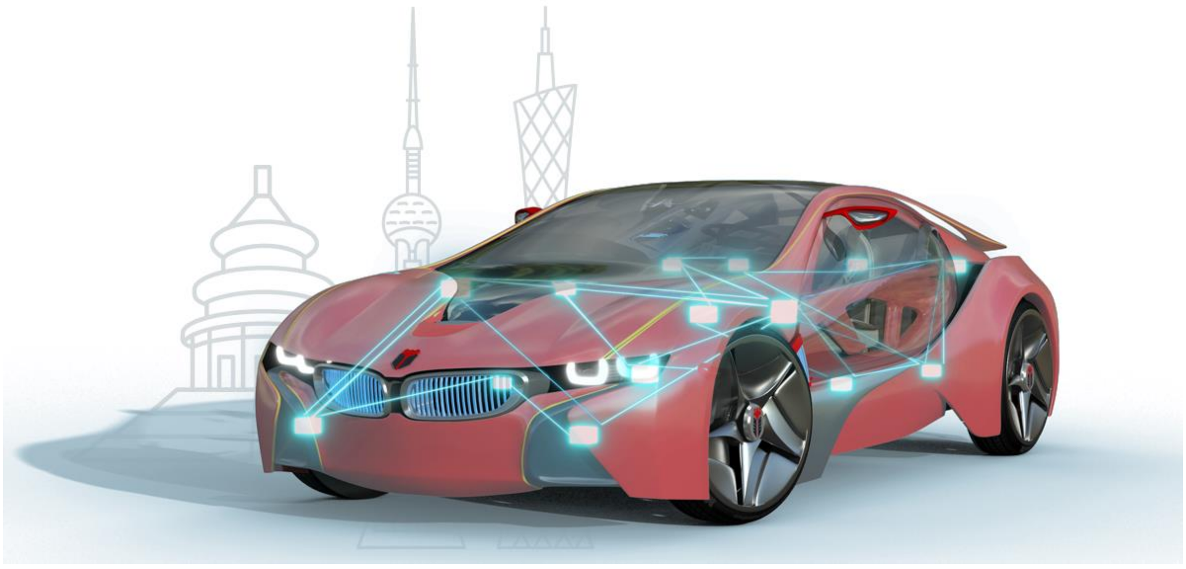
知从青龙队列刷写方案介绍

ZC.QINGLONG PIPELINE PROGRAM SOLUTION INTRODUCTION

知从青龙 FOTA 平台

ZC.QingLong FOTA platform



知从青龙队列刷写方案介绍

ZC.QINGLONG PIPELINE PROGRAM SOLUTION INTRODUCTION

知从青龙 FOTA 平台

ZC.QingLong FOTA platform

1 技术背景 TECHNICAL BACKGROUND

在传统的 Bootloader 刷写流程中，数据下载（Download）与数据编程（Program）是严格的串行执行模式：Bootloader 必须等待一整块数据（例如一个完整的 Flash 扇区数据）通过 CAN、以太网等总线完全接收并校验无误后，才能启动对 Flash 存储器的擦除和写入操作。在此期间，高速的数据总线在等待相对低速的 Flash 操作完成时处于空闲状态，而 Flash 存储器在等待下一块数据接收时也处于闲置状态，这种“等待-忙碌”交替的模式造成了硬件资源利用率低下，整体刷写时间被显著拉长，成为了提升效率的主要瓶颈。

In the traditional bootloader flashing process, data download and data programming follow a strictly serial execution model: the bootloader must wait for an entire data block (e.g., a complete Flash sector) to be fully received and verified via buses like CAN or Ethernet before initiating erase and write operations on the Flash memory. During this period, the high-speed data bus remains idle while waiting for the relatively slow Flash operation to complete, and the Flash memory also sits idle awaiting the next data block. This alternating “wait-busy” pattern results in low hardware resource utilization and significantly extends the overall programming time, becoming a major bottleneck for efficiency improvement.

