FIRMDRIVE®及 PXIE-1010
总线控制器使用手册

2018-8-31

简仪科技有限公司

info@jytek.com

目录

图例索引

表格索引

1 概要

1.1 简介

FirmDrive®与 PXIe-1010 总线控制模块是上海简仪科技有限公司和上海聚星仪器有限公司联合研制开发的。FirmDrive®是基于 Xilinx FPGA 芯片的 PCIe、PXIe 的软件驱动架构，针对测试测量应用做了大量的优化。PXIe-1010 是 PXIe 的总线控制模块（母板）。FirmDrive®与 PXIe1010 同时使用构成了完整的 PXIe 总线控制器。硬件工程师可以在 PXIe 总线控制器上开发自定义的 PXIe 功能板卡（子板）。图 1-1 FirmDrive®架构图。

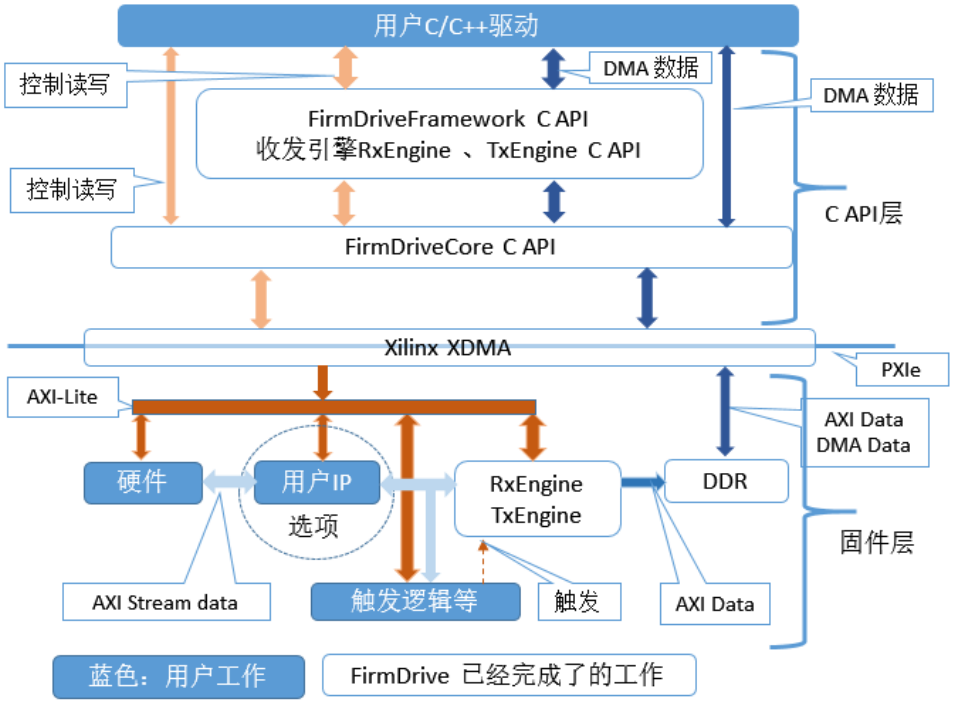


图 1-1 FirmDrive®架构图

简仪科技完成了固件层次的收发引擎等 IP 的制作。用户只要将 AXI-Stream Data 连接到这些固件就可以完成数据的传输。

FirmDriveCore 是一组基本的 C API。它完成对寄存器的读写、DMA 数据的读写等。它对硬件工程师的调试工作有很大的帮助。但在编写驱动时，FirmDriveFramework 更加容易使用。

FirmDriveFramework C API 内部使用 FirmDriveCore 和其它工具。它完成了对 DDR 环形缓存区的管理，同时也完成了数据的传输管理。在通常情况下，用户的 C/C++驱动可以在 FirmDriveFramework C API 上编制。简仪科技提供了编制 C/C++驱动的范例，便于用户开发 C/C++驱动。

所有固件层次的 IP 都是跨平台的。所以 FirmDrive®的结构本身就时一个跨平台的架构。

硬件工程师只要把设计好的硬件以 AXI-Stream Data 的形式连接到 FirmDrive®提供的引擎 IP 接口，就完成了数据的传输工作。

综上所述，使用上海简仪提供的 FirmDrive®和 PXIe-1010 总线控制器有以下几个优点：

- 硬件设计师无需对 PXIe 总线有细致的了解
- 大大简化了 PXIe 板的设计工作量
- 设计师无需对 C 软件驱动程序担忧
- 大大提高了可维护性
- 降低了研发和维护成本
- 可嵌入属于自己的知识产权（FPGA 代码）
- 加快设计进程

该文档结合简仪提供的 PXIe-1010DK 开发套件详细介绍 PXIe1010 总线控制器模块、FirmDrive®驱动架构中的 C API 函数、固件 IP；同时向硬件设计师介绍如何使用 FirmDrive®和 PXIe-1010 设计您需要的 PXIe 模块；按照简仪提供的参考设计创建 C 驱动程序和 C#驱动程序；最后一并介绍跨平台的驱动设计。除了您需要和硬件设计需要的文档资料外，简仪力争这是您唯一需要的文档。

简仪科技欢迎您对此文档提出宝贵的意见，我们也将不断地完善此文档，以便更好地帮助您的设计工作。您可以在 www.jytek.com 注册后下载最新的文档。

1.2 如何使用本手册

本手册的对象是硬件设计工程师。您应该首先阅读本章节的介绍，了解 FirmDrive®的工作原理和流程。您今后最多的工作是如何将您设计的硬件通过 AXI-Lite 总线连接到 XDMA，并将符合 Xilinx AXIS 数据格式的原始数据流连接到 FirmDrive®提供的一些固件模块，如收发引擎 RxEngine、TxEngine 等。您是不需要知道 PXIe 总线的工作原理及细节的，也无需对 Xilinx 的 XDMA 有非常深入的了解，更不需要了解 FirmDrive®固件接口的内部工作机制。您只需要了解 XDMA AXI-Lite 总线如何控制您的硬件，知道 FirmDrive®模块接口的工作机制和如何连接。

在阅读了本章之后，您可以直接跳到第三章。第三章通过简仪提供的 PXIe-1010DK 开发套件介绍了一个典型的硬件设计和连接，能帮助您更好地了解如何完成上述地任务。在该章中您会接触到 FirmDrive®提供的接口模块和 XDMA 的接口模块。第四章对这些模块有单独的介绍。这些介绍应该能够满足您使用 FirmDrive®完成您设计任务的需求。

当您理解了 FirmDrive®的工作原理后，您可以通过第九章提供的硬件接口信息开始设计您的硬件。当您的硬件成功地连接到 XDMA 和 FirmDrive®后，您就可以通过第二章提供的 FirmDriveCore C API 和第四章介绍的 FirmDriveFramework C API 来调试您的硬件了。这两组 C API 提供了 C/C++驱动的基本单元。您可以根据第五、六、七、八章来编制更适合测量应用的 C/C++、C#、及跨平台的驱动。简仪科技同时提供快速的上手服务来帮助您完成驱动的撰写。您一旦熟悉了 FirmDrive®的架构后，您今后的设计效率会有大幅度的提高。

1.3 从应用到驱动的跨平台总体设计

自上而下的软件架构：锐视开源测控平台到FirmDrive®驱动

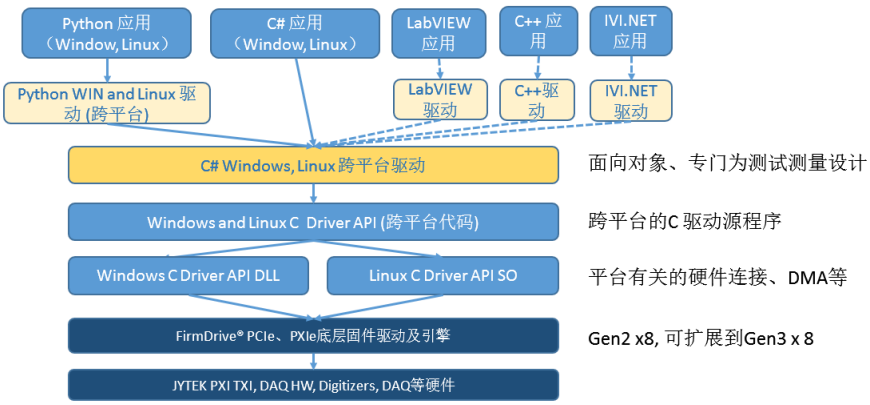


图 1-2 简仪科技的驱动结构

传统的驱动设计往往是由计算机软件工程师来完成的。它是一个自下而上的过程。软件工程师把硬件的特点以 C/C++的接口形式开放给应用工程师，虽然很多时候这些硬件功能和测试测量并没有非常直接的关系。应用工程师使用诸多与测试测量有关与无关的 C/C++API 来完成应用软件的编制。再者 C/C++驱动靠近操作系统的底层，编制困难，维护困难。在当今炙手可热的软件行当，测试领域里称职的 C/C++驱动工程师可谓凤毛麟角。

