Embedded systems security
Hardware architecture/microarchitecture attacks

Prof. Stefano Di Carlo

DMA Attacks

Lecture credits:
Elias Michael
Marino Lorenzo
Sansone Giacomo
Valenza Arianna

Acknoledgments

This material was initially developed as part of an assignment for the Operating Systems for embedded systems course delivered at Politecnico di Torino by Prof. Stefano Di Carlo during the academic year 2022/20023.

Credits for the preparation of this material go to:

- Elias Michael (`https://github.com/mik300`)
- Marino Lorenzo (`https://github.com/iken1997`)
- Sansone Giacomo (`https://github.com/pcineverdies`)
- Valenza Arianna

Contents

1 Structure of the work **3**

2 Overview on DMA attacks **4**

2.1 Prerequisites 4

2.1.1 What is the DMA 4

2.1.2 Memory protection 5

2.2 An introduction to DMA attacks 5

2.3 Some examples of DMA attacks 6

2.4 Countermeasures 6

3 Documents from literature **7**

3.1 Paper 1 7

3.2 Paper 2 8

3.3 Paper 3 9

3.4 Paper 4 10

4 Documentation **12**

4.1 UART Library 12

4.1.1 uart_parameters 12

4.1.2 init_uart 13

4.2 DMA Library 14

4.2.1 dma_transfer 14

4.2.2 dma_lli_element 15

4.2.3 init_DMA 15

4.2.4 transfer_DMA 15

4.3 Application 15

4.3.1 Setup 15

4.3.2 Micrium OS 16

4.3.3 APP task 16

4.3.4 Lock task 17

4.3.5 Display task 17

4.3.6 DMA_IRQHandler 17

Chapter 1

Structure of the work

This document collects material and resources on DMA attacks.

This document has the following structure:

- chapter 2 is a simple introduction to Direct Access Memory (DMA) attacks. This type of attack is not particularly difficult to understand, as the interesting part is to perform (and try to avoid) it, so that is only a small part of the work;
- chapter 4 explains the two libraries we wrote for the LPC1768 microprocessor to activate and use the UART peripheral and the GPDMA controller;
- chapter 3 presents brief overviews of some of the relevant papers we have identified on this topic.

Chapter 2

Overview on DMA attacks

2.1 Prerequisites

2.1.1 What is the DMA

Direct Memory Access is a feature that allows different peripherals to access the memory directly, independently of the CPU. This frees up the CPU to perform other tasks and can significantly boost the system's performance. Many hardware systems use DMA, including disk drive controllers, graphics cards, and network cards. It is a hardware technology that works independently from the operating system. Therefore, since it works with physical addresses, it represents a source of nondeterministic behaviors and a possible backdoor for malware attacks. DMA is typically composed of the following:

- A DMA controller coordinates data transfer between the memory and peripheral devices.
- A set of DMA channels: allows multiple devices to perform DMA transfers simultaneously.
- A DMA buffer: a small amount of memory reserved for the DMA controller to store data during transfers temporarily.

We can have two different organizations of the DMA engine:

- Third-party DMA: The host CPU configures (source and destination address) the central DMA controller to perform a DMA transfer. The DMA controller interrupts the host CPU after the DMA transfer. Hence, the host CPU is aware of a third-party DMA transfer.
- First-party DMA: The peripheral device can configure its own DMA engine. The device acts as a bus master to get control of the system bus to perform a DMA transfer. The device can interrupt the host CPU when the device has completed the transfer. The transfer also works if the device does not interrupt the host CPU at the end of the DMA transfer. In this case, the CPU is entirely unaware of the DMA transfer.

