分　类　号：U463.61　　单位代码：10183
研究生学号：2018424049　　密　　级：公开

吉 林 大 学

硕士学位论文

(专业学位)

基于多片多核处理器的自动驾驶控制软硬件架构实现研究

Research on the Realization of Automatic Driving Control Software and Hardware Architecture Based on Multiple Multi-core Processors

作 者 姓 名：　李明
类　　别：　工程硕士
领域（方向）：　车辆工程
指 导 教 师：　张建伟 副教授
培 养 单 位：　汽车工程学院

2021 年 5 月

基于多片多核处理器的自动驾驶控制软硬件架构实现研究

Research on the Realization of Automatic Driving Control Software and Hardware Architecture Based on Multiple Multi-core Processors

作 者 姓 名：李明

领域（方向）：车辆工程

指 导 教 师：张建伟 副教授

类　　别：工程硕士

答 辩 日 期：2021 年 5 月 23 日

吉林大学硕士学位论文原创性声明

本人郑重声明：所呈交学位论文，是本人在指导教师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：李明

日期：2021 年 6 月 3 日

摘　要

基于多片多核处理器的自动驾驶控制软硬件架构实现研究

随着自动驾驶的不断发展，对自动驾驶控制器的算力提出了越来越高的要求，传统的单核处理器已经无法满足自动驾驶的算力需求，而多核处理器开始得到更多的应用。另外，功能安全也是自动驾驶需要解决的一个重要问题，为了提高自动驾驶控制器的功能安全，冗余技术开始逐渐应用于自动驾驶控制器的设计上。因此，本文对多核处理器在自动驾驶控制中的应用以及自动驾驶控制器的冗余设计方法进行研究，提出了基于多核处理器的双冗余自动驾驶控制器硬件架构，并基于该硬件架构进行了自动驾驶控制软件架构的设计。

本文首先对自动驾驶控制器进行了硬件设计。针对控制器的算力和功能安全需求，选择了能够达到最高功能安全等级 ASIL-D 的多核处理器 TC297，并在此基础上对控制器进行了双系统冗余设计。两个系统之间通过 HSSL、SPI、CAN 和 ERU 进行通信，一方面使两系统可以相互进行故障检测以实现冗余功能，另一方面使两个处理器能够协同运行，从而进一步提高控制器的算力。在硬件架构设计的基础上，进行了控制器的电路设计，包括电路原理图的设计和印制电路板的设计，并通过 PCB 的加工和元器件焊接，完成了自动驾驶控制器的实物设计，从而为自动驾驶控制软件架构的开发提供了硬件基础。

针对自动驾驶控制器的系统启动和应用程序升级需求，设计了系统基础软件 Bootloader。为了使控制器正常启动，设计了系统启动程序，用于对处理器的时钟系统和内存等进行设置，以建立正确的应用程序运行环境，并对各内核的寄存器、堆栈及 Cache 等进行设置，以保证多核能够正常启动和运行；为了解决应用程序可执行文件的下载、格式转换及烧写等问题，设计了应用程序可执行文件的下载程序；为了使 Bootloader 在完成系统启动和应用程序升级工作后顺利跳转到应用程序入口并开始运行应用程序，采用相关的指令设计了跳转程序；为了使 Bootloader 合理地使用程序存储器空间，进行了程序存储器的空间分配方案设计。

针对自动驾驶任务的实时运行需求和多核协同运行需求，进行了多核处理器基础软件的设计。为了使自动驾驶任务的运行具有较高的实时性，选择了非对称多处理作

为多核处理器的运行模式，并在处理器上移植了实时操作系统 FreeRTOS；为了满足多核协同运行时的核间任务同步需求，采用核间中断设计了核间任务同步机制；为了解决多核协同运行时由于同时访问共享资源而发生相互冲突的问题，采用处理器的专用硬件指令设计了核间共享资源互斥访问机制；为了满足多核协同运行时的数据通信需求，采用共享内存的方式，结合核间任务同步机制设计了核间通信机制。

针对自动驾驶控制器的功能安全需求，在双冗余控制器硬件平台的基础上，进行了双冗余系统软件设计。为了使控制器中互为冗余的两个系统在启动后能够获取自身及对方的工作状态信息，并使两系统进入正确的运行状态，设计了系统状态信息管理程序；为了使两系统同步运行以避免故障误判，设计了系统间的同步协议和相应的同步程序；为了保证系统出现故障时能够快速的检测到故障的发生并进行正确的处理，设计了系统之间和系统内部的系统故障检测及处理程序。

为了验证软硬件设计的正确性，分别针对 Bootloader、多核处理器基础软件和双冗余系统软件的各功能模块设计了不同的测试用例并进行了测试。根据相关测试结果，验证了本文所设计的自动驾驶控制器软硬件的正确性。

关键词：

自动驾驶控制器，多核处理器，冗余系统，软件架构

ABSTRACT

Research on the Realization of Automatic Driving Control Software and Hardware Architecture Based on Multiple Multi-core Processors

With the continuous development of automatic driving, higher and higher requirements have been placed on the computing power of the automatic driving controller. Traditional single-core processors have been unable to meet the computing power requirements of automatic driving, and multi-core processors have begun to get more and more application. In addition, functional safety is also an important issue that needs to be solved in automatic driving. In order to improve the functional safety of automatic driving controller, redundancy technology has begun to be gradually applied to the design of automatic driving controllers. Therefore, this paper studies the application of multi-core processors in automatic driving control and the redundant design method of automatic driving controllers, and proposes a dual-redundant automatic driving controller hardware architecture based on multi-core processors, and based on this hardware architecture, designed the automatic driving control software architecture.

This paper firstly carried out the hardware design of the automatic driving controller. In view of the controller's computing power and functional safety requirements, the multi-core processor TC297, which can reach the highest functional safety level ASIL-D, was selected, and on this basis, the controller was designed with dual-system redundancy. The two systems communicate through HSSL, SPI, CAN, and ERU. On the one hand, the two systems can perform fault detection with each other to achieve redundant functions. On the other hand, the two processors can work together to further improve the controller’s computing power. On the basis of the hardware architecture design, the circuit design of the controller was carried out, including the design of the circuit schematic diagram and the design of the printed circuit board, and the physical design of the automatic driving controller was completed through PCB processing and component welding, so as to provide a hardware foundation for the development of automatic driving control software architecture.

In response to the system startup and application upgrade requirements of the automatic driving controller, the system basic software Bootloader is designed. In order to make the

controller start normally, a system startup program is designed to set the clock system and memory of the processor to establish the correct application program operating environment, and to set the registers, stacks and cache of each core. In order to ensure that the multi-core can start and run normally; in order to solve the problems of application executable file download, format conversion and programming, the application executable file download program is designed; in order to enable the Bootloader to complete the system startup and application upgrade work After smoothly jumping to the application program entrance and starting to run the application program, the jump program was designed with related instructions; in order to make the Bootloader use the program memory space reasonably, the program memory space allocation scheme was designed.

In response to the real-time operation requirements of automatic driving tasks and the requirements of multi-core cooperative operation, the basic software of multi-core processors is designed. In order to make the operation of automatic driving tasks have high real-time performance, asymmetric multi-processing is selected as the operating mode of the multi-core processor, and the real-time operating system FreeRTOS is transplanted on the processor; in order to meet the inter-core tasks during multi-core cooperative operation For synchronization requirements, the inter-core task synchronization mechanism is designed by using the inter-core interrupt; in order to solve the problem of mutual conflicts due to simultaneous access to shared resources during multi-core cooperative operation, the special hardware instructions of the processor are used to design the mutual exclusive access mechanism of shared resources between cores; In order to meet the data communication requirements of multi-core cooperative operation, the method of shared memory is adopted, and the inter-core communication mechanism is designed in combination with the inter-core task synchronization mechanism.

Aiming at the functional safety requirements of the automatic driving controller, the dual-redundant system software design is carried out on the basis of the dual-redundant controller hardware platform. In order to enable the two redundant systems in the controller to obtain the working status information of themselves and each other after startup, and to make the two systems enter the correct operating state, the system status information management program is designed; in order to make the two systems run synchronously In order to avoid misjudgment of faults, a synchronization protocol and corresponding synchronization procedures between systems are designed; in order to ensure that the occurrence of the fault can be quickly detected and handled correctly when the system fails,

the system faults between and within the system are designed Testing and processing procedures.

In order to verify the correctness of the software and hardware design, different test cases were designed and tested for each functional module of Bootloader, multi-core processor basic software and dual-redundant system software. According to the relevant test results, the correctness of the software and hardware of the automatic driving controller designed in this paper is verified.

Key words:

automatic driving controller, multi-core processor, redundant system, software architecture

目　录

第 1 章 绪论

1.1 课题背景及研究意义

随着汽车电子控制技术的不断发展，汽车越来越倾向于智能化，自动驾驶已经成为当前汽车发展的主要方向[1]。为了实现自动驾驶，通常需要大量的传感数据来识别车辆的周围环境，如雷达数据和摄像头数据，另外还需要惯性导航数据和高精地图数据等判断车辆的位置和运动状态[2]，并且为了提高感知信息的准确性，需要将各种传感数据进行融合[3]，因此在传感数据处理方面，要求自动驾驶控制器的处理器具有较强的算力。在规划和控制方面，为了获得良较好的规划和控制效果，通常需要运行较为复杂的规划和控制算法[4]，因此也对控制器的处理器性能提出了更高的要求。

在多核处理器出现之前，通常采用提高主频的方法来提高单核处理器的性能。但随着主频的不断提高，芯片上晶体管的集成度已经接近上限，同时，随着芯片上晶体管集成度的提高，也导致处理器功耗不断上升，以致于普通的冷却方式无法有效地对处理器进行散热，因此将主频提高到一定程度后，无法继续通过这种方式来提升单核处理器的性能[5]，为了解决上述问题，多核处理器应运而生。多核处理器通过在单芯片上集成多个内核的方式，实现了在不提高主频的情况下提升性能，并且相对于提高主频的方式，多核处理器能够在功耗相同的情况下获得更高的性能[6]。因此，在自动驾驶对处理器算力要求不断提高的现状下，对多核处理器在自动驾驶中应用的研究具有重要意义。

另外，随着自动驾驶的不断发展，车辆的控制任务越来越多地由自动驾驶控制器来完成，为了保证乘员和车辆的安全，对控制器的功能安全提出了更高的要求。L3 级以上的自动驾驶通常要求各功能模块提供冗余，以实现功能安全目标。自动驾驶控制器是自动驾驶中的关键部分，因此对其冗余设计的研究具有重要意义。

本课题来源于实验室与国内某车企合作的 L3-级自动驾驶开发项目。在研究内容上，主要针对自动驾驶在算力和功能安全方面的需求，对多核处理器在自动驾驶控制中的应用及自动驾驶控制器的冗余设计方法进行研究，提出了基于多核处理器的双冗余自动驾驶控制器硬件架构设计方案，通过两片多核处理器解决算力需求问题，同时

以两片处理器为基础构成双冗余系统，实现 Fail-Operational 的功能安全目标。该设计方案通过两片多核处理器同时解决了控制器的算力和功能安全问题，降低了控制器的硬件开发成本，为自动驾驶控制器的设计提供了可行方案，具有一定的创新意义和较强的工程应用价值。

1.2 多核处理器的发展

在多核处理器出现之前，为提高处理器的性能，通常采用两种方法：（1）通过改进制造工艺，提高处理器内核的主频；（2）通过增加内核每个周期的指令执行数目，提高指令执行效率[7]。但是，这两种方法都存在各自的问题。

第一种方法根据的是摩尔定律，即“大约每隔 18 个月，芯片上可容纳的晶体管数目便会增加一倍，性能也将提升一倍[8]。”通过第一种方法，可以使处理器芯片上的电路集成度更高，从而使处理器具有更高的时钟频率，但电路集成度和时钟频率的提高所带来的问题是处理器的功耗越来越高[9]。图 1.1 所示为处理器时钟频率、功耗及性能的发展趋[10]，从图中可以看出，在 2005 年之前，随着时钟频率的提高，处理器的性能也随之提高，但到了 2005 年左右，处理器时钟频率几乎不再提高，原因是当时 AMD 和 Intel 等处理器厂商将处理器时钟频率提高到 4GHz 后，发现处理器功耗超过了 100W，由此引起了严重的处理器发热问题，以致于普通的计算机冷却系统难以进行有效的散热。因此，不能再简单地通过提高时钟频率来获得性能的提升；在第二种方法中，通常使用指令流水线、超标量结构、超长指令字和超线程等技术来提高处理器的指令执行效率，但由于单核处理器的指令执行能力有限，其性能并不能获得显著的提高[11]。由于单核处理器的性能已经达到了瓶颈，人们开始探索新的提高处理器性能的方法，于是将目光投向了多核处理器[12]。

虽然内核设计的复杂度已经难以提高，但随着技术的不断发展，单个处理器上可以集成更多的晶体管，使得单个处理器上可以集成更多的内核，从而可以通过多核的并行运行来提高整个处理器的性能。于是人们开始通过在单处理器上集成更多简单内核来提高处理器性能，而不再进行功耗巨大的复杂内核的设计。图 1.2 所示为处理器晶体管数量及核数的发展趋势，可以看到 2005 年后单个处理器上集成的内核数量开始迅速增长，处理器逐步进入多核时代。

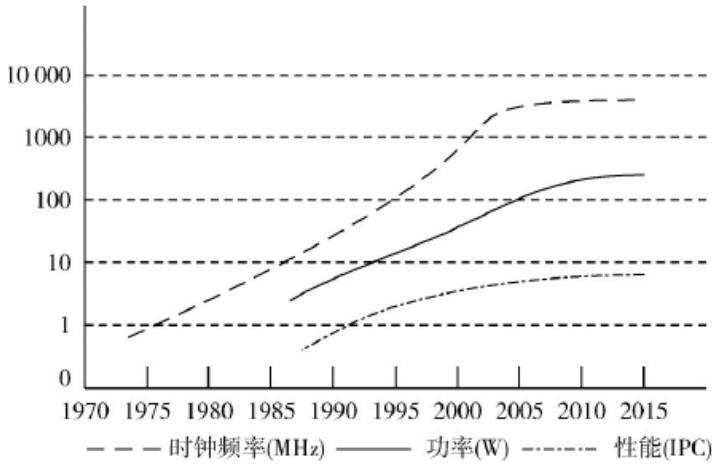


图 1.1 处理器时钟频率、功率及性能的发展趋势

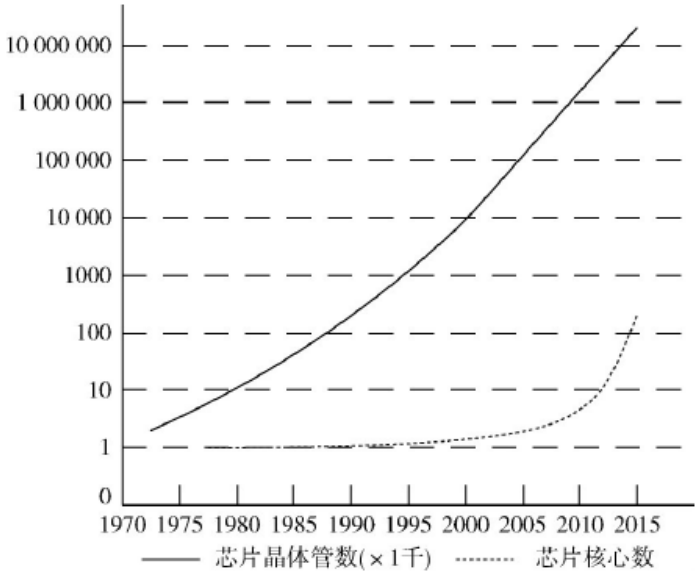


图 1.2 处理器晶体管数量及芯片核心数量的发展趋势

多核处理器也可以叫做单芯片多处理器或片上多处理器（Chip Multi-Processor，CMP）。多核处理器通过集成多个内核，使得整个处理器并行执行的线程数成倍提高，从而使其性能也获得了极大的提高。除了性能上的提升，多核处理器还兼具如下优点：（1）多核处理器的功耗随着器件密度的减小而降低；（2）多核处理器在设计上结构简单，只需要在单核处理器的基础上增加内核数量并扩展相应的总线，因此具有良好的扩展性；（3）由于多个内核集成在同一芯片上，核与核之间的通信效率较高，从而保证了多核线程之间的高并行度；（4）多个内核可以有效地共享片上资源，从而使资源的利用率得到了提高[13][14]。多核处理器的这些优势使其逐渐取代单核处理器成为主流。

多核处理器相比于传统的单核处理器，在整体结构设计上具有很好的灵活性，各处理器厂商可以根据各自的实际应用场景设计出结构完全不同的多核处理器。但根据

处理器上集成的各个内核的架构是否相同，可大致将多核处理器分为同构多核处理器与异构多核处理器两种[15][16]。同构多核处理器各个内核具有相同的架构，且每个内核的功能完全相同，典型的同构多核处理器有斯坦福大学研发的 4 核处理器 Hydra 和麻省理工学院研发的 16 核处理器 RAW；异构多核处理器通常由通用内核和专用内核组成，其中通用内核为传统的处理器内核，而专用内核通常为用于某些特定领域的内核，如用于数字信号处理的 DSP 内核、用于图像处理的 GPU 内核以及近年来出现的用于人工智能的 AI 内核等。通过将不同架构和功能的内核进行组合，能够使异构多核处理器实现性能的最优化组合，同时有效地降低功耗。典型的异构多核处理器有 IBM、索尼和东芝共同研发的 Cell Broad-Band Engine[17]以及 ARM 研发的 big.LITTLE[18]。

Hydra 处理器是由美国斯坦福大学研发的一种多核处理器，也是首个研制成功的多核处理器，其组成结构如图 1.3 所示[19][20]。该处理器集成了 4 个架构相同的通用百万指令级内核，每个内核具有私有的一级指令缓存、数据缓存和内存控制器，各个内核之间通过总线共享片上二级缓存、主存储器接口和 I/O 总线接口，因此可以采用总线共享缓存的方式进行核间通信。由于 Hydra 处理器的 4 个内核均通过总线对片上资源进行访问，因此，为了防止多核同时访问资源时发生总线冲突，在各个内核之间还配置了中央总线仲裁器[21]。

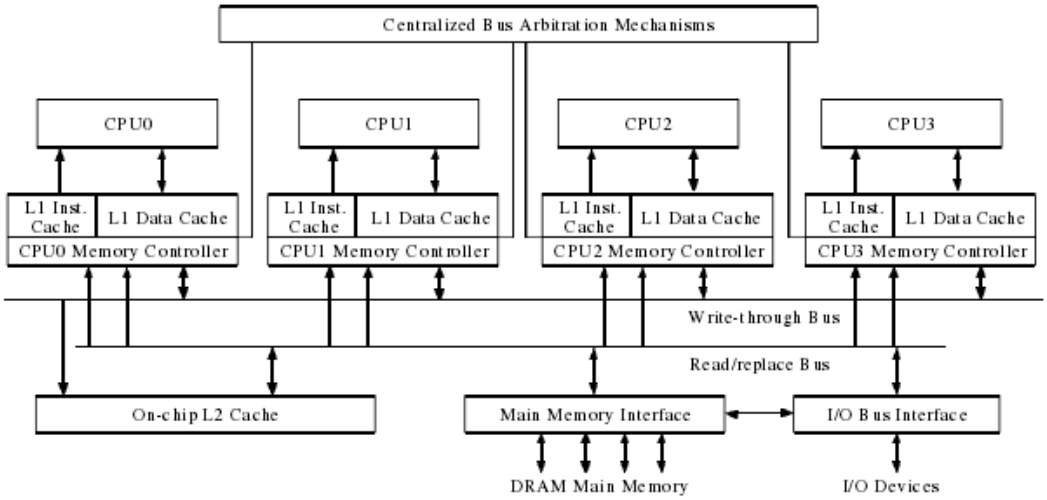


图 1.3 Hydra 处理器结构

RAW 处理器是美国麻省理学院研发的一种同构多核处理器，具有结构简单、功耗低和可扩展性强的特点，其多核之间的连接采用了一种新型的片上网络互连的方式，具体结构如图 1.4 所示[22]。整个处理器由 16 个 Tile 内核组成，且各个内核之间通过可编程网络相互连接，用户可以通过编程改变内核之间的互连结构。通过片上网络中的

同步网络端口，可以进行高效的核间通信，其通信速度能够接近寄存器的访问速度，内核对外部存储器资源和外部 I/O 接口的访问也通过片上网络进行[23][24]。

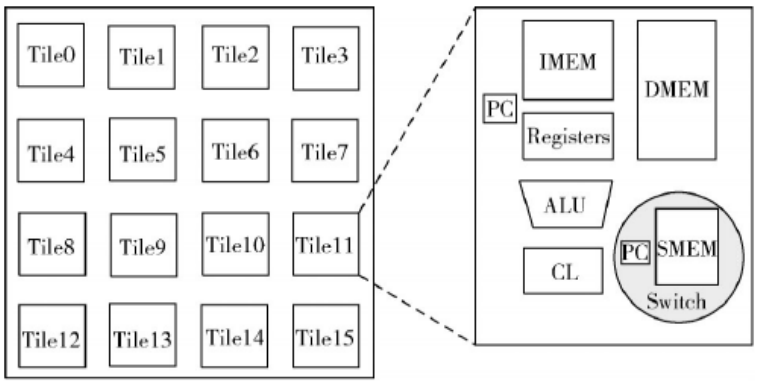


图 1.4 RAW 处理器结构

Cell 处理器是 IBM 与索尼、东芝合作开发的一种多核处理器，主要是为了能够满足新一代宽带多媒体与图形应用的高效率、低功耗的需求，该处理器已成功应用于微软 XBOX 和索尼 PS3 产品中，其组成结构如图 1.5 所示[25]。Cell 处理器由 9 个内核、一个内存控制器和一个总线接口控制器组成，并通过片上总线将各部件相互连接。该处理器属于异构多核处理器，9 个内核中包括 1 个通用处理单元（Power Processing Element，PPE）和 8 个协处理单元（Synergistic Processing Element，SPE），其中 PPE 为 64 位 PowerPC 架构的双线程 RISC 处理单元，时钟频率高达 3.2GHz。PPE 属于通用内核，主要用于控制和处理，通常运行操作系统，负责系统资源的管理和 SPE 核的任务分配；SPE 是 32 位的基于 RISC 的 SIMD 处理单元，主要用于处理计算密集型任务。Cell 处理器内部采用统一编址，并通过内部总线进行核间通信[26]。

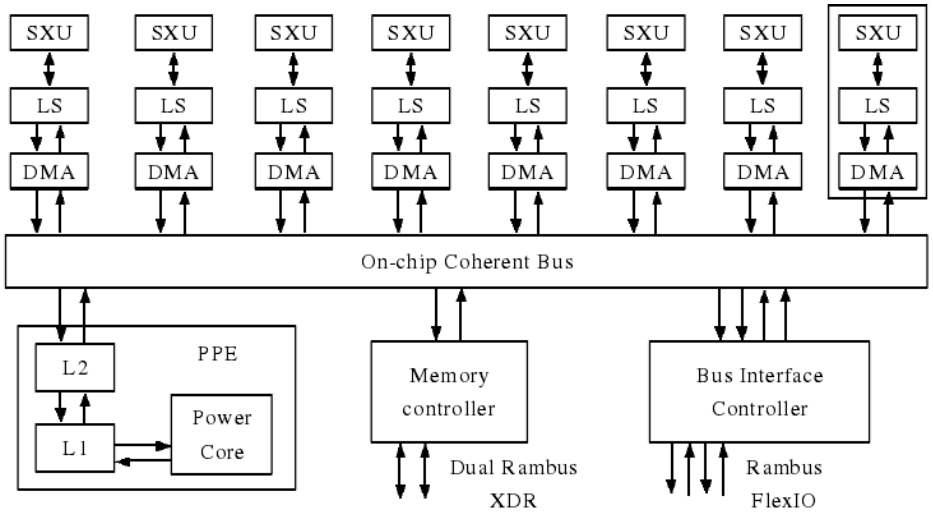


图 1.5 Cell 处理器结构

big.LITTLE 架构是 ARM 提出的一种异构多核处理器架构，其结构如图 1.6 所示。在该架构中，处理器由指令集相同但性能不同的 2 种内核组成，其中高性能内核采用大型超标量内核 Cortex-A15，而性能较低的内核采用高能效的小型内核 Cortex-A7。每种内核具有各自的 L2 级缓存，并分别通过各自的总线接口与内存设备和 I/O 等片上资源进行连接。通过两种内核的结合使用，能够实现在功耗和性能之间进行平衡，当处理器负载较高时，可以单独使用高性能内核或同时开启两个内核，以满足性能需求；而当处理器负载较小时，可以只使用较低性能的内核，既能够满足性能需求，又能够降低处理器的功耗[27]。

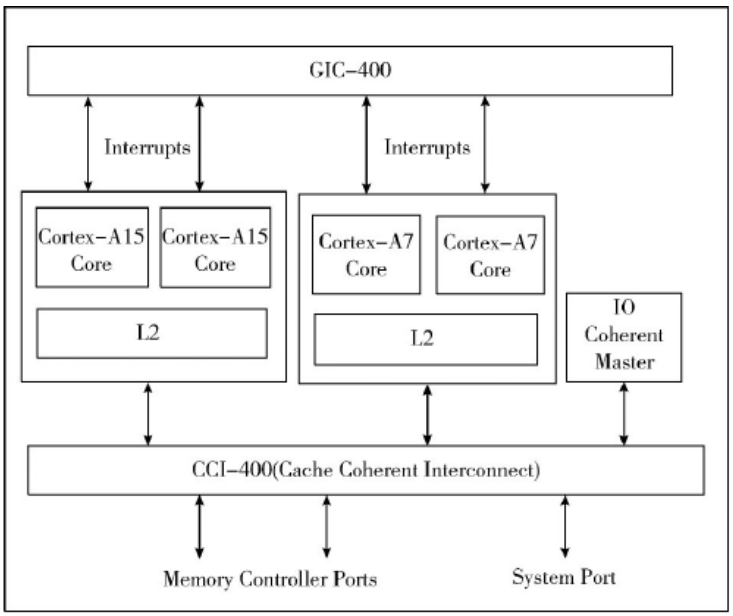


图 1.6 big.LITTLE 架构

1.3 多核关键技术研究现状

1.3.1 核间通信的研究

多核处理器的各个内核通常不是各自独立运行的，而是通过多核的协同运行来发挥每个内核的性能，从而进行大量数据的处理或复杂算法的运行等。各内核之间通常需要进行数据交换，因此核间通信是多核处理器的一项关键技术，核间通信的效率会直接影响到处理器的整体性能输出[28]。多核处理器在核间通信上的结构设计主要有三种：共享缓存总线结构、共享总线结构和片上网络结构[29][30]。

（1）共享缓存总线结构。该结构是最早使用的一种片上通信结构，在该结构中，

每个内核将各自的二级或三级缓存通过总线进行共享，或每个内核通过总线共享同一个二级或三级缓存，如图 1.7(a)和(b)所示。通过这两种方式可以实现缓存数据的共享，从而实现核间通信[31]。共享缓存总线结构的优点是设计简单、易于实现且通信速度快，但缺点是扩展性较差，通常只用于内核集成数量较少的同构多核处理器中。比较典型的采用共享缓存总线结构的处理器有 Hydra 和 Intel 的酷睿处理器等。

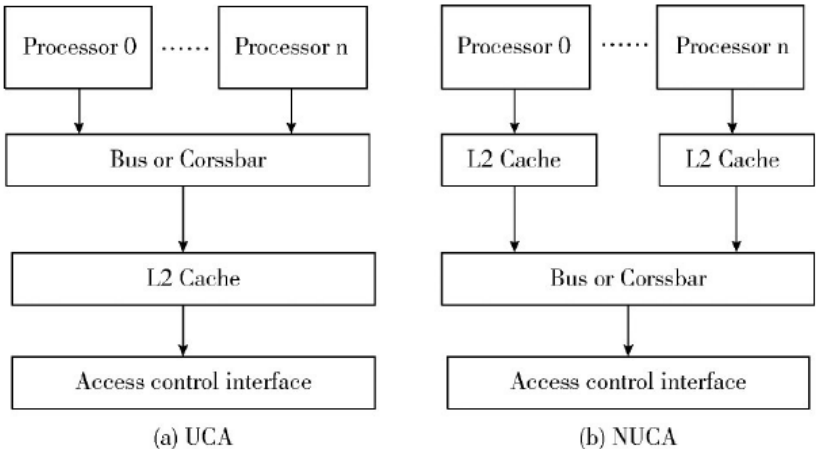


图 1.7 共享缓存总线结构

（2）共享总线结构。在该结构中，每个内核通过交叉开关与核间总线连接，并通过访问公共邮箱实现相互通信。典型的共享总线为 Cell 处理器的元件互联总线（Element Interconnect Bus，EIB），其结构如图 1.8 所示。该结构中包含数据仲裁器、共享命令总线及 4 个连接各个内核的高速环形总线，并通过高速环形总线实现核间通信。共享总线结构的优点为总线拓扑结构简单、扩展性强，而缺点是通信延迟较高、系统无法进行全局同步[32]。