与传统驱动设计不同，简仪科技的驱动是根据多年测试测量的经验来设计的。这是一个自上而下，从应用到底层的设计。C#层的驱动是简仪科技的核心。在这一层次，简仪吸取了国外测量软件的长处，定义了测试测量所需要的方法和属性。使用这些方法、属性再加上面向对象的编程环境就可以非常有效率地编制应用软件。所有其它的语言类（Python，Lab VIEW，C++）的驱动都是建立在 C#驱动上的。C#层的驱动是一个跨平台的驱动。因此简仪科技只需要维持一个稳定的 C#驱动就可以支持不同平台的应用。

根据 C#驱动的要求，简仪设计了跨平台的 C 驱动。这个驱动和 Windows 有关库（DLL）连接就生成了 Windows 的 C 驱动；与 Linux .SO 库连，就生成 Linux 的 C 驱动。

无论是 Windows 的 DLL 库还是 Linux 的.SO 库都调用同一个底层的 FirmDrive®驱动库。FirmDrive®驱动库是简仪和聚星联手开发的。使用简仪的架构能够避免对其它诸如 VISA 等驱动软件的依赖，大大提供驱动性能，提高效率。

1.4 FirmDriveCore 与 FirmDriveFramework

FirmDrive® C API 包括了两个部分：FirmDriveCore 底层 C API 接口库和更加切合测量应用的 FirmDriveFramework 高层 C API 接口库。

FirmDriveCore C API 接口库使用 C 语言来访问寄存器，控制 DMA，从寄存器或 DMA 获取或输出数据。这些 C API 大大方便了硬件工程师的调试工作，降低调试难度，节省时间，同时也给有必要直接调用驱动底层的用户提供了直接的接口。

FirmDriveFramework 高层 C API 是简仪和聚星根据测试测量应用的特点专门设计的高层次的 C 接口。FirmDriveFramework 包括了数据收引擎 RxEngine、数据发引擎 RxEngine、可变功能接口 PFI、路由矩阵、触发管理、计数器管理等。在收发引擎

RxEngine、TxEngine 中还建立了环形缓冲器管理。使用 FirmDriveFramework 可以避免了底层 C API 很多没有必要的功能，和测试应用吻合，用户完全无需考虑寄存器、DMA 和缓冲器设置等问题。FirmDriveFramework C API 和 C/C++驱动 API 几乎完全吻合。在 FirmDriveFramework C API 基础上编制 C/C++驱动大大提高了效率，提高了可维护性，降低了对 C/C++驱动工程师的要求的门槛。

1.5 FirmDrive®的适用范围

FirmDrive®目前支持 PXIe Gen2 x 8、PCIe Gen2 x 8；同时支持 Windows 7、Windows 10 和 Linux。根据 Xilinx 的介绍，FirmDrive®可以在未来支持 PCIe Gen3 x 8 系统。FirmDrive® 是专门为了测试测量设计的，目前的版本还暂时没有对闭环控制系统的应用进行优化，也暂时不支持中断的处理。所以对一些要求高的闭环控制系统可能会有效率下降的现象。值得一提的是，Xilinx 的 XDMA 架构是可以支持中断功能的。

1.6 对硬件工程师的要求

由于 FPGA 在硬件设计中的重要性日益重要，当今的硬件设计师必然要会进行基本的 FPGA 设计。FirmDrive® 对硬件工程师有以下要求:

- 熟悉专业的传统硬件设计；
- 必须会使用 Xilinx Vivado 工具；
- 必须对 AXI 系列接口及模块有一定的理解；
- 了解 AXI-Stream 数据格式；
- 能够通过 Xilinx FPGA 控制设计硬件
- 会一些基本的 C 编程语言并用 C 语言对所设计的硬件进行调试。

简仪认为以上的要求并没有对硬件设计师提出更多和更苛刻的要求。在以下的文档里，简仪认定您已经具备了以上的能力。在没有具备以上能力之前，简仪不建议您使用简仪的 FirmDrive®和 PXIe-1010 总线控制器模块。简仪科技无法对以上工具提供咨询和服务。

1.7 与传统硬件设计的区别

如图 1-3 所示，传统的 PXIe 板设计需要三方面的技术人员合作：硬件设计师、固件工程师和驱动工程师。设计的分工如下：

硬件工程师：了解硬件功能和部分 PXIe 总线细节，了解一些固件和硬件的连接和控制，了解硬件固件的寄存器使用，使用的设计工具主要是如 Cadence 等软件。但对固件的详细用途并无需太了解；

固件工程师：对固件和开发工具有相当的了解，配合硬件设计师完成数据的传输和对硬件的控制。主要使用的工具是如 Vivado 等 FPGA 设计工具。由于个性化设计日益重要，几乎任何复杂一些的数据采集卡和模块仪器都会用到 FPGA 固件，所以固件工程师是不可缺少的；

软件驱动工程师：配合固件工程师完成诸如 DMA 功能的实现，主要的工具是微软 WDM 等接触到系统内核的开发工具。由于这一些工具非常专业，开发难度大，所以很多驱动工程师倾向于使用诸如 VISA 软件或者 WinDriver 等封装过的底层开发工具。但是使用这些软件也带来一些缺点。比如说 VISA 驱动软件并不一定开放所有的底层功能，所以在功能上会受到一些限制；又比如说， WinDriver 软件对于用量不大的客户是相当昂贵的。

综上所述，在使用 FPGA 固件设计数据采集，模块仪器硬件时，传统的方法是要使用三种不同的开发工具，具备三种不同的技术背景的人员参与。

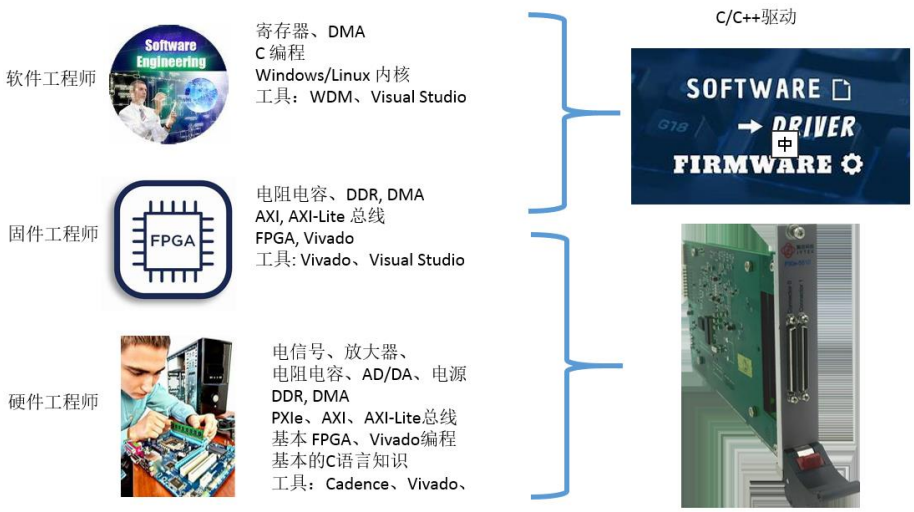


图 1-3 传统的测试测量硬件设计

使用 FirmDrive® 设计硬件

FPGA 厂商也在不断的努力减轻开发 PCIe 板卡的难度。Xilinx 研制了 XDMA 模块并使用了 ARM 的 AXI 接口技术就是为了达到这一目的。虽然如此，由于 PCIe 总线的应用非常广泛，Xilinx 也无法针对测试行业设计一款 PXIe 的控制器。上海简仪和聚星两家公司合力将数十年的测试测量经验、Xilinx 的 XDMA 模块及 AXI 技术成功地柔和在一起，完成了专门解决测试测量 PXIe 和 PCIe 的总线驱动架构，即 FirmDrive®架构。

图 1-4 与图 1-5 都表示硬件工程师使用 FirmDrive®来设计硬件。前者是在简仪的辅助下完成 C 驱动的编制，往往是第一次使用 FirmDrive®。后者是在熟悉了 FirmDrive®后单独完成。