2.1.2 Memory protection

Memory protection is crucial to computer systems, especially for modern operating systems, as it helps prevent errors and security breaches. Instead of using the physical addresses of the memory space, we use *virtual addresses*, which will be translated to the real ones by proper hardware devices. Since there is always an entity (MMU - Memory Management Unit) between us and the memory, it is impossible to access the memory freely. It can be implemented in different ways, such as:

- Memory segmentation: This technique divides memory into distinct segments, each with its specific purpose and access control permissions. This helps prevent programs from accessing memory locations they are not authorized to access.
- Paging: Paging is a technique where the physical memory is divided into fixed-sized blocks called pages, which can be individually assigned to different processes. This allows the operating system to control and monitor the memory access of each process.
- Access control: Access control is a mechanism that defines which processes or users are allowed to access specific memory locations and what actions they are permitted to perform. This helps prevent unauthorized access to sensitive information.
- Virtual memory: Virtual memory is a technique that allows a process to use more memory than is physically available by temporarily transferring data from physical memory to a disk. This helps ensure that processes don't interfere with each other's memory usage.

The implementation can vary depending on the operating system, hardware, and security requirements.

Despite its many advantages, in many embedded systems, this implementation is avoided to reduce costs, complexity, and time to market.

2.2 An introduction to DMA attacks

DMA attacks are a mighty class of attacks, especially for embedded systems. These attacks enable a potential attacker to read and write memory off a victim system directly, bypassing the main CPU and OS. Achieving DMA attacks in embedded systems is more accessible than in a general-purpose computer due to the absence of memory protection and more dangerous because often flat architecture operating systems are implemented in that kind of systems, which allows the attacker to access reserved and critical areas, like the ones of the OS itself.

By overwriting memory in a DMA transfer, for instance, just increasing the size of the burst transfer, the attacker can modify and control reserved memory regions, performing any manner of malicious activity.

Direct Memory Access is a capability implemented in every system that needs to speed up the transfer from/to peripherals. It permits the execution of the transfer without the aid of the CPU or Operating System after setting up the DMA so that the peripherals can directly communicate with the system's memory and save time.

Despite the possibility of accelerating the system, DMA exposes the system to various attacks, for instance:

- Access and modify reserved information;
- Extend control over the execution of the kernel itself;
- Malware injection.

2.3 Some examples of DMA attacks

One of the main tools used to perform DMA attacks is pcileech. Most of the works you can find on the internet regarding DMA attacks are proof of concept, though there are some cases in which hackers could inject malicious code inside a computer once they have access to the hardware.

2.4 Countermeasures

DMA attacks can be difficult to prevent because they occur at a hardware level and are not visible to the operating system or CPU. However, some measures can be taken to reduce the risk of DMA attacks and make them more challenging to execute.

One way to prevent DMA attacks is to use an IOMMU (Input/Output Memory Management Unit). An IOMMU hardware component acts as a memory management unit for input/output (I/O) operations. It allows the system to restrict the memory addresses that I/O devices can access, making it more difficult for an attacker to perform a DMA attack.

A different approach is to use hardware-enforced memory isolation. This technique involves dividing the system's memory into other regions, with each region assigned a specific level of access. This can help to prevent an attacker from gaining access to sensitive areas of memory.

Another measure to prevent DMA attacks is to use a secure boot. Secure boot is a process that verifies the integrity of the system's boot process. It ensures that only trusted software can run when the system boots up, making it more difficult for an attacker to inject malicious code.

Other approaches to preventing DMA attacks include using hardware-based firewalls, implementing role-based access control, and hardware-based encryption.

Chapter 3

Documents from literature

We used the following papers to understand better how DMA attacks are performed.

3.1 Paper 1

The paper's title is *Characterizing, exploiting, and detecting DMA code injection vulnerabilities in the presence of an IOMMU* [1].

This paper primarily focuses on the challenges of conducting a DMA attack when an Input/Output (IO) Memory Management Unit (MMU) is present (an IO MMU is similar to a regular MMU but only deals with IO accesses). This paper may be challenging to understand if you are unfamiliar with virtual memory on x86 systems. Since our system does not have virtual memory or protection, we aim to summarize some of the key ideas from the article using only physical address space.

The first half of the paper is the most important, as it provides an overview of DMA attacks and the necessary conditions to perform code injection. In the second half, the authors present a tool they developed to perform static analysis of the Linux kernel and identify vulnerabilities of a particular type.