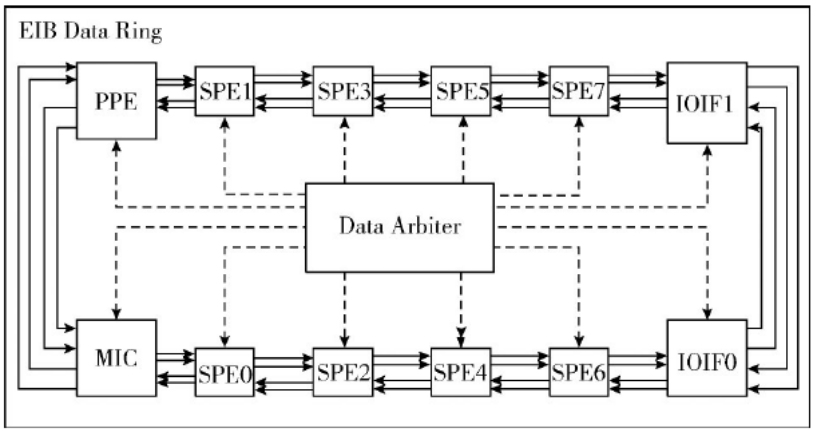


图 1.8 Cell 处理器的 EIB 共享总线结构

（3）片上网络结构。片上网络的设计参考了并行计算机的互联网络技术，将互连

网络技术应用到多核处理器设计上，以解决多核处理器片上组件之间的通信问题。片上网络支持包通信，提供透明的传输服务且具有良好的扩展性。相比于共享缓存总线结构和共享总线结构，片上网络可以连接更多的组件，且具有功耗低、可靠性高和可扩展性强等优点，因此适用于规模较大的多核处理器的核间通信。其缺点是片上网络的布置会占用较大的空间，并且需要设计复杂的通信协议。典型的采用片上网络进行核间通信的多核处理器是麻省理工学院设计的 RAW 处理器[33]。

1.3.2 同步互斥机制的研究

在单核处理器时代，操作系统的引入使得处理器能够并发执行程序，而由此带来了多线程的同步互斥问题。而进入多核处理器时代，由于多个内核共享处理器片上资源且各内核并行运行，因此同步互斥问题更加凸显[34]。典型的多核同步互斥的例子是 Linux 操作系统的大内核锁（Big Kernel Lock，BKL），由于 Linux 对多核的支持采用对称多处理的方式，同一时刻只能有一个内核运行操作系统，因此采用大内核锁将操作系统作为临界段进行保护，这里的大内核锁就是一种同步互斥机制[35]。

在单核处理器中，通常可以简单地使用关中断并禁止任务调度的方法实现并发线程间的同步互斥[36]。但对于多核处理器，由于多个内核同时并行运行，且每个内核只能关闭自身的中断而不能关闭其他内核的中断，因此关中断并禁止任务调度的方法并不适用用于多核处理器。多核间同步互斥机制的实现通常采用处理器提供的专用机器指令，如比较并交换指令 CAS、测试并加锁指令 TSL 和交换指令 XCHG 等，这些指令在硬件层面上保证对内存访问的互斥性，当多核处理器中某一内核通这些指令对内存进行访问时，会将内存总线锁死，任何其他内核都无法在该指令执行结束前访问内存，并且这些指令均为原子指令，其执行只需要一个指令周期，因此在指令执行过程中不会被本地中断所打断[37]。

比较并交换指令 CAS 是一种被广泛使用的专用机器指令，几乎所有的处理器家族如 Intel 的 x86 和 IA64、IBM 的 z 系列、TI 和 Sun 合作研发的 SPARC 等都支持该指令，且大多数操作系统都通过该指令作为实现同步互斥机制的基本原语。

TLS 指令和 XCHG 指令本质上与 CAS 指令相似。TLS 指令将一个内存单元的值读取到寄存器中，然后将该内存单元的值设置为一个非零值[38]；而 XCHG 指令将一个寄存器的值和一个内存单元的值进行交换，Intel 的 IA32、IA64 和 x86 系列处理器都

通过该指令实现底层的互斥机制。

1.3.3 多核任务调度的研究

任务调度分为静态任务调度和动态任务调度，静态任务调度算法可进一步分为依赖任务调度算法和独立任务调度算法，依赖任务调度算法用于各个任务之间存在依赖关系的情况，任务之间的依赖关系通常是指各任务需要按照一定的顺序进行，如一个任务的运行需要另一个任务的运行结果，从而使得两个任务的运行存在先后顺序；而独立任务调度算法用于任务之间没有依赖关系的情况。

静态任务调度在程序编译过程中或程序运行前进行，根据任务在处理器上的执行时间、各任务节点的依赖关系和任务节点之间的通信负载等任务节点调度信息，合理地进行任务分配设计，并将任务分配到相应的内核上运行。任务在分配到指定内核后不能进行更改，直到该任务在所在内核上执行完毕；动态任务调度在运行时对程序的执行情况和各内核的负载进行实时监测，然后根据这些动态运行信息，在各内核之间进行任务的分配与调度[39]。

静态任务调度具有实现简单、对任务分配的控制强、调度算法开销小的优点，但由于任务在各内核上的调度是在程序运行前进行的，因此难以实现负载均衡；而动态任务调度是在程序执行时根据系统的实时运行状态进行调整，因此能够很好地实现负载均衡，但动态调度的实现难度较大，且由于调度算法的时间复杂度较高，会使系统产生较大的开销，动态任务调度一般用于并行计算[40][41]。

1.4 冗余控制系统研究现状

国外早在上个世纪 60 年代开始提出冗余设计思想，主要用于对功能安全要求较高的军事和航空领域，当时的冗余设计通常是将双机系统进行简单的并联[42]。而在 80 年代以后，随着计算机技术的发展和应用以及硬件成本的降低，冗余技术的应用越来越广泛，从最开始的军事和航空领域发展到服务器、电力、汽车和民用客机等新的领域，很多大型民用客机如 A320、A330、A340 和波音 777 等均采用了冗余技术[43][44]。在无人机设计上也通常采用冗余设计，如美国的全球鹰无人机对除了发动机以外的所有其余系统都进行了双冗余设计，以保证其可靠性[45]。在可编程逻辑控制器（PLC）

领域，许多国外 PLC 厂商推出了采用冗余设计的系统，如西门子公司推出的 S7-300 软件冗余系统和 S7-400H 硬件冗余系统、罗克韦尔公司推出的 ControlLogix 软件冗余系统和硬件冗余系统等[46][47]。

随着国外冗余技术的不断发展，国内也对冗余技术进行了研究，国内的研究主要集中于冗余系统的架构设计。南车株洲电力机车公司的刘豫湘等对机车制动控制单元进行了双冗余设计，通过两套完全相同的制动控制板构成双冗余系统，并采用了热备份的冗余备份方案，实现了系统故障时的快速切换，提高了机车制动系统的可靠性[48]；中国科学院上海微系统与信息技术研究所的吕聪等对卫星星务管理中的通信系统进行了双冗余设计，不同于常用的同构双冗余系统，其采用了两个异构处理器构成双冗余系统，其中主系统采用性能较高但可靠性略低的处理器，而备用系统采用性能略低但可靠性较高的处理器，通过两种处理器的结合实现了性能与可靠性的互补，具有一定的创新性[49]；清华大学的周树桥和李铎通过对双冗余系统的状态分析明确了冗余设计中可能存在的“双主”冲突问题和备用系统失效问题，并针对此问题提出了基于双通信通道的冗余切换方案和从机状态报告机制[50]；南华大学电气工程学院的刘冲和付江梅基于罗克韦尔公司的 ControlLogix 系统设计了了一种可通过网络控制的控制器冗余方案[51]。清华大学的王鼎提出了一种应用于浮动式核电站中的基于 DeviceNet 总线的冗余控制系统，构建了高可靠性的核电站专用分布式控制系统[52]。

1.5 本文主要研究内容

本文主要研究了多核处理器在自动驾驶控制中的应用以及自动驾驶控制器的冗余设计方法，主要内容分为自动驾驶控制器的硬件设计和软件设计，具体研究内容如下：

第一章首先明确课题的背景及研究意义，然后分别对多核处理器的发展、多核关键技术的研究现状以及冗余控制系统的研究现状进行总结，最后进行全文内容及章节的安排。

第二章主要针对算力和功能安全需求对自动驾驶控制器进行硬件设计。首先对控制器的整体硬件架构进行设计，然后在硬件架构的基础上进行主要器件的选型，包括处理器、电源管理芯片和 CAN 收发器等。最后对控制器进行电路设计，包括电路原理图和印制电路板的设计。

第三章主要进行系统基础软件 Bootloader 的设计，以保证控制器的正常启动并满足控制器的应用程序升级需求。首先进行系统启动程序的设计，主要解决系统启动时的环境设置和多核的启动引导问题。然后进行应用程序下载的相关程序设计，包括 Bootloader 与上位机软件的通信设计、应用程序可执行文件的下载、解析与烧写程序设计以及 Bootloader 到应用程序的跳转程序设计。最后对程序存储器的空间分配方案进行设计，以保证 Bootloader 与应用程序的地址空间不发生冲突，并且使程序存储器的空间得到合理的分配和使用。

第四章进行多核处理器基础软件的设计。首先针对自动驾驶任务的实时性要求，进行多核基础软件架构的设计，主要工作包括：多核运行模式的选择、各内核的功能划分以及实时操作系统的移植与配置等。然后针对多核协同运行问题，进行相关软件机制的设计，包括核间任务同步机制、核间共享资源互斥访问机制、核间通信机制等，为多核的协同运行提供必要条件，从而使多核处理器的性能得以充分发挥。

第五章对控制器进行双冗余系统软件设计，以实现控制器的冗余功能。首先对双冗余控制器的总体冗余策略进行设计。其次，为了保证互为冗余的两个系统之间具有正确的状态关系，对系统状态信息管理程序进行设计。然后，针对两系统协同运行时的同步需求，进行同步协议和同步程序的设计。最后，在以上设计的基础上，进行系统故障检测及处理机制的设计，以保证系统出现故障时能够检测到故障的发生并立即做出响应，保证控制器稳定、可靠地工作。

第六章主要进行相关软件的测试，以验证软件设计的正确性。首先进行测试环境的搭建，然后设计相应的测试方法或编写相应的测试用例，分别对 Bootloader、多核处理器基础软件及双冗余系统软件进行测试，最后根据测试结果评估软件设计的正确性。

第七章对本文所做的工作进行总结，并对下一步的工作方向和内容进行展望。

第 2 章 自动驾驶控制器硬件设计

自动驾驶控制器硬件是进行软件设计并实现控制器相应功能的基础，因此本文首先对自动驾驶控制器的硬件进行设计，具体设计主要满足以下要求：

（1）充足的算力。在自动驾驶中，控制器通常需要处理大量的传感信息并运行较为复杂的算法，因此需要控制器具有充足的算力来保障；

（2）实现失效可用。在 L3 级以上的自动驾驶中，由于车辆的驾驶任务主要由自动驾驶控制器来完成，一旦控制器失效，会给乘员带来安全风险，因此需要自动驾驶控制器实现失效可用（Fail-Operational）的功能安全目标。

2.1 整体硬件架构设计

针对自动驾驶控制器的设计要求，首先对控制器进行了整体硬件架构的设计。由于多核处理器相较于单核处理器具有功耗低、性能强、多核线程间并行度高等优点，因此可以采用多核处理器来提高控制器的算力。而为了实现 Fail-Operational 的功能安全目标，通常采用冗余设计方法。

在冗余设计上，首先需要进行冗余度的选择。冗余度是指为了实现冗余所用的重复部件的数量，冗余度一方面影响控制器的可靠性，另一方面决定了系统的硬件成本。图 2.1 所示为冗余度与系统可靠性的关系曲线[53]。从图中可以看出，当冗余度从 1 增加到 2 时，系统的可靠性获得了大幅提升，但随着冗余度的继续增加，系统的可靠性提升越来越少。考虑到随着冗余度增加而带来的硬件成本的提高，文本采用了冗余度为 2 的双冗余设计方案。

结合多核处理器和双冗余设计方案，设计了如图 2.2 所示的自动驾驶控制器硬件架构。该架构的主要特点如下：

（1）采用多核处理器，用于提高控制器的算力；

（2）采用系统级冗余设计，即设计两套完全相同且完整的系统互为冗余，而不是分别对每个元器件单独进行冗余；

（3）在供电方式设计上，对两个系统分别进行独立供电，以防止电源的单点故障导致两个系统全部失效。

（4）两个系统之间通过相应的通信方式进行连接，一方面使两系统之间能够进行数据交互，从而协同运行以提高控制器的整体算力；另一方面使两系统可以相互监视对方的运行状态并进行故障检测，使控制器在出现故障时能够立即检测到故障的发生并快速进行响应，从而实现 Fail-Operational 的功能安全目标。

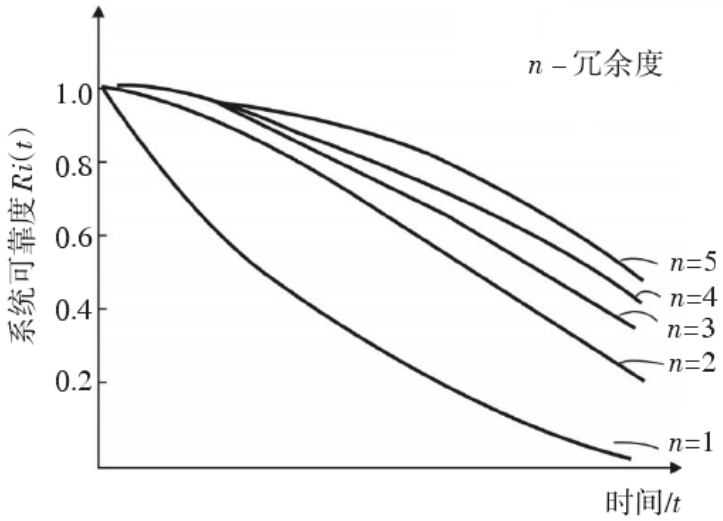


图 2.1 冗余度与系统可靠性关系曲线

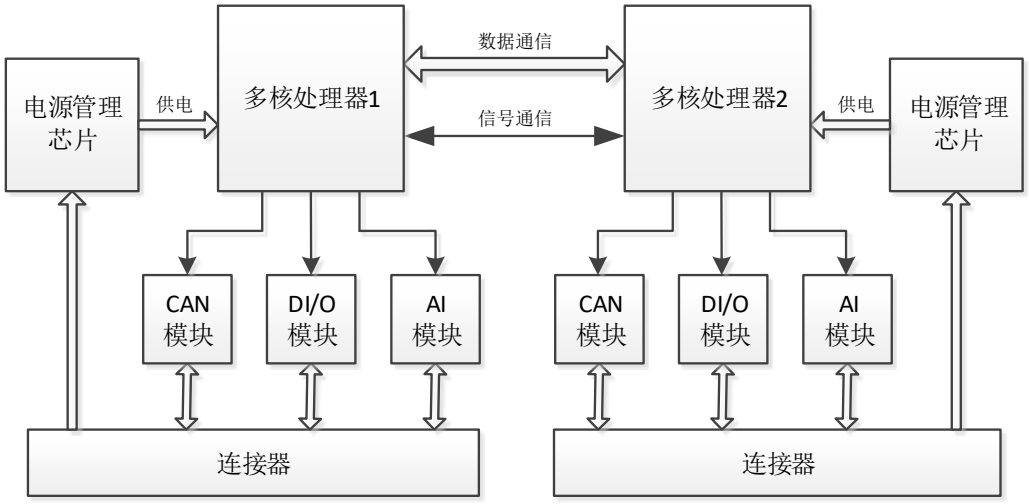


图 2.2 自动驾驶控制器硬件架构

2.2 主要元器件的选型

在自动驾驶控制器硬件架构的基础上，需要对架构中的主要元器件进行选型，包括多核处理器、电源管理芯片和 CAN 收发器。元器件的选型主要依据自动驾驶控制器的设计要求进行。

2.2.1 多核处理器的选型

多核处理器的选型主要考虑以下要求：

（1）充足的算力。处理器的算力决定了整个控制器的算力，因此为了保证自动驾驶控制器具有足够的算力，需要选择算力较强的处理器。

（2）处理器的功能安全特性。通过双系统冗余的方式实现控制器的功能安全目标，首先要保证两个系统自身满足一定的功能安全要求，而处理器是系统中最主要的部件，因此其选型需要考虑功能安全特性。

英飞凌公司的 TC297 是同构三核处理器，三个内核均为 Tricore-1.6P 架构，单核运行频率高达 300MHz，且每个内核都具有专用的浮点计算单元（Float Point Unit，FPU），用于处理浮点型运算，具有较强的算力。在功能安全方面，TC297 的其中一个内核带有锁步功能，可以用于诊断内核的指令执行是否存在错误；此外，TC297 能够达到 ISO26262 最高安全等级 ASIL-D。

综上所述，TC297 符合上述的多核处理器选型要求，因此本文采用 TC297 作为自动驾驶控制器的处理器。TC297 的芯片封装如图 2.3 所示。

图 2.3 TC297 芯片封装

2.2.2 电源管理芯片的选型

电源管理芯片的选型主要考虑以下要求：

（1）多轨供电。自动驾驶控制器中的各个部件通常具有不同的工作电压，因此需要电源管理芯片能够提供不同电压的多轨供电。

（2）芯片的功能安全特性。由于电源管理芯片负责系统的供电，因此对系统的功能安全具有较大的影响，电源管理芯片的功能安全机制能够与处理器的相关机制配合

使用，从而进一步提高系统的功能安全。因此在进行电源管理芯片的选型时，除了其供电功能以外，还要考虑其功能安全特性。

英飞凌公司的 TLF35584 是一款电源管理芯片，能够提供数字 3.3V、数字 5V 和模拟 5V 的多轨供电，且能够对每一路输出电压进行监视，从而为处理器和 CAN 收发器等提供稳定可靠的供电。除了基础的供电功能，还具有窗口看门狗、功能看门狗和安全状态控制等功能安全相关的机制，其中窗口看门狗和功能看门狗能够对处理器进行工作状态监视，并对处理器的异常状态做出响应；类似地，安全状态控制功能能够检测处理器主动报告的错误状态，从而进行相应的处理。

综上所述，TLF35584 不仅能够提供多轨供电，同时还提供了多种功能安全相关的机制，满足上述电源管理芯片的选型要求，因此本文采用 TLF35584 作为电源管理芯片。TLF35584 的芯片封装如图 2.4 所示。

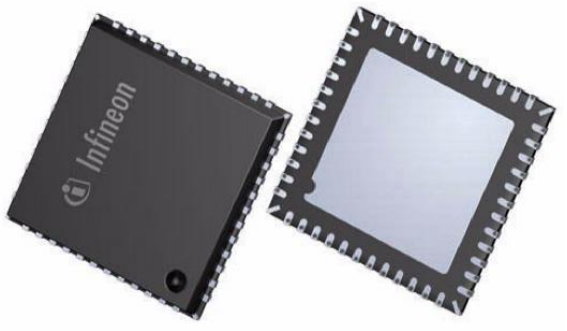


图 2.4 TLF35584 芯片封装

2.2.3 CAN 收发器的选型

CAN 收发器的选型主要考虑以下要求：

（1）支持 CAN FD。在 CAN 协议下，CAN 总线的最高传输速率只能达到 1Mbit/s，而在自动驾驶中，随着传感数据量的增加，对 CAN 总线的通信速率提出了更高的要求，因此在进行 CAN 收发器的选型时，需要考虑 CAN 收发器是否支持 CAN FD 协议，从而提高 CAN 总线的通信速率，以应对传感器的大量数据传输需求。

（2）封装紧凑。自动驾驶控制器通常具有较多的 CAN 通道，如果 CAN 收发器的封装较大，则相应的 CAN 通信电路在电路板上会占用较大的空间，不利于整个电路板的元器件布置，因此 CAN 收发器的选型应尽量选择紧凑型封装。

英飞凌的 TLE9251VLE 是一款高速 CAN 收发器，支持 CAN FD，最高传输速率

可达 5Mbit/s，符合 ISO11898-2 标准，此外还符合 SAE 标准 J1939 和 J2284-4/5。在芯片封装上，采用了紧凑型封装 PG-TSON-8，具有占用空间小的特点。

TLE9251VLE 满足上述 CAN 收发器的选型要求，因此本文选择 TLE9251VLE 作为控制器的 CAN 收发器。TLE9251VLE 的芯片封装如图 2.5 所示。

图 2.5 TLE9251VLE 芯片封装

2.3 控制器电路设计

在整体硬件架构设计和主要元器件选型的基础上，需要进行自动驾驶控制器硬件设计的最后一步工作，即电路设计。控制器电路设计的主要工作包括电路原理图设计和印制电路板（Printed Circuit Board，PCB）设计。

2.3.1 电路原理图设计

电路原理图确定了各个元器件之间的电路连接关系，是进行 PCB 设计的基础和依据。在控制器电路原理图的设计上，采用了自顶向下的设计方法，首先对顶层原理图进行了设计，确定了整个电路中包含的功能模块以及各模块之间的电路连接关系，然后分别针对底层的各个模块进行了电路设计。

顶层原理图的设计如图 2.6 所示。由于控制器采用双冗余设计，因此在顶层原理图中设计了两个相同的系统，两个系统之间通过 HSSL、SPI 和 ERU 硬线进行连接，以实现两个系统之间的通信。系统中的主要功能模块包括：系统供电模块、处理器最小系统模块、CAN 通信模块、数字信号输入处理模块、数字信号输出模块和模拟信号输入处理模块。

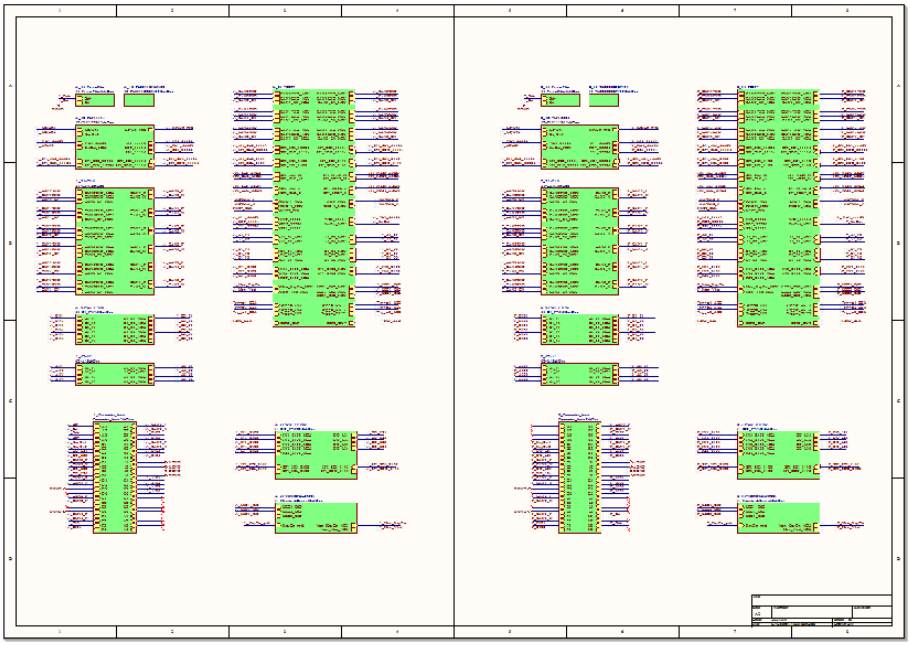


图 2.6 控制器顶层电路原理图

系统供电模块为整个系统提供电源，其电路主要围绕电源管理芯片 TLF35584 进行设计，另外还包括电源输入滤波电路的设计以及 TLF35584 与处理器之间的 SPI 通信电路的设计，对应的电路原理图如图 2.7 所示。

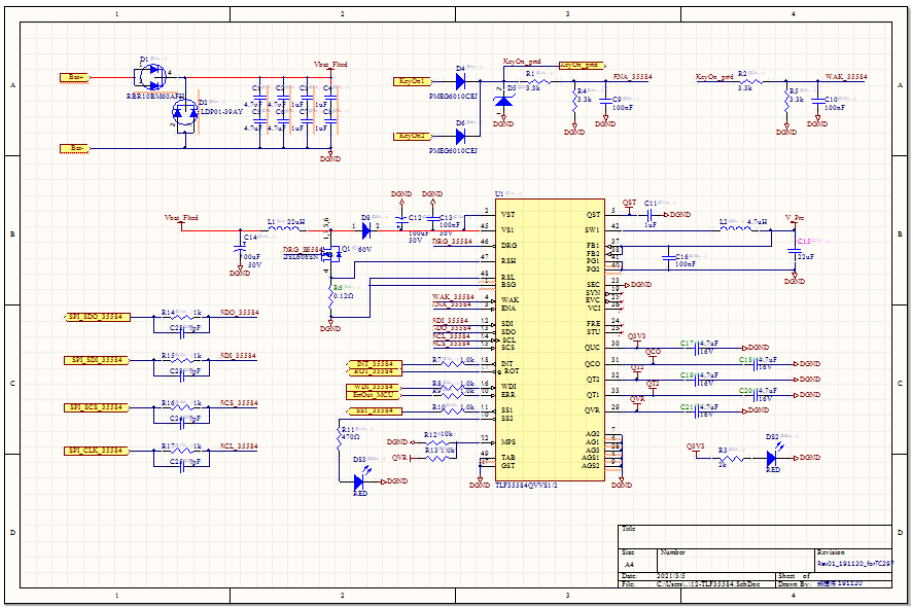


图 2.7 系统供电模块电路原理图

在进行处理器最小系统模块的电路设计时，将处理器按功能划分为 5 个部分，包括：供电、配置、通信、数字量输入输出和模拟量输入，并分别进行设计，如图 2.8 所示。各部分的具体设计如下：

（1）处理器供电部分。该部分电路设计的主要工作为：选择合适容值的电容，分

别为内核、外设以及晶振的供电引脚进行去耦，以过滤电源中存在的噪声，使供电电压更加平稳，从而保证处理器稳定运行；

（2）处理器配置部分。该部分电路设计的主要工作包括：1）通过将处理器配置引脚与电源或地相连的方式，对处理器的启动模式和供电模式进行设置；2）为处理器提供外部晶振作为时钟源；3）根据程序下载器的接口标准，将处理器的程序下载相关引脚与外接端子进行连接，从而为处理器程序下载提供接口；

（3）通信部分。该部分电路设计的主要工作为：将 CAN、SPI 和 HSSL 等通信外设对应的 I/O 引脚引出，以便与外围电路进行连接。另外，由于 CAN 和 HSSL 使用差分信号进行通信，因此对其进行了差分阻抗的配置；

（4）数字量输入输出部分。该部分电路设计的主要工作为：将用到的数字量输入输出引脚引出，以便与外围电路进行连接；

（5）模拟量输入部分。该部分电路设计的主要工作为：为模数转换器（Anolog-to-Digital Converter，ADC）设置参考电压，并将用到的 ADC 引脚引出，以便与外围电路进行连接。

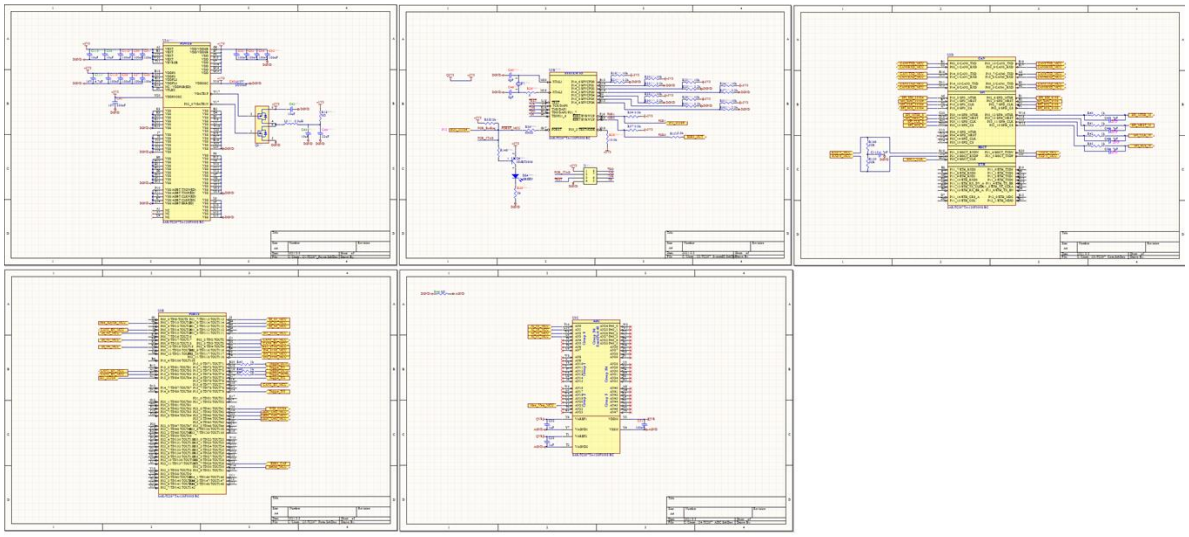


图 2.8 处理器最小系统模块电路原理图

CAN 通信模块包含的主要元器件包括 CAN 收发器和扼流圈，CAN 收发器用于处理器数字信号与 CAN 总线差分信号之间的信号类型转换，而扼流圈用于降低差分信号中的共模噪声以提高信号质量，对应的电路原理图如图 2.9 所示。