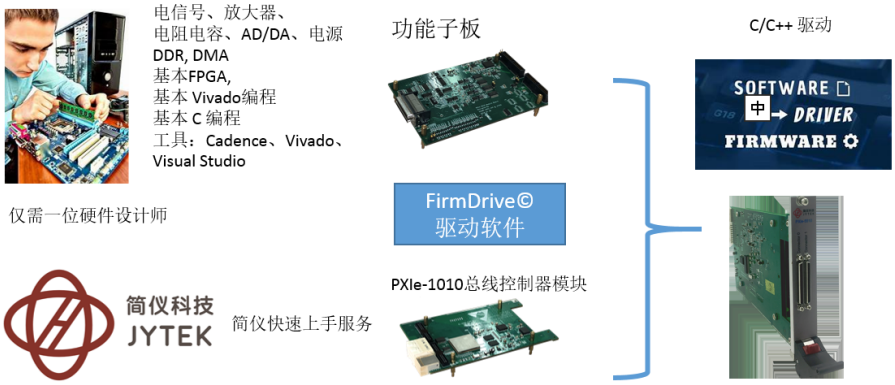


图 1-4 设计硬件在简仪科技辅助下完成 C 驱动

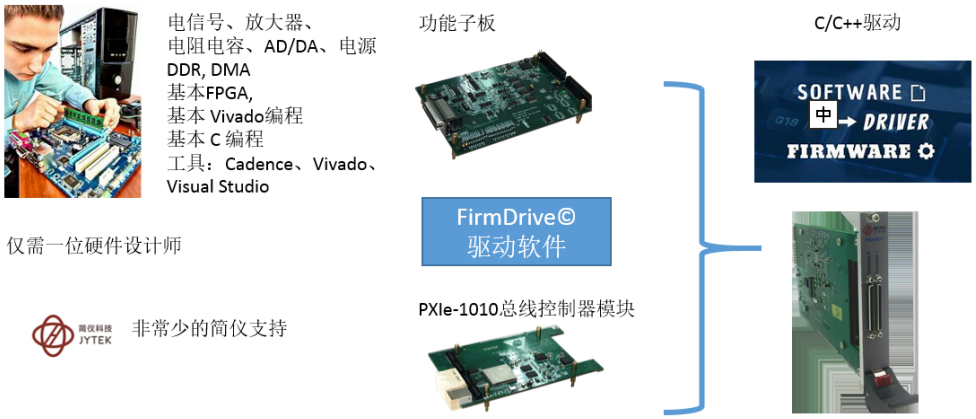


图 1-5 单独完成全部硬件设计和驱动 C 编制

传统硬件设计师经过短暂的 Vivado 工具培训和 C 语言的培训，即可以按照简仪、聚星提供的固件 IP 轻松地完成在 PCIe、PXIe 总线上进行数据传输以及对 PCIe、PXIe 硬件的控制。更加值得提出的是 FirmDrive®已经提供了 C 语言的 API。这些 C-API 和 FirmDrive®提供的固件 IP 配合使用。通过这些 C-API 就可以轻松地编制传统的 C/C++驱动，并在此基础上来编写简仪提出的 C#驱动。这一做法效率高，可维护性好。

在硬件工程师掌握了 FirmDrive®的技术并了解了如何使用 Vivado 工具将设计的硬件和固件连接到 FirmDrive®提供的收发引擎后，就能够自己产生 C 的驱动。

为了使您更方便地了解和使用简仪的 FirmDrive®和 PXIe-1010 总线控制器模块，简仪科技还提供了 PXIe-1010DK 的开发套件。该套件提供了全部的源程序和设计，可以是您起步的参考。

1.8 典型设计流程

错误!未找到引用源。显示一个典型的设计流程。

物理及电气连接：FPGA 芯片是 PXIe-1010 已经选定的，FPGA 的引脚和 PXIe-1010 上的接插件的连接也确定好了的，这两部分无需改动。硬件设计师通过功能子板上的接插件把自己的硬件与母板上的 PXIe 总线、FPGA、电源相连接。

连接成功后您开始进行带有 FPGA 控制的硬件设计。在这个过程中您需要使用 Vivado 来编制一些底层的固件程序来完成硬件的读写及控制功能，形成一些您自己的固件 IP。通常您的硬件是通过一些寄存器读写来控制的。为了方便您的设计，简仪科技提供了一组底层的 C API 通用接口库 FirmDriveCore 来完成这一任务。只要是按照 FirmDrive®规范连接的固件 IP 都可以使用 FirmDriveCore C API 来控制。

除了底层的 FirmDriveCore 之外，FirmDrive®还提供了一组专门为测试测量应用设计的高层次的 C API，FirmDriveFramework。FirmDriveFramework 包括数据收发引擎（RxEngine、TxEngine）、缓冲器管理、可变功能接口 PFI、路由矩阵、触发管理、计数计时。这组 C API 都有自己对应的固件 IP。您只要将您的硬件 IP 和 FirmDrive®提供的固件 IP 连接后，您就可以非常轻松地使用 C API 来调用调试您的硬件和 FirmDrive®的高级功能，从而完成对硬件的控制和数据传输。这一方式可以避免您要在固件上做过多的开发，减少甚至避免了您对固件制作的依赖。

当您的调试结束后，您使用的 FirmDriveFramework C API 已经形成了您硬件的 C/C++驱动的雏形。您可以自如的选择需要的 C/C++驱动接口。简仪科技建议您参照简仪科技的方式来封装 C/C++驱动。因为这可以降低您今后封装 C#、Python 等驱动的工作量。

最后您可以按照简仪科技提供的参考方案生成 C#的驱动，从而完成自硬件到软件驱动的设计和调试。C#驱动使得您的硬件可以在简仪科技锐视测控平台得到非常好的使用。您也可以在 C#驱动上封装 Python、LabVIEW 等其它驱动。

以上各步的具体详情在在以下的章节做详细的介绍。

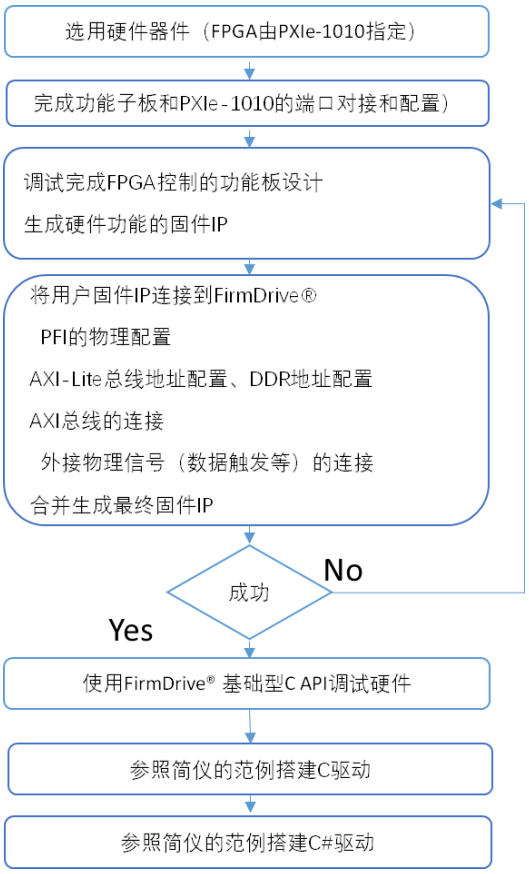


图 1-6 典型设计流程

1.8.1 模拟输入的上行通道示意图

上行通道是指数据从用户硬件端口经过收引擎 RxEngine、PXIe 总线传输到上位机（PC）。用户在使用简仪科技 FirmDrive®构架时还可以嵌入自己的 IP。A/D 接口模块、触发逻辑、路由矩阵、用户自定义 IP 都是由 AXI-Lite 控制的。A/D 接口模块的输出是不带地址信息的数据流，AXI-Stream 的接口形式。您可以在收引擎 RxEngine 之前嵌入自己的 IP 对数据进行预处理。用户 IP 的输出必须仍旧是 AXI-Stream。无地址的 AXI-Stream 数据经过 RxEngine（及配合的 AXI DataMover）之后，被转换为带有地址信息的 AXI 数据。RxEngine 提供了环形缓冲区的功能管理能力，AXI 数据会被自动地写入到指定地址处的环形缓冲区。RxEngine C API 提供了设置、读缓冲区数据的工具，从而自动完成 DMA 数据从 FPGA 到上位机 PC 的任务。

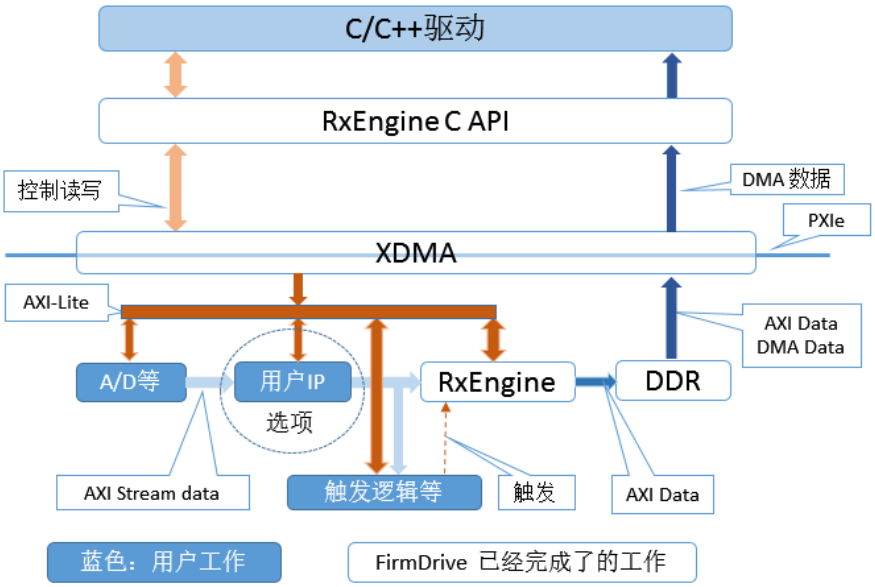


图 1-7 上行：典型模拟输入的设计

在上行通道里，用户 IP 必须置于 RxEngine 之前，对 AXI-Stream 数据进行操作。用户 IP 和 A/D 接口模块一样有自己的配置寄存器地址空间。用户 IP 的参数可以由 AXI-Lite 总线读写配置寄存器来完成。如果没有嵌入 IP，可以忽略这一步，将 A/D 接口模块的 AXI-Stream 接口可以直接连接到 RxEngine。