A system with a DMA controller is always vulnerable to DMA attacks. These attacks can occur concurrently with normal CPU execution, making it difficult for the system to detect them. DMA attacks can often be executed by exploiting firmware bugs. Various attacks can be carried out using DMA, including injecting and executing code on the target system, stealing sensitive information such as secret cipher keys, and dumping the entire memory.

As mentioned, the paper discusses some characteristics of modern operating systems that can help defend against DMA attacks. We will briefly summarize them here to explain why they are irrelevant in our context.

- IOMMU: an Input/Output Memory Management Unit (IOMMU) allows peripherals to access only certain virtual addresses (IOVAs) and prevents them from accessing physical addresses directly. A single physical address can be mapped to multiple IOVAs, just as in the CPU's virtual memory. In our system, we don't use virtual addresses, as we do not have an MMU: both the CPU and the peripherals always use physical addresses without any form of protection.

- NX-BIT: in a system with virtual memory, pages have certain bits that describe the operations that can be performed on them. For example, if the W bit is 1, you can write to that page using a specific virtual address (VA). If you try to perform this operation on a page with the W bit set to 0, you will receive an exception (likely a segmentation fault). One bit that is present is the NX bit, which determines whether the content of that page can be used as executable code or not. If a peripheral can inject malicious code into memory and that memory area has NX=0, the code cannot be executed. If you try to run it, you will receive an exception. This is not an issue in our system since we do not have any protection. If we can write to the Program Counter register, we can execute malicious code from any location in memory as long as we have write access.

- (K)ASLR: in many DMA attacks, we need to know a specific memory address to steal data or inject malicious code. This is easy if we already know the memory layout. Some systems randomize the memory layout and the location of critical data structures at each boot to prevent attackers from guessing the locations based on previous execution. Since our system does not have this feature, locating addresses using the .mem file may be easy after compiling/flashing the board.

The paper's authors often refer to sub-page vulnerabilities, which occur when the target buffer of a DMA transaction is smaller than a page, and other data on that page may be modified or stolen. There is a classification of attacks based on this characteristic. Since we are not concerned with pages, we can think about this classification as physical memory only:

- The IO buffer (target of the DMA transaction) is part of a larger data structure that includes function pointers. These pointers may be used as function callbacks to execute our malicious code.

- There is some operating system metadata in addition to the IO buffer.

- The IO buffer is next to some dynamically allocated data structure. This is difficult to discover since we are not responsible for managing the heap.

To perform a code injection, we need the following three attributes:

1. A buffer filled with malicious code. We also know its physical address if we can write to that buffer.

2. Write access to a function callback pointer, which can alter the CPU control flow and cause it to execute the malicious code.

3. A time window exists when the device can modify the callback pointer, and the CPU will subsequently jump to the pointed code.

3.2 Paper 2

The paper's title is *Stealthy Information Leakage Through Peripheral Exploitation in Modern Embedded Systems* [4].

At the beginning of the work, the authors emphasize that security is often overlooked in the embedded domain because time-to-market is considered more valuable (and engineers must focus on implementing functionalities rather than making them secure). One way to detect infiltrations in an embedded device is to profile the CPU: if its usage is higher than usual, there may be malware inside. This is why DMA attacks are powerful: DMA does not interact with the CPU, so its activity is hidden from any profiling tools within the system (the suggested solution is to detect malware by examining DMA transactions).

The authors use the Device Tree, a data structure provided by Embedded Linux that contains information about all connected devices to make the attack as general as possible. By parsing its content, they can automate choosing the appropriate output analog device and initialize the DMA transactions with the correct values.

To carry out the attack, the board's firmware must be modified, which can be done physically (by changing the content of the boot memory) or remotely (by accessing the board through a network connection and gaining root access). There are various ways to verify the integrity of the original code, such as checking the hash of the entire code against a value stored in a secure memory (such as ROM, which cannot be modified). x86 systems are introducing the Secure Boot system, which cannot be implemented in already operational embedded systems. This system is unlikely to be used in all new devices due to the area and power constraints in this domain.