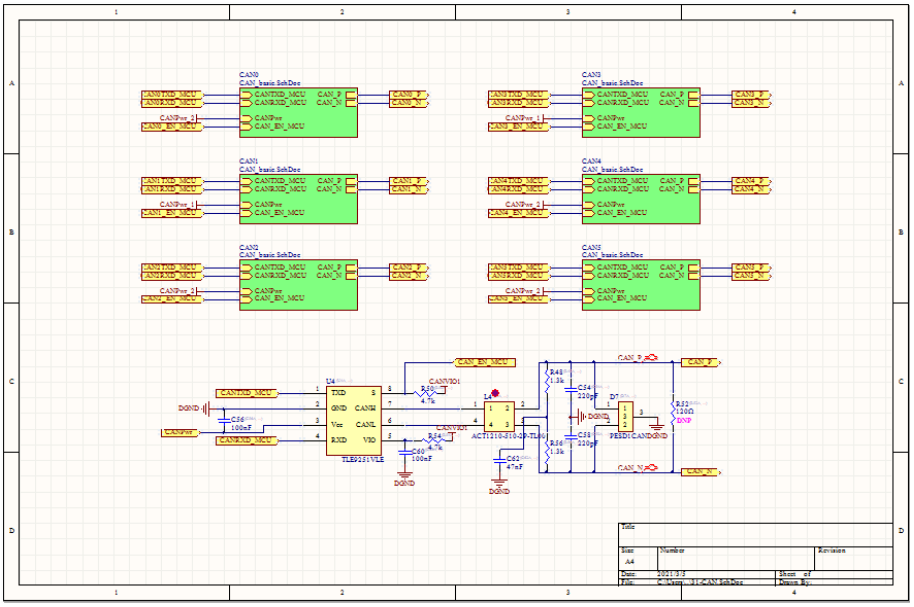


图 2.9 CAN 通信模块电路原理图

数字信号输入处理模块主要对输入信号进行滤波和分压处理，并对电路进行保护，以防止输入信号过压，对应的电路原理图如图 2.10 所示。

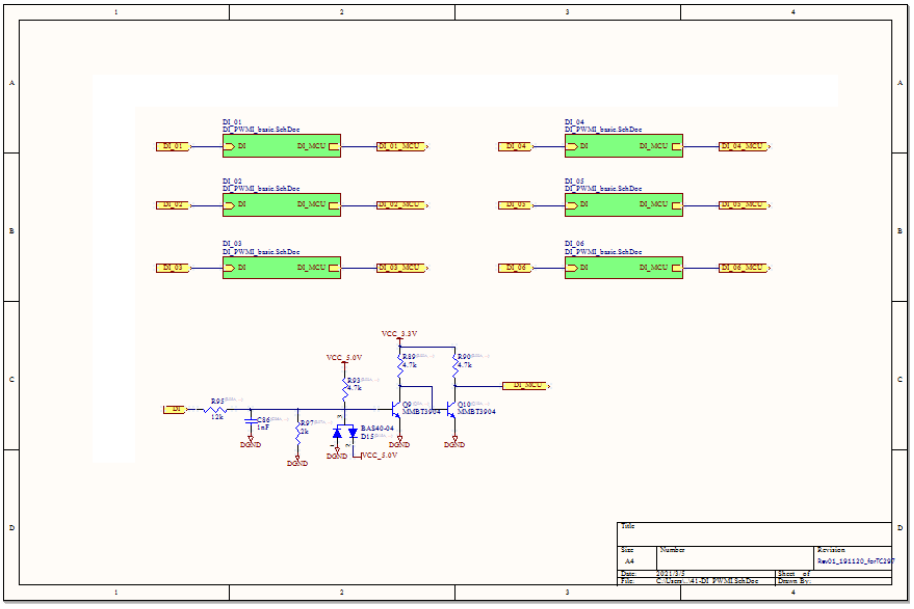


图 2.10 数字信号输入处理模块电路原理图

数字信号输出模块采用低边开关芯片 TLE8018 进行设计，处理器通过 SPI 通信控制该芯片，从而控制数字信号的输出，对应的电路原理图如图 2.11 所示。

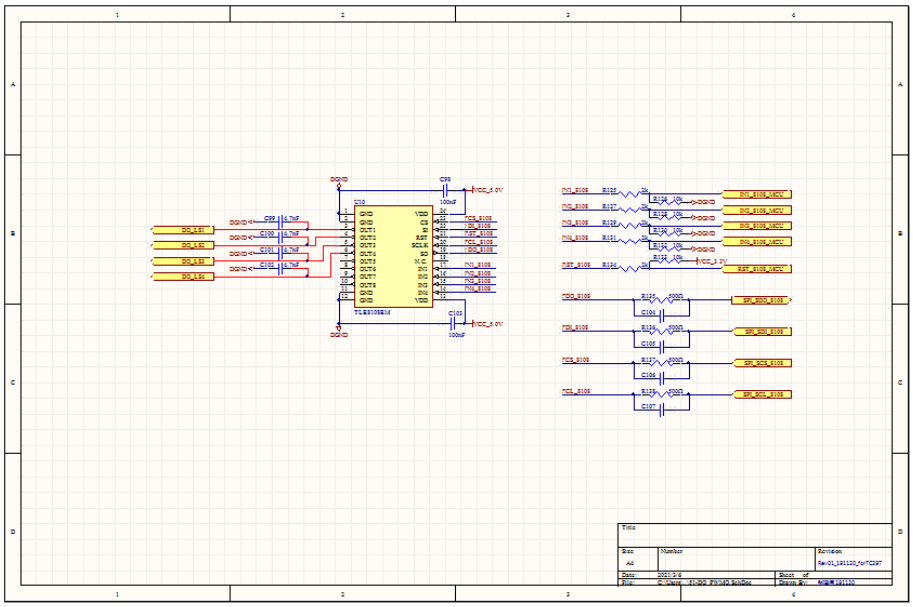


图 2.11 数字信号输出模块电路原理图

模拟信号输入处理模块主要对输入信号进行滤波和分压处理，并对电路进行保护，以防止输入信号过压，对应的电路原理图如图 2.12 所示。

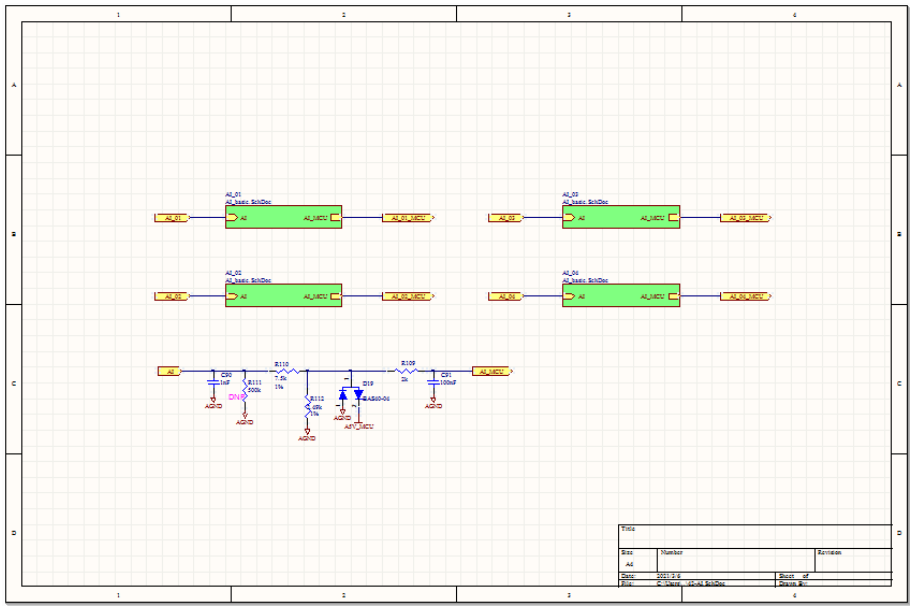


图 2.12 模拟信号输入处理模块电路原理图

2.3.2 印制电路板设计

在电路原理图的基础上，进行了印制电路板 PCB 的设计。PCB 设计的主要工作包括：PCB 基板的设置、元器件的布置和元器件之间的布线，具体设计如下：

（1）PCB 基板设置。在 PCB 基板的设置上，采用了 6 层板设计，分别为顶层（Top

Layer）、地层（GND）、中间层 1（Mid Layer1）、中间层 2（Mid Layer2）、电源层（POWER）和底层（Bottom Layer），如图 2.13 所示。其中顶层和底层用于布置器件和走线；中间层用于走线，主要是为了防止顶层和底层走线空间不足；电源层用于布置电源轨，包括为处理器内核供电的 1.3V 电源轨、为 CAN 收发器供电的 5V 电源轨、为处理器外设及其他元器件供电的 3.3V 电源轨以及专门为 ADC 提供参考电压的模拟 5V 电源轨。当元器件需要供电时，可以通过过孔直接从相应的电源轨引出电源，减小了回流路径，因此电源层的设计有利于提高 PCB 的电磁兼容性（Electromagnetic Compatibility，EMC）[54]；地层分为数字参考地（Digital Ground，DGND）和模拟参考地（Anolog Ground，AGND），分别用于为数字器件和模拟器件提供参考地，地层的设计也有利于提高 PCB 的电磁兼容性。

（2）元器件布置。在元器件的布置上，由于电源电路中有较多的铝质电容和电感，这些元器件重量较大且高度较高，在振动较强的车辆使用环境下容易发生焊点开焊或焊盘脱落等情况，因此，为了避免此类情况的发生，将电源电路布置在 PCB 板的边缘，从而减轻振动以提供更好的固定。另外，由于某些元器件在工作过程中容易发热，若散热不良可能会导致元器件发生故障，因此将发热器件全部布置在 PCB 顶层，并在器件底层设置了散热盘，然后通过导热硅脂与控制器的金属壳体相接触，从而使器件能够良好地散热。

（3）PCB 布线。在 PCB 的布线上，主要考虑电源相关电路的载流强度和通信电路的特殊布线要求。电源供电部分由于载流较大，因此采用较宽的走线或用覆铜的方式代替走线，从而提高载流能力；而一般的信号线载流较小，因此使用正常的走线宽度。在通信电路中，由于 SPI 具有严格的时序要求，因此采用了等长走线的方式进行布线；CAN 和 HSSL 均采用差分信号进行通信，因此采用了差分走线的方式进行布线。

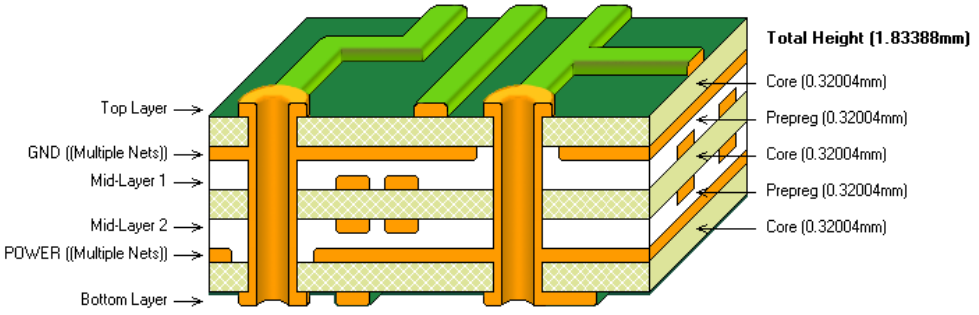


图 2.13 PCB 基板设置图

经过以上过程，完成了控制器的 PCB 设计，如图 2.14 所示。经过 PCB 加工和元器件焊接后，得到的控制器 PCB 实物如图 2.15 所示。最终将控制器 PCB 安装到相应的壳体中，并进行了封胶防水处理，如图 2.16 所示。

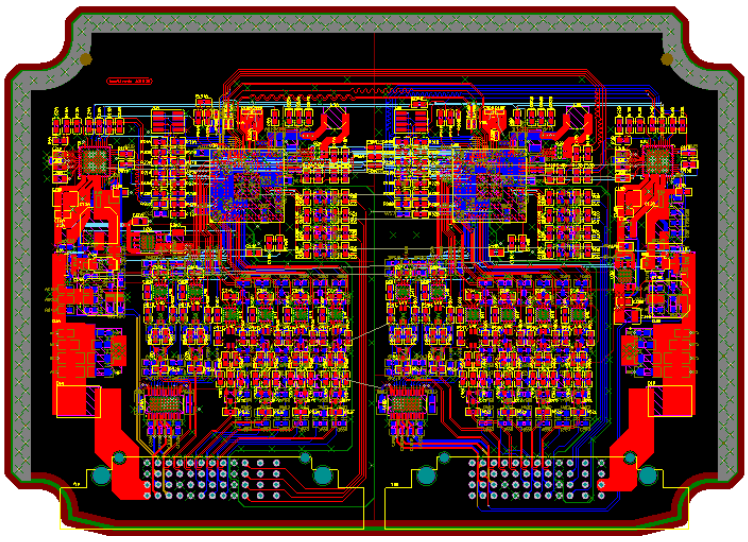


图 2.14 控制器 PCB 设计图

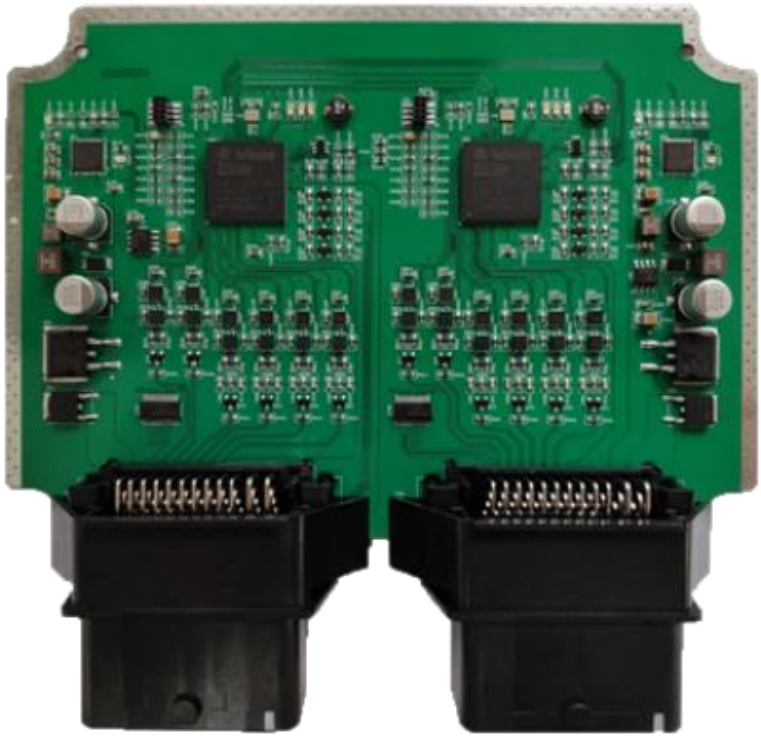


图 2.15 控制器 PCB 实物图

图 2.16 控制器实物图

2.4 本章小结

本章主要进行了自动驾驶控制器的硬件设计。首先根据自动驾驶控制器的功能需求进行了整体硬件架构的设计，设计了基于多核处理器的双冗余硬件架构，为控制器的硬件设计提供了基础框架。然后在硬件架构的基础上，分别根据相关需求对处理器、电源管理芯片和 CAN 收发器等主要元器件进行了选型。最后进行了控制器的电路设计，分别对控制器的系统供电模块、处理器最小系统模块、CAN 通信模块、数字信号输入处理模块、数字信号输出模块和模拟信号输入处理模块等进行了电路原理图设计，并在此基础上完成了 PCB 的设计，进而完成了整个自动驾驶控制器的硬件设计。

第 3 章 引导加载程序 Bootloader 设计

引导加载程序 Bootloader 是控制器上电后系统运行的的第一段程序，也是控制器最基础的软件程序。Bootloader 主要解决以下问题：

（1）控制器的正常启动。控制器上电后，首先必须对系统进行相应的设置，并建立正确的程序运行环境，才能保证应用程序的正常运行；

（2）应用程序的下载。由于自动驾驶控制器经常需要对应用程序进行升级，因此需要自动驾驶控制器提供应用程序下载功能，以实现应用程序升级。

3.1 Bootloader 总体设计

引导加载程序 Bootloader 的工作模式分为启动加载模式和下载模式，分别用于系统的启动和应用程序的下载[55]。在启动加载模式下，Bootloader 首先引导系统启动并为应用程序建立正确的运行环境，然后跳转到应用程序的起始地址并执行应用程序；在下载模式下，Bootloader 首先引导系统启动并为应用程序建立正确的运行环境，然后通过上位机软件下载应用程序可执行文件，并将可执行文件烧写到程序存储器中，在完成应用程序的下载与烧写后，跳转到应用程序的起始地址并执行应用程序。Bootloader 的总体工作流程如图 3.1 所示。

Bootloader 设计的主要工作包括：

（1）系统启动程序设计；

（2）应用程序的下载程序设计；

（3）Bootloader 到应用程序的跳转程序设计。

其中，系统启动程序主要对系统进行设置，以建立正确的程序运行环境，并引导多核正常启动；应用程序的下载程序用于对控制器进行应用程序升级；Bootloader 到应用程序的跳转程序用于使 Bootloader 在完成相应的工作后，跳转到应用程序并使应用程序开始运行。

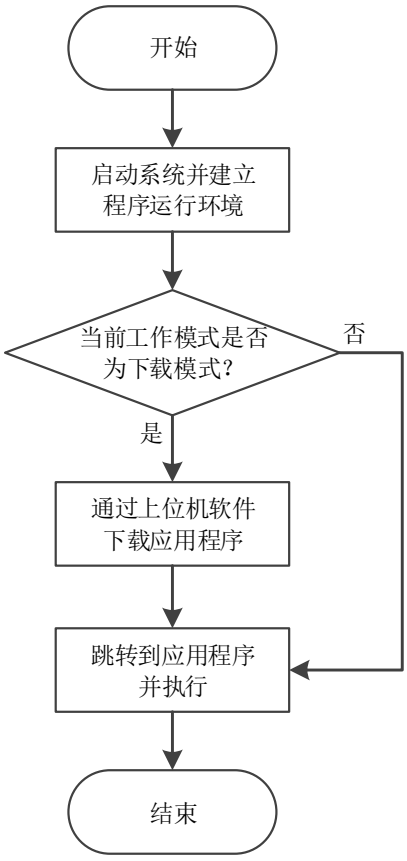


图 3.1 Bootloader 总体工作流程

3.2 多核处理器的启动

单核处理器在启动过程中主要对内核寄存器、堆栈、系统时钟和看门狗等进行初始化设置，而多核处理器由于包含多个内核，相比于单核处理器，其系统启动设置还需要考虑多核的启动与初始化问题。

TC297 为同构三核处理器，各内核具有相同的架构，如图 3.2 所示。其中 Core0 为引导核，负责整个系统的启动设置及另外两个内核的启动引导。系统上电后会首先启动 Core0，而 Core1 和 Core2 处于挂起状态，Core0 完成自身的初始化和系统启动设置后引导 Core1 和 Core2 启动并初始化，从而完成三个内核的启动。

系统启动程序的具体工作流程如图 3.3 所示。系统上电后首先初始化程序存储器 Flash，以保证内核能够从 Flash 中读取程序代码并执行，然后设置 Core0 的程序计数器 PC，使其指向 Core0 的初始化程序入口，接着启动 Core0 使其执行初始化程序。Core0 的初始化程序一方面对内核自身进行设置，包括看门狗的设置、内核寄存器的设置、Cache 的开启与关闭设置和上下文链表的设置；另一方面对系统整体进行设置，

主要包括 RAM 的初始化和系统时钟的初始化。其中看门狗的设置主要是将看门狗暂时关闭，以防止系统启动与设置过程中发生看门狗复位而导致系统重启；内核寄存器的设置主要包括程序状态字寄存器 PSW、上文信息寄存器 PCXI、用户堆栈寄存器 SP、中断堆栈寄存器 ISP、中断向量基地址寄存器 BIV 和陷阱向量基地址寄存器 BTV 的设置；上下文链表的设置主要是对上下文存储区 CSA 进行初始化，以保证程序上下文能够正常切换；RAM 的初始化主要是将程序存储器中存储的各全局变量的值分别拷贝到全局变量在 RAM 中的地址，从而完成全局变量的初始化。除此之外还需要将运行在 RAM 中的程序代码从程序存储器拷贝到 RAM 中，以保证程序代码的正常运行；系统时钟的初始化主要包括时钟源的选择和锁相环 PLL 的设置。当 Core0 完成自身和系统的设置后，对 Core1 和 Core2 的启动进行引导：首先设置内核的程序计数器 PC，使其指向内核初始化程序的入口，然后启动内核并执行初始化程序。完成 Core1 和 Core2 的启动引导后，Core0 跳转到其对应的主函数开始执行任务。Core1 和 Core2 启动后分别运行各自的初始化程序，对看门狗、内核寄存器、Cache 和上下文链表进行设置，并在初始化完成后跳转到各自对应的主函数开始执行任务。

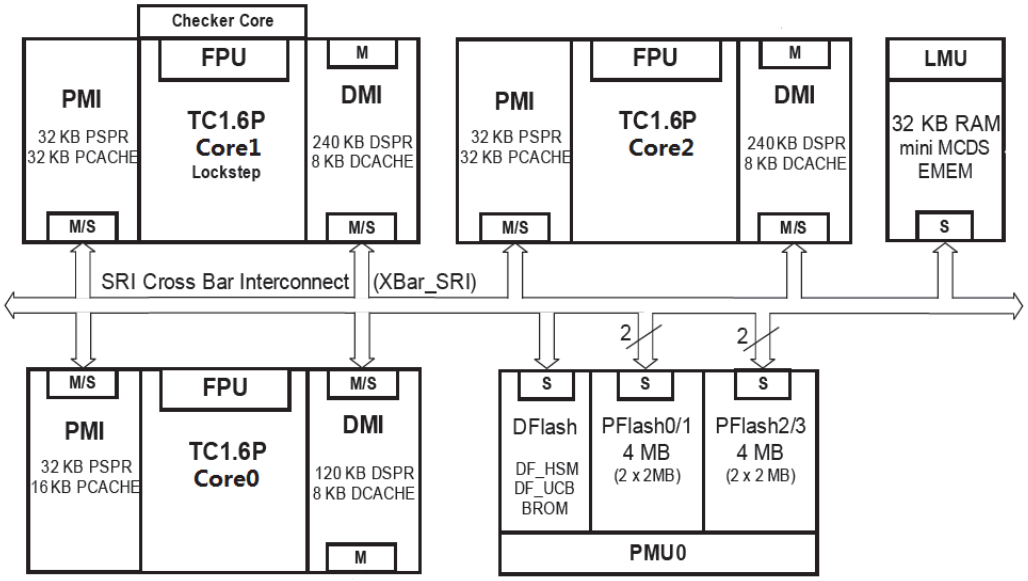


图 3.2 TC297 多核架构

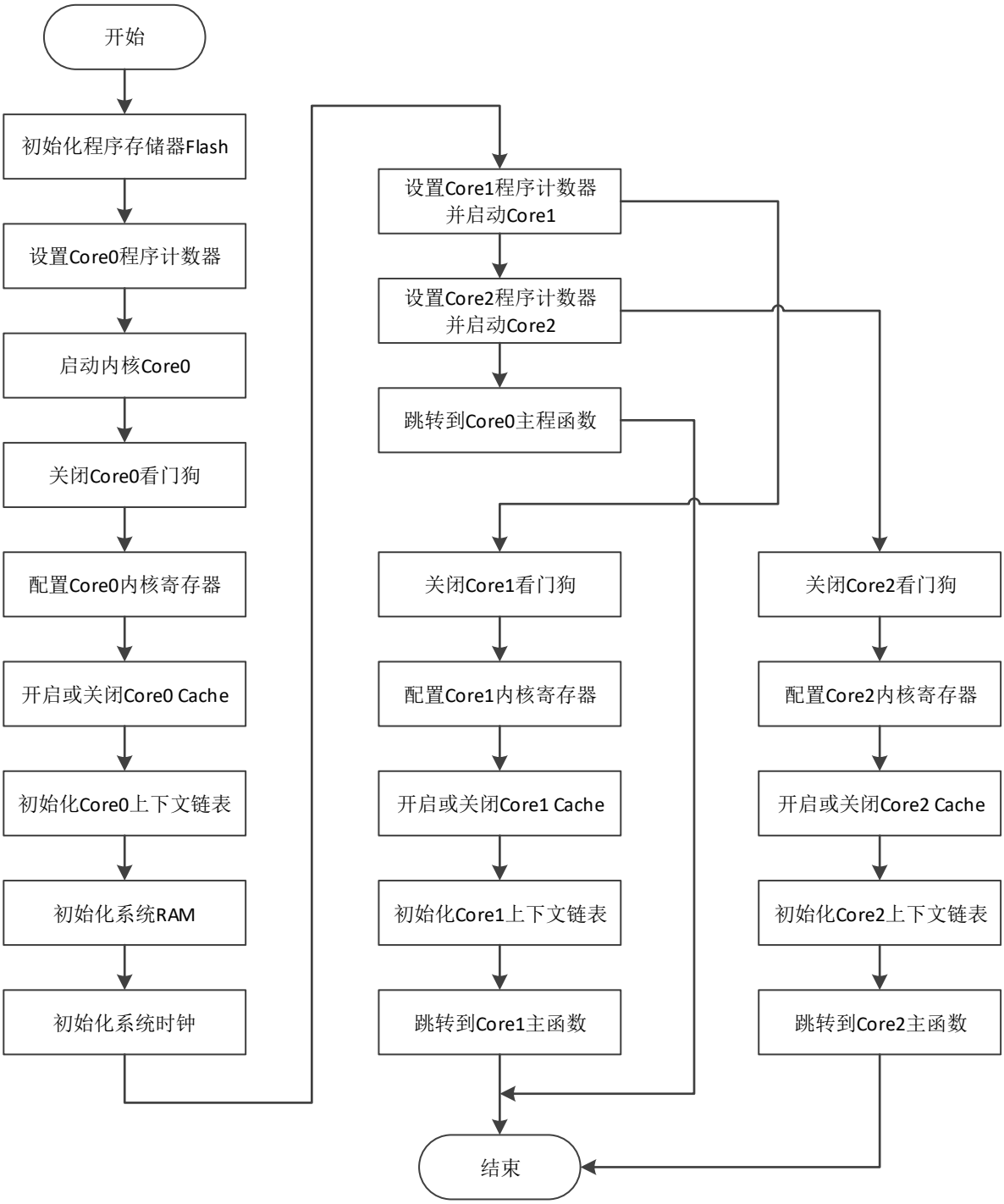


图 3.3 系统启动程序工作流程

3.3 应用程序的下载

3.3.1 可执行文件格式 Srec

应用程序的可执行文件格式有多种，包括 Elf、Hex、Bin 及 Srec 格式等，本文采

用 Srec 作为可执行文件的格式。Srec 格式是摩托罗拉公司制定的一种可执行文件格式标准，其目的是为了在不同的计算机平台之间传输程序代码和数据，图 3.4 所示为 Srec 格式可执行文件（以下简称 Srec 文件）的部分片段。

```
S01600004D756C7469636F72655F746573742E737265633B
S315A00E002091E000FAD9FF0422DC0F00900000000038
S315A00E0040000000000000000000000000000000FC
S315A00E0050000000000000000000000000000000EC
......
S315A00E01400D00000202F491E000FAD9FF94E2DC0F52
S315A00E0150008000000000000000000000000000006B
S315A00E81B000000000000000000000000000000000B
S313A00F1FE00D00000291E000EAD9EEA612DC0E6B
S315A00F60000D00000202F491E000FAD9FF9EC2DC0F48
S30DA00F6B4040110060000000000E7
S705A00E00202C
```

图 3.4 Srec 格式可执行文件示例

在 Srec 文件中，程序和数据文件以 ASCII 格式进行编码。Srec 文件的每一行都由 ASCII 字符‘S’开头，因此 Srec 文件的一行被称为一条“S 记录”，S 记录都由类型段、长度段、地址段、数据段、校验段 5 个部分组成，并以字符‘\r’和‘\n’结尾。S 记录的结构如图 3.5 所示。

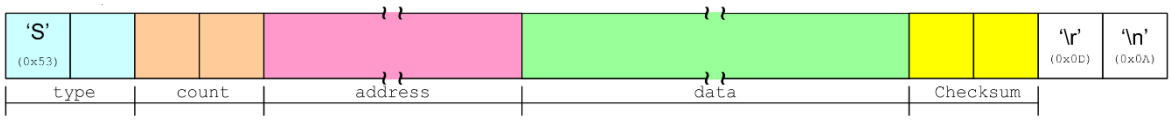


图 3.5 S 记录的组成结构

其中类型段描述了 S 记录的类型，S 记录的类型有 9 种，每种类型的 S 记录如下：

（1）S0 记录：Srec 文件的第一条记录，其地址段无效，而数据段记录了名称、版本号、修订版本号和文本注释等信息。S0 记录只是用来记录 Srec 文件的部分属性，不包含程序代码和数据，因此不需要烧写到程序存储器中；

（2）S1 记录：包含可执行文件的代码和数据信息，地址段占 2 个字节；

（3）S2 记录：包含可执行文件的代码和数据信息，地址段占 4 个字节；

（4）S3 记录：包含可执行文件的代码和数据信息，地址段占 8 个字节；

（5）S5 记录：记录了 Srec 文件中 S1、S2 和 S3 记录的总数，不包含数据段；

（6）S7 记录：记录了整个可执行文件的入口地址，地址长度为 8 个字节；

（7）S8 记录：记录了整个可执行文件的入口地址，地址长度为 4 个字节；

（8）S9 记录：记录了整个可执行文件的入口地址，地址长度为 2 个字节。

长度段占 2 个字节，表示地址段、数据段和校验段所占的字节总数；地址段占用

的字节数由 S 记录的类型决定，该段记录了 S 记录数据段中的指令代码和数据在程序存储器中的地址；数据段占用的字节数可以根据 S 记录的类型段和长度段进行计算，该段包含可执行文件的指令代码和数据，其内容需要写入到程序存储器中；校验段占 2 个字节，记录了整条 S 记录的校验值，用于保证 S 记录的可靠传输，防止 S 记录的数据在传输过程中发生错误而导致 S 记录失效，校验值的计算所涵盖的范围包括：长度段、地址段和数据段，其校验方法为和校验。

3.3.2 上位机软件工作流程

在 Bootloader 下载模式中，应用程序可执行文件的下载需要上位机软件的支持，上位机软件的主要工作为：将可执行文件解析成 S 记录并逐条发送给 Bootloader。

Bootloader 与上位机软件之间通过 CAN 总线进行通信，其通信协议如表 3.1 所示。Bootloader 以 0x0x0CFF0127 为 ID 向上位机软件发送不同类型的 CAN 帧，包括连接确认、S 记录接收确认帧和 S 记录重发请求帧；上位机软件以 0x0x0CFF0126 为 ID 向 Bootloader 发送不同类型的 CAN 帧，包括连接请求帧和 S 记录数据帧。在下载模式中，上位机软件通过连接请求帧向 Bootloader 发出连接请求，Bootloader 在收到连接请求后，通过回复连接确认帧与上位机软件建立通信连接，上位机通过 S 记录数据帧将可执行文件的 S 记录逐条发送给 Bootloader，Bootloader 通过回复 S 记录接收确认帧表示正确地收到了 S 记录，并通知上位机发送下一条 S 记录，当接收到的 S 记录存在数据错误时，Bootloader 可以通过重发请求帧向上位机软件请求重发本条 S 记录。