随着 ECU 功能日益复杂，其程序存储器的容量从几百 KB 激增至数 MB 甚至数十 MB，这种串行模式下的耗时问题在生产线下线和售后升级场景中变得愈发突出。为了最大限度地压榨硬件性能、缩短整体刷写时间，“队列刷写” 技术应运而生。其核心背景思想是引入一个数据缓冲区队列（通常是双缓冲区或环形缓冲区），将线性的串行任务重构为并行的队列作业。具体而言，Bootloader 将整个数据流划分为多个连续的数据块（Block）。当第一个数据块接收完成后，并不等待全部数据，而是立即启动对该块的 Flash 编程操作；与此同时，第二个数据块的后台接收过程利用总线带宽，与前一个数据块的编程操作并行进行。如此循环往复，形成了“接收块 N+1”与“编程块 N”在时间轴上的重叠，从而将原本被浪费的总线空闲时间和

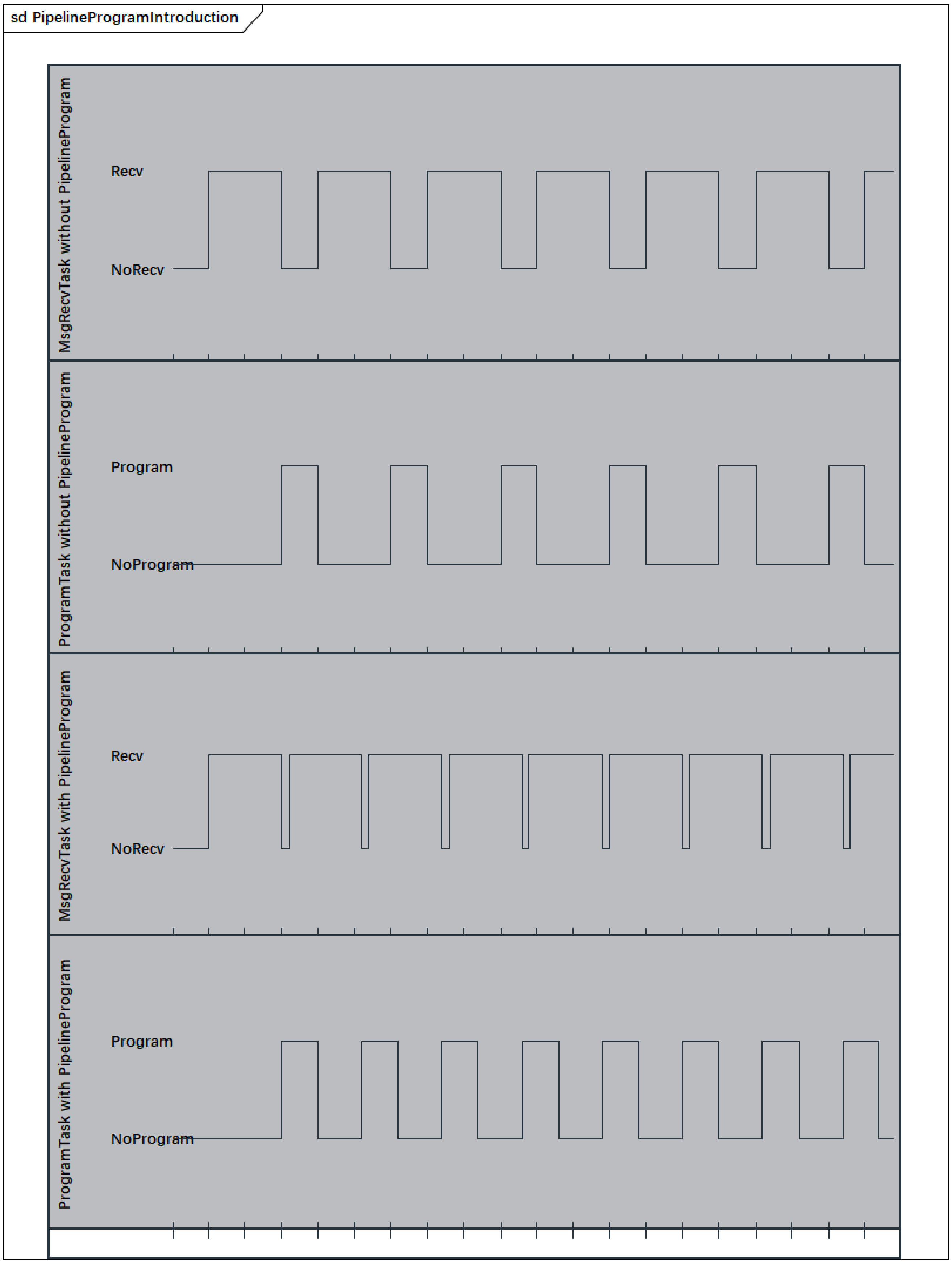
Flash 等待时间有效利用起来，使得数据总线的传输时间和 Flash 的编程时间得以部分抵消，最终实现了对单一控制器刷写过程的速度优化，这在处理大容量 Flash 或使用相对低速通信接口时带来的时间收益尤为显著。这项技术是嵌入式 Bootloader 设计中一种经典的、以空间（缓冲区）换时间（刷写耗时）的优化策略。