We are almost finished after introducing the new kernel with our malware (some code that is periodically executed to exfiltrate data). After reconnaissance and considering the attack objectives, the attacker must decide which memory addresses to leak. In general, embedded systems do not implement Address Space Layout Randomization (ASLR), so finding application data addresses, such as the control runtime process and leak control-related information, is easy.

The remaining sections of the article discuss the actual implementation, mitigation techniques, and experimental results that demonstrate that DMA attacks do not affect CPU time.

3.3 Paper 3

The title of the paper is *Attacking TrustZone on devices lacking memory protection* [2].

ARM Trustzone offers a Trusted Execution Environment (TEE) embedded into the processor core. It establishes a new CPU operating mode called "secure mode"; therefore, it creates a distinction between a "secure" world and a "normal" world. The separation is artificial: both environments use the same CPU and RAM; the Non-Secure bit (NS) gives the distinction.

In this paper, the authors exploit missing hardware feature on real embedded systems to execute arbitrary code in a secure world: to work correctly, TrustZone need an additional dedicated hardware controller, i.e., TZASC (TrustZone Address Space Controller) and TZPC (TrustZone Protection Controller).

The attack is performed on a Raspberry Pi 3 Model B. This board has an SoC with ARM11 core and TrustZone support but lacks the abovementioned controller.

On this SoC, the authors run OP-TEE (a Trusted Execution Environment that implements ARM TrustZone) without NS-bit support. Therefore the secure memory is accessible through DMA transactions. Their goal is to change the opcodes that return error values of key functions without changing the stack; in this way, they can bypass OP-TEE OS Trusted Application (TA) signature validation and gain control of every TA of the system. To reach this goal, first, they read the memory pages from the RAM, then compare this memory with a compiled version of the secure OS and ARM Trusted Firmware to locate similar functions. Because the OS uses Address Space Layout Randomization, they used the address from the memory dump to override trusted applications' signature validations through the DMA; after doing so, the TEE OS succeeded in validating their malicious TA, making possible the arbitrary code execution.

3.4 Paper 4

The provided document is a chapter of a conference proceeding of the 9th *GI International Conference on Detection of Intrusions and Malware & Vulnerability Assessment*; the chapter name is *Understanding DMA Malware*[3].

This paper explains how they implemented malware exploiting DMA on x86 architectures, particularly executing malicious code on the processor provided by *Intel's Manageability Engine*.

The malware is called **DAGGER** (**D**m**A** based keystroke lo**GGER**), and in the implementations described in the paper, it has been used to find and monitor the keyboard buffer.

Controlling the DMA engine is not enough to perform this kind of attack because the target memory address is usually unknown. The attacker must overcome *address randomization* since the data structures are saved in different locations at every reboot. Moreover, the OS works with virtual addresses whether the DMA uses physical ones: the memory mapping is saved in so-called *page tables*, in which the physical address is held in a register not visible to the DMA engine. Therefore the attacker needs to find a way to restrict the search space. Otherwise, it needs to scan the whole host memory. In the document, the authors suggest analyzing if the OS places the desired data structures in approximately the same region or implementing OS memory management mechanisms.

DAGGER performs a stealthy attack structured in three stages corresponding to three components:

1. Search: find the address of valuable data in the host memory via DMA.

2. Process Data: read valuable data within the regions identified in the search process.

3. Exfiltration: exfiltrate information in an invisible way to the host (e.g., using the Network Interface Card).

The authors implemented two prototypes to attack two different OS targets: Windows and Linux. The search phase differs according to the OS to attack:

- Linux: the search phase is based on a signature scan compared with a signature derived analyzing the source code of the OS. The scan is restricted only to the kernel space found in the first gigabyte of the system RAM. Exploiting GRUB, the kernel identifier is located at a constant physical memory address; using this value, it is possible to derive the offset used by the OS to map the virtual addresses to the physical ones. The authors successively restrict the search space further by empirically analyzing which memory area the kernel places the needed data structure, which in this case is *USB Request Block*, in particular the field *transfer_ dma*.