表 3.1 Bootloader 与上位机软件的通信协议

消息发送方	消息 ID	CAN 帧数据段	CAN 帧类型
Bootloader	0x0cff0127	0x0706050403020100	连接确认帧
		0x0000000000214b1d	S 记录接收确认帧
		0x0000000000534f53	S 记录重发请求帧
上位机软件	0x0cff0126	0x00000000083e572d	连接请求帧
		S 记录数据	S 记录数据帧

上位机软件的工作流程如图 3.6 所示。

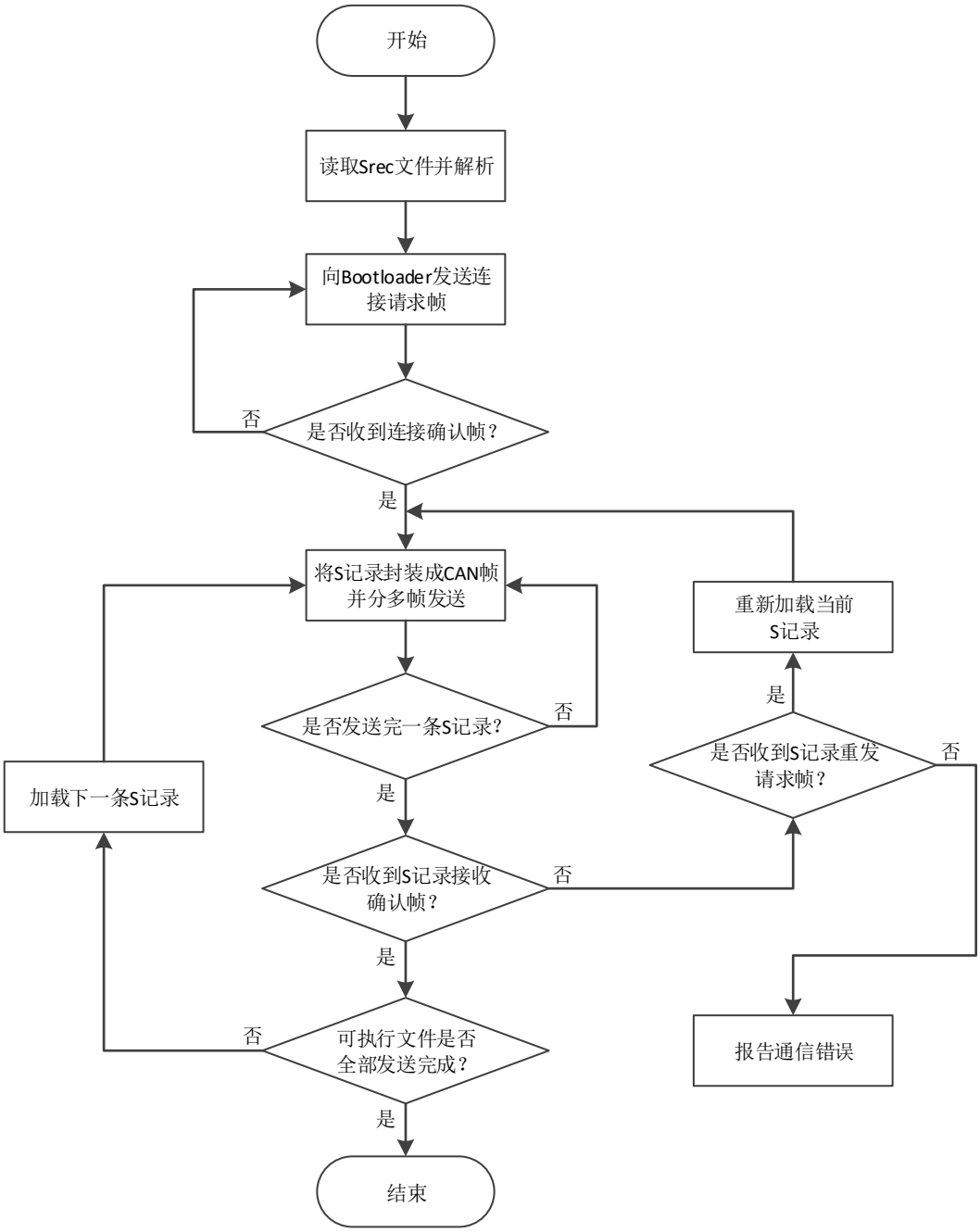


图 3.6 上位机软件工作流程

上位机首先读取应用程序可执行文件并解析成 S 记录，接着向 Bootloader 发送连接请求帧，请求与 Bootloader 建立通信连接，当收到 Bootloader 回复的连接确认帧后，通信连接建立完成，然后开始向 Bootloader 发送 S 记录。由于一个数据帧只能携带 8 个字节的数据，通常无法一次完成整条 S 记录的发送，因此一条 S 记录需要通过多个数据帧进行发送。当完成一条 S 记录的发送后，检查是否收到 Bootloader 回复的 S 记录接收确认帧，若收到则进一步检查应用程序可执行文件的所有 S 记录是否全部发送完成，如果所有 S 记录全部发送完成，则整个应用程序下载过程完成，否则加载下一

条 S 记录并发送；如果没有接到到 Bootloader 回复的 S 记录接收确认帧，则判断是否收到 S 记录重发请求帧，如果收到 S 记录重发请求帧，则重新发送本条 S 记录，否则说明 Bootloader 与上位机软件在通信过程中发生了错误，于是对该错误进行报告。

3.3.3 可执行文件的下载与烧写

上位机软件向 Bootloader 发送 S 记录时，将 S 记录分成多个数据帧进行发送。为了能够获得到正确的 S 记录数据，Bootloader 需要将接收到的多个数据帧还原为一条完整的 S 记录。本文设计了 S 记录下载缓冲区，用于存储数据帧中的 S 记录片段并将各片段重组，从而还原成一条完整的 S 记录，对应的程序工作流程如图 3.7 所示。

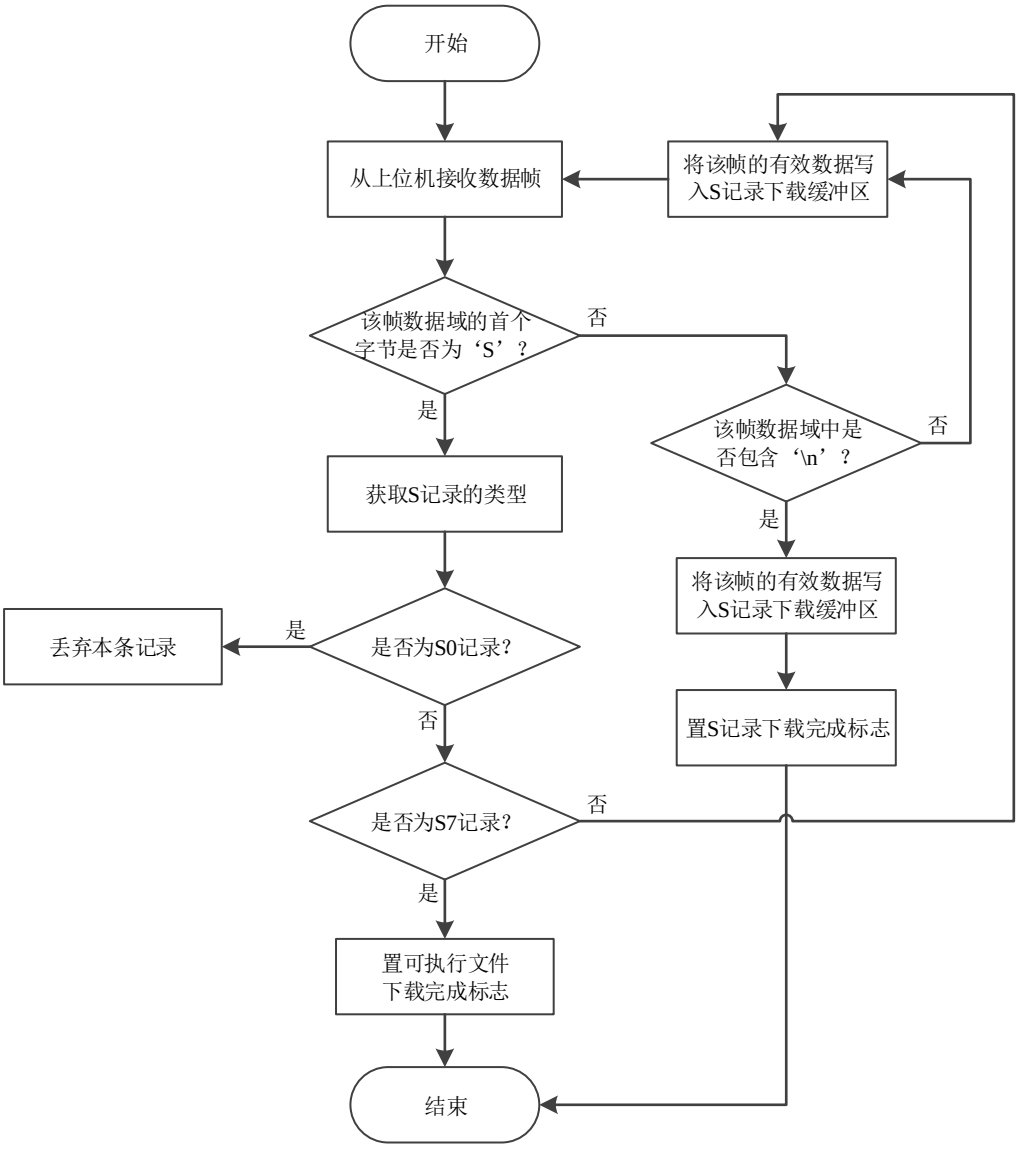


图 3.7 S 记录下载程序工作流程

Bootloader 在接收到上位机发送的数据帧时，首先判断该帧数据域的首个字节数据是否为字符‘S’，从而确定该帧是否为 S 记录的第一帧，如果是第一帧，则进一步根据该帧数据域的第二个字节数据即 S 记录的类型段判断本条 S 记录的类型，如果本条 S 记录为 S0 记录，则丢弃本条记录，因为 S0 记录并不包含有效的代码和数据；如果本条记录为 S7 记录，则说明整个可执行文件已经下载完成，于是置下载结束标志并结束本次下载；如果本条 S 记录的类型是 S0 或 S7 以外的其他记录类型，如 S1、S2 或 S3 等包含有效代码和数据的 S 记录，则将该帧中的有效数据存储到 S 记录下载缓冲区。如果该帧数据域的首字节数据不是字符‘S’，则进一步判断该帧数据域中是否包含字符‘\n’，从而判断该帧是否为 S 记录的最后一帧，如果不是最后一帧，则将该帧中的有效数据存储到 S 记录下载缓冲区，从而与本条 S 记录的其他片段进行重组；如果是最后一帧，则首先将其中的有效数据存储到 S 记录下载缓冲区，然后置 S 记录下载完成标志，于是完成了一条完整 S 记录的下载与格式还原。

由于此时得到的 S 记录的格式为 ASCII 编码的字符串格式，而可执行文件在程序存储器中需要以十六进制数的形式存储，因此目前得到的 S 记录不能直接烧写到程序存储器中，而需要进行格式转换。字符串到十六进制数的转换方式为每 2 个 ASCII 字符转换成一个字节的十六进制数，转换过程中所用的数据结构定义如下。

```
typedef struct
{
    uint32 addr;
    uint32 dtLen;
    uint32 sdata[16];
} S19_t;
```

该数据结构用来存放完成格式转换后的 S 记录，并按照 S 记录中的各主要段进行结构划分，其中 addr 用于存储 S 记录的地址，dtLen 用于存储 S 记录中数据段的长度，sdata 用于存储 S 记录中数据段的内容，即可执行文件的指令代码和数据。具体的 S 记录格式转换过程如下：

（1）首先根据 S 记录的类型段确定 S 记录的地址段长度，然后根据该长度对地址段进行格式转换，从而确定本条 S 记录的指令代码和数据在程序存储器中的地址，并将该地址存储到 addr 中；

（2）将长度段的两个字符转换成十六进制数，从而确定地址段、数据段和校验段的总长度，单位为字节；

（3）根据步骤（2）中获取的总长度和步骤（1）中获取的地址段长度，计算出数据段的长度并存储到 dtLen 中，具体方法为：数据段长度 = 总长度 – 地址段长度 – 校验段长度，其中校验段长度始终为 1。在计算得到数据段的长度后，即可对 S 记录的数据段进行格式转换，得到指令代码和数据并存储到数组 sdata 中。

由于 S 记录在传输过程中可能会产生数据错误，因此为了确保 S 记录的完整性和正确性，需要对其进行校验，具体校验过程为：

（1）将 S 记录的长度段、地址段和数据段的值按照字节划分，并对划分后的各字节数据进行求和，然后取反；

（2）将取反后得到的值与 S 记录中校验段的值进行比较，如果二者相等，则说明本条 S 记录正确；不相等则说明本条 S 记录存在数据错误，此时需要向上位机请求重发本条 S 记录。

在完成 S 记录的格式转换后，得到了符合程序存储器烧写格式要求的 S 记录，但除了格式要求以外，程序存储器对写入数据还有最小写入单位的要求，最小写入单位即一次性向程序存储器写入的最少字节数。S 记录的数据段大小通常小于程序存储器的最小写入单位，因此无法将一条 S 记录直接烧写到程序存储器中。为此，本文设计了 S 记录烧写缓冲区，用于解决此问题。S 记录烧写缓冲区的主要工作原理为：每当接收到一条 S 记录并完成格式转换与校验后，将其存储到 S 记录烧写缓冲区中，并与缓冲区中的其他 S 记录组合成新的、更长的 S 记录，当缓冲区中 S 记录的数据段长度达到程序存储器的最小写入单位要求时，将其烧写到程序存储器中。S 记录烧写缓冲区的数据结构如下所示。

```
typedef struct
{
    uint32   addr;
    uint32   dtLen;
    uint32   data[16];
    uint32   nextAddr;
    uint8    *pdata;
} SlineBuffer_t;
```

其中 addr 为缓冲区中组合得到的新 S 记录的地址；dtLen 为新 S 记录的数据段长度；数组 data 用于存储新 S 记录的数据段内容；nextAddr 为当前接收到的 S 记录的预期地址，用于判断当前接收到的 S 记录与缓冲区中 S 记录的地址是否连续，如果当前接收到的 S 记录的地址与预期地址 nextAddr 相等，则说明其与缓冲区中 S 记录的地

址是连续的，可以进行组合。否则说明其与缓冲区中 S 记录的地址是不连续的，不能够进行组合，此时需要对缓冲区中的 S 记录和当前接收到的 S 记录进行特殊处理。nextAddr 的值可以通过缓冲区中 S 记录的地址 addr 和数据段长度 dtLen 计算得到；pdata 为指向 data 数组中空闲位置的指针，用于缓冲区中 S 记录的组合。

S 记录烧写程序的工作流程如图 3.8 所示。当接收到一条 S 记录并进行格式转换后，首先将此 S 记录的地址与 nextAddr 进行比较，以判断此 S 记录与缓冲区中 S 记录的地址是否连续，如果地址连续，则将此 S 记录存储到缓冲区并与缓冲区中的 S 记录进行组合，然后判断缓冲区中 S 记录的数据段长度是否达到最小写入单位，若达到最小写入单位，则根据 S 记录的地址，将缓冲区中长度为最小写入单位的数据烧写到程序存储器中，然后更新 addr、dtLen、nextAddr 和 pdata 等缓冲区相关信息，并向上位机发送接收确认帧，以通知上位机发送下一条 S 记录；如果缓冲区中 S 记录的长度尚未达到程序存储器的最小写入单位，则只需更新缓冲区的相关信息。如果此 S 记录与缓冲区中 S 记录的地址不连续，则可能的情况有两种：

（1）当前接收到的 S 记录为 Srec 文件中的第一条有效记录；

（2）新的 S 记录包含中断向量或陷阱向量等信息，从而使该记录的地址相对于其上一条记录发生跳变。

情况（1）的产生的原因为缓冲区在存储第一条 S 记录之前其结构体中的各字段均为默认值 0，因此 nextAddr 也为 0，而 S 记录的地址值一定是非 0 的，所以会在收到第一条 S 记录时判断其地址与缓冲区 S 记录的地址不连续的。针对该情况，由于缓冲区当前为空，且一条 S 记录的数据段长度小于最小写入单位，不满足烧写条件，因此将该 S 记录存储到缓冲区中准备与其他 S 记录进行组合，然后更新缓冲区相关信息，并向上位机发送接收确认帧，以通知上位机发送下一条 S 记录；

对于情况（2），由于接收到了地址不连续的 S 记录，不能与缓冲区中的 S 记录进行组合，因此需要先将缓冲区中的 S 记录清空，然后将当前接收到的 S 记录存储到缓冲区中，使缓冲区从新的地址开始存储 S 记录，并向上位机发送接收确认帧，以通知上位机发送下一条 S 记录。清空缓冲区 S 记录的方法为：将缓冲区中 S 记录的数据段的空闲部分用数值 0 进行填充，以满足最小写入单位要求，然后将填充后的 S 记录烧写到程序存储器中。

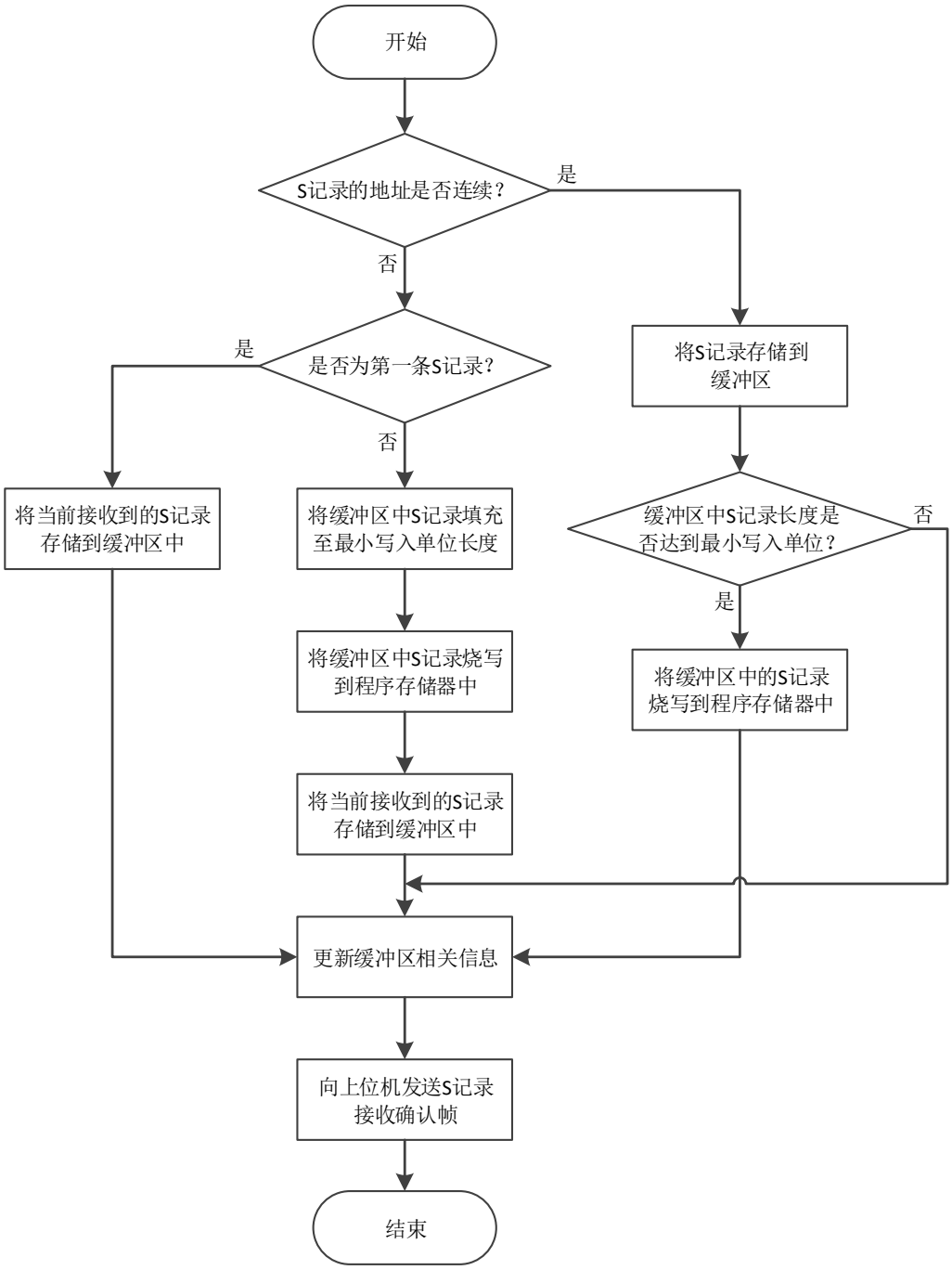


图 3.8 S 记录烧写程序工作流程

3.4 Bootloader 到应用程序的跳转

在引导加载模式中，系统启动后需要从 Bootloader 跳转到应用程序；同样地，在下载模式中，完成应用程序的下载后，系统也需要从 Bootloader 跳转到应用程序，因此需要设计跳转程序以满足上述需求。跳转程序设计的主要工作包括：跳转准备程序

的设计和跳转代码的设计[56]。

3.4.1 跳转前的准备

跳转前的准备工作主要是将 Bootloader 运行时对系统所做的设置或修改清除，并将用到的外设复位到初始状态，目的是防止对应用程序的运行产生影响。跳转准备程序的主要工作如下：

（1）清空随机存储器 RAM。TC297 的随机存储器 RAM 分为程序 RAM（PRAM）和数据 RAM（DRAM），分别用于存放程序代码和数据，Bootloader 中用于可执行文件烧写的 Flash 驱动程序存储在 PRAM 中，而 Bootloader 执行过程中产生的全局变量和局部静态变量等存储在 DRAM 中，因此在 Bootloader 退出时需要分别将 PRAM 和 DRAM 进行清空，以免其中的无效数据对应用程序的运行产生影响。清空 RAM 的具体方法是从起始地址开始将 RAM 的整个地址空间的值设置为 0；

（2）复位已开启的外设。Bootloader 中开启的外设主要是用来与上位机通信的 CAN 模块，因此需要对 CAN 模块进行复位，具体方法为将 CAN 节点和 CAN 消息对象等的寄存器设置为默认值；

（3）关闭全局中断。在 Bootloader 到应用程序的跳转过程中，如果发生中断则会打断跳转过程而导致跳转失败，因此需要在跳转之前关闭全局中断。

3.4.2 跳转程序的设计

Bootloader 到应用程序的跳转方法有两种，第一种方法为将应用程序入口地址转换为函数指针，然后以函数调用的方式执行应用程序，从而完成 Bootloader 到应用程序的跳转，采用该方法设计的跳转代码如下所示。

```
void JumpToApp( uint32 appEntry )
{
    ( *(void (*)(void))(appEntry) )();
}
```

在该函数中，参数 appEntry 为应用程序入口地址，函数首先通过强制类型转换的方式将 appEntry 转换成函数指针类型 void (*)(void)，然后将该指针解引用为函数名，并通过函数调用的方式执行该函数，从而间接地执行应用程序。

第二种方法为通过专用的跳转指令跳转到应用程序入口地址并执行。采用该方法

设计的跳转代码如下所示。

```
void Jump_Asm( uint32 appEntry )
{
    __asm( "movh.a %a15,hi:%0\n"
    "\tlea %a15,[%a15]lo:%0\n"
    "\tji %a15"::"d"(appEntry) );
}
```

这里采用了内嵌汇编语言，其中 movh.a 指令的作用为将 16 位常数的值复制到目标寄存器的高 16 位中，并将目标寄存器的低 16 位设置为 0；lea 指令的作用为计算有效地址并将其加载到目标寄存器中；hi 的作用为取某一数值的高 16 位，lo 的作用为取某一数值的低 16 位；ji 的作用为将地址寄存器的值加载到程序计数器 PC 中，然后跳转到该地址并执行程序。跳转程序的具体工作过程为：首先通过 movh.a 指令和 lea 指令将应用程序入口地址 appEntry 的值写入通用寄存器 a15 中，然后调用 ji 指令跳转到 a15 寄存器所记录的地址并从该地址开始执行程序。

第一种方法由于采用了函数调用的方式，因此在堆栈中保留了函数的返回地址，该返回地址属于 Bootloader 产生的数据，对于应用程序来说属于无效数据，且占用 RAM 空间导致内存浪费。而第二种方法通过跳转指令直接修改 PC 的值，不会遗留不必要的数据，因此本文采用第二种方法设计的跳转程序。

3.5 程序存储器的空间分配

在完成 Bootloader 程序的设计后，需要为 Bootloader 分配合适的存储空间以存放程序代码和数据。由于应用程序通常包含大量的数据和算法，会占用较大的存储空间，因此 Bootloader 存储空间的分配需要考虑如何提高程序存储器的空间利用率，为应用程序保留更多的存储空间。

TC297 的程序存储器采用了闪存 Flash，其内存空间结构及地址如图 3.9 所示。Flash 由 4 个物理扇区组成，分别为 PS0、PS1、PS2 和 PS3，每个扇区大小为 512KB，4 个物理扇区进一步划分为 27 个逻辑扇区，且每个物理扇区的划分粒度不同。物理扇区 PS0 被划分为 18 个逻辑扇区，每个扇区的大小为 16KB、32KB 或 64KB 不等；物理扇区 PS1 划分为 5 个逻辑扇区，每个扇区的大小为 64KB 或 128KB 不等；物理扇区 PS2 和 PS3 均划分为 2 个 256KB 大小的逻辑扇区。每个逻辑扇区又进一步被划分为多个页，每一页的大小为 32B，该页大小即为 Flash 的最小写入单位。Flash 在使用上

具有写前擦除的特性，即向 Flash 某一地址空间写入数据之前，必须先将该地址空间所在的逻辑扇区擦除，然后才能写入数据。

Logical Sector	Phys. Sub-Sector	Size	Offset Address[1)]
S0	PS0 (512 KB)	16 KB	$00'0000_H$
S1		16 KB	$00'4000_H$
S2		16 KB	$00'8000_H$
S3		16 KB	$00'C000_H$
S4		16 KB	$01'0000_H$
S5		16 KB	$01'4000_H$
S6		16 KB	$01'8000_H$
S7		16 KB	$01'C000_H$
S8		32 KB	$02'0000_H$
S9		32 KB	$02'8000_H$
S10		32 KB	$03'0000_H$
S11		32 KB	$03'8000_H$
S12		32 KB	$04'0000_H$
S13		32 KB	$04'8000_H$
S14		32 KB	$05'0000_H$
S15		32 KB	$05'8000_H$
S16		64 KB	$06'0000_H$
S17		64 KB	$07'0000_H$
S18	PS1 (512 KB)	64 KB	$08'0000_H$
S19		64 KB	$09'0000_H$
S20		128 KB	$0A'0000_H$
S21		128 KB	$0C'0000_H$
S22		128 KB	$0E'0000_H$
S23	PS2 (512 KB)	256 KB	$10'0000_H$
S24		256 KB	$14'0000_H$
S25	PS3 (512 KB)	256 KB	$18'0000_H$
S26		256 KB	$1C'0000_H$

图 3.9 TC297 程序存储器空间结构

在对 Bootloader 进行存储空间分配时，需要先确定其所用地址空间在整个程序存储器空间中的位置，然后为其分配合适大小的逻辑扇区。Bootloader 的地址空间可以分配在存储器的低地址区、中地址区或高地址区，但首先应该排除中地址区的分配方案，因为将 Bootloader 放在中地址区会导致程序存储器被分割成两块不连续的空间，从而使得其实际可以用空间减小；而将 Bootloader 存储在低地址区或高地址区时，由于其占用的地址空间位于程序存储器空间的两端，因此不会产生存储器空间被分割的问题。

经过编译后，得到了 Bootloader 的可执行文件，其大小为 25.3KB。将可执行文件存储在高地址区时，会占用逻辑扇区 S26 的后 25.3KB 大小的地址空间，此时逻辑扇区 S26 还剩余 230.7KB 大小的空闲地址空间，但由于 Flash 的写前擦除特性，如果想要向扇区 S26 的剩余地址空间写入数据，必须先将整个 S26 扇区擦除，而一旦擦除 S26

扇区，则存储在该扇区的 Bootloader 程序会被破坏，从而导致系统无法正常启动，因此 Bootloader 所在扇区的剩余地址空间不能被使用。因此将 Bootloader 分配在高地址区的方案会间接地造成存储空间的浪费，使实际可用空间减小。

将 Bootloader 的地址空间分配在低地址区时，可以使用低地址空间的 2 个 16KB 的逻辑扇区 S0 和 S1 来存储 Bootloader 程序，而将 S2 到 S26 的逻辑扇区空间留给应用程序使用。S0 和 S1 共提供 32KB 的存储空间，在存储 25.3KB 的 Bootloader 程序后所剩余的空间大小只有 6.7KB，空间利用率较高，并且使用扇区 S0 和 S1 时不会影响其他扇区的正常使用，因此这里采用了将 Bootloader 分配到低地址区的方案。

3.6 本章小结

本章进行了系统基础软件 Bootloader 的设计，首先对 Bootloader 进行了总体工作流程的设计，同时明确了 Bootloader 设计的主要工作。然后针对启动加载模式设计了系统启动程序，用于系统的启动设置和多核的启动引导。针对下载模式设计了应用程序可执行文件下载程序，以解决可执行文件的下载、格式转换、校验和烧写问题，实现了应用程序的下载和升级功能。另外还采用处理器的专用指令设计了跳转程序，实现了 Bootloader 到应用程序的跳转。最后进行了存储空间的分配方案设计，使 Bootloader 存储到合适的位置，从而为应用程序保留充足的存储空间。

第 4 章 多核处理器基础软件设计

多核处理器的基础软件设计主要解决以下两个问题：

（1）自动驾驶任务的实时运行。由于自动驾驶涉及危险场景，如果自动驾驶任务无法实时运行，则会导致车辆的控制产生滞后，从而给乘员带来安全风险，因此在自动驾驶中，对任务运行的实时性具有较为严格的要求；