设计师的任务是把设计好的硬件模块，此例是 AD 模块、触发逻辑、及用户 IP 连接到 XDMA 的 AXI-Lite 控制线上，再将 AXI Stream Data 连接到 RxEngine 就可以了。设计师完全不需要知道 PXIe 的工作机制，也无需知道 RxEngine、XDMA 等工作机制。FirmDrive®会完成数据从 RxEngine 通过 DDR 到上位机的全部任务。

1.8.2 模拟输出的下行通道示意图

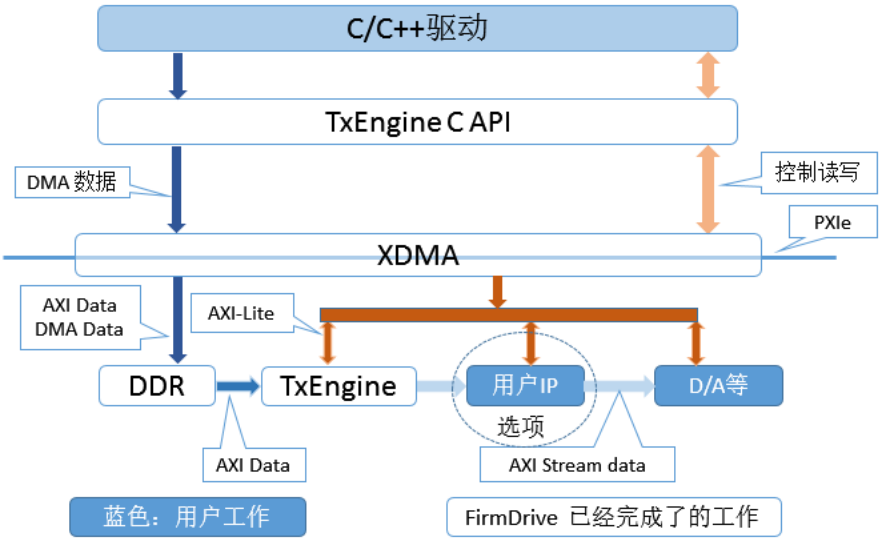


图 1-8 下行：典型模拟输出的设计

下行通道是指数据从上位机（PC）经过 PXIe 总线、发引擎 TxEngine，传输至用户的户硬件端口。与上一节的模拟输入类似，上位机的数据通过如**错误!未找到引用源。**所示完成。TxEngine 也有自己的输出环形缓冲区，并可以通过 C API 来设置。在下行通道里，用户嵌入的 IP 都必须置于在 TxEngine 的出口之后。所以用户 IP 都是对 AXI-Stream 数据进行操作的。用户 IP 和 D/A 接口模块一样有自己的配置寄存器地址空间。用户 IP 的参数可以由 AXI-Lite 总线读写配置寄存器来完成。如果没有嵌入 IP，可以忽略这一步。TxEngine 的 AXI-Stream 接口直接连接到 D/A 接口模块上。

设计师的任务是把设计好的硬件模块，此例是 D 模块、触发逻辑、及用户 IP 连接到 XDMA 的 AXI-Lite 控制线上，再将 AXI Stream Data 连接到 RxEngine 就可以了。设计师完全不需要知道 PXIe 的工作机制，也无需知道 RxEngine、XDMA 等工作机制。FirmDrive®会完成数据从 RxEngine 通过 DDR 到上位机的全部任务。

1.9 简仪科技提供 C/C++的快速上手服务

如 1.8 节所述，您的固件开发工作主要是硬件接口的设计、用户 IP 的设计、和使用 AXI-Lite 总线来控制这些固件 IP。如果这是您第一次使用 FirmDrive®，简仪科技可以向您提供一项收费合理的增值服务。这项服务包括以下三个方面：1）帮助将您设计的硬 IP 与 FirmDrive 的 IP 互联并规划分配地址；2）根据您的硬件模块地址，简仪科技帮助您建立 C API 的工程并提供源程序，使您能可以使用 FirmDriveCore 的底层 C API 来调用调试您的硬件。3）帮助您建立 C/C++驱动的工程并提供 C/C++的源程序。这些服务会大大地减少您在研制驱动需要花费的时间。您可以在简仪提供的 C-API 样板上调试您的硬件，改善您的硬件驱动。您无需向简仪科技提供您的设计细节。简仪只需要了解固件的地址空间即可。

这项服务的目的是使您快速上手。简仪科技不保证也不提供对最终驱动程序的支持与服务。这一点特在此申明。

2 FirmDriveCore：底层 C API 通用接口库

FirmDriveCore 是 FirmDrive®架构中一组比较底层的 C API 通用接口库。使用这一组 C API，硬件工程师就可以方便地调用按照简仪提供的接口方式和 Xilinx Vivado 工具开发的硬件固件。这极大地方便了硬件的开发调试。对于测试测量板卡的驱动开发，简仪和聚星还提供了更为方便使用的上层软件工具 FirmDrive®的收发引擎，路由矩阵等。对于有一些特殊闭环控制需要的应用，可以直接使用本章的 C API。由于简仪的主要精力在测试测量行业，本文的主要目的是测试测量的硬件设计，所以虽然我们的架构是可以支持高效的闭环应用的，我们没有仔细的优化和闭环有关的设计，在此特别指出。

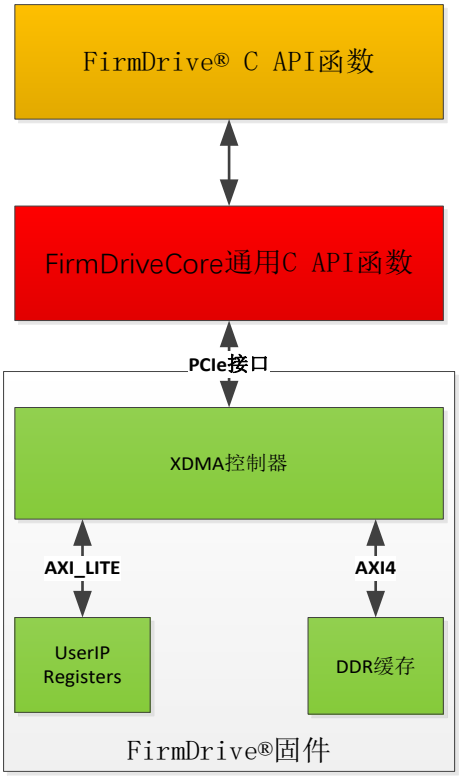


图 2-1 FirmDriveCore 通用 C API 的作用

如图 2-1 FirmDriveCore 通用 C API 的作用所示，FirmDriveCore C API 向下负责了 PCIe 接口的通讯工作，完成寄存器、DDR 缓存的数据读写，向上为 FirmDrive®架构的 C API 函数提供了基础调用函数。用户通过对 FirmDriveCore 通用 C API 函数的调用，实现对硬件的控制。

FirmDriveCore C API 通过与 XDMA(Xilinx DMA/Bridge Subsystem for PCI Express® (PCIe™))进行通讯完成对硬件的控制，如图 2-2 XDMA 用户侧接口所示，XDMA 的用户侧接口分别为：DMA 数据接口-AXI-Stream C2H(板卡到上位机)、AXI-Stream H2C(上位机到板卡)、AXI4_MM(按地址进行读写)；寄存器读写接口 AXI4_LITE；DMA Bypass(绕开 DMA 进行按地址的数据读写)接口。FirmDriveCore C API 函数分别对应于这些接口的数据读写操作。

