart UART is not busy.
 *
 * @param huart the UART handler
 * @param strRingBuf the string ring buffer handler
 */
void startDMATransfer(UART_HandleTypeDef *huart, ringbufferType *strRingBuf);
```

To complete the new functions, implement the `sendString()` function.

sendString()

7. Implement the `sendString()` function
   a. Enqueue the string ring buffer with the new data
   b. Trigger a possible new DMA transfer with `startDMATransfer()`

Possible Function Header

```
/**
 * @brief Enqueues the given string format into the string ring buffer.
 *        After saving, the function starts the DMA process
 *        if the UART is not busy to send the next message
 *        in the string ring buffer.
 *
 * @param huart HAL UART handler
 * @param strRingBuf string ring buffer handler
 * @param format String format
 * @param ... additional arguments
 */
void sendString(
    UART_HandleTypeDef *huart,
    ringbufferType *strRingBuf,
    const char *format,
    ...);
```

Hint

You can use the same functionality as in `printf` or `snprintf` with the format string `const char *format` and the additional arguments `...` by using the following functions:

1. Include standard header files `<stdarg.h>` and `<stdio.h>`
2. Create a `va_list` instance.
3. Link the variable list with `va_start()`.
4. Create the string with the arguments from the function header with `vsnprintf()`. Use a buffer size of your choice.
5. Clean up argument list with `va_end()`.

```
va_list vl;
va_start(vl, format);
vsnprintf(buf, sizeof(buf), format, vl);
va_end(vl);
```

Using the new functions, implement all three tasks with their corresponding jobs to test the behaviour of your implementation.

Tasks initialisation and configuration

8. Create and initialise a string ring buffer instance and buffer with your defined length and size.
9. Implement the super-loop task with a user defined message.
10. Implement the `HAL_TIM_PeriodElapsedCallback()` interrupt task with a different message.
11. Implement the `HAL_UART_TxCpltCallback()` interrupt task.
12. Before testing your implementation, answer the following questions with your expectations:

    Question

    a. Do you expect to receive all messages from both tasks?

    b. What could be a problem?

    c. What are the limitations of your implementation?

13. Now test your implementation and discuss your expectations from the results:

    Question

    a. Are you receiving a message from both tasks?

    b. What is the problem?

    c. What are the limitation?

    Solution

    The problem is the current speed at which the CPU can fill the buffer. Although the DMA is fast when it is sent, it is not as fast as the current required CPU load. Sooner or later, the ring buffer will overfill.

This problem can be solved by sending periodic messages in the superloop. A first quick solution can be implemented with the `HAL_Delay()` function. This is not recommended as it blocks the CPU. A better, more efficient and more deterministic solution is to use the SysTick. The SysTick increments a counter variable with a default frequency of $1\,\mathrm{kHz}$, which can be read with `HAL_GetTick()`.

SysTick in superloop

14. Add a condition to send a message periodically in the superloop without blocking the CPU.

> **Note**
>
> The counter value should not be changed because it can be used by other parts of the code. Using the modulo operator can be a solution, but in this case it has two problems:
>
> - Missing a period in case of high CPU load
> - Several events in one period if the CPU load is low.
>
> A different solution with a greater than or equal condition can solve these problems.

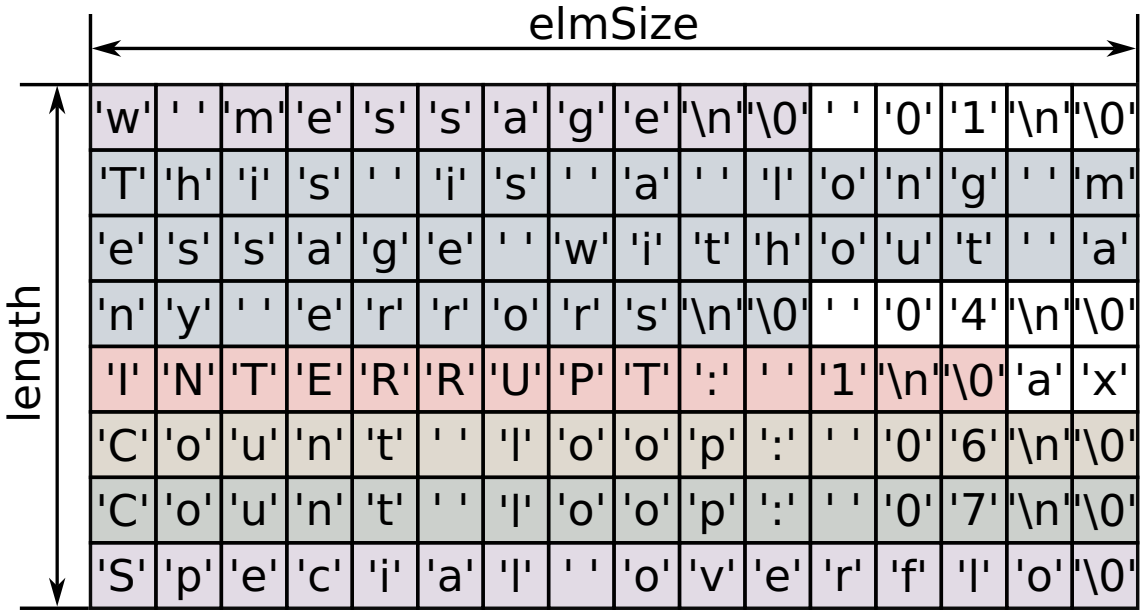
15. Reflect on the implementation of the previous exercises and answer the following questions:

> **Question**
>
> a. Have all the problems been solved?
>
> b. How could you improve your solution?
>
> c. How do you determine the size of the ring buffer?

(optional) Optimize string ring buffer

Optionally, you can tweak your string ring buffer to add a new feature. Currently the strings have a maximum size of the string length variable from the buffer array. This can be optimised by using the continuous array behaviour by writing directly to the next buffer element and incrementing the end position twice. The same principle can be used for dequeuing. The problem with this feature is handling the overflow from the last position to the new element, as it is forbidden to overwrite to the next element there.

Optional Buffer Management



Buffer memory block example

16. Redesign `strRingbuffer_enqueueStr()` to add the possibility described above.

> **Hint**
>
> Do not forget to handle the critical last element, which must not overflow the buffer element size.

17. Rewrite `strRingbuffer_dequeueStr()` to return the string with the correct incremented head position.
18. Test your implementation with a small buffer element size and check for the critical overflow.