（2）多核的协同运行。自动驾驶控制器的算力取决于其处理器的性能，因此，为了保证自动驾驶控制器具有充足的算力，需要使多核处理器的各内核协同运行，从而充分发挥多核处理器的性能，提高控制器的算力。

4.1 总体软件架构设计

针对自动驾驶任务的实时运行问题和多核处理器的多核协同运行问题，设计了如图 4.1 所示的多核处理器总体软件架构。

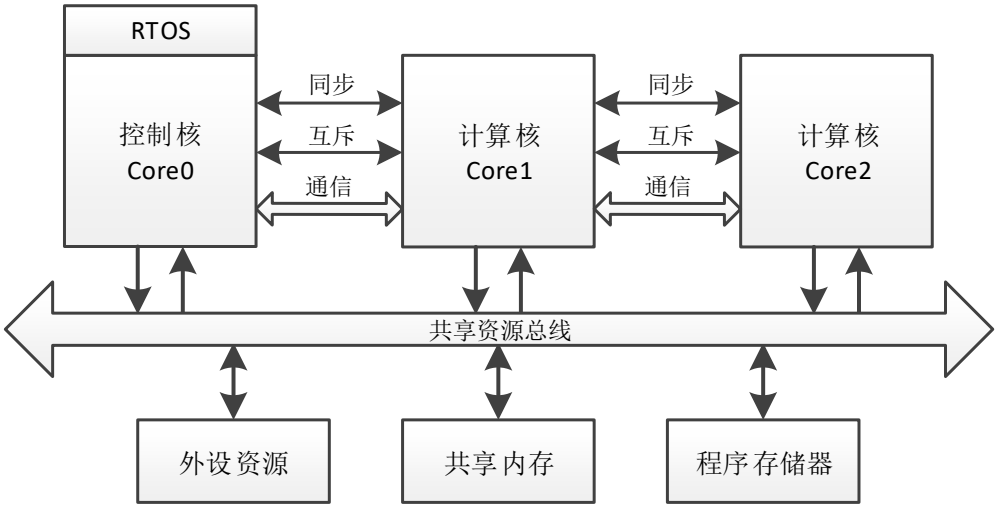


图 4.1 多核处理器总体软件架构

该架构的主要特点如下：

（1）采用非对称多处理作为多核处理器的运行模式。多核处理器的运行主要分为两种：对称多处理（Symmetric Multiprocessing，SMP）和非对称多处理（Asymmetric Multiprocessing，AMP）。在采用 SMP 模式时，处理器的多个内核运行同一个操作系统，各内核的地位均等并通过操作系统共享内存空间和外设等片上资源，任务在各个

内核上的分配是在运行时决定的；而采用 AMP 模式时，各内核的使用相比于 SMP 模式更为灵活，通常运行各自的操作系统或不运行操作系统，且操作系统可以相同也可以不同，各内核之间的关系可以为均等关系，也可以是主从关系等，任务在各个内核上的分配是在程序编译和链接时决定的。SMP 模式的优点是易于实现多核的负载均衡且片上资源的利用率较高；缺点是由于多核的任务分配发生在程序运行时，因而增加了系统运行时的不可预测性，且为了实现多核的负载均衡，通常需要较为复杂的任务分配算法，从而使得进行任务分配时产生较大的系统开销，并可能会破坏系统的实时性。AMP 模式不易于实现负载均衡，但由于 AMP 模式任务的分配是在编译和链接时确定的，因此系统的运行具有更好的可预测性和实时性。因此，为了解决自动驾驶任务的实时运行问题，采用了 AMP 模式；

（2）处理器各内核按功能划分为两个计算核和一个控制核。自动驾驶任务的类型分为两种：计算密集型任务和 I/O 密集型任务，例如，感知融合、运动分析和轨迹规划等属于计算密集型任务，而轨迹跟随控制则属于 I/O 密集型任务。当计算密集型任务和 I/O 密集型任务运行于同一内核时，二者之间会产生运行效率上的相互制约，计算密集型任务的运行需要内核不断地进行运算，而 I/O 密集型任务的运行则需要频繁地产生中断，中断的产生会打断内核的正常运行而进入中断处理程序，从而打断计算密集型任务的运行，使计算密集型任务的运行效率降低。因此，该架构将 TC297 的三个内核划分为计算核和控制核，计算核运行计算密集型任务，而控制核运行 I/O 密集型任务，从而防止两种类型的任务相互干涉，以提高任务的运行效率，进而更好地发挥多核处理器的性能；

（3）控制核搭载实时操作系统，计算核裸机运行。实时操作系统提供了多任务实时调度支持，利用实时操作系统有利于实现自动驾驶任务的实时运行，因此本文在多核处理器上搭载了实时操作系统。实时操作系统在运行过程中会通过中断来实现任务调度，控制核由于运行 I/O 密集型任务，其运行不受操作系统中断的影响，因此适合搭载实时操作系统，而计算核运行计算密集型任务，搭载实时操作系统会导致内核频繁发生中断，从而打断计算任务的运行，降低计算任务的运行效率，且运行实时操作系统会产生一定的系统开销，因此计算核采用裸机运行的方式而不搭载实时操作系统；

（4）不同内核上的任务之间通过核间任务同步机制进行同步。任务同步是指两个任务的运行存在时序关系，一个任务在另一个任务运行结束之前需要保持等待状态，

而当另一个任务运行结束后将其唤醒并运行，这个过程称为任务同步，在任务同步中，称时序上先后运行的两个任务分别为同步引导任务和同步等待任务。例如，内核 A 与内核 B 协同运行，内核 A 上运行感知融合任务，内核 B 上运行运动分析任务。由于运动分析需要感知融合得到的数据，因此两个任务在运行时序上存在先后关系而需要进行同步，其中感知融合任务为同步引导任务，而运动分析任务为同步等待任务，运动分析任务在感知融合任务完成之前需要保持等待状态，而直到感知融合任务运行结束后将其唤醒并运行，从而完成同步。核间任务的同步机制是多核协同运行的必要条件，因此该架构中设计了核间任务同步机制；

（5）不同内核上的任务通过核间共享资源互斥访问机制来保证对共享资源的访问不发生冲突。多核处理器运行时，各内核共享内存和外设等资源，而由于各个内核并发运行，所以多个内核可能会在同一时间对同一共享资源进行访问而产生竞争条件，所谓竞争条件，是指两个或多个进程使用共享资源时，最后的执行结果取决于进程运行的精确时序，由此引起的两个进程间的资源竞争。产生竞争条件时，可能导致共享资源访问失败或共享资源被破坏。因此为了保证多核协同运行时各内核能够有序地进行共享资源访问，该架构中设计了核间共享资源互斥访问机制；

（6）各内核之间通过核间通信机制进行数据交互。多核的协同运行需要不同内核上的任务之间能够进行数据交互，例如，内核 A 上运行感知融合任务，内核 B 上运行运动分析任务，二者之间不仅存在同步关系，还存在数据依赖关系，运动分析任务需要获得感知融合任务产生的数据才能正常运行，所以两个任务之间需要进行数据交互。因此，该架构中设计了核间通信机制。

下面将进行该架构的具体实现，主要包括实时操作系统在控制核上的移植以及核间任务同步机制、核间共享资源互斥访问机制和核间通信机制的设计。

4.2 实时操作系统的移植与配置

4.2.1 实时操作系统 FreeRTOS

FreeRTOS 是一个开源、支持多平台的轻量级嵌入式实时操作系统，支持任务管理、时间管理、内存管理、信号量和消息队列等功能，且具有运行开销小的优点，能够满足自动驾驶应用场景下的任务实时调度需求，因此本文选择 FreeRTOS 作为控制

核的实时操作系统。其主要特点如下[57]：

（1）同时支持基于优先级的抢占式任务调度和时间片轮转调度；

（2）操作系统内核可配置为可剥夺型内核或不可剥夺型内核；

（3）用户可根据实际需要对操作系统功能进行裁剪；

（4）可以创建任意数量的任务；

（5）具有堆栈溢出检测功能；

（6）采用优先级继承的方法防止优先级反转的发生；

（7）源代码的编写遵循 MISRA-C 标准。

4.2.2 FreeRTOS 的移植

FreeRTOS 源代码的文件结构如图 4.2 所示。其中 Source 目录包含了 FreeRTOS 操作系统内核源文件，include 目录包含了操作系统内核的头文件，与操作系统移植有关的文件位于 portable 目录下，该目录包含了 FreeRTOS 支持的各处理器平台的移植文件，移植文件主要包括 port.c、portmacro.c 和 porttrap.c 三个源文件。

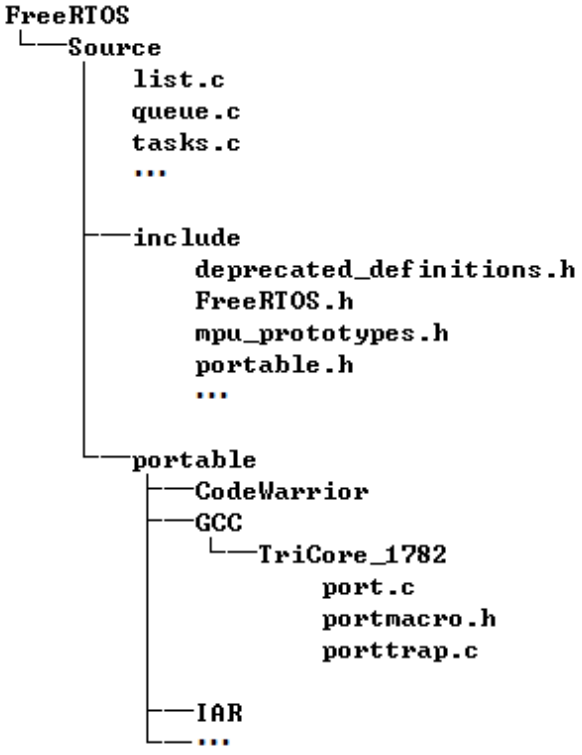


图 4.2 FreeRTOS 源代码文件结构

FreeRTOS 的任务调度方式有三种，分别为周期性执行的任务调度、在任务中执行

的任务调度和在中断中执行的任务调度，FreeRTOS 移植的主要工作是为这三种任务调度方式提供处理器平台的支持。

周期性执行的任务调度用于调度具有精确执行周期的任务，其实现需要处理器提供时钟中断，当时钟中断产生时，操作系统会在中断处理函数中进行任务调度，让具有最高优先级的就绪任务获得 CPU 使用权，从而实现周期性任务调度。TC297 提供了多种片上时钟资源，包括 STM、GTM 和 GPT12，本文采用系统定时器 STM 为周期性任务调度提供时钟中断，STM 的相关设置如下：

（1）设置 STM 的时钟中断频率。时钟中断的频率决定了操作系统进行任务调度的时间粒度，频率太高会导致内核频繁进出中断并不断发生上下文切换，从而产生较大的开销，影响系统性能；而频率太低可能无法满足任务周期的精度要求。本文综合两方面因素，将时钟中断频率设置为 1kHz，即任务调度周期为 1ms；

（2）选择时钟中断服务对应的内核。FreeRTOS 运行在控制核 Core0 上，因此需要将该时钟中断服务挂载到 Core0；

（3）设置时钟中断处理函数。port.c 中定义了函数 prvSystemTickHandler()，该函数为 FreeRTOS 针对不同处理器平台定义的周期性任务调度的接口函数，主要进行任务调度相关的工作。通过将该函数设置为处理器的时钟中断处理函数，能够使内核周期性进入该函数，从而进行周期性任务调度，因此需要将 STM 的时钟中断处理函数设置为 prvSystemTickHandler()；

在任务中执行的任务调度用于主动进行任务切换的场景，如某一任务向消息邮箱发送消息后，主动执行任务调度，唤醒等待消息的任务。该任务调度方式对应的函数包括 portYIELD()、taskYIELD()和 portYIELD_WITHIN_API()，在执行这些函数时，操作系统会立即进行任务调度，让具有最高优先级的就绪任务获得 CPU 使用权。为了实现主动任务调度，需要处理器平台提供相应的机制。TC297 提供了系统调用指令 syscall，该指令是一种陷阱指令，当执行该指令时，内核会立即进入陷阱处理函数，因此可以用于实现主动任务调度。对 syscall 指令及其陷阱处理函数的设置如下：

（1）将任务调度函数与系统调用指令进行关联。通过宏定义将 portYIELD()定义为 syscall(0)，使操作系统在调用 portYIELD()时间接调用 syscall 指令，从而进入系统调用的陷阱处理程序，该宏的定义如下所示。函数 taskYIELD() 和 portYIELD_WITHIN_API()为 portYIELD()的别名，因此不需要另做设置。

```
#define portYIELD()        _syscall( 0 )
```

（2）设置系统调用的陷阱处理函数。port.c 中定义了函数 prvTrapYield ()，该函数为 FreeRTOS 针对不同处理器平台定义的在任务中执行的任务调度的接口函数，主要进行任务调度相关的工作，通过将该函数设置为系统调用的陷阱处理函数，可以使 portYIELD()或 portYIELD_WITHIN_API()函数被调用时，间接调用 syscall 指令并进入该函数，从而进行任务调度。因此需要将系统调用的陷阱处理函数设置为 prvTrapYield ()。

在中断中执行的任务调度用于延迟任务调度，相应的任务调度函数包括 portYIELD_FROM_ISR()和 taskYEILD_FROM_ISR()。延迟任务调度是指将中断处理函数中需要进行的任务调度延迟到中断退出后进行。当中断发生时，会保存中断上下文并进入中断处理函数，如果此时操作系统进行任务调度，会导致上下文的切换发生错误。为了解决这一问题，需要一种机制将任务调度的执行暂时挂起，当退出中断后再执行。TC297 提供了软件中断资源 GPSR（General Purpose Service Request），可用于实现上述机制，具体设置如下：

（1）将任务调度函数与软件中断的触发指令进行关联。通过宏定义将 portYIELD_FROM_ISR() 函数定义为软件中断触发指令，使操作系统在调用 portYIELD_FROM_ISR()时，间接触发软件中断，并进入软件中断处理程序，该宏的定义如下所示。函数 taskYEILD_FROM_ISR()为 portYIELD_FROM_ISR()的别名，不需要另做设置。

```
#define portYIELD_FROM_ISR( xHigherPriorityTaskWoken )  \
        {\
            if( xHigherPriorityTaskWoken != pdFALSE )\
            {\
                INT_SRB0.B.TRIG0 = 1;\
                _isync();\
            }\
        }
```

（2）设置软件中断的中断处理函数。port.c 中定义了函数 prvInterruptYield ()，该函数为 FreeRTOS 针对不同处理器平台定义的在中断中执行的任务调度的接口函数，通过将该函数设置为软件中断的处理函数，可以使 portYIELD_FROM_ISR() 或 taskYEILD_FROM_ISR()函数被调用时，间接执行软件中断触发指令并进入该函数，从而进行任务调度。因此需要将软件中断的处理函数设置为 prvInterruptYield ()。

（3）设置软件中断的优先级。利用软件中断实现在中断中进行任务调度的原理为：在中断中进行任务调度时会执行函数 portYIELD_FROM_ISR() 或 taskYIELD_FROM_ISR()，进而触发软件中断，当软件中断发生时，先将自身挂起，等到当前所在中断退出后再进入软件中断处理程序进行任务调度。为了使软件中断发生时能够将自身挂起而不是抢占其所在中断，需要设置软件中断的优先级低于其所在中断的优先级，因此需要将软件中断的优先级设置为最低中断优先级，从而使基于软件中断的延迟任务调度可用于所有优先级的中断中。TC297 的中断优先级范围为 1 至 255，数值越大，优先级越高，因此需要将相应软件中断的优先级设置为 1。

4.2.3 FreeRTOS 的配置

在完成 FreeRTOS 的移植后，还需要对其进行相应的配置以适应处理器平台和实际应用场景。主要配置工作包括调度方式的选择、与处理器平台相关的参数配置及操作系统功能配置等，相应的配置在文件 FreeRTOSConfig.h 中进行，如下所示。

```
#define configUSE_PREEMPTION                    1
#define configCPU_CLOCK_HZ                      ( 300000000UL )
#define configPERIPHERAL_CLOCK_HZ               ( 100000000UL )
#define configTICK_RATE_HZ                      ( 1000UL )
#define configUSE_COUNTING_SEMAPHORES           1
#define configUSE_MUTEXES                       1
#define configUSE_QUEUE_SETS                    1
```

其中，configUSE_PREEMPTION 选项用于配置是否使用基于优先级的抢占式任务调度，这里设置为 1 表示使用该调度方式，从而可以根据优先程度为任务指定不同的优先级，并根据优先级进行任务调度；参数 configCPU_CLOCK_HZ 用于配置内核的时钟频率，TC297 的内核时钟频率为 300MHz，因此这里设置为 300000000；参数 configPERIPHERAL_CLOCK_HZ 用于配置外设时钟频率，TC297 的外设时钟频率为 100MHz，因此设置为 100000000；参数 configTICK_RATE_HZ 用于配置操作系统的时钟节拍频率，应该与 STM 时钟中断频率相同，因此设置为 1000；configUSE_MUTEXES、configUSE_TIMERS 及 configUSE_COUNTING_SEMAPHORES 分别用于配置操作系统是否使用互斥量、软件定时器和计数信号量，这里均设置为 1 以启用这些功能。

4.3 核间任务同步机制设计

4.3.1 核间同步信号的传递

为了实现核间任务的同步，首先需要使内核之间能够相互传递同步信号，从而使同步引导任务通过同步信号唤醒同步等待任务。核间同步信号传递的实现方式主要有两种：核间中断和共享内存。核间中断是一种内核之间相互触发中断的机制，一个内核可以通过触发目标内核产生中断的方式来实现向目标内核发送同步信号；通过共享内存实现核间信号传递的方法为：在共享内存中定义一个全局变量作为同步信号标志，各内核都能够对其进行访问，一个内核向目标内核传递同步信号时将该标志置位，以表示同步信号的发出。

采用核间中断机制实现同步信号传递时采用了触发的方式，因此具有实时性高的优点；而采用共享内存的方式虽然在实现上较为简单，但由于目标内核需要轮询检测同步信号标志的状态，同步信号传递的实时性较差，且轮询检测会产生一定的系统开销，导致系统性能降低，因此本文采用核间中断机制实现核间同步信号的传递。

4.3.2 核间任务同步程序设计

本文采用核间中断设计了核间任务同步机制，其工作原理如图 4.3 所示。内核 A 与内核 B 协同运行，其中“同步消息邮箱 1”用管理内核 A 上同步引导任务与内核 B 上同步等待任务之间的同步过程，其实质是一个类型为 sync_mailbox_t 的全局共享变量，其具体定义如下。其中数组 tasks 记录了内核 B 上所有与内核 A 存在同步关系的同步等待任务，task[0]不记录任务。tasks 数组的元素类型为 task_entity_t，该类型中记录了指向任务函数及其参数的指针；sync_task_id 记录了当前需要被唤醒的同步等待任务的 ID，其默认值为 0。ID 为 i 的任务记录在数组 tasks 的第 i 个位置，即 tasks[i]；指针 intercore_int 指向内核 A 用于向内核 B 发送同步信号的核间中断资源。

```
typedef struct
{
    task_entity_t *tasks[MAX_TASKS];
    tid_t sync_task_id;
    volatile Ifx_SRC_SRCR *intercore_int;
} sync_mailbox_t;
```

利用核间任务同步机制实现内核 A 上同步引导任务与内核 B 上同步等待任务之间的同步的具体过程如下：

（1）内核 A 上的同步引导任务运行结束后，在“同步消息邮箱 1”中设置 sync_task_id，即与其存在同步关系而需要被唤醒的内核 B 上的同步等待任务的 ID；

（2）同步引导任务通过同步消息邮箱中指向核间中断资源的指针 intercore_int 触发内核 B 产生中断，作为内核 A 向内核 B 发送的同步信号；

（3）内核 B 在中断处理程序中读取内核 A 设置的同步等待任务 ID，然后根据 ID 从同步消息邮箱的 tasks 数组中查找并运行对应的同步等待任务，从而完成内核 A 与内核 B 之间的任务同步。

类似地，内核 B 上同步引导任务与内核 A 上同步等待任务之间的同步采用“同步消息邮箱 2”进行管理，这里不再赘述。

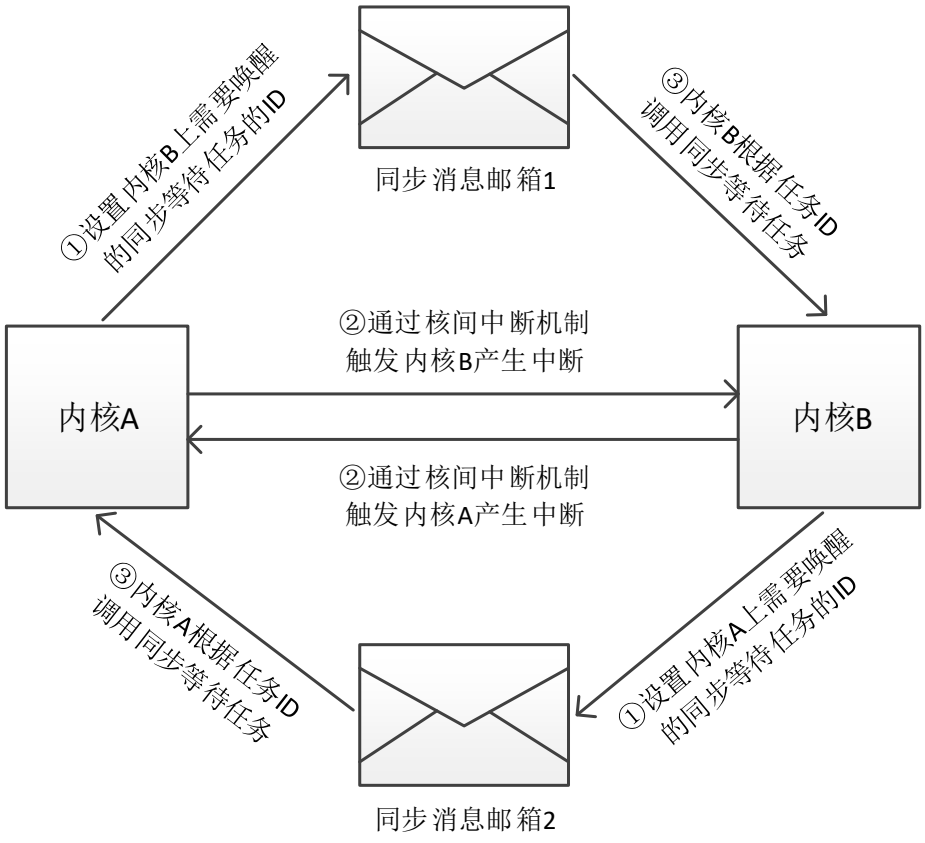


图 4.3 核间任务同步机制工作原理

核间任务同步机制的相关函数主要包括：init_sync_mailbox()、sync_task_add()和 task_sync()，各函数的定义如下：

（1）init_sync_mailbox()。该函数用于对同步消息邮箱进行初始化，以保证同步消

息邮箱具有正确的初始状态，其函数原型如下。其中，参数 p_mailbox 为指向需要初始化的消息邮箱的指针；p_intsrc 指向该同步消息邮箱所用的核间中断资源；serv_func 指向核间中断对应的中断处理函数。

```
init_sync_mailbox( sync_mailbox_t *p_mailbox, Ifx_SRC_SRCR *p_intsrc,
                                              void (*serv_func)( void * ) );
```

该函数的主要工作包括：1）为同步消息邮箱分配核间中断资源，并设置核间中断对应的中断处理函数。首先将指向核间中断资源的指针 p_intsrc 赋给同步消息邮箱中的成员变量 intercore_int，然后将 serv_func 指向的函数设置为核间中断的中断处理函数；2）初始化同步消息邮箱中的变量 sync_task_id 为默认值 0。

（2）sync_task_add()。该函数用于将对方内核上的同步等待任务添加到同步消息邮箱中，其函数原型如下。其中，参数 p_mailbox 指向要添加到的同步消息邮箱；id 是为当前添加的同步等待任务设置的 ID，在对同步等待任务进行唤醒时，会根据其 ID 从同步消息邮箱的 tasks 数组中查找对应的任务；func 为同步等待任务对应函数的指针，arg 为指向函数参数的指针，在对同步等待任务进行唤醒时，通过这两个指针调用同步等待任务对应的函数，从而使同步等待任务开始运行。

```
void sync_task_add( sync_mailbox_t *p_mailbox, tid_t id,
                                    void (*func)( void * ), void *arg );
```

（3）task_sync()。该函数用于唤醒对方内核上的同步等待任务，其函数原型如下。其中，参数 p_mailbox 指向同步等待任务所在的同步消息邮箱，id 为要唤醒的同步等待任务的 ID。该函数首先将同步消息邮箱中的 sync_task_id 设置为需要唤醒的同步等待任务的 ID，然后通过同步邮箱中指向核间中断资源的指针 intercore_int 触发内核 B 产生中断以作为同步信号。对方内核发生核间中断后进入中断处理函数，读取同步消息邮箱中的 sync_task_id 以获取被唤醒的同步等待任务的 ID，然后根据该 ID 从同步消息邮箱中查找对应的任务函数并运行。

```
void task_sync( sync_mailbox_t *p_mailbox, tid_t id );
```

4.4 核间共享资源互斥访问机制设计

为了保证多核处理器的各内核在使用共享资源时不产生竞争条件，设计了核间共享资源互斥访问机制，在该机制下，各内核通过互斥锁实现共享资源的有序访问，当多个内核同时访问共享资源时，只有获得锁的一方才能使用共享资源，而没有获得锁

的内核无法使用共享资源，从而保证同一时刻共享资源只被一个内核访问。

互斥锁的设计可以采用处理器的专用硬件指令，如比较与交换指令及测试并加锁指令等；也可以使用软件进行设计，如 Peterson 算法和 Dekker 算法。但由于 Peterson 算法核 Dekker 算法只适用于两个进程之间的互斥，不适用于本文所采用的三核处理器，因此本文采用了 TC297 提供的专用硬件指令——比较与交换指令（Compare and Swap，CAS）进行互斥锁的设计，该指令的典型定义如下。

```
int compare_and_swap( int *mem, int testval, int newval )
{
    int oldval;
    oldval = *mem;
    if ( testval == oldval )
        *mem = newval;
    return oldval;
}
```

CAS 指令首先用一个测试值（testval）对一个内存单元（*mem）进行检查，如果该内存单元的当前值等于测试值，则用一个新值（newval）与其进行交换，否则保持其原值不变。该指令返回内存单元的原值，因此当返回值与测试值相同时，表示内存单元的值与测试值发生了交换，否则表示内存单元的值没有与测试值发生交换。整个过程按原子操作执行，不受中断影响。基于 CAS 指令设计的互斥锁程序如下所示，互斥锁程序主要包含两个函数：getLock()和 releaseLock()，其定义如下：

（1）getLock()。该函数用于获取锁，其工作过程为：首先在 while 循环中执行 CAS 指令对应的函数 compare_and_swap()，将指针 lock 指向的内存单元的值与 0 进行比较，如果不相等则返回 0，相等则返回 1，compare_and_swap()函数会不断执行直到返回值为 0，此时退出 while 循环。退出循环后会再次检查该返回值，为 0 表示成功获得锁，函数返回 TRUE，若不为 0 则表示获取锁失败，函数返回 FALSE；

（2）releaseLock()。该函数用于释放锁，具体方法是将 lock 指向的内存单元的值设置为 0。

```
boolean getLock( int *lock )
{
    uint32 lockVal = 1;
    while( lockVal != 0 )
        lockVal = compare_and_swap( lock, 0, lockVal );
    if( lockVal == 0 )
        return TRUE;
    else
        return FALSE;
```

```
}
void releaseLock( int *lock )
{
      *lock = 0;
}
```

getLock()函数和 releaseLock()函数必须成对使用，否则会导致“死锁”的发生。

4.5 核间通信机制设计

本文采用共享内存与核间任务同步机制结合的方式实现核间通信，其中共享内存用于实现两内核之间的数据传递，而核间任务同步机制用于对进行数据通信的两个内核上的任务进行同步。一个内核与另一内核进行数据通信时，首先将数据写入到共享内存中，然后通过核间任务同步机制唤醒对方内核上的任务对共享内存中的数据进行读取，从而完成两个核之间的数据通信。

4.5.1 共享内存空间的创建

TC297 每个内核都有各自的私有内存 DRAM，用于存储数据和堆栈等。每个内核既可以访问自身的私有 DRAM，也可以访问其他内核的 DRAM，内核对私有 DRAM 的访问不需要经过总线，因此访问速度较快，无延迟；但对其他内核的 DRAM 访问需要经过总线，因此访问时存在一定的延迟。除了各内核的私有内存，TC297 片上还提供了公共内存 LMU（Local Memory Unit），各内核都可以通过总线对其进行访问。

由于各内核的私有内存能够被其他内核访问，因此可以作为共享内存使用，但这样会使内核的可用内存空间减小，容易导致堆栈溢出。因此，为了不影响内核的正常运行，共享内存的创建不使用私有内存，而是在 LMU 中创建内存空间用作共享内存。共享内存空间的创建通过编译器指令#pragma section 和链接器脚本（Linker Script）来完成。具体的创建过程如下：

（1）创建共享内存空间管理结构。本文通过定义类型为 shdmem_mang_t 的全局共享变量 g_shdmem 对共享内存进行管理，该类型的定义为如下。其中数组 data_field 的地址空间即为共享内存空间，其大小由宏 SHDMEM_SIZE 定义；size_total、size_remain 和 avail_addr 用于共享内存空间的管理，其中 size_total 表示整个共享内存空间的大小，size_remain 表示剩余共享内存空间的大小，avail_addr 表示当前可用内

存空间的起始地址。

```
typedef struct {
    uint32   size_total;
    uint32   size_remain;
    uint32   *avail_addr;
    uint32   data_field[SHDMEM_SIZE];
} shdmem_mang_t;
```