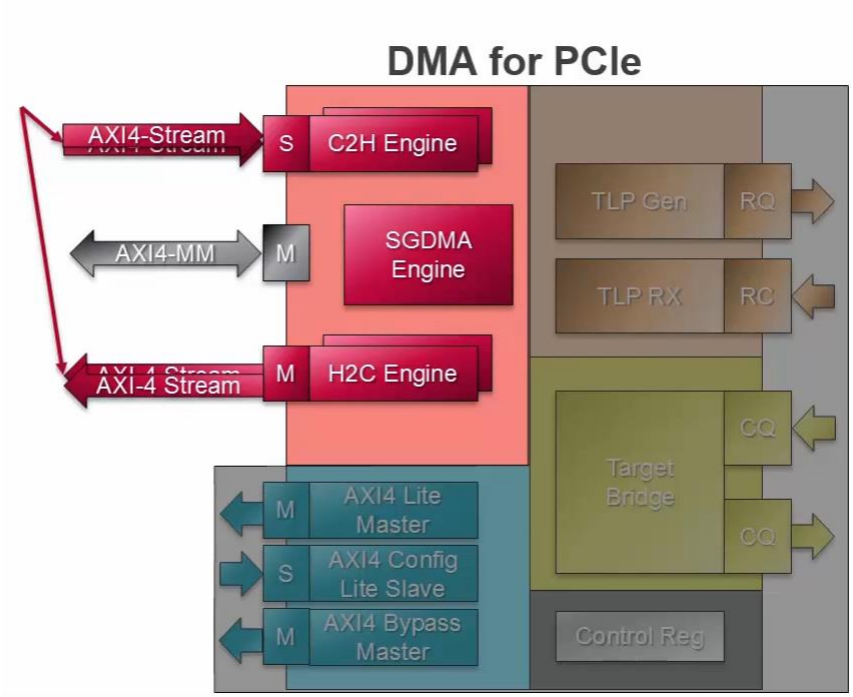


图 2-2 XDMA 用户侧接口

2.1 FirmDriveCore C API 具体函数

以下是 FirmDriveCore 提供的基础型 C API，主要包含设备初始化、设备关闭，寄存器读写、DMA Bypass 接口读写，DMA 读写，获取最近一次的错误信息等函数。硬件设计师在 Vivado 环境下完成配置后可以直接使用这些 C API 来调用所设计的硬件进行通讯。

C API 函数如下所示：

2.1.1 FirmDrive_Init()

```
/// ************************************************************************************
/// <summary>
/// 打开并初始化设备，返回设备handle. vendorIDIndex必须根据要求指定正确的参数，deviceID与slotNum可以仅指定其中一个，另外一个为0表示该参数任意。
/// </summary>
/// <param name="vendorIDIndex">设备VendorId序号，JXI=0,JY=1,ADLINK=2，支持列表请查看驱动inf描述文件的设备列表</param>
/// <param name="deviceID">设备DeviceId，默认为0时，打开第一块符合要求的板卡</param>
/// <param name="slotNum">设备所在PXI机箱卡槽号，使用PXI2PCIe转接板时，该值为31</param>
/// <returns>成功： DEVICE HANDLE; 失败：NULL</returns>
/// <created>Yasing,2017/10/19</created>
/// <changed>Yasing,2017/12/14</changed>
/// ************************************************************************************
EXTERN_C DLL_API FirmDriveDevice FirmDrive_Init(UINT32 vendorIDIndex, UINT32 deviceID, UINT32 slotNum);
```

注意：vendorIDIndex 是必填的参数，deviceID 和 slotNum 至少要填写一个，另外一个可填 0，表示该参数为任意。在对板卡进行任何操作之前，需要先调用该函数进行设备的初始化，获取设备的 handle。如果初始化失败，可以使用 FirmDrive_GetLastErr()获取最近一次的错误信息，根据文本信息和错误码定位问题。

VID:全称 Vendor Identification，又称 Vendor ID，是代表发明设备的专利所有者(技术厂商)的识别码，即常说的厂商 ID，这个 ID 是 PCI-SGI 组织统一编制命名的，是唯一的厂商标识。FirmDrive_Init()中的 vendorIDIndex 是驱动支持的 vendor ID index，比如 0 表示 JXIstrumentation VID=0xfee5、1 表示 JYTEK VID=0x1e2a，更多支持列表请参考最新驱动发布说明。

PID:全称 Production/Device Identification，又称 Device ID，是针对设备本身标识的代码，即常说的设备 ID。这个 ID 标识主要区别同类设备的不同型号，一般由技术发明厂商按 PCI 规范命名，不同厂商的设备可以有重名(由于不同厂商都有唯一的 VID，因此并不会混淆身份)。

Slot number: 对于 PXIe 板卡，除了 VID PID 外，卡槽号用于区分不同插槽的同系列板卡，0 表示任意符合 VID PID 参数的板卡，2 表示 2 槽的某型号板卡。

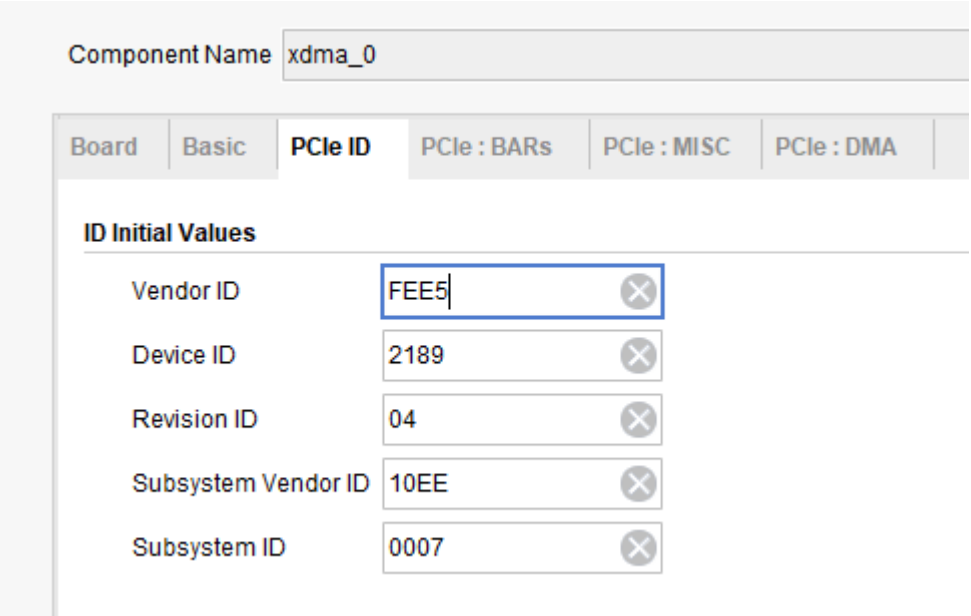


图 2-3 固件的 VID PID 设置

DMA 控制器使用 Xilinx 的 XDMA 控制器，可以在 XDMA 的配置页面中设定合理的 VID PID 值，配合驱动使用。如图 2-3 固件的 VID PID 设置所示，VID 设置为 0xFEE5，PID 设置为 0x2189，历史版本号设置为 04。为了得到板卡的 slot number，固件开发人员需要指定 GA 引脚(PXI 规范中用于识别卡槽号的 5 位引脚)对应的 AXI_GPIO 模块的基地址为 0x0000_0000。如图 2-4 GA 引脚的 AXI4_LITE 基地址设置为 0x0000_0000 所示。

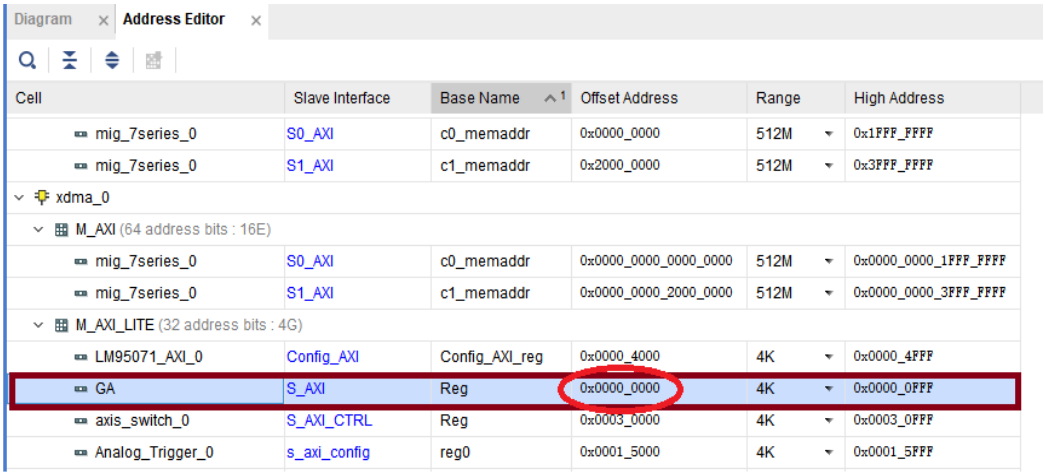


图 2-4 GA 引脚的 AXI4_LITE 基地址设置为 0x0000_0000

2.1.2 FirmDrive_Close()

```
/// ************************************************************************************
/// <summary>
/// 关闭指定设备，释放资源
/// </summary>
/// <param name="hDev">FirmDrive设备HANDLE</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Yasing,2017/10/19</created>
/// <changed>Yasing,2017/12/14</changed>
/// ************************************************************************************
EXTERN_C DLL_API int FirmDrive_Close(FirmDriveDevice hDev);
```