As ECU functionality grows increasingly complex, program memory capacity has surged from hundreds of kilobytes to several megabytes or even tens of megabytes. This time-consuming serial approach has become increasingly problematic in production line-off and aftermarket upgrade scenarios. To maximize hardware performance and minimize overall programming time, “pipeline programming” technology emerged. Its core concept involves introducing a data buffer queue (typically a double-ended buffer or circular buffer) to transform linear serial tasks into parallel pipeline operations. Specifically, the Bootloader divides the entire data stream into multiple contiguous data blocks. Upon completing reception of the first block, it immediately initiates Flash programming for that block without waiting for all data. Simultaneously, the background reception process for the second block utilizes bus bandwidth to run in parallel with the programming operation of the previous block. This iterative process creates temporal overlap between “receiving block N+1” and “programming block N.” Consequently, the previously wasted bus idle time and Flash wait time are effectively utilized, partially offsetting the data bus transfer time and Flash programming time. Ultimately, this achieves speed optimization for the single-controller flashing process, yielding particularly significant time savings when handling large-capacity Flash or using relatively low-speed communication interfaces. This technique represents a classic optimization strategy in embedded bootloader design, trading space (buffer) for time (programming duration).

2 方案介绍 SOLUTION INTRODUCTION

队列刷写是一种针对单一 ECU 控制器的高效固件更新方案。其核心思想是打破传统刷写中数据传输与 Flash 编程的串行依赖，通过引入数据缓冲队列和任务重叠执行机制，将二者并行化，从而显著压缩整体刷写时间。该方案本质上是计算机体系结构中“队列”概念在嵌入式刷写领域的具体应用，通过“以空间换时间”（牺牲少量 RAM 作为缓冲区）来最大化利用总线带宽与 Flash 编程时间。下图为不使用队列刷写以及使用队列刷写的时序差异图：

Queue-based flashing constitutes an efficient firmware update solution for individual ECU controllers. Its core principle lies in breaking the serial dependency between data transmission and Flash programming inherent in conventional flashing methods. By introducing data buffering queues and task overlapping execution mechanisms, both processes are parallelised, thereby significantly reducing overall flashing duration. This approach essentially applies the “queue” concept from computer architecture to the embedded programming domain. It maximises utilisation of bus bandwidth and Flash programming time by trading space for time (sacrificing a small amount of RAM as buffer space). The diagram below illustrates the timing difference between non-queued and queued programming:


sd PipelineProgramIntroduction
MsgRecvTask without PipelineProgram
Recv
NoRecv
ProgramTask without PipelineProgram
Program
NoProgram
MsgRecvTask with PipelineProgram
Recv
NoRecv
ProgramTask with PipelineProgram
Program
NoProgram

流水线刷写方案的实施始于精细的初始化配置与准备工作。在服务器端，完整的固件镜像被预先划分为一系列连续的、大小固定的数据块，块大小的设定需综合考量 Flash 扇区的物理特性、通信协议的单帧负载上限以及目标 ECU 内可用 RAM 资源，以达成传输效率与内存开销的最佳平衡。与此同时，在 ECU 的 Bootloader 中，需要开辟至少两个独立的 RAM 缓冲区，即经典的双缓冲结构，并设计一个明确的状态机来管理整个刷写生命周期。通信协议也需进行相应扩展，在标准数据下载服务的基础上，增加用于协调发送与接收节奏的流控制机制。

The implementation of the pipeline flashing solution begins with meticulous initial configuration and preparatory work. On the server side, the complete firmware image is pre-partitioned into a series of contiguous, fixed-size data blocks. The block size must be determined by comprehensively considering the physical characteristics of Flash sectors, the single-frame payload limit of the communication protocol, and the available RAM resources within the target ECU, to achieve an optimal balance between transmission efficiency and memory overhead. Simultaneously, within the ECU's Bootloader, at least two independent RAM buffers must be allocated—forming a classic dual-buffer structure—alongside a dedicated state machine to manage the entire flashing lifecycle. The communication protocol requires corresponding enhancements, adding a flow control mechanism to coordinate transmission and reception rhythms on top of standard data download services.