- Windows: Unlike Linux, the memory mapping does not just use an offset, but *page tables* created by the kernel. The authors empirically determined the physical address of these tables and traversed them, looking for a structure named *DeviceExtension* containing the last-pressed key buffer. The algorithm to find a starting point for the search is more elaborate concerning the previous case and requires more steps.

In the paper, some possible countermeasures are discussed in addition to the performance results of the malware. It is claimed that the attack has not been detected by any anti-virus or anti-malware software present on the OS. As for the I/OMMU, the researchers enabled Intel VT-d and successfully blocked the attack on Linux but not Windows. They also state that to prevent the attack; an I/OMMU needs an appropriate configuration and additional hardware that prevents the same malware from changing its configuration.

Chapter 4

Documentation

In this chapter, the developed project simulating the DMA Attack through UART peripheral will be described in detail. All the code developed is intended to be used with the microprocessor LPC1768. The user manual for this microprocessor can be found here. This was the primary source for the project: as it is known, firmware development mainly consists in understanding and applying concepts from it, with the aim of creating a working system. Other interesting sources to write our code were this and this (though part of this presentation is in Spanish).

The code is meant to be clear and modular through macros.

4.1 UART Library

The following subsections describe the composition of the UART library and how to enable the peripheral for communication (both sending and receiving). This library can be used with peripherals 0, 2, and 3. UART 1 has a different structure.

4.1.1 uart_parameters

```c
struct uart_parameters{
  uint32_t  uart_n;
  uint32_t  clock_selection;
  uint32_t  DLL;
  uint32_t  DLM;
  uint32_t  DivAddVal;
  uint32_t  MulVal;
  uint32_t  fifo_enable;
  uint32_t  dma_mode;
  uint32_t  trigger_level;
  uint32_t  rdv_int;
  uint32_t  word_size;
  uint32_t  stop_bit_n;
  uint32_t  parity_active;
  uint32_t  parity_type;
};
```

Listing 4.1: Structure definition for UART parameters

This structure is used to define the communication parameters. Each field of the structure can take either a numeric value or a specific value as specified in the LPC1768 user manual. We provide many macros with different values to make it easier for the user to use the library without knowing all the low-level details.

- `uart_n`: this value selects which of the three UART peripherals you want to use. Possible values are `UART0`, `UART1` and `UART3`.
- `clock_selection`: these values determine which clock frequency the peripheral uses, selected from four different submultiples of the CPU clock (`CCLK`): `UART_CCLK_DIV_4` (CCLK / 4),`UART_CCLK_DIV_1` (CCLK) ,`UART_CCLK_DIV_2` (CCLK / 2), and `UART_CCLK_DIV_8` (CCLK / 8).
- `DLL`, `DLM`, `DivAddVal`, `MulVal`: these four values are used to set the baud rate for the UART connection, as explained in the LPC1768 user manual.
- `fifo_enable`: this value potentially enables the FIFO mode required to use DMA. Its values are `UART_EN_FIFO` and `UART_NO_FIFO`.
- `dma_mode`: this value potentially enables the DMA mode. Its values are `UART_DMA` and `UART_NO_DMA`.
- `trigger_level`: the trigger level is the number of elements that need to be received before an interrupt is raised, and in case the DMA is enabled, it will trigger a DMA request interrupt, resulting in a DMA transfer. Possible values are `UART_TR_LEVEL_1` (trigger level 1), `UART_TR_LEVEL_4` (trigger level 4), `UART_TR_LEVEL_8` (trigger level 8), and `UART_TR_LEVEL_16` (trigger level 16).
- `rdv_int`: this value enables an interrupt when new data is received. Possible values are `UART_RECV_DV_INT` and `UART_NO_INT`;
- `word_size`: This value determines the size of a word in the communication; possible values are `UART_WORD_5` (5), `UART_WORD_6` (6), `UART_WORD_7` (7) and `UART_WORD_8` (8).
- `stop_bit_n`: this value determines the number of stop bits we want, between 1 (`UART_STOP_1` and 2 (`UART_STOP_2`).
- `parity_active`: this element allows to enable (`UART_PARITY`) or disable (`UART_NO_PARITY`) the parity field in the transmission.
- `parity_type`: this determines the type of parity we want, between odd (`UART_ODD_PARITY`) and even (`UART_EVEN_PARITY`).