2.1.3 FirmDrive_GetLastErr()

```
/// ************************************************************************************
/// <summary>
/// 获取最近一次指定板卡操作错误的详细信息
/// </summary>
/// <param name="hDev">指向设备的句柄</param>
/// <param name="pnErrorCode">返回错误码</param>
/// <param name="lpszErrorMessage">以文本格式返回详细错误信息</param>
/// <param name="nStrLen">用于存放错误信息的缓冲区大小</param>
/// <returns>成功：0； 错误： 错误码</returns>
/// <created>Yasing,2017/10/20</created>
/// <changed>yasing,2018/5/2</changed>
/// ************************************************************************************
EXTERN_C DLL_API int FirmDrive_GetLastErr(FirmDriveDevice hDev, int *pnErrorCode, char
*lpszErrorMessage, UINT32 nStrLen);
```

详细错误码定义请参照头文件，比如 FAILED_OPEN_USER_DEV，在 FirmDriveCore.h 的末尾处。

```
#define ERRO_ADDRESS_OFFSET -5000
#define FAILED_OPEN_USER_DEV        ERRO_ADDRESS_OFFSET+0
```

2.1.4 FirmDrive_GetDevInfo()

```
/// ************************************************************************************
/// <summary>
/// 获取设备信息
/// </summary>
/// <param name="hDev">FirmDrive设备HANDLE</param>
/// <param name="DeviceInfo">设备信息，具体包含信息参考结构体FirmDriveInfo</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Yasing,2017/10/19</created>
/// <changed>Yasing,2017/12/14</changed>
/// ************************************************************************************
EXTERN_C   DLL_API   int   FirmDrive_GetDevInfo(FirmDriveDevice   hDev,   FirmDriveInfo
*DeviceInfo);
```

FirmDriveInfo设备信息结构体定义如下

```
typedef struct
{
#ifdef WINDOWS_IMPL
    GUID guid;//WINDOWS专用
#endif
    UINT32 VID;//vendor ID
    UINT32 PID;//product ID
    UINT32 SUBsysID;//Further qualifies the manufacture of the device or application,
this value is typicaly the same as the device ID.
    UINT32 RevisionID;//可以在XDMA IP核里面进行设置，表示当前固件版本号
    UINT32 SlotNum;
    BOOL fStreaming;//是否是stream接口
    UINT64 dmaBufSize;
    UINT32 bypassSize;//bypass地址空间大小
    UINT32 userBarSize;
}FirmDriveInfo;
```

2.1.5 FirmDrive_WriteReg()

```
/// ************************************************************************************
/// <summary>
/// 写入指定偏移地址寄存器
/// </summary>
/// <param name="hDev">FirmDrive设备HANDLE</param>
/// <param name="offset">偏移地址，一定是4的整数倍</param>
/// <param name="regVal">要写入寄存器的值</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Yasing,2017/10/19</created>
/// <changed>yasing,2018/1/16</changed>
/// ************************************************************************************
EXTERN_C DLL_API int FirmDrive_WriteReg(FirmDriveDevice hDev, UINT32 offset, INT32 regVal);
```

用于向 FPGA 固件上指定地址处写入寄存器值。如图 2-5 XDMA AXI4_LITE 与 User IP AXI4_LITE 连接方式所示：用户 IP 通过寄存器实现对 LED 灯状态的控制。寄存器读写通过 AXI4_LITE 总线挂接在 AXI Interconnect/Smart Connect 上实现总线的连接。

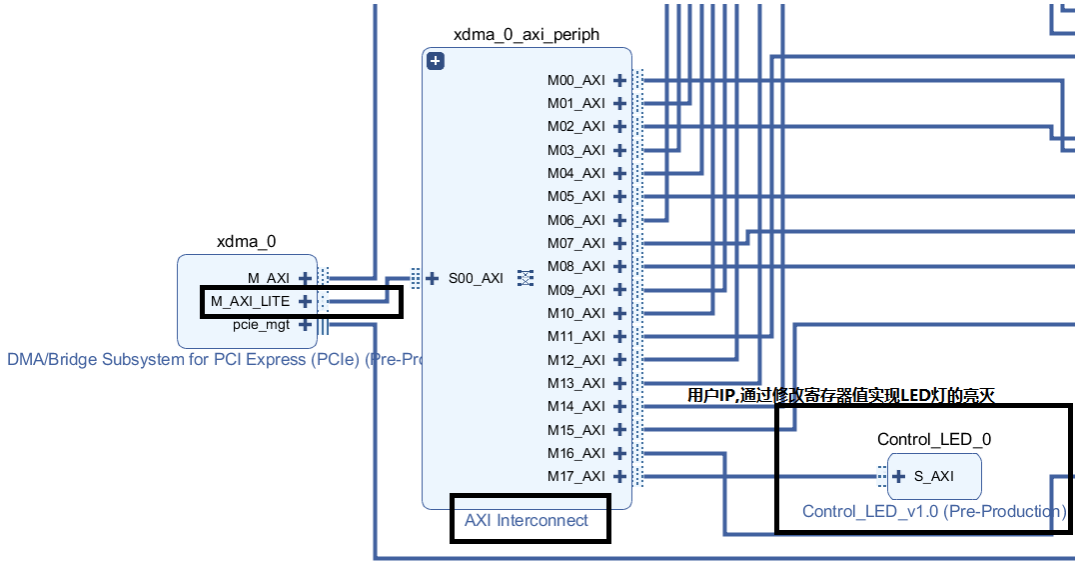


图 2-5 XDMA AXI4_LITE 与 User IP AXI4_LITE 连接方式

进行对应的连接后，还需要进行各个模块基地址的设置。如图 2-6 User IP 基地址设定所示，通过 VIVADO 中 Block Design 提供的 Address Editor 可以分配用户 IP 的地址空间。某寄存器对应的偏移地址如果是 0x4，那么其实际地址 offset=base_address+0x4，指定相应的寄存器提供的寄存器读写函数即可读写对应寄存器。

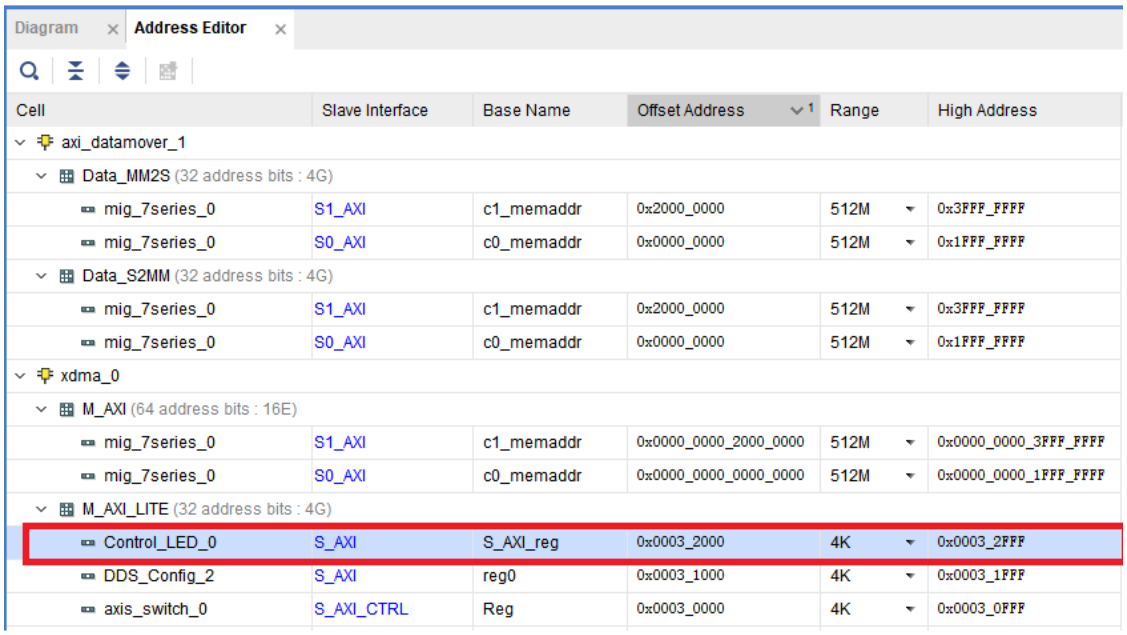


Diagram | Address Editor

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
axi_datamover_1					
Data_MM2S (32 address bits : 4G)					
mig_7series_0	S1_AXI	c1_memaddr	0x2000_0000	512M	0x3FFF_FFFF
mig_7series_0	S0_AXI	c0_memaddr	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
mig_7series_0	S1_AXI	c1_memaddr	0x2000_0000	512M	0x3FFF_FFFF
mig_7series_0	S0_AXI	c0_memaddr	0x0000_0000	512M	0x1FFF_FFFF
xdma_0					
M_AXI (64 address bits : 16E)					
mig_7series_0	S1_AXI	c1_memaddr	0x0000_0000_2000_0000	512M	0x0000_0000_3FFF_FFFF
mig_7series_0	S0_AXI	c0_memaddr	0x0000_0000_0000_0000	512M	0x0000_0000_1FFF_FFFF
M_AXI_LITE (32 address bits : 4G)					
Control_LED_0	S_AXI	S_AXI_reg	0x0003_2000	4K	0x0003_2FFF
DDS_Config_2	S_AXI	reg0	0x0003_1000	4K	0x0003_1FFF
axis_switch_0	S_AXI_CTRL	Reg	0x0003_0000	4K	0x0003_0FFF