具体刷写流程启动后，主机首先传输第一个数据块，Bootloader 将其数据存入缓冲区 A 并进行实时校验。一旦缓冲区 A 被有效数据填满且校验通过，整个方案的核心并行逻辑便立刻激活：Bootloader 在向主机发送确认并请求下一数据块的同时，几乎同步地启动将缓冲区 A 内数据编程至 Flash 存储器的操作。从此刻起，系统进入高效的流水线作业状态：当主机正在传输第二个数据块至缓冲区 B 时，Flash 驱动器正独立地处理缓冲区 A 的数据编程；待缓冲区 B 接收完毕且缓冲区 A 编程完成时，二者角色瞬间切换——缓冲区 B 转交给 Flash 进行编程，而刚刚腾空的缓冲区 A 则准备接收第三个数据块。如此往复，数据的接收与 Flash 的编程在两个缓冲区之间“乒乓”切换，形成了稳定的接力循环，使得通信总线与 Flash 存储器得以长时间并行工作，资源利用率达到最大化。

Upon initiating the specific flashing process, the host first transmits the initial data block. The Bootloader stores this data in Buffer A and performs real-time verification. Once Buffer A is filled with valid data and verification succeeds, the core parallel logic of the entire solution activates immediately: The Bootloader sends an acknowledgment to the host and requests the next data block while simultaneously initiating the programming of Buffer A's data into Flash memory. From this point, the system enters a highly efficient pipeline operation: while the host transmits the second data block to Buffer B, the Flash driver independently processes the programming of

Buffer A's data. Upon Buffer B's reception completion and Buffer A's programming finish, their roles instantly switch—Buffer B is handed over to Flash for programming, while the newly freed Buffer A prepares to receive the third data block. This cycle repeats, with data reception and Flash programming “ping-ponging” between the two buffers to form a stable relay loop. This enables the communication bus and Flash memory to operate in parallel for extended periods, maximizing resource utilization.

流程的收尾阶段需要特别处理。当接收到最后一个数据块后，流水线将完成其最后一次“接收-编程”的交替。随后，Bootloader 不再请求新数据，但必须等待最后一块数据的编程操作彻底完成。此后，系统执行最终的全局验证，例如计算整个已刷写区域的 CRC 校验和，并与主机端提供的权威摘要进行比对，以此确保整个固件镜像的完整性与正确性。只有通过此终极验证，Bootloader 才会更新版本信息等元数据，标志着此次流水线刷写任务安全、成功地结束。

The final phase requires special handling. Upon receiving the last data block, the pipeline completes its final “receive-program” cycle. The Bootloader then ceases requesting new data but must wait for the programming of this final block to fully conclude. Following this, the system performs a final global verification. This includes calculating the CRC checksum for the entire programmed region and comparing it against the authoritative digest provided by the host. This ensures the integrity and correctness of the entire firmware image. Only upon passing this ultimate verification does the Bootloader update metadata such as version information, signifying the safe and successful completion of the pipeline programming task.

3 方案运行注意事项 PRECAUTIONS FOR PLAN IMPLEMENTATION

由于软硬件差异，不同环境下使用队列刷写方案可能会出现不同的使用现象，部分使用场景可能导致队列刷写失败或者无法达到预期减少刷写时间的目的，以下为使用队列刷写方案可能遇到的问题点以及推荐的解决方案：

Due to variations in hardware and software configurations, employing queue-based write optimisation schemes may yield differing outcomes across environments. Certain usage scenarios may result in failed queue writes or failure to achieve the intended reduction in write latency. The following outlines potential issues encountered when implementing queue-based write optimisation, alongside recommended solutions:

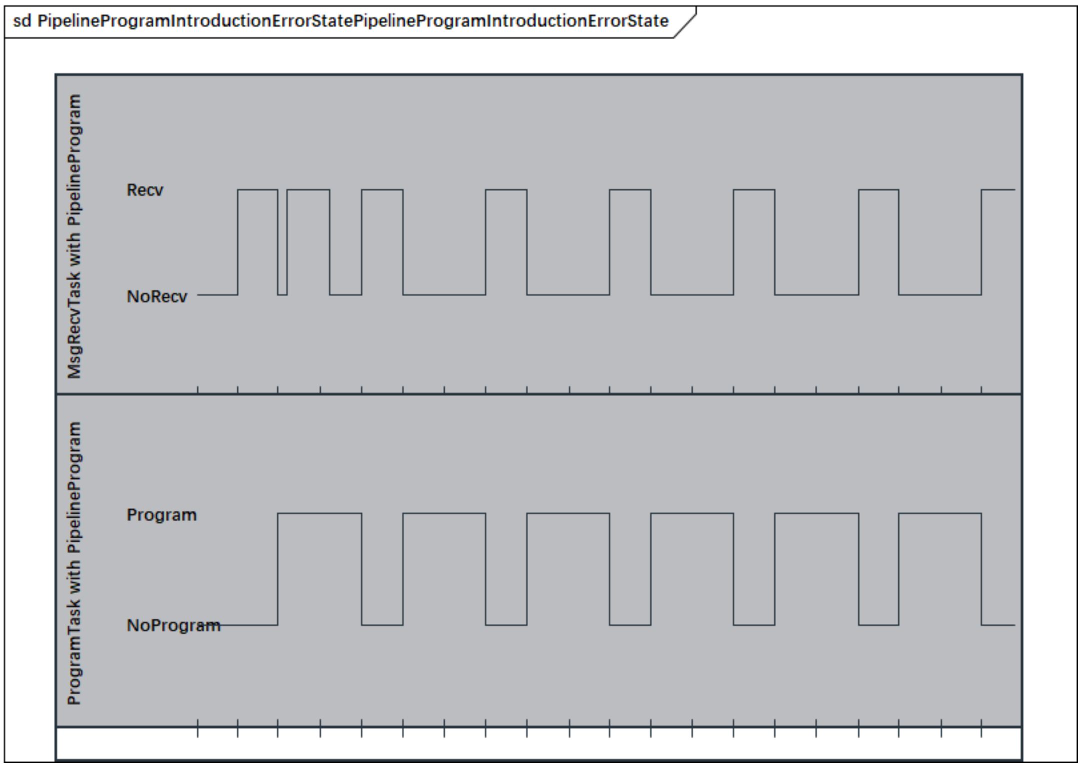
- 队列停滞：

队列刷写方案中期望数据刷写时间是小于期望数据传输时间的，但是在实际使用场景下，数据刷写时间可能出现远大于数据传输时间的情况，这会导致缓冲区很快被填满，主机必须等待，并行效率低下。推荐的解决方案如下：

1. 优化 Flash 驱动：使用更快的刷写模式。

2. 增大缓冲区：允许接收更多数据块以覆盖编程延迟。

3. 调整块大小：减小块大小，使编程任务更快完成，更频繁地释放缓冲区。


sd PipelineProgramIntroductionErrorStatePipelineProgramIntroductionErrorState
MsgRecvTask with PipelineProgram
Recv
NoRecv
ProgramTask with PipelineProgram
Program
NoProgram

- Queue Stagnation:

In queue-based flash programming schemes, the expected data write time is typically shorter than the expected data transfer time. However, in practical scenarios, the data write time may significantly exceed the data transfer time. This can rapidly fill the buffer, forcing the host to wait and resulting in poor parallel efficiency. Recommended solutions are as follows:

1. Optimise the Flash driver: Employ a faster write mode.

2. Increase buffer size: Allow reception of more data blocks to compensate for programming delays.

3. Adjust block size: Reduce block size to accelerate programming tasks and enable more frequent buffer clearance.

- 数据不同步

队列刷写过程涉及多组数据处理，可能由于缓冲区切换逻辑错误或标志位竞争导致数据被覆盖或编程了错误的数据，最终导致刷写文件校验失败。推荐的解决方案如下：

1. 严格的临界区保护：使用关中断或互斥锁。

2. 状态机验证：在切换前进行多重条件检查。

3. 增加序列号：为每个数据块附加序列号，编程前进行核对。

- Data asynchrony

The queue flashing process involves multiple data handling operations. Errors in buffer switching logic or flag contention may cause data overwriting or incorrect programming, ultimately resulting in failed file verification. Recommended solutions are as follows:

1. Strict critical section protection: Employ interrupt disabling or mutual exclusion locks.

2. State machine validation: Implement multi-condition checks prior to switching.

3. Sequence number incrementation: Append a sequence number to each data block and verify it prior to programming.