4.1.2 init_uart

This function takes a pointer to a `uart_parameters` structure as an argument and initializes the relevant registers with the provided values. It is also responsible for powering on the peripheral through the register `PCONP`.

4.2 DMA Library

4.2.1 dma_transfer

```c
struct dma_transfer{
  uint32_t  channel;
  uint32_t  size;
  uint32_t  dest;
  uint32_t  source;
  uint32_t  source_burst_size;
  uint32_t  dest_burst_size;
  uint32_t  source_increment;
  uint32_t  dest_increment;
  uint32_t  source_transfer_width;
  uint32_t  dest_transfer_width;
  uint32_t  terminal_count_enable;
  uint32_t  transfer_type;
  uint32_t  source_peripheral;
  uint32_t  dest_peripheral;
  uint32_t  lli_head;
};
```

Listing 4.2: DMA transfer struct

Like with the UART definition, each value can be either a numeric value or a specific value expressed through a macro.

- `channel`: this value determines which channel we want to use. There are 8 channels available in the GPDMA, numbered from 0 to 7, with the lower numbers having higher priority. The macros `DMA_CHANNEL_x` can fill this field.
- `size`: how many elements we want to transfer.
- `dest`: address of the destination.
- `source`: address of the source.
- `source_burst_size`, `dest_burst_size`: the burst size refers to the number of data items transferred in a single burst during a DMA transfer. It is typically measured in the number of items transferred per DMA transfer request. For example, a burst size of 8 means that 8 data items are transferred per DMA request. The burst size can affect the performance of the DMA transfer and may need to be optimized for the specific application. The values these two fields can assume are all the power of 2 between 1 and 256, using the macros `DMA_BURST_SIZE_x`.
- `source_increment`, `dest_increment`: these values determine if, after each transfer, we want to increment the source and destination address. The possible values are `DMA_NO_INCREMENT` and `DMA_INCREMENT`.
- `source_transfer_width`, `dest_transfer_width`: width of each element of source and destination. Possible values are 8 (byte), 16 (half word), and 32 (word), expressed through the macros `DMA_TRANSFER_WIDTH_8`, `DMA_TRANSFER_WIDTH_16` and `DMA_TRANSFER_WIDTH_32`.
- `terminal_count_enable`: with the value `DMA_TERMINAL_COUNT_ENABLE`, we generate an interrupt at the end of the transaction, otherwise (using `DMA_TERMINAL_COUNT_NOT_ENABLE`) no.

- `transfer_type`: there are 4 kinds of transfer: memory to memory (`DMA_M2M`), memory to peripheral (`DMA_M2P`), peripheral to memory (`DMA_P2M`) and peripheral to peripheral (`DMA_P2P`).
- `source_peripheral`, `dest_peripheral`: these fields determine the source or destination peripheral of the transfer. This field specifies when to initiate a transaction based on the peripheral status. Given many peripherals, we have not defined a macro for each one. Possible values can be found on page 592 of the user manual.
- `lli_head`: if we want to use DMA with the scatter gather technique, we fill this field with the pointer to the head of the linked list (whose structure is defined in `dma_lli_element`).

4.2.2 dma_lli_element

```
struct dma_lli_element{
  uint32_t            src;
  uint32_t            dest;
  struct dma_lli_element*    next_lli;
  uint32_t            ctrl;
};
```

Listing 4.3: DMA transfer struct

This structure is used to perform a scatter gather operation.

4.2.3 init_DMA

This function activates the GPDMA through the register `PCONP` and enables the associated interrupt.

4.2.4 transfer_DMA

This function uses a pointer to `dma_struct` as an argument and initializes a DMA transaction.