图 2-6 User IP 基地址设定

比如想要点亮 Control_LED 模块对应的第一个 LED，则只需要向该模块的指定控制寄存器写入 LED 的对应位置为 1 即可。打开板卡后，调用代码 FirmDrive_WriteReg(hDev, 0x32000, 0x1)即可。

2.1.6 FirmDrive_ReadReg()

```
/// ********************************************************************************************
/// <summary>
/// 读取指定偏移地址寄存器,单次读取4个字节
/// </summary>
/// <param name="hDev">FirmDrive设备HANDLE</param>
/// <param name="offset">偏移地址，一定是4的整数倍</param>
/// <param name="regVal">寄存器的读出值</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Yasing,2017/10/19</created>
/// <changed>yasing,2018/1/16</changed>
/// ********************************************************************************************
EXTERN_C DLL_API int FirmDrive_ReadReg(FirmDriveDevice hDev, UINT32 offset, INT32 *regVal);
```

使用方法同 FirmDrive_ReadReg().

2.1.7 FirmDrive_Read_Memory()

```
/// ********************************************************************************
/// <summary>
/// 使用XDMA上的DMA Bypass接口读取指定偏移地址数据
/// </summary>
/// <param name="hDev">FirmDrive设备HANDLE</param>
/// <param name="pUserBuf">用户缓存</param>
/// <param name="offset">偏移地址，一定是4的整数倍</param>
/// <param name="length">读出数据长度</param>
/// <param name="pnActualBytesRead">实际读取的数据长度,bytes</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Yasing,2017/10/19</created>
/// <changed>yasing,2018/6/6</changed>
/// ********************************************************************************
EXTERN_C DLL_API int FirmDrive_Read_Memory(FirmDriveDevice hDev, PVOID pUserBuf, UINT32 offset, UINT32 length, UINT32 *pnActualBytesRead);
```

当固件开发人员使能 XDMA IP 的 DMA bypass 功能，并连接对应的存储资源。上位机允许用户以 Memory Map 的方式读写 bypass 映射的内存空间，使能 DMA Bypass 接口如图 2-7 DMA bypass 的使能所示：

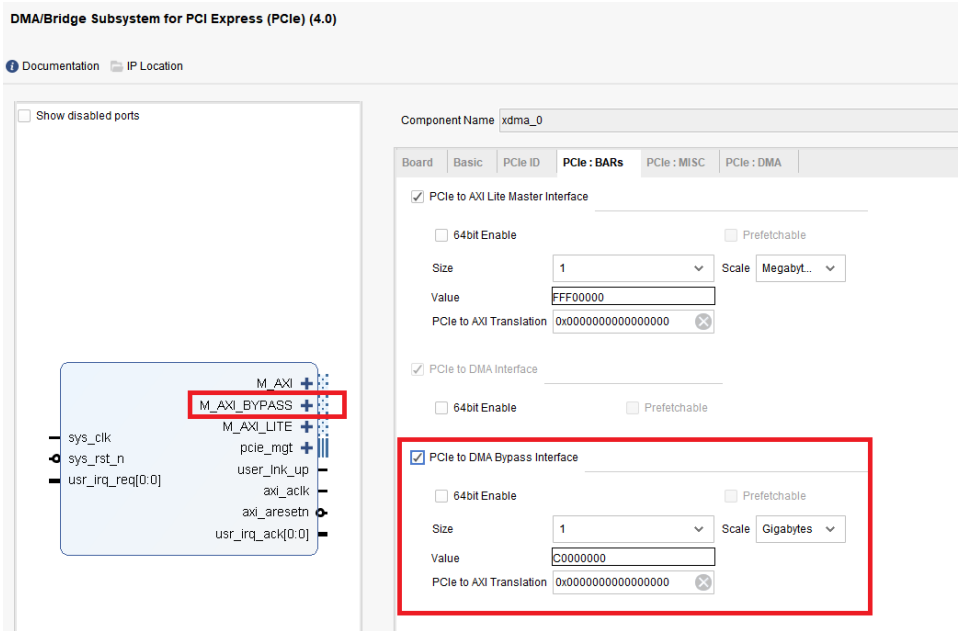


图 2-7 DMA bypass 的使能

2.1.8 FirmDrive_Write_Memory()

```
/// ************************************************************************************
/// <summary>
///使用XDMA上的DMA Bypass接口向指定地址处的数据
/// </summary>
/// <param name="hDev">FirmDrive设备HANDLE</param>
/// <param name="pUserBuf"></param>
/// <param name="offset">偏移地址，一定是4的整数倍</param>
/// <param name="length">寄存器的读出数据buffer</param>
/// <param name="pnActualBytesWritten">实际写入的数据长度，bytes</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Yasing,2017/10/19</created>
/// <changed>yasing,2018/6/6</changed>
/// ************************************************************************************
EXTERN_C DLL_API int FirmDrive_Write_Memory(FirmDriveDevice hDev, PVOID pUserBuf, UINT32
offset, UINT32 length, UINT32 *pnActualBytesWritten);
```

使用方法同 FirmDrive_Read_Memory()

2.1.9 FirmDrive_DMA_S_Start()

```
/// ************************************************************************************
/// <summary>
/// XDMA的DMA接口分为Memory Map和Streaming两种方式,FirmDrive_DMA_S_Start()用于启动
Streaming模式下的DMA传输
/// </summary>
/// <param name="hDev">FirmDrive设备HANDLE</param>
/// <param name="DMA_Direction">DMA方向，只能是DMA_C2H DMA_H2C</param>
/// <param name="ID">DMA通道，默认为0，DMA通道个数取决于固件开发人员XDMA中DMA通道数的配
置</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Yasing,2017/10/19</created>
/// <changed>Yasing,2017/12/19</changed>
/// ************************************************************************************
EXTERN_C DLL_API int FirmDrive_DMA_S_Start(FirmDriveDevice hDev, DMA_DIR DMA_Direction,
UINT32 ID);
```

用于开启 Streaming 模式下的 DMA 数据传输。

2.1.10 FirmDrive_DMA_S_Stop()

```
/// ************************************************************************************
/// <summary>
/// XDMA的DMA接口分为Memory Map和Streaming两种方式,FirmDrive_DMA_S_Stop()用于停止Streaming模式DMA传输
/// </summary>
/// <param name="hDev">FirmDrive设备HANDLE</param>
/// <param name="DMA_Direction">DMA方向, DMA_C2H或者DMA_H2C</param>
/// <param name="ID">DMA通道，默认为0，DMA通道个数取决于固件开发人员XDMA中DMA通道数的配置</param>
/// <returns>成功：0；失败： 错误码</returns>
/// <created>Yasing,2017/10/19</created>
/// <changed>Yasing,2017/12/19</changed>
/// ************************************************************************************
EXTERN_C DLL_API int FirmDrive_DMA_S_Stop(FirmDriveDevice hDev, DMA_DIR DMA_Direction, UINT32 ID);
```

2.1.11 FirmDrive_DMA_S_ReadC2H()

```
/// ************************************************************************************
/// <summary>
/// XDMA的DMA接口分为Memory Map和Streaming两种方式,FirmDrive_DMA_S_ReadC2H()用于在Streaming模式下从FPGA读取DMA数据
/// </summary>
/// <param name="hDev">FirmDrive设备HANDLE</param>
/// <param name="ID">DMA通道，默认为0，DMA通道个数取决于固件开发人员XDMA中DMA通道数的配置</param>
/// <param name="pUserBuf">用户内存</param>
/// <param name="nBytesToRead">单次指定读取数据大小，可以比config 时声明的DMA buffer size大</param>
/// <param name="nTimeout">延时时间，可根据当前实际传输速度设定，过小会导致无法在规定时间内读走所要求全部数据，发生数据错误</param>
/// <param name="pnActualBytesRead">当前读取结束后，实际读取到的数据量</param>
/// <param name="pnDataRemaining">当前读取结束前剩余可读数量</param>
/// <returns>成功：0； 错误：错误码</returns>
/// <created>Yasing,2017/10/31</created>
/// <changed>Yasing,2017/12/19</changed>
/// ************************************************************************************
EXTERN_C DLL_API int FirmDrive_DMA_S_ReadC2H(FirmDriveDevice hDev, UINT32 ID, PVOID pUserBuf, UINT32 nBytesToRead, UINT32 nTimeout,UINT32 *pnActualBytesRead, UINT32 *pnDataRemaining);
```