（2）通过#pragma section 指令将全局共享变量 g_shdmem 定义为一个数据段。#pragma section 是一种编译器指令，用于定义程序或数据的段属性，通过该指令进行数据段定义的程序如下所示。这里将 g_shdmem 变量的段名定义为“.shdmem_sec”，并定义了该段的可写权限和数据对齐长度。

```
#pragma section ".shdmem_sec" aw 4
shdmem_mang_t g_shdmem;
#pragma section
```

（3）在链接脚本中添加相应的脚本命令，使数据段.shdmem_sec 存储在 LMU 中。首先通过 MEMORY 命令定义 LMU 的内存空间地址和大小，然后用 SECTION 命令将数据段.shdmem_sec 存储到 LMU 中，从而在 LMU 中创建共享内存空间，相应的脚本如下所示。

```
MEMORY
{
    lmu (w!xp): org = 0xb0000000, len = 32K
}
SECTIONS
{
    .data_shdmem : FLAGS(awl)
    {
        . = ALIGN(4) ;
        *(.shdmem_sec)
        *(.shdmem_sec.*)
    } > lmu AT> pfls0
}
```

4.5.2 共享内存的获取

在进行核间通信时，首先需要通过函数 shdmem_get()获取共享内存，该函数的原型如下。其中，n_words 表示申请的内存空间大小，单位为字；返回值的类型为 shdmem_t。

```
shdmem_t shdmem_get( uint32 n_words );
```

该函数的返回值类型 shdmem_t 的定义如下。其中 lock 为互斥锁，用于防止两个

相互通信的内核同时对共享内存进行写入和读取而导致数据一致性错误，在进行共享内存访问时应先获取 lock，而访问结束后释放 lock；size 表示获取到的共享内存的大小；pdata 指向获取到的共享内存的起始地址。

```
typedef struct
{
    uint32    lock;
    uint32    size;
    uint32    *pdata;
} shdmem_t;
```

函数 shdmem_get()的工作过程为：首先将参数 n_words 与共享内存空间管理结构中的 size_remain 进行比较，判断当前申请的共享内存大小是否超过剩余可用大小，若超过剩余可用大小，则无法成功获取共享内存，此时将返回值中的 size 设置为 0 并返回，表示共享内存获取失败；若不超过剩余可用大小，则首先设置返回值中的 lock 为 0，即互斥锁的初始状态设置为释放状态，然后设置 size 为所获取的内存大小，将 pdata 的值设置为共享内存空间管理结构中的 avail_addr，即当前可用内存空间的起始地址。在函数返回之前还需要对共享内存空间的管理变量 g_shdmem 进行更新，包括其中的当前可用地址 avail_addr 和剩余空间大小 size_remain。

4.5.3 核间通信的同步

为了保证核间通信的实时性，当发起通信的内核向共享内存中写入数据后，需要对方内核立即对数据进行读取，即需要对两个内核之间的通信进行同步。核间通信的同步通过核间任务同步机制来实现，具体过程如下：

（1）首先获取合适大小的共享内存用于核间数据传递；

（2）将数据接收内核上的数据处理任务添加到任务同步消息邮箱中，并将该任务函数的参数设置为步骤（1）中所获取的共享内存的起始地址；

（3）将数据写入到共享内存，并通过核间任务同步机制唤醒对方内核上的数据处理任务，使其对数据进行读取和处理。

4.6 本章小结

本章针对自动驾驶任务在多核处理器上的实时运行的问题和多核协同运行问题，

进行了多核处理器基础软件的设计。首先进行了多核处理器总体软件架构的设计，然后根据总体软件架构，在处理器上移植了实时操作系统 FreeRTOS，以实现自动驾驶任务的实时调度和运行；为了使多核能够协同运行，采用核间中断设计了核间任务同步机制，采用处理器硬件指令设计了核间共享资源互斥访问机制，采用共享内存并结合核间任务同步机制设计了核间通信机制，为多核的协同运行提供了必要条件。

第 5 章 双冗余系统软件设计

在控制器双冗余硬件架构的基础上，双冗余系统软件主要用于实现控制器的冗余功能，进而实现 Fail-Operational 的功能安全目标。

5.1 冗余策略总体设计

本文设计了双系统冗余的自动驾驶控制器，其冗余策略的总体设计如下：

（1）在正常情况下，控制器工作在双系统协同运行模式，此时互为冗余的两系统同时运行，具有较强的算力，因此运行功能较为复杂的控制任务。两系统之间通过 HSSL、SPI 或 CAN 进行数据通信，并通过 ERU 实现相互的工作状态监视，以判断对方是否发生故障，从而及时进行相应的故障处理。

（2）当某一系统发生故障时，控制器切换到单系统运行模式，此时只有一个系统能够正常运行，计算能力降低，因此控制器进行功能降级，以保证车辆控制的安全性。

5.2 系统状态信息的管理

双冗余控制器的两个系统在运行时需要读取和记录系统状态信息，如本系统是否失效、对方系统是否失效及本系统的复位次数等信息，系统需要根据这些信息进行相应的决策，以保证系统的正常工作。

5.2.1 状态信息管理数据结构

为了记录系统的状态信息，设计了如下所示的数据结构。

```
typedef struct
{
    uint8       is_master;
    boolean     sys_failure;
    uint8       failure_mode;
    boolean     cpsys_failure;
    uint32      abn_resets;
    uint32      total_resets;
} sys_states_t;
```

其中 is_master 为主系统标志，系统根据该标志判断自身为主系统还是从系统，在进行系统同步时，主系统负责同步信号的发起，而从系统则等待接收同步信号；sys_failure 为本系统失效标志，用于标识系统自身是否失效，当系统检测到自身发生 CAN 通信电路失效、I/O 模块失效或异常复位故障等导致的系统失效时，通过设置此标志表示本系统已失效，使系统下一次启动时通过该标志获知系统失效的发生，从而将系统挂起而停止运行，以免对控制器的正常工作产生影响；failure_mode 用于记录本系统的失效模式，便于对失效系统进行失效分析；cpsys_failure 为对方系统失效标志，用于标识对方系统是否发生失效，当检测到对方系统发生失效时将该标志置位，从而使系统下一次启动时可以直接通过该标志获知对方系统的失效情况并进行相应的决策；abn_resets 为异常复位计数，用于记录系统的异常复位次数，而 total_resets 为总复位计数，用于记录系统的总复位次数，通过这两个计数值可以计算系统异常复位的频率，从而评估系统工作的稳定性。

5.2.2 状态信息的存储与更新

系统在控制器上电启动时需要获取上一次运行时的状态信息，从而确定系统的主从身份及系统失效标志等。如果使用全局变量记录状态信息，则会由于全局变量存储在 RAM 中而导致信息随着控制器的断电而全部丢失。为了使系统的状态信息能够在控制器断电后保存下来，需要将这些状态信息存储在非易失性存储器中。

TC297 提供了非易失性存储器 DFlash 用于存储数据，将系统状态信息存储在 DFlash 中可以保证这些信息不会在控制器断电后丢失。在系统状态信息的管理上，系统启动时从 DFlash 中读取状态信息到 RAM，以便于状态信息的修改，而当系统状态发生改变时，将新的系统状态写回到 DFlash 中，以保证所记录的系统状态始终为最新状态。状态信息的管理流程如图 5.1 所示。

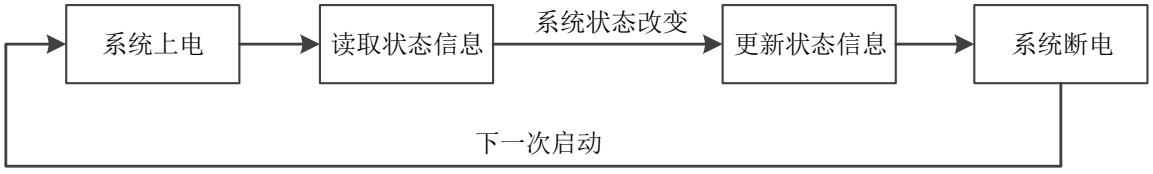


图 5.1 系统状态信息的管理流程

5.2.3 状态信息的初始化

在控制器首次运行之前，需要对两个系统的状态信息进行初始化，以保证系统初始状态的正确性，状态信息的初始值通过 DFlash 驱动程序进行设置。其中，主从系统标志 is_master 的初始值设置需要保证两个系统中一个为主系统，另一个为从系统，而不能使两系统身份相同，从而使两个系统具有正确的主从关系；本系统失效标志 sys_failure 和对方系统失效标志 cpsys_failure 均初始化为 0，表示本系统和对方系统状态正常；系统失效模式设置为未失效；系统首次运行前没有复位发生，因此异常复位计数值 abn_resets 和总复位计数值 total_resets 都初始化为 0。通过以上设置，可以确保控制器首次上电后从正确的初始状态开始工作。

5.3 主从系统之间的同步

当两个系统状态均正常时，控制器可以工作在双系统协同运行模式下，但两个系统在协同运行之前需要先进行同步，以防止两系统因开始运行的时间不同而导致发生故障误判。例如一个系统开始运行并执行故障检测程序时，另一系统尚未开始运行，则前者会检测到后者的非运行状态并认为后者发生了故障，于是产生故障误判的情况。因此设计了同步程序对两系统的运行进行同步，以保证控制器正常工作。两系统之间的同步通过同步信号来完成，并在同步过程中通过相应的协议保证同步的正确性。

5.3.1 同步信号的传递

在双冗余控制器的电路设计上，为两个系统之间提供了 3 种通信连接，分别为 HSSL、SPI 和 ERU 。其中 HSSL 为高速通信接口，其通信协议分为物理层、数据链路层和传输层，能够在处理器之间进行大量数据的可靠传输；SPI 为串行通信接口，能够在处理器之间进行全双工高速通信；ERU（External Request Unit）为外部请求单元，通过 ERU 可以对 I/O 引脚进行电平边沿检测，当 I/O 引脚产生相应类型的电平边沿时会触发 ERU 中断，从而可以通过控制 I/O 引脚电平触发对方系统产生 ERU 中断的方式实现两系统间的信号通信。

HSSL 和 SPI 主要用于大量数据的传输，通信协议相对复杂且实时性不高；而采

用 ERU 时，两系统之间通过硬线连接，且信号的传输不需要通信协议，因此具较高的实时性和可靠性，并且实现上也较为简单。因此这里采用 ERU 实现两系统之间的同步信号传输。

5.3.2 同步协议设计

本文设计了如图 5.2 所示的同步协议。在该协议中，主系统与从系统之间的信号分为握手信号和同步信号，两种信号分别采用不同的电平边沿类型。握手信号主要用于检查两系统之间能否正常进行信号的传输；同步信号用于两系统之间的同步，该信号是系统开始执行任务的标志信号。

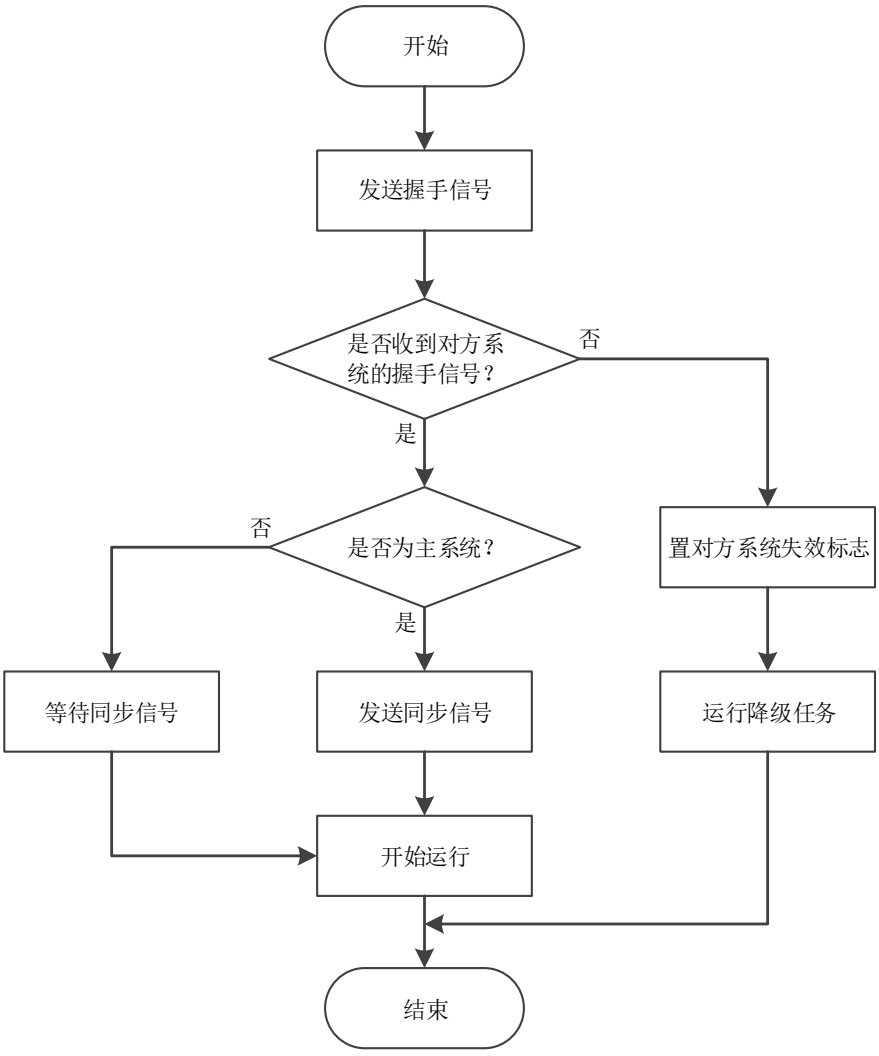


图 5.2 主从系统同步协议

双冗余控制器启动后，主系统与从系统相互发送握手信号，以判断双方能否正常进行信号传输，如果两系统均收到了对方系统的握手信号，则说明主系统与从系统之

间能够正常进行信号传输，此时主系统向从系统发送同步信号并开始执行任务，而从系统收到主系统的同步信号后也开始执行任务；如果主系统没有收到从系统的握手信号，则说明从系统发生故障，此时需要将对方系统失效标志 cpsys_failure 置位并对控制器进行功能降级，以保证车辆控制的安全性；同样地，如果从系统没有接收到主系统的握手信号，则说明主系统发生故障，此时同样需要将对方系统失效标志 cpsys_failure 置位并对控制器进行功能降级。

5.4 故障的检测与处理

控制器的故障检测分为系统间的故障检测和系统内部的故障检测，系统间的故障检测主要针对导致对方系统处理器工作异常的故障，如处理器硬件故障；而系统内部的故障检测主要针对处理器外围电路的故障，以及处理器可以自己检测到的故障，如处理器异常复位故障。通过两种故障检测方式的结合，能够较为全面地检测控制器的故障并进行相应的处理。

5.4.1 系统之间的故障检测与处理

系统间的故障检测主要用于检测对方系统是否发生了处理器硬件故障、电源供电故障及处理器异常复位故障等。为了使两系统之间能够相互进行故障检测，设计了基于“心跳信号”的故障检测机制。“心跳信号”是指两系统之间周期性向对方发出的用于表示系统正常运行的信号，当一个系统无法检测到对方系统的心跳信号时，可以判定对方系统发生了故障。两系统之间通过控制 I/O 引脚电平触发对方系统产生 ERU 中断的方式实现心跳信号的发送和接收。

基于“心跳信号”的故障检测机制由三个功能模块实现，包括心跳信号发送模块、心跳信号接收模块和心跳信号检测模块。心跳信号发送模块周期性向对方系统发送心跳信号并计数，每发送一次心跳信号，将发送计数值增加一；心跳信号接收模块接收对方系统的心跳信号并计数，每次接收到心跳信号后，将接收计数值增加一；心跳信号检测模块周期性检测对方系统的心跳信号是否正常，并判断对方系统是否发生故障，其工作原理为：两系统经过同步后同时开始运行，且按照相同的周期相互发送心跳信号，因此同一时刻系统发送和接收到的心跳信号次数应该是相同的，如果接收计数与

发送计数相等，说明对方系统正常运行；如果检测到心跳信号的接收计数小于发送计数，则说明对方系统发生了故障。

为了提高信号传输的可靠性，防止因单信号传输通道失效而导致发生故障误判，采用了双通道冗余的方式进行心跳信号的传输。采用双传输通道时，心跳信号发送模块需要同时通过两路通道发送心跳信号并分别计数；同样地，心跳信号接收模块也需要同时对两路通道接收到的心跳信号分别进行计数；而心跳信号检测模块则需要通过两路通道的心跳信号发送和接收计数判断对方系统心跳信号是否正常，从而判断对方系统是否发生故障。心跳信号检测模块的的工作流程如图 5.3 所示。首先对传输通道 1 的心跳信号发送和接收计数进行比较，若相等，则说明对方系统正常运行，并进入下一个检测周期；否则说明对方系统可能发生故障，但具体情况需要进一步根据通道 2 的心跳信号计数进行判断，如果通道 2 的心跳信号发送和接收计数相等，则说明对方系统正常运行，于是进入下一个检测周期；否则说明对方系统发生了故障，于是进入相应的故障处理程序。

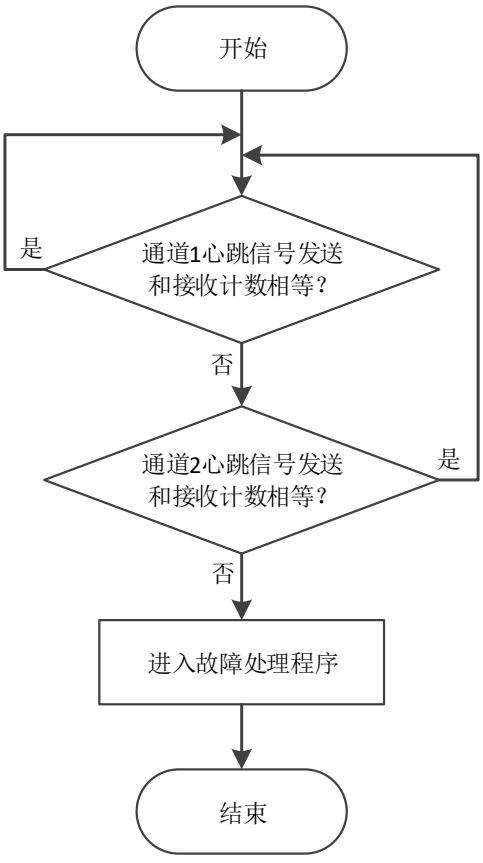


图 5.3 心跳信号检测程序工作流程

故障处理程序的主要工作包括控制器的功能降级及相关系统状态信息的更新等，

具体工作流程如图 5.4 所示。当检测到对方系统发生故障时，首先置对方系统失效标志 cpsys_failure 并立即切换到降级任务，从而实现控制器的功能降级，保证车辆安全。在降级任务运行的过程中，需要检测对方系统是否从故障中恢复正常，以应对对方系统的故障模式为处理器异常复位的情况。处理器异常复位故障发生的原因主要是程序跑飞，在处理器复位重启后，系统通常能够正常运行，因此当对方系统发生故障后，需要进一步判断其是否发生了处理器异常复位故障并且从故障中恢复正常，如果对方系统恢复正常，则将对方系统失效标志 cpsys_failure 复位，从而使控制器下一次上电启动时能够恢复双系统协同运行模式；否则说明对方系统发生了永久性失效，此后控制器只能工作在单系统运行模式下。

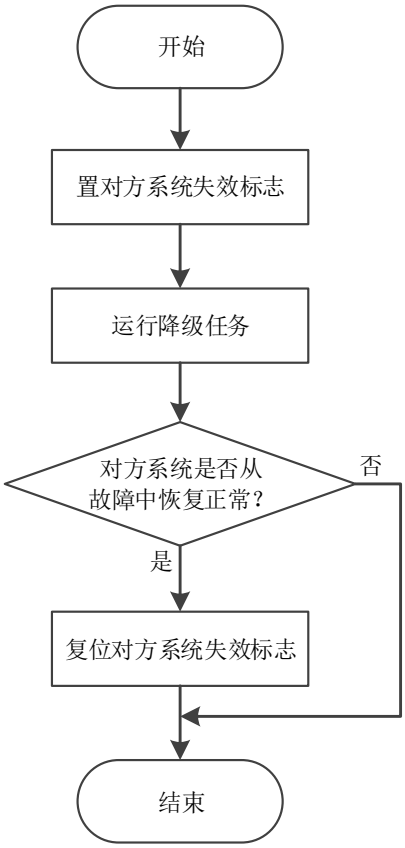


图 5. 4 故障处理程序工作流程

5.4.2 系统内部故障的检测与处理

系统内部故障的检测主要针对处理器异常复位故障，可以通过读取相关状态寄存器的值检测系统是否发生了处理器异常复位故障，TC297 的复位状态寄存器 RSTSTAT 记录了处理器最近一次复位的类型，通过该寄存器的值可以判断处理器的本次复位是

否为异常复位。

处理器发生异常复位并重启后通常能够正常运行，因此一般不会导致系统失效，但如果处理器频繁发生异常复位，则会导致整个系统无法稳定运行，从而给控制器的运行带来很大的不稳定性，此时需要将处理器异常复位频率过高的情况当做系统失效来处理，相应的故障处理程序如图 5.5 所示。

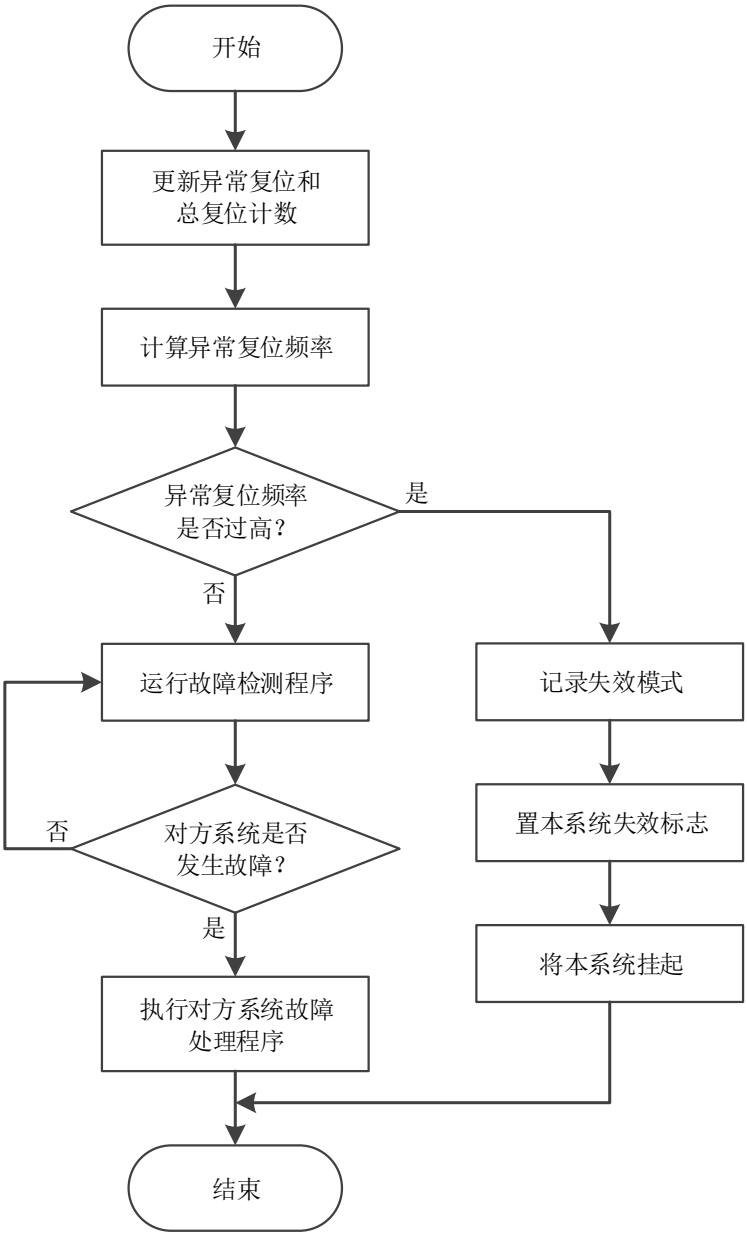


图 5.5 异常复位处理程序工作流程

发生处理器异常复位故障的系统重启并检测到自身发生了异常复位后，首先对系统状态中的异常复位计数 abn_resets 和总复位计数 total_resets 进行更新，然后根据这两项计数值计算出处理器异常复位的频率，进而评估系统运行的稳定性。如果异常复

位频率在正常范围内，则本系统仍可以正常运行，此时运行故障检测程序，一方面，重新开始向对方系统发送心跳信号，以表示本系统已经从处理器异常复位故障中恢复正常，当对方系统检测到本系统发出的心跳信号后会将相应的系统失效标志复位，从而使控制器下一次上电启动时恢复双系统协同运行模式；另一方面，作为当前正在运行的对方系统的备份，一旦检测到对方系统发生故障，立即执行故障处理程序，并接替对方系统运行，从而使控制器正常工作。若处理器异常复位频率过高，则将本系统当做失效进行处理，首先设置本系统的失效模式 failure_mode 为异常复位失效，然后置本系统失效标志 sys_failure，使系统下一次启动时通过该标志得知本系统已失效，从而进入挂起状态，最后使本系统进入挂起状态并停止运行。

除了异常复位故障，系统内部故障检测还针对外围电路故障，如 I/O 模块及 CAN 通信模块等的故障。其中 I/O 模块的故障可以通过读取 I/O 控制芯片 TLE8108 的故障诊断相关的寄存器进行检测；CAN 通信模块的故障检测主要针对断线故障，可以通过测试 CAN 模块能否正常监听总线消息进行检测。当系统发生以上两种故障时，相应的功能模块无法正常工作，从而导致系统因失去相应的功能而失效，因此对应的故障处理程序的主要工作为：首先设置相应的失效模式 failure_mode，然后置本系统失效标志 sys_failure，最后使本系统进入挂起状态并停止运行。

5.5 双冗余系统整体工作流程

双冗余控制器中两个系统的总体工作流程如图 5.6 所示。在系统上电复位时，首先检查处理器本次复位是否为异常复位，从而判断本系统是否发生了处理器异常复位故障，若本系统发生了异常复位故障，则执行相应的故障处理程序；否则系统正常启动并根据系统失效标志 sys_failure 检查本系统是否失效。若本系统失效，则进入挂起状并停止运行；否则进一步根据对方系统失效标志 cpsys_failure 检查对方系统是否失效。若对方系统失效，则此时控制器中只有一个系统能够正常运行，于是运行降级任务对控制器进行功能降级；若对方系统未失效，则说明两系统状态均正常且能够以双系统协同运行模式工作。在进入双系统协同运行模式之前，对两系统进行同步，若同步成功，则两系统开始协同运行；否则说明某一系统发生故障，于是对控制器进行功能降级。在双系统协同运行模式中，两系统始终相互监视对方系统的运行状态，一旦检测到对方系统发生故障，立即运行相应的程序对故障进行处理。

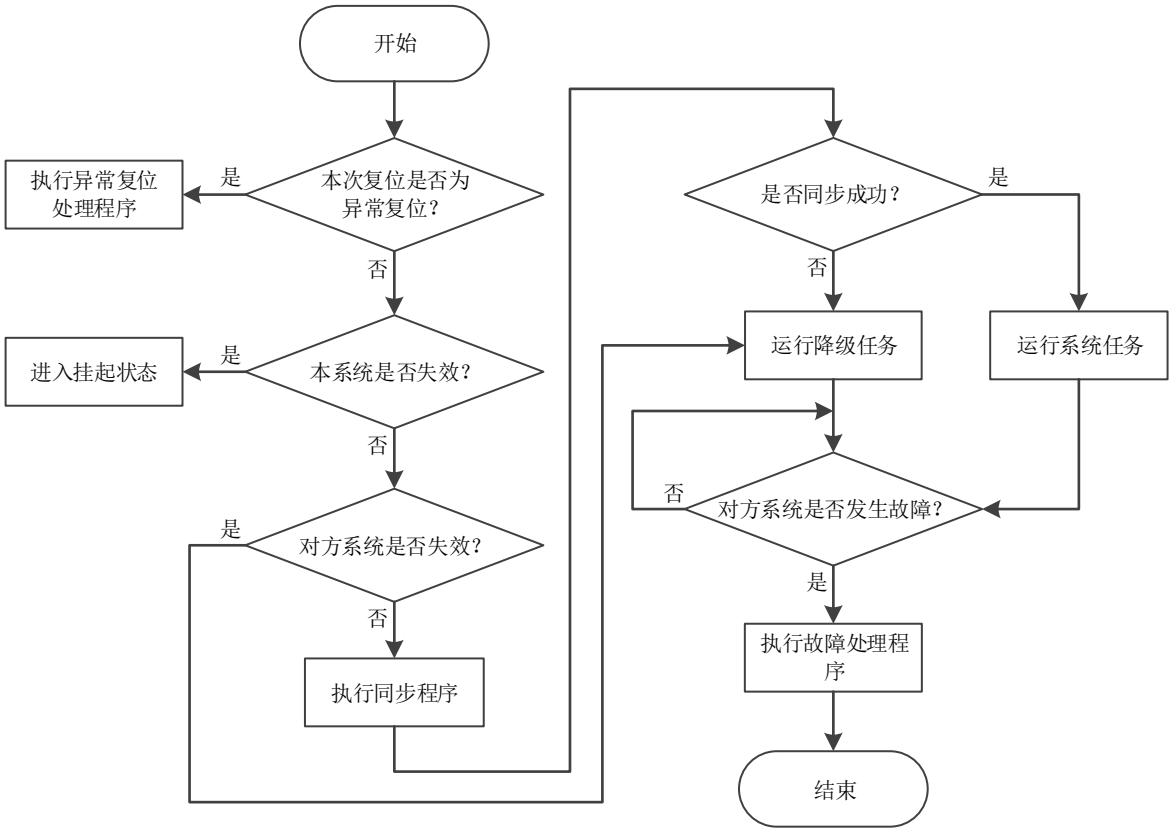


图 5.6 双冗余控制器整体工作流程

5.6 本章小结

本章进行了双冗余系统的软件设计，以实现控制器的冗余功能和 Fail-Operational 的功能安全目标。首先对双冗余控制器的总体冗余策略进行了设计，然后设计了系统的状态信息管理程序。针对双系统协同运行模式下两系统之间的同步需求，采用 ERU 实现了系统之间的信号通信，并在此基础上设计了基于握手信号和同步信号的同步协议，实现了两系统之间的同步；针对控制器的主要故障模式如处理器硬件故障、系统供电故障、处理器异常复位故障、CAN 通信故障及 I/O 模块故障等，设计了控制器的故障检测与处理机制，包括系统之间以及系统内部的故障检测方法设计及相应的处理程序设计；最后，综合上述设计的各功能模块，对双冗余控制器的整体工作过程进行了总结。