4.3 Application

An application based on Micrium OS has been developed to test the libraries and simulate the DMA attack on a real system; in this case, the target board is the LandTiger V2.0 NXP LPC1768 ARM development board.

The simulated scenario of the application is a DMA attack on the control unit charged to monitor the state of the payload system in a spacecraft. The aim is to cause the mission's failure, changing the sensors' parameters to shut down the onboard computer OBC, which cannot operate over specific pressure and temperature values.

4.3.1 Setup

The setup used for the development of the application has been the following:

- LandTiger V2.0 NXP LPC1768 ARM development board

- Keil uVision 5 running on a General Purpose PC
- Putty terminal emulator open source
- Uart to USB cable

4.3.2 Micrium OS

The application is built above the operating system Micrium, which is preemptive, deterministic, royalty-free, and based on a flat architecture.

As already discussed in 2.2, DMA attacks are even easier to perform on this kind of architecture because there is no distinct division between the application and operating system, so accessing reserved memory areas through the application is possible.

Following the Task Model typical of Micrium, few tasks have been created, and relative TCB and Stack have been allocated. These are:

- APP task
- Lock task
- Display task

4.3.3 APP task

The purpose of the application is to communicate with sensors and take actions depending on the data, so it is required to set up UART communication and DMA transfer. This task has the highest priority.

A struct `uart_parameters` has to be defined and instantiated to initialize a UART communication. Here all the parameters are set; in particular, the most important ones are:

- `Clock frequency` : (CPU Clock)/4
- `Baud rate`: 9645 (Calculated using algorithm explained in the user manual)
- `DMA mode`: ON
- `Trigger level`: 1 (as soon as an element arrives, a DMA request is sent)
- `Word size`: 8 bits (1 Byte)

After the initialization of the UART connection, the task is suspended and will be resumed by the `lock_task`.

As soon as `app_task` is resumed by `lock_task`, it initializes the DMA transfer, setting all the required parameters in an endless loop. The key ones are:

- `Destination`: Struct handling the data coming from the sensor
- `Source`: Receiver Buffer Register of UART

- `Burst size Dest/Source`: 1 (Just transfer one data item at a time)
- `Terminal Count Enable`: ON (Generate interrupt at the end of the transfer)
- `Transfer Type`: Peripheral to Memory (UART to memory)

After the parameters are set up, `transfer_DMA` function is called, and the DMA transfer is initialized. Then the task is suspended again.

4.3.4 Lock task

Data from the sensor has a CRC field calculated before sending the data to the board. `Lock task` recalculates the CRC and compares it to the received one to ensure the data was sent correctly.

If the data is accepted, which means the calculated CRC corresponds to the received one, the condition to open the door is checked, updating the related flag. Depending on its final value, the door is opened or closed, and the state is saved.

The `App_task` is resumed, and the `Lock_task` is suspended.

4.3.5 Display task

This task has the lowest priority among the others and is responsible for displaying the data coming from the sensor on the screen.

4.3.6 DMA_IRQHandler

This DMA handler is called when the DMA transfer is completed. The door's flag is reset, and the LCD is cleared.

Depending on the result of the redundancy check, the data might be taken or not. If it is, no action is performed because Lock_task already did it, but if the CRC condition is not satisfied, the door's state still needs to be decided, which is done by restoring the previous state.

Finally, it resumes the `Lock_task`, restarting the flow of the application.

Bibliography

[1] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. Characterizing, exploiting, and detecting dma code injection vulnerabilities in the presence of an iommu. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 395–409, New York, NY, USA, 2021. Association for Computing Machinery.

[2] Ron Stajnrod, Raz Yehuda, and Nezer Zaidenberg. Attacking trustzone on devices lacking memory protection. *Journal of Computer Virology and Hacking Techniques*, 18, 09 2022.

[3] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In Ulrich Flegel, Evangelos Markatos, and William Robertson, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[4] Dimitrios Tychalas, Anastasis Keliris, and Michail Maniatakos. Stealthy information leakage through peripheral exploitation in modern embedded systems. *IEEE Transactions on Device and Materials Reliability*, 20(2):308–318, 2020.