第 6 章 软件架构实车应用与测试

6.1 软件架构实车应用介绍

本文根据实验室与国内某乘用车公司合作的 L3-级自动驾驶系统开发项目的实际需求，进行了基于多核处理器的双冗余自动驾驶控制器的硬件设计和软件架构开发，并应用于实车，如图 6.1 所示。该项目在 L3-级自动驾驶的实现上，采用 5R1V 作为传感器配置方案，即采用 5 个毫米波雷达（包括 1 个前向雷达和 4 个角雷达）和 1 个前视摄像头作为传感器，并通过布置在背舱的自动驾驶控制器进行传感融合、运行分析、场景分析、行为决策、轨迹规划和运动控制等工作。

图 6.1 实验车及自动驾驶控制器

本文所设计的自动驾驶控制器及软件架构在实车上的具体应用如下：

（1）Bootloader。通过将 Bootloader 程序固化到自动驾驶控制器中，使得控制器能够通过 CAN 总线进行应用程序的在线下载与升级。如图 6.2 所示，将 PC 通过 USB-CAN 卡接入 CAN 总线，即可使上位机软件通过 CAN 总线与控制器中的 Bootloader 进行通信，从而将应用程序下载到控制器中，实现应用程序的升级。

（2）多核处理器及其基础软件。为了充分发挥多核处理器的性能优势，在实车应用中根据自动驾驶任务的类型，将任务在不同的内核上进行了分配，如图 6.3 所示。其中计算核 Core1 裸机运行，主要负责采集雷达和摄像头的传感数据，并根据传感数据进行感知融合、运动分析和场景分析，从而得到周围目标物信息，然后通过核间通信机制将目标物信息发送给计算核 Core2。由于雷达和摄像头的工作周期均为 50ms，

因此 Core1 上的任务以 50ms 为周期运行；计算核 Core2 裸机运行，主要负责根据周围目标物信息进行行为决策和运动规划，从而得出运动规划信息，然后通过核间通信机制将运动规划信息发送给控制核 Core0。由于 Core2 任务的运行需要 Core1 任务得出的数据，因此 Core2 任务和 Core1 任务之间存在同步关系，二者之间的同步采用核间任务同步机制来实现；Core0 搭载实时操作系统，主要负责根据运动规划信息，以 10ms 为周期对整车进行实时运动控制。此外 Core0 还负责自动驾驶功能状态机的管理，并在控制器进行功能降级时负责 L2 级自动驾驶功能的运行。

图 6.2 上位机软件与控制器的通信连接

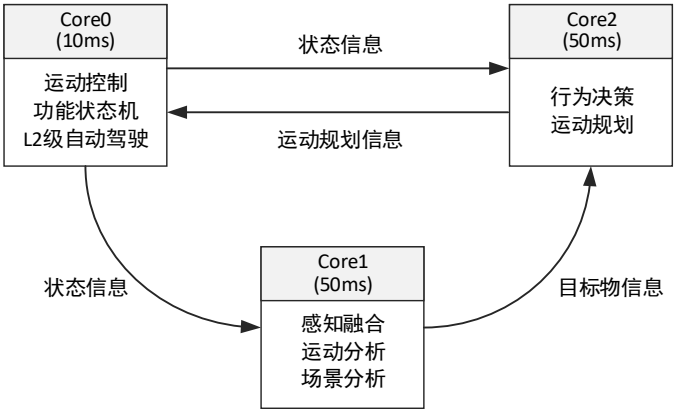


图 6.3 多核处理器各内核的任务分配

（3）双冗余系统。双冗余控制器中的两个系统互为冗余，通过本文设计的故障检测及处理程序实现故障的实时检测与故障发生时的系统切换，提高了控制器的功能安全。在实车应用上，两系统都通过各自的 CAN 通信接口接入到 CAN 总线，且均接收总线信息并运行 L3-级自动驾驶程序，但只有主系统向总线发送车辆控制信号，而备用系统关闭控制信号输出。当主系统发生故障而需要备用系统代替其对车辆进行控制

时，备用系统只需开启控制信号输出，即可迅速接管车辆的控制。

6.2 软件架构测试环境

软件架构的测试环境如图 6.4 所示，测试所用到的主要设备及软件包括：示波器、USB-CAN 卡、PC 端上位机软件 iLoader 和 CAN Master。

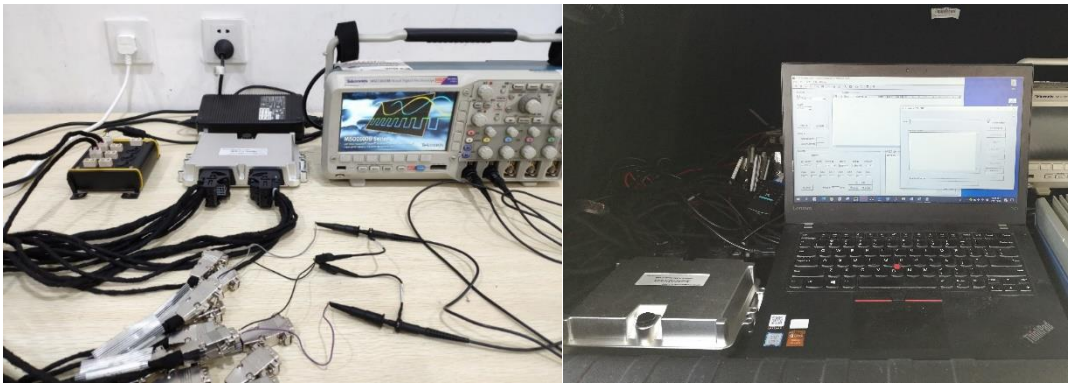


图 6.4 软件架构测试环境

示波器主要用于测量控制器的数字量输出（Digital Output，DO）通道的电平信号，从而观察控制器相关任务的执行过程，且由于示波器能够精确地对信号进行时间测量，因此可以用于测量信号的周期以及不同信号之间的时序关系。本文采用的示波器为 Tektronix 公司的 MSO2024B，如图 6.5 所示。该示波器具有 4 路模拟信号通道和 16 路数字信号通道，模拟带宽 200 MHz，采样率 1 GS/s，记录长度为 1 M 点。

图 6.5 MSO2024B 示波器

USB-CAN 卡用于 USB 信号与 CAN 信号之间的信号类型转换，通过 USB-CAN 卡可以使控制器与 PC 端通过 CAN 总线进行通信，本文所用的 USB-CAN 卡为广州致远科技的 USBCAN-I-mini，如图 6.6 所示。该 USB-CAN 卡具有 1 路 CAN 总线接口，符合 CAN2.0A/B 规范，支持 5Kbps~1Mbps 之间的任意波特率，单通道最高数据流量为：接收时 14000 帧/秒，发送时 3000 帧/秒。

图 6.6 USBCAN-I-mini 型 USB-CAN 卡

iLoader 是一款 Bootloader 上位机软件，主要用于应用程序可执行文件的下载，其使用界面如图 6.7 所示。该软件能够对应用程序可执行文件进行解析，并通过 USB-CAN 卡与 Bootloader 进行通信，将可执行文件发送给 Bootloader，从而使 Bootloader 完成控制器应用程序的下载和升级。

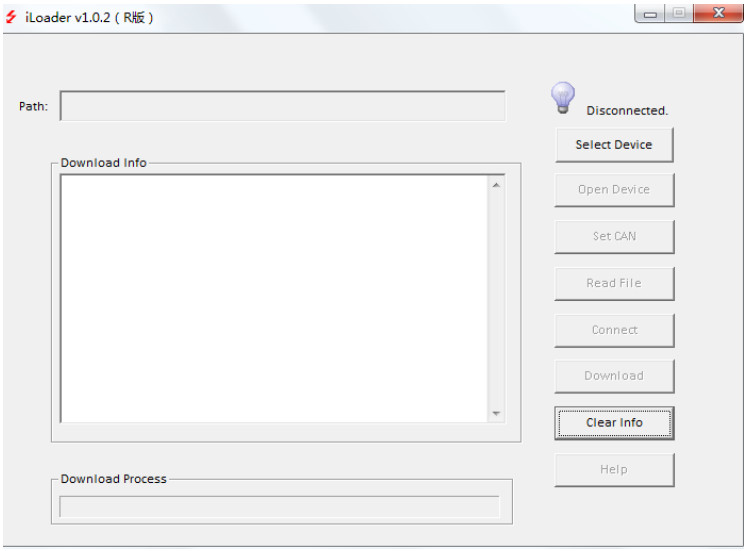


图 6.7 上位机软件 iLoader

CAN Master 是一款能够用来接收、发送、分析和记录 CAN 报文的上位机软件，支持 USBCAN-I-mini 型 USB-CAN 卡，如图 6.8 所示。本文主要使用该软件接收和记录控制器发出的 CAN 报文，从而获得控制器的工作信息。

在通信连接和信号线路连接方面，首先通过 USB-CAN 卡将 PC 端的 USB 接口与控制器的 CAN 通道进行连接，从而使控制器与上位机软件之间能够进行通信。然后将示波器探头与控制器的 DO 通道进行连接，从而通过 DO 通道的电平状态和电平变化判断相应软件是否运行正确，以验证软件设计的正确性。

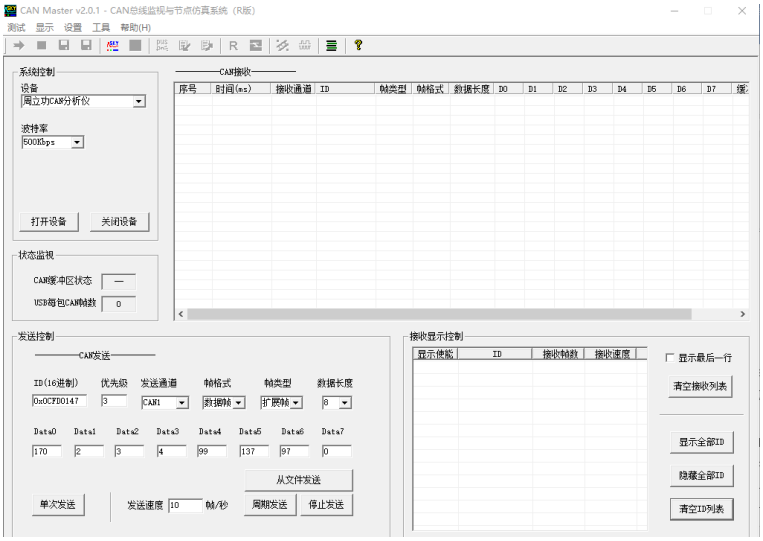


图 6.8 上位机软件 CAN Master

6.3 Bootloader 测试

Bootloader 的测试主要验证：（1）Bootloader 能否正确地进行系统设置并引导各内核正常启动；（2）能否与上位机软件 iLoader 建立通信连接并下载应用程序；（3）在完成系统启动或下载工作后，能否正确跳转到应用程序入口并运行应用程序。测试的具体步骤如下：

（1）设计如下应用程序作为测试用例：内核 Core0、Core1 和 Core2 进入各自的主函数后，分别以各自的内核 ID 作为 CAN 报文的 ID 和数据，通过 CAN 通信接口发送报文到上位机软件 CAN Master，以表示内核正常启动和运行。完成应用程序的设计后进行编译，生成 Srec 格式的应用程序可执行文件；

（2）打开上位机软件 iLoader 进行应用程序的下载。首先点击“Select Device”选择 USB-CAN 卡的型号，然后点击“Open Device”启用 USB-CAN 卡，接着点击“Set CAN”设置 CAN 通信的波特率为 500kbps，保证与 Bootloader 的 CAN 模通信波特率相同，然后点击“Read File”选择步骤（1）中生成的应用程序可执行文件，点击“Connect”申请与 Bootloader 建立连接，并将控制器上电。Bootloader 在检测到 iLoader 发出的连接请求后会与其建立通信连接，如果通信连接建立成功则 iLoader 会显示“Connect Success”，然后点击“Download”开始下载应用程序。

（3）应用程序下载完成后打开 CAN Master 并查看是否同时接收到 3 个内核发出的报文，从而判断 Bootloader 是否正确实现了相应的功能。

经过以上测试步骤得到了如图 6.9 和图 6.10 所示的测试结果。从 iLoader 软件的信息提示栏中可以看到，从读取可执行文件到建立通信连接，再到应用程序的下载，整个过程顺利完成，说明 Bootloader 成功进行了应用程序的下载。从 CAN Master 接收到的信息可以看到，3 个内核均通过控制器的 CAN 通信接口正确地发送了报文，说明 Bootloader 正确地完成了系统启动设置和 3 个内核的启动引导，并在应用程序下载完成后，实现了 Bootloader 到应用程序的跳转和应用程序的执行，证明了 Bootloader 设计的正确性。

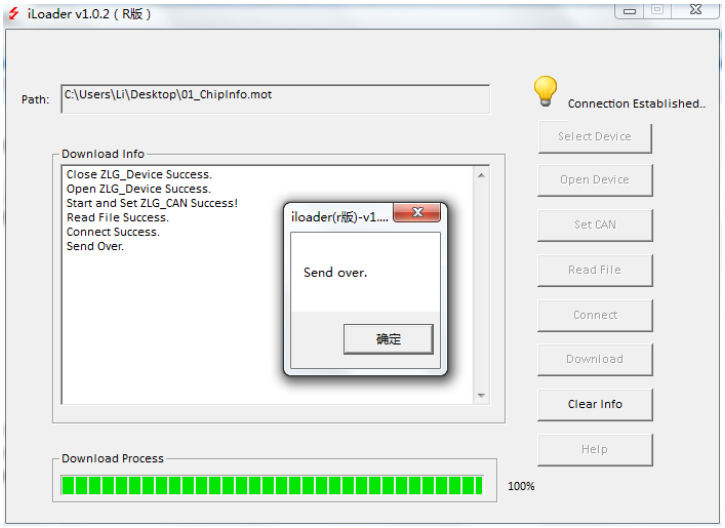


图 6.9 iLoader 下载结果

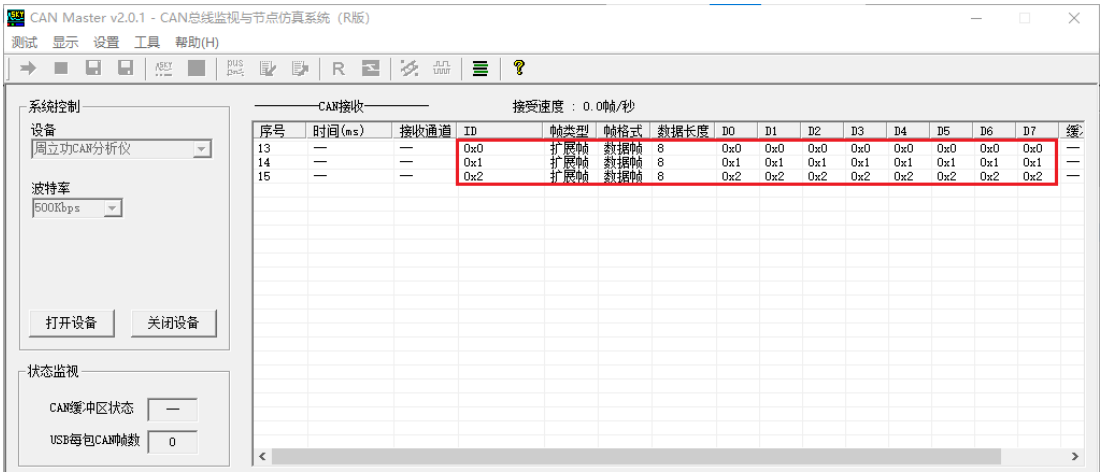


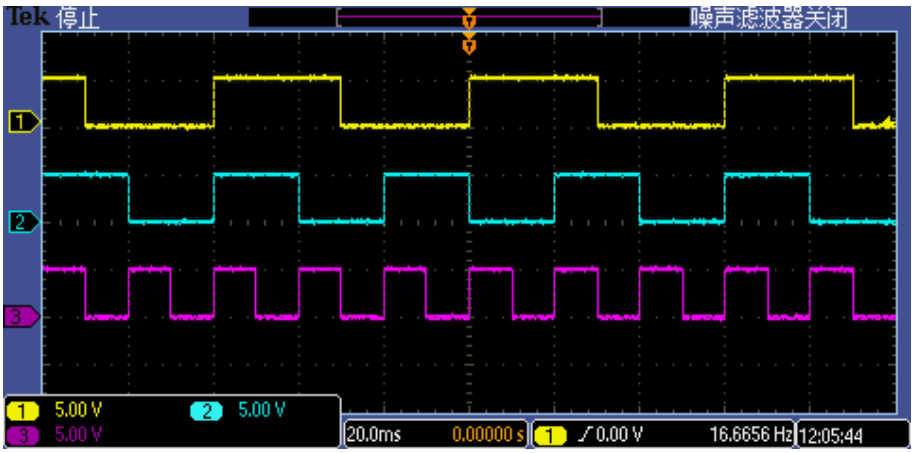
图 6.10 CAN Master 接收到的信息

6.4 多核处理器基础软件测试

6.4.1 FreeRTOS 移植测试

FreeRTOS 测试主要验证 FreeRTOS 实时多任务调度功能和基于优先级的抢占式任务调度功能的正确性，进而验证 FreeRTOS 是否移植成功。

首先进行实时多任务调度功能测试，测试方法为：通过任务创建函数 xTaskCreate() 创建 3 个周期分别为 60ms、40ms 和 20ms 的任务，每个任务周期性地对相应的 DO 通道进行电平翻转。通过示波器对各任务对应的 DO 通道电平进行测量，从而判断 3 个任务是否按设定周期实时运行，最后得到的测试结果如图 6.11 所示，其中信号 1 为 60ms 任务对应 DO 通道的电平信号，信号 2 为 40ms 任务对应 DO 通道的电平信号，信号 3 为 20ms 任务对应 DO 通道的电平信号。



6.11 实时多任务调度测试结果

通过示波器的测量功能，分别对 3 个任务对应 DO 通道的电平信号进行周期测量，如图 6.12、6.13 和 6.14 所示。从图中可以看到三个信号的周期分别为 60ms、40ms 和 20ms，与设定的任务周期相同，说明 FreeRTOS 成功进行了多任务调度，且各任务的运行具有良好的实时性，验证了 FreeRTOS 多任务实时调度功能的正确性。

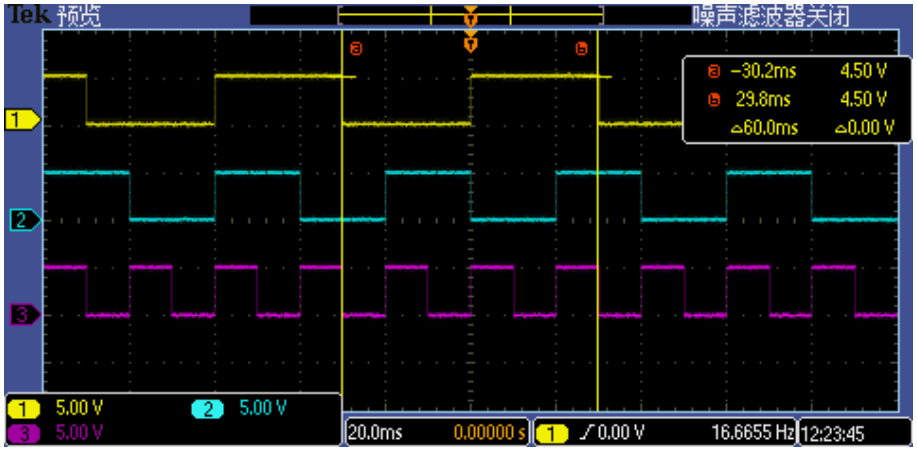


图 6.12 60ms 任务对应 DO 通道电平信号的周期测量结果

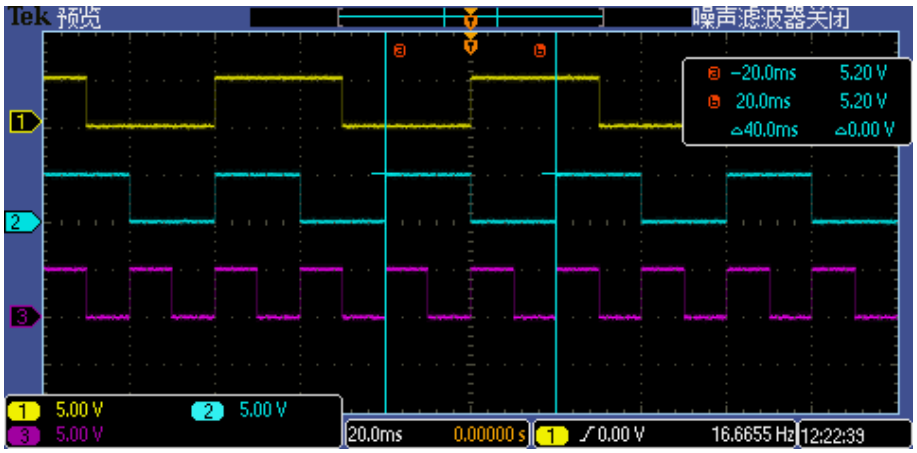


图 6.13 40ms 任务对应 DO 通道电平信号的周期测量结果

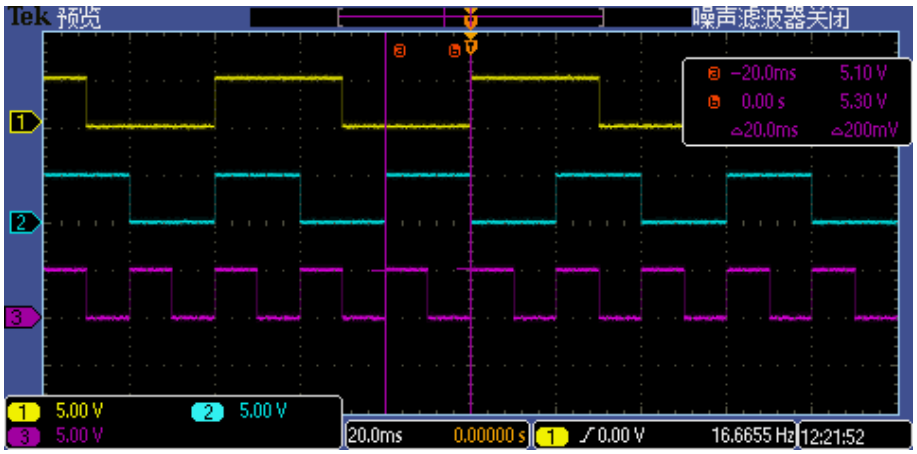


图 6.14 20ms 任务对应 DO 通道电平信号的周期测量结果

接着进行基于优先级的抢占式任务调度功能测试，测试方法为：通过 xTaskCreate() 函数创建优先级从高到低分别为 3、2 和 1 的周期任务，任务周期分别为 30ms、20ms 和 10ms，每个任务周期性对相应 DO 通道进行电平翻转。其中优先级最高的任务在将对应 DO 通道电平拉高后，通过 nop 指令执行 20ms 的空操作，然后将 DO 通道电平

拉低，并通过 vTaskDelayUntil()函数将自身挂起，直到下一个运行周期到来时再继续运行。通过示波器对各任务对应的 DO 通道电平进行测量，从而观察不同优先级任务之间是否发生抢占。最后得到的测试结果如图 6.15 所示，其中信号 1 优先级为 3 的任务对应 DO 通道的电平信号，信号 2 优先级为 2 的任务对应 DO 通道的电平信号，信号 3 优先级为 1 的任务对应 DO 通道的电平信号。

从图中可以看出，优先级最高的任务按照设定的周期正常运行，而优先级为 1 和 2 两个任务均没有按照设定周期执行电平翻转操作，且最高优先级任务对应的 DO 通道电平为高时，两个低优先级任务均没有执行电平翻转操作，而只有当最高优先级任务对应 DO 通道电平为低时，两个低优先级任务才执行了电平翻转操作。原因是当两个低优先级的任务在到达运行周期并准备执行电平翻转操作时，内核使用权被最高优先级的任务所抢占，且最高优先级的任务通过运行 nop 指令一直占用内核的使用权，当最高优先级任务将对应 DO 通道电平拉低并释放内核使用权后，两个低优先级任务才获得内核使用权并执行 DO 通道电平翻转操作，由此说明基于优先级的抢占式任务调度功能正确运行。

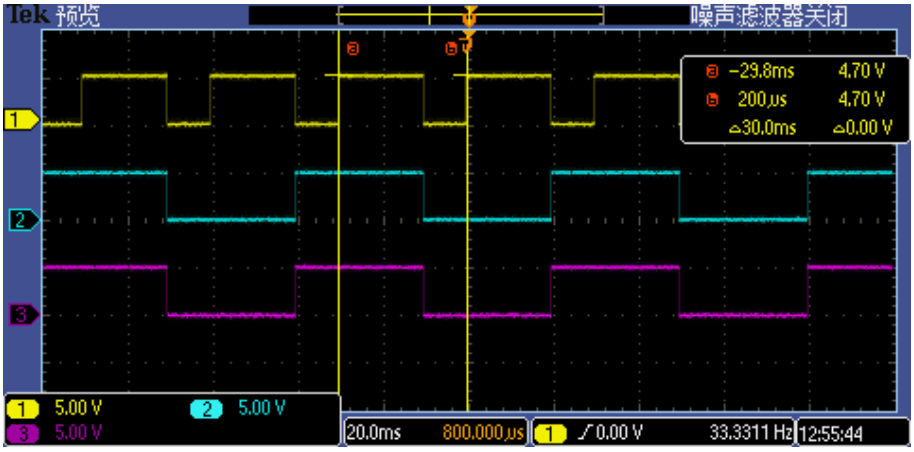


图 6.15 基于优先级的任务抢占功能测试结果

通过以上测试，验证了 FreeRTOS 实时多任务调度和基于优先级的抢占式任务调度功能的正确性，证明 FreeRTOS 移植成功。

6.4.2 核间通信及任务同步机制测试

该测试主要验证核间通信机制设计的正确性和核间任务同步机制的实时性，测试的具体步骤如下：

（1）编写测试用例，使内核 Core0 通过核间通信机制向内核 Core1 发送 4 个字节大小的数据，并通过核间任务同步机制唤醒 Core1 上的数据接收任务对数据进行读取。整个过程循环进行，且每次发送不同的数据。为了验证核间通信过程中数据传输的正确性和核间任务同步机制的实时性，Core0 在发送数据的同时，通过 CAN 将所发送的数据和系统定时器 STM0 的当前计数值封装成一条 CAN 报文发送至上位机软件 CAN Master。同样地，Core1 在接收到数据的同时，也通过 CAN 将所收到的数据和系统定时器 STM0 的当前计数值封装成一条 CAN 报文发送至上位机软件 CAN Master。在进行报文发送时，Core0 和 Core1 分别以各自的内核 ID 作为 CAN 报文 ID，从而对两个内核发送的报文进行区分。

（2）运行测试用例，并查看 CAN Master 接收到的报文，检查每一次 Core1 收到的数据与 Core0 发送到的数据是否相同，从而判断核间通信机制设计的正确性。同时，计算每一次 Core0 发送数据时的 STM0 计数值和 Core1 收到数据时的 STM0 计数值之差，即任务同步的响应时间，从而判断核间任务同步机制是否具有良好的实时性。

最后得到的测试结果如图 6.16 所示，其中 CAN 帧数据域的低 4 字节数据为 Core0 和 Core1 发送或接收的数据，而高 4 字节数据为 Core0 和 Core1 发送或接收数据时刻的 STM0 计数值。从测试结果可以看出，Core1 每次接收到的数据与 Core0 发送的数据相同，证明了核间通信机制设计的正确性。同时，通过计算可知，每一次从 Core0 发送数据到 Core1 收到数据之间的 STM0 计数值之差均为 66。系统定时器 STM0 的计数频率为 100MHz，因此可得从 Core0 发送数据到 Core1 收到数据的响应时间为 660ns，即同步响应时间为 660ns，证明核间任务同步机制具有良好的实时性。

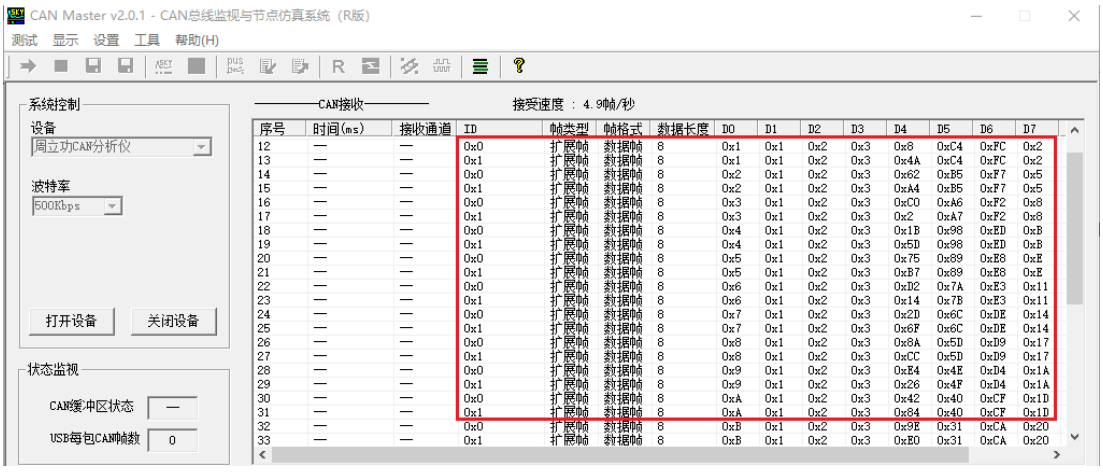


图 6.16 核间通信及任务同步机制测试结果

6.4.3 核间共享资源互斥访问机制测试

该测试主要验证核间共享资源互斥访问机制设计的正确性，测试的具体步骤如下：

（1）编写测试用例，使内核 Core0 和 Core1 同时对共享内存中的变量 counter 从 0 开始进行递增操作。测试分为对照组和实验组，在对照组中，Core0 和 Core1 在对 counter 进行递增操作时不使用互斥锁。而在实验组中，Core0 和 Core1 在对 counter 进行递增操作时使用互斥锁。为了观察变量 counter 的数值变化，2 个内核在完成递增操作后将 counter 的值通过 CAN 发送至上位机软件 CAN Master。

（2）运行测试用例，并查看 CAN Master 接收到的报文，比较对照组与实验组中变量 counter 的数值变化过程，从而判断核间共享资源互斥访问机制是否正确地实现了相应的功能。

两组测试的结果分别如图 6.17 和图 6.18 所示。从图 6.17 中可以看到，在不使用互斥锁时，counter 的数值没有正确地进行递增，其原因为：递增操作的过程为读取-修改-写回，在不使用互斥锁的情况下，Core0 和 Core1 能够同时访问变量 counter，此时 2 个内核同时读取到 counter 的初始值 0，然后分别将修改后的值 1 写回。因此，虽然 2 个内核对变量 counter 共进行了 2 次递增操作，但递增后的值不是预期的 2 而是错误值 1，该错误被称为数据一致性错误；从图 6.18 可以看到，在使用互斥锁时，变量 counter 的数值正确地进行递增，因为互斥锁能够保证同一时刻只有 1 个内核对变量 counter 进行访问，从而保证了变量 counter 的数据一致性，由此证明了核间共享资源互斥访问机制设计的正确性。

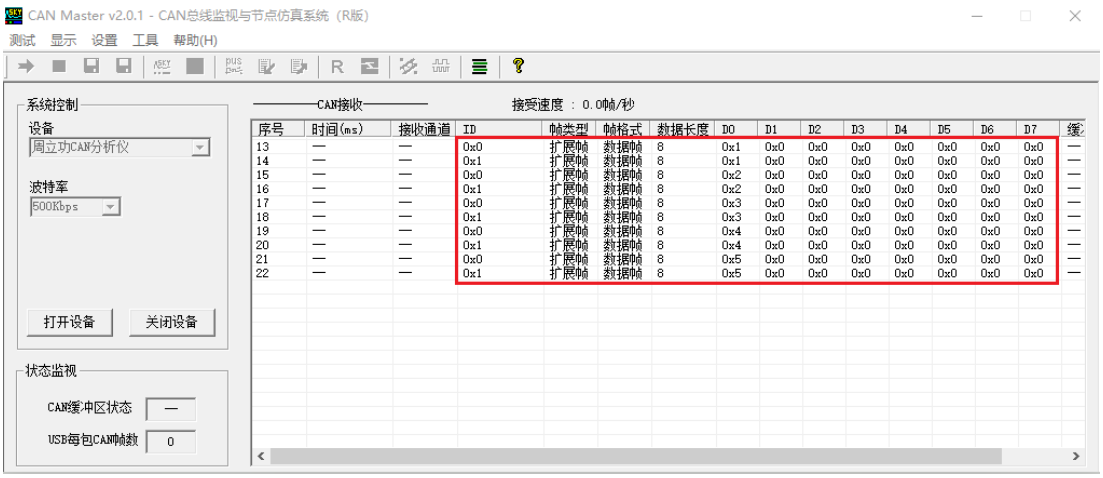


图 6.17 不使用互斥锁的测试结果

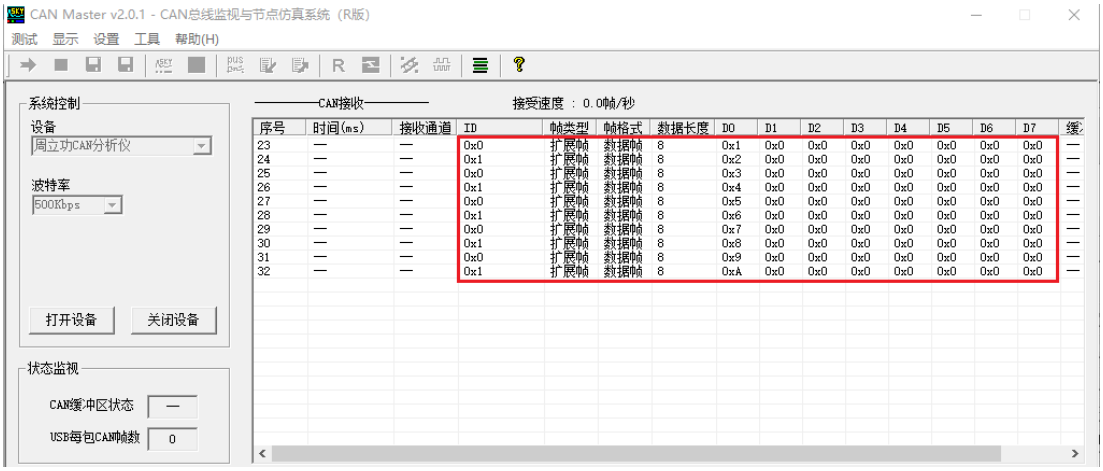


图 6.18 使用互斥锁的测试结果

6.5 双冗余系统软件测试

6.5.1 主从系统同步测试

该测试主要验证同步机制能否实现主从系统之间的同步运行。主从系统均以 DO 通道电平信号作为系统执行相关操作的标志。系统上电时引脚电平默认为高电平，因此第一个上升沿表示系统上电启动，系统启动后首先将 DO 通道电平拉低。为了便于通过示波器观察系统的同步状态及同步时序，设计了测试用例，使系统开始准备同步时将 DO 通道电平拉高，而同步完成后将 DO 通道电平拉低，并且为了更好地测试同步机制的有效性，使主从系统不同时上电，而是相隔一段时间先后上电。最后得到的测试结果如图 6.19 和图 6.20 所示，其中信号 1 为主系统对应 DO 通道的电平信号，信号 2 为从系统对应 DO 通道的电平信号。

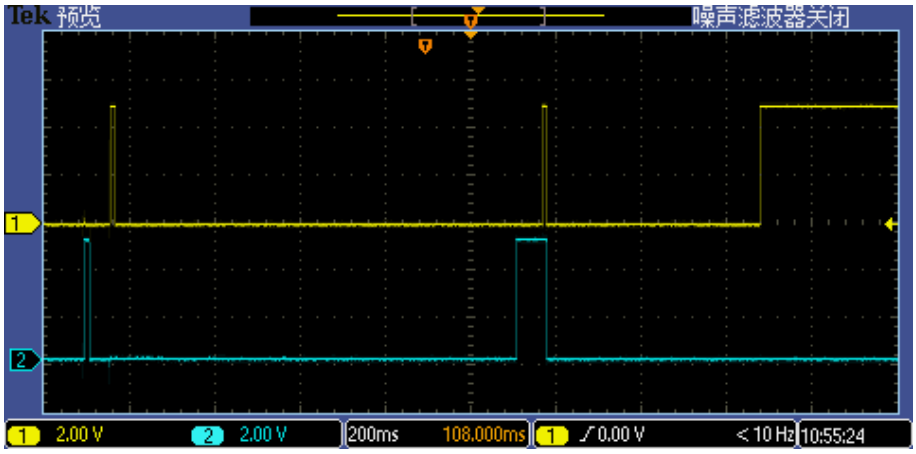


图 6.19 主从系统同步测试结果

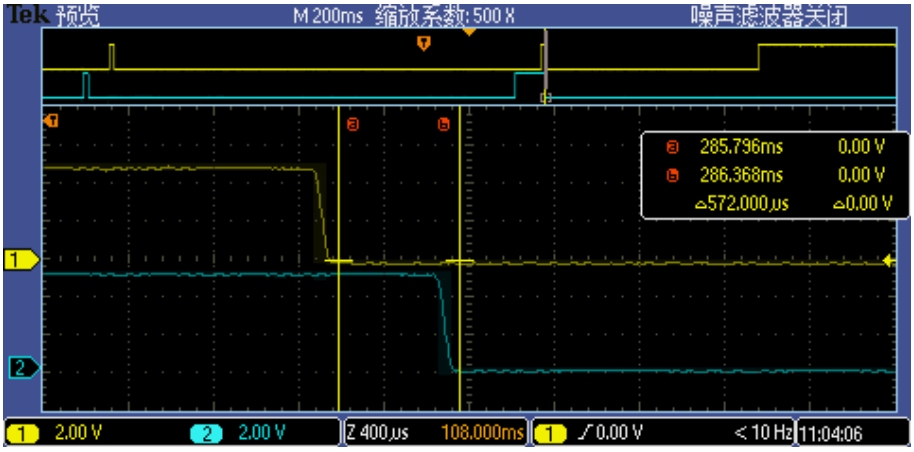


图 6. 20 主从系统同步测试的信号测量结果

从示波器波形图中可以看到，从系统对应 DO 通道的第一个上升沿信号和第二个上升沿信号均早于主系统，说明从系统的启动和开始同步的时间均早于主系统，但通过之后的下降沿可以看到主从系统几乎同时完成同步，且根据图 6.20 所示的测量结果可知，主从系统同步后，二者的系统时间只相差 572us，能够满足防止系统故障误判的相应要求，证明了同步机制设计的正确性。

6.5.2 故障检测及处理测试

该测试主要验证故障检测及处理机制能否在系统出现故障时，及时检测到故障的发生并进行正确的处理。为了同时测试系统之间和系统内部故障检测及处理机制，模拟了处理器异常复位故障和系统供电故障，并以 DO 通道电平信号作为系统执行相关操作的标志。具体的测试用例设计如下：主系统在同步成功后，将对应的 DO 通道电平拉高，表示控制器进入双系统协同运行模式，而从系统保持对应的 DO 通道为低电平。主系统运行一段时间后定时触发软件复位，以模拟处理器异常复位故障的发生。从系统检测到主系统发生故障时，将对应的 DO 通道电平拉高，表示成功检测到对方系统发生故障并进行了相应的处理。主系统发生异常复位故障并重启后，保持 DO 通道为低电平。在主系统发生异常复位故障后，经过一段时间将从系统的电源关闭，以模拟从系统发生供电故障。如果主系统发生异常复位故障后正确地进行了故障处理并检测到从系统供电故障的发生，则将对应的 DO 通道电平拉高。通过以上测试用例，得到了如图 6.21 所示的测试结果，图中信号 1 为主系统 DO 通道的电平信号，信号 2 为从系统 DO 通道的电平信号。

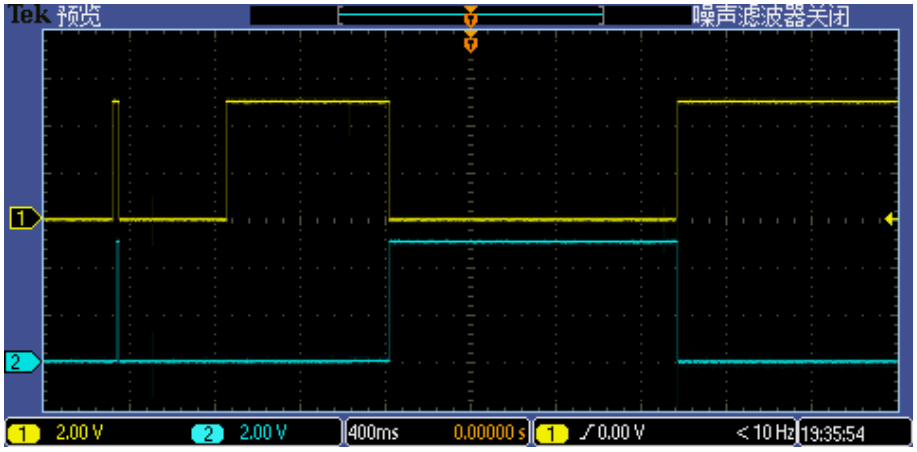


图 6.21 故障检测机制测试结果

从图 6.21 中可以看出，完成同步后主系统对应 DO 通道电平拉高，说明控制器进入了双系统协同运行模式。运行一段时间后，主系统对应 DO 通道变为低电平，说明其按照测试用例的设定模拟了处理器异常复位，与此同时，从系统对应的 DO 通道立即变为高电平，说明从系统检测到了主系统异常复位的发生，且通过图 6.22 所示的测量结果，得到故障响应时间为 2.08ms，满足响应时间要求。经过一段时间后，将从系统电源关闭以模拟系统供电故障，其对应的 DO 通道因断电而变为低电平，而此时主系统对应的 DO 通道立即变为高电平，说明主系统发生处理器异常复位故障后正确地进行了故障处理，并作为从系统的备用系统继续运行，且当从系统发生故障时成功检测到了故障的发生并替代从系统运行。由图 6.23 所示的测量结果可得相应的故障响应时间为 4.04ms，而车辆控制信号的周期为 10ms，因此满足故障响应时间的要求。

以上测试结果表明，在发生故障时，主从系统均能够及时检测到故障的发生并进行正确的故障处理，证明了故障检测及处理机制设计的正确性。

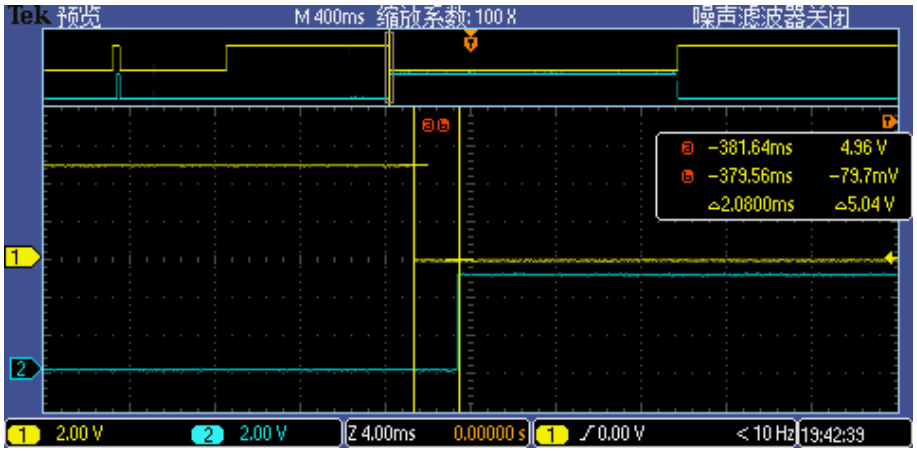


图 6.22 故障检测机制测试的测量结果 1

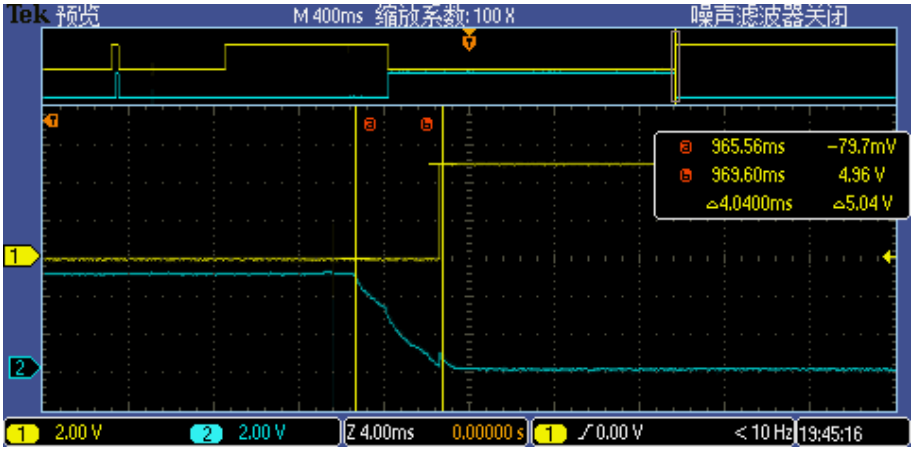


图 6. 23 故障检测机制测试的测量结果 2

6.6 本章小结

本章主要对自动驾驶控制器软件进行了测试，以验证软件设计的正确性。首先对 Bootloader 进行了应用程序下载测试，验证了 Bootloader 的系统启动、应用程序下载和程序跳转功能的正确性。然后对多核处理器基础软件进行了测试，通过对 FreeRTOS 的多任务实时调度和基于优先级的抢占式任务调度功能的测试，验证了 FreeRTOS 移植的正确性；通过相关测试用例，分别验证了核间任务同步机制、核间共享资源互斥访问机制和核间通信机制设计的正确性。最后对双冗余系统软件进行了测试，通过两系统间的同步测试，验证了主从系统同步机制设计的正确性；通过模拟异常复位故障和系统供电故障，对冗余系统的故障检测与处理机制进行了测试，验证了该机制设计的正确性。

第 7 章 总结与展望

7.1 全文总结

本文主要针对目前自动驾驶在算力和功能安全方面的需求，研究了多核处理器在自动驾驶控制中的应用以及自动驾驶控制器的冗余设计方法，提出了基于多核处理器的双冗余自动驾驶控制器硬件架构，并基于该硬件架构进行了自动驾驶控制软件架构的设计与实现，主要研究工作总结如下：

（1）完成了自动驾驶控制器的硬件设计。针对自动驾驶在算力和功能安全方面的需求，设计了基于多核处理器的双冗余自动驾驶控制器硬件架构，并在硬件架构的基础上完成了主要器件的选型，包括处理器、电源管理芯片和 CAN 收发器等，然后进行了控制器电路原理图和印制电路板的设计，并通过印制电路板的加工和元器件焊接，完成了控制器的实物设计，为后续的软件设计提供了硬件平台。

（2）完成了系统基础软件 Bootloader 的设计。首先设计了系统启动程序，解决了系统启动时的环境设置和多核的启动引导问题，保证了系统的正常启动。接着进行了应用程序下载功能相关的程序设计，包括 Bootloader 与上位机软件间通信的设计、应用程序可执行文件的下载、解析与烧写程序的设计以及 Bootloader 到应用程序的跳转程序的设计，实现了应用程序下载功能。最后进行了程序存储器的空间分配方案设计，保证了 Bootloader 与应用程序的地址空间不发生冲突，并使程序存储器的空间得到合理的分配和使用。通过进行应用程序的下载测试，验证了 Bootloader 系统启动和应用程序下载功能的正确性。

（3）完成了多核处理器基础软件的设计。首先根据自动驾驶应用场景的实际需求，进行了多核处理器的总体软件架构设计，明确了任务实时运行的条件以及多核协同运行所需的相关机制。然后进行了实时操作系统 FreeRTOS 的移植，保证了自动驾驶任务的实时运行。最后设计了核间任务同步机制、核间共享资源互斥访问机制和核间通信机制，为多核的协同运行提供了必要条件。通过相关测试，验证了 FreeRTOS 的成功移植以及多核协同运行相关机制设计的正确性。

（4）完成了双冗余系统软件的设计。首先对双冗余控制器的总体冗余策略进行了

设计，接着设计了系统状态信息管理程序，保证了互为冗余的两个系统之间具有正确的状态关系。然后设计了同步协议和同步程序，实现了双系统协同运行模式下两系统之间的同步。最后设计了系统故障检测及处理机制，保证了系统出现故障时能够检测到故障的发生并立即做出响应。通过测试，验证了双冗余系统软件设计的正确性。

7.2 全文展望

本文还存在以下方面有待进一步完善：

（1）在核间通信机制的设计上，本文采用了共享内存的核间通信方式，当一个任务需要共享内存进行核间通信时，通过内存获取函数从内存池中申请所需的共享内存空间。但本文未设计相应的共享内存回收机制，因此下一步工作可以对内存回收机制进行设计，以提高共享内存使用的灵活性；

（2）在系统内部故障的检测方法设计上，本文主要针对处理器异常复位故障、CAN 通信故障和 I/O 模块故障进行设计，而在其他内部故障的检测方法设计上，还有待进一步的完善，从而更加全面地对系统内部故障进行检测。

参考文献

[1] 王金强,黄航,郅朋,等.自动驾驶发展与关键技术综述[J].电子技术应用,2019,45(06):28-36.

[2] 罗为明,袁建华,陆文杰,等.自动驾驶技术解读--自动驾驶汽车环境感知系统(下)[J].道路交通科学技术,2019(4):3-5.

[3] 王贺.雷达摄像头数据融合在智能辅助驾驶的应用[D].吉林大学,2019.

[4] 管家意.自动驾驶汽车轨迹跟踪控制方法研究[D].武汉理工大学,2017.

[5] 多核系列教材编写组.多核程序设计[M].清华大学出版社,2007.

[6] Blake G, Dreslinski R G, Mudge T. A survey of multicore processors[J]. IEEE Signal Processing Magazine, 2009, 26(6): 26-37.

[7] Imtiaz S Y, Hameed A, Min-Allah N. Multi-core Technology: An overview[C]// 32nd Annual Conference on Artificial Intelligence KI-2009, September 2009. 2009.

[8] Chien A A, Karamcheti V. Moore's law: The first ending and a new beginning[J]. Computer, 2013, 46(12): 48-53.

[9] Bulusu N, Heidemann J, Estrin D. GPS-less low-cost outdoor localization for very small devices[J]. IEEE personal communications, 2000, 7(5): 28-34.

[10] 周楠,胡娟,胡海明.多核处理器发展趋势及关键技术[J].计算机工程与设计,2018,39(02):393-399+467.

[11] Breternitz M, Wu Y, Sassone P, et al. Instruction boundary prediction for variable length instruction set: U.S. Patent 9,223,714[P]. 2015-12-29.

[12] Vachharajani N, Iyer M, Ashok C, et al. Chip multi-processor scalability for single-threaded applications[J]. ACM SIGARCH Computer Architecture News, 2005, 33(4): 44-53.

[13] 史莉雯,樊晓桠,张盛兵.单片多处理器的研究[J].计算机应用研究,2007(09):46-49.

[14] 刘必慰,陈书明,汪东.先进微处理器体系结构及其发展趋势[J].计算机应用研究,2007(03):16-20+26.

[15] Hayashi A. Studies on automatic parallelization for heterogeneous and homogeneous multicore processors[D]. Waseda University, 2012.

[16] Hyari A. A comparative study on heterogeneous and homogeneous multiprocessors[J].

University of Jordan, Department of Computer Engineering, Submission date, 2009.

[17] Zhou R, Chen H, Liu Q, et al. A server model for reliable communication on cell/BE[C]//2013 42nd International Conference on Parallel Processing. IEEE, 2013: 1020-1027.

[18] Greenhalgh P. Big.little processing with arm cortex-a15 & cortex-a7[J]. ARM White paper, 2011, 17.

[19] Hammond L S. Hydra: a chip multiprocessor with support for speculative thread-level parallelization[M]. Stanford University, 2002.

[20] Hammond L, Hubbert B A, Siu M, et al. The stanford hydra cmp[J]. IEEE micro, 2000, 20(2): 71-84.

[21] Taylor M B. The raw processor specification[R]. Technical Memo, CSAIL/Laboratory for Computer Science, MIT, 2004.

[22] Olukotun K, Nayfeh B A, Hammond L, et al. The case for a single-chip multiprocessor[J]. ACM Sigplan Notices, 1996, 31(9): 2-11.

[23] Taylor M. The Raw Prototype Design Document V5. 01[J]. 2004.

[24] Abellán J L, Ros A, Fernández J, et al. Econo: Express coherence notifications for efficient cache coherency in many-core cmps[C]//2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). IEEE, 2013: 237-244.

[25] 黄国睿,张平,魏广博.多核处理器的关键技术及其发展趋势[J].计算机工程与设计,2009(10):80-84.

[26] Kahle J A. Introduction to the Cell multiprocessor IBM[J]. Journal of research and development, 2005, 49(4/5).

[27] Jeff B. Big.LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration[C]//DAC. 2012: 1143-1146.

[28] Qadri M, Sangwine S. Multicore Technology: Architecture, Reconfiguration, and Modeling[M]. CRC Press, Inc. 2013.

[29] Qin Y. Overview of inter-communication mechanism on multi-core processor[C]//2010 International Conference on Computer Application and System Modeling (ICCASM 2010). IEEE, 2010, 7: V7-187-V7-190.

[30] LI J, WANG J, ZHANG Q. Design of Inter-core Communication Mechanism Adapting to Multi-core Processor[J]. Intelligent Computer and Applications, 2011, 4.

[31] 盛肖炜.多核处理器内部核间通信研究[D].沈阳理工大学,2013.

[32] 杨国芳.多核处理器核间通信技术研究[D].哈尔滨工程大学,2011.

[33] Ganguly A, Chang K, Deb S, et al. Scalable hybrid wireless network-on-chip architectures for multicore systems[J]. IEEE Transactions on Computers, 2010, 60(10): 1485-1502.

[34] 孙晨.基于 AMP 架构的多核通信系统研究[D].华北电力大学(北京),2019.

[35] Love R. Linux kernel development[M]. Pearson Education, 2010.

[36] 孔帅帅.基于嵌入式多核处理器的通信及中断问题的研究[D].电子科技大学,2011.

[37] 布赖恩特,奥哈洛伦,奕利,等.深入理解计算机系统[M].中国电力出版社,2004.

[38] 陈向群,马洪兵.现代操作系统[M].机械工业出版社,2009.

[39] 耿晓中.基于多核分布式环境下的任务调度关键技术研究[D].吉林大学,2013.

[40] 李静.基于多核的任务调度策略研究[D].哈尔滨工程大学,2011.

[41] 刘林东,刘波.一种多核处理器调度策略研究[J].广东第二师范学院学报,2014,34(05):90-94.

[42] 齐爽.控制器双冗余设计与实现[D].哈尔滨工业大学,2015.

[43] Brière D, Favre C, Traverse P. A family of fault-tolerant systems: electrical flight controls, from Airbus A320/330/340 to future military transport aircraft[J]. Microprocessors and Microsystems, 1995, 19(2): 75-82.

[44] Yeh Y C. Triple-triple redundant 777 primary flight computer[C]//1996 IEEE Aerospace Applications Conference. Proceedings. IEEE, 1996, 1: 293-307.

[45] Lippiello V, Ruggiero F. Exploiting redundancy in Cartesian impedance control of UAVs equipped with a robotic arm[C]//2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE, 2012: 3768-3773.

[46] 张立杰,张蕊,孙荣廷,等.基于 S7-400H PLC 冗余系统的烧结温度控制[J].自动化与仪表,2012,27(01):44-47.

[47] 黄洁.Control Logix 控制系统及其应用[J].冶金自动化,2003(S1):211-212.

[48] 刘豫湘,刘剑锋.机车制动控制单元的双 CPU 控制板冗余设计[J].电力机车与城轨车辆,2013,36(05):1-4.

[49] 吕聪,李华旺,常亮.星务管理中的双 CPU 冗余通信设计与实现[J].电子设计工程,2017,25(14):92-95+100.

[50] 周树桥,李铎.双冗余控制器的失效状态分析及面向高可靠度的设计[J].清华大学学报(自然科学版),2017,57(04):399-404.

[51] 刘冲,付江梅.双重冗余 PLC 控制系统的可靠性与可用性研究[J].自动化仪表,2010,31(09):44-46+53.

[52] 王鼎,李铎.浮动式核电站专用控制器中 CPU 冗余技术研究[J].原子能科学技术,2010,44(01):44-47.

[53] 蔡敬海,张振权.电子系统冗余设计及可靠性分析[J].光电技术应用,2016,31(01):64-67.

[54] 李小坚,郝晓丽.Protel DXP 电路设计与制版实用教程:Applications of Protel DXP[M].人民邮电出版社,2015.

[55] 胡尔佳.深入理解 BootLoader[M].机械工业出版社,2016.

[56] 袁磊,朱怡安,兰婧.嵌入式系统 BootLoader 设计与实现[J].计算机测量与控制,2009,17(02):389-391.

[57] 张龙彪,张果,王剑平,等.嵌入式操作系统 FreeRTOS 的原理与移植实现[J].信息技术,2012,36(11):31-34.

致谢

冰雪消融，暖意初来，在这春回大地的三月，我们已经渐行在毕业的路上，三年的研究生时光匆匆而逝，在此，我要感谢一路上给予我帮助的人。

感谢我的导师张建伟老师，他工作认真，一丝不苟，做事负责，事必躬亲，在科研上对我的言传身教让我积累知识，并教给我学习方法以及解决问题的思路，每当遇到问题时，他总能一针见血的帮我指出问题关键。除此之外，在生活中上他处处关心我的心态、身体状况。另外，感谢课题组丁海涛老师，每当有困难寻找老师帮助时，他总是仔细聆听，耐心帮我解决，他虚心务实，博学广识，总是会给我们提供一系列主题讲座，帮助我开拓视野、充实生活。在此，向两位恩师表达最诚挚的敬意，祝福他们工作顺利，阖家幸福。

感谢肖遥师兄、郑敏师兄和李斌师兄在科研和学习上对我的指导，为我之后的工作提供了很大的帮助。感谢李冬、孟强、袁佳威、刘灿、潘威、吴钰繁等师兄在职业规划上给予我的指导和建议，让我有了开阔的视野和明确的目标。

感谢孙林、杨鑫、冉德智、王雪犇和李鹤的陪伴，让我度过了愉快的研究生生活，我们一起在实验室努力学习，也一同在业余时光谈笑风生。感谢室友李赞、成泉岳和武建君对我学习和生活的关照，我们共同营造了良好的寝室氛围，让平日的生活充满乐趣。同时感谢实验室其他同学，三年时光的点点滴滴仍旧历历在目，感谢的人太多，衷心祝福你们事事顺利，身体健康。

最后，感谢父母二十多年的养育之恩，你们含辛茹苦的抚养、培养我，才让我在学习之余无后顾之忧，谢谢你们的默默付出。

2021 年 5 月于长春