2.1.12 FirmDrive_DMA_S_WriteH2C()

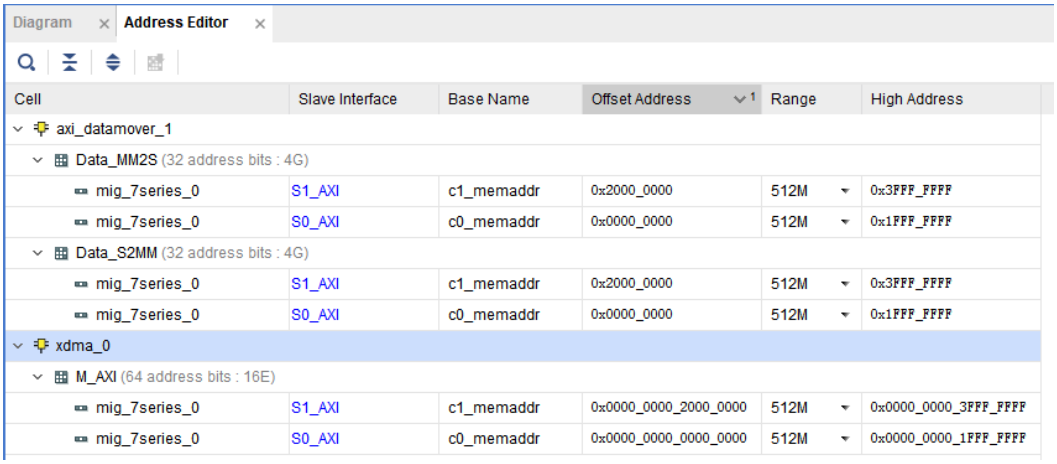
```
/// ************************************************************************************
/// <summary>
/// 在Streaming模式下向FPGA写入数据，目前暂未实现环形缓冲区
/// </summary>
/// <param name="hDev">FirmDrive设备HANDLE</param>
/// <param name="ID">DMA通道，默认为0，DMA通道个数取决于固件开发人员XDMA中DMA通道数的配置</param>
/// <param name="pUserBuf">用户内存</param>
/// <param name="nBytesToWrite">单次指定读取数据大小，可以比config 时声明的DMA buffer size大</param>
/// <param name="nTimeout">延时时间，可根据DMA 最大速度设定，由于DMA速度很快，一般设置100ms足够使用，设置过大并没有意义</param>
/// <param name="nActualBytesWriten">当前读取结束后，实际读取到的数据量</param>
/// <param name="nBufAvailable">当前读取结束前最后一次剩余可读数量</param>
/// <returns>成功：0； 错误：错误码</returns>
/// <created>Yasing,2017/10/31</created>
/// <changed>Yasing,2017/12/19</changed>
/// ************************************************************************************
EXTERN_C DLL_API int FirmDrive_DMA_S_WriteH2C(FirmDriveDevice hDev, UINT32 ID, PVOID pUserBuf, UINT32 nBytesToWrite,UINT32  nTimeout, UINT32* nActualBytesWriten, UINT32 *nBufAvailable);
```

2.1.13 FirmDrive_DMA_MM_ReadC2H()

```
// ************************************************************************************
/// <summary>
/// 在Memory map方式下从FPGA侧缓存指定地址处读取数据
/// </summary>
/// <param name="hDev">设备handle</param>
/// <param name="ID">DMA通道，默认为0，DMA通道个数取决于固件开发人员XDMA中DMA通道数的配置</param>
/// <param name="pUserBuf">用户内存</param>
/// <param name="offset">读取数据起始偏移地址</param>
/// <param name="length">请求读取数据长度</param>
/// <param name="pnActualBytesRead">实际读取数据量</param>
/// <returns>成功 0，失败 错误码</returns>
/// <created>Yasing,2017/11/28</created>
/// <changed>yasing,2018/8/9</changed>
// ************************************************************************************
EXTERN_C DLL_API int FirmDrive_DMA_MM_ReadC2H(FirmDriveDevice hDev, UINT32 ID, PVOID pUserBuf, UINT32 offset, UINT32 length, UINT32 *pnActualBytesRead);
```

使用指定的 DMA 通道从板卡缓存指定地址处读取固定长度的数据，并返回实际读取到的数据长度。比如在某型号板卡固件中，使用该函数 DMA 读取 DDR 上的数据，

memory 地址分配如图 2-8 DDR 地址分配所示，DDR 地址空间(图中 mig_7series_0 为该板卡 DDR)为 0x0000_0000 到 0x3FFF_FFFF，用户指定的读写起始地址 offset+length 需要落在该范围内。



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
axi_datamover_1					
Data_MM2S (32 address bits : 4G)					
mig_7series_0	S1_AXI	c1_memaddr	0x2000_0000	512M	0x3FFF_FFFF
mig_7series_0	S0_AXI	c0_memaddr	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
mig_7series_0	S1_AXI	c1_memaddr	0x2000_0000	512M	0x3FFF_FFFF
mig_7series_0	S0_AXI	c0_memaddr	0x0000_0000	512M	0x1FFF_FFFF
xdma_0					
M_AXI (64 address bits : 16E)					
mig_7series_0	S1_AXI	c1_memaddr	0x0000_0000_2000_0000	512M	0x0000_0000_3FFF_FFFF
mig_7series_0	S0_AXI	c0_memaddr	0x0000_0000_0000_0000	512M	0x0000_0000_1FFF_FFFF

图 2-8 DDR 地址分配

2.1.14 FirmDrive_DMA_MM_WriteH2C()

```
// ********************************************************************************
/// <summary>
///  在Memory map方式下向FPGA侧缓存指定地址处写入数据
/// </summary>
/// <param name="hDev">设备handle</param>
/// <param name="ID">DMA通道，默认为0，DMA通道个数取决于固件开发人员XDMA中DMA通道数的配置</param>
/// <param name="pUserBuf">用户内存</param>
/// <param name="Offset">写入地址偏移量</param>
/// <param name="length">请求写入数据长度</param>
/// <param name="pnActualBytesWriten">实际写入数据量</param>
/// <returns>成功 0，失败 错误码</returns>
/// <created>Yasing,2017/11/28</created>
/// <changed>yasing,2018/8/9</changed>
// ********************************************************************************
EXTERN_C DLL_API int FirmDrive_DMA_MM_WriteH2C(FirmDriveDevice hDev, UINT32 ID, PVOID pUserBuf, UINT32 Offset, UINT32 length, UINT32* pnActualBytesWriten);
```

用法同 FirmDrive_DMA_MM_ReadC2H()。

2.1.15 FirmDrive_DMA_S_CheckStatus()

```
// ********************************************************************************
/// <summary>
/// 当XDMA运行在Streaming格式下，检查DMA状态
/// </summary>
/// <param name="hDev">FirmDrive设备HANDLE</param>
/// <param name="DMA_Direction">DMA_TYPE: DMA_C2H或者DMA_H2C</param>
/// <param name="ID">DMA通道，默认为0，DMA通道个数取决于固件开发人员XDMA中DMA通道数的配置</param>
/// <param name="pnDataRemaining">DMA_C2H 缓冲区剩余可读数据大小</param>
/// <param name="pnBufAvailable">DMA_H2C 剩余的可写入数据大小，即内部缓冲区的空余空间大小</param>
/// <param name="pnDataTransfered">DMA 已经传输完成(Read/Write)的数据大小</param>
/// <param name="pfOverflow">DMA缓冲区是否已经停止/DMA是否已经停止</param>
/// <returns>成功： 0； 错误： 错误码</returns>
/// <created>Yasing,2017/10/19</created>
/// <changed>yasing,2018/8/9</changed>
// ********************************************************************************
EXTERN_C DLL_API int FirmDrive_DMA_S_CheckStatus(FirmDriveDevice hDev, DMA_DIR DMA_Direction, UINT32 ID, UINT32 *pnDataRemaining,UINT32 *pnBufAvailable, UINT64 *pnDataTransfered, BOOL *pfOverflow);
```

3 功能板和 PXIe-1010 总线控制器的连接

当您的硬件初步设计完毕后，您需要将您的硬件连接到简仪科技提供的 PXIe-1010 总线控制器，详见第 9 章。这有几个步骤：1）功能板的 AD/DA 等模块的控制信号按照按照简仪科技提供的方式连接到 AXI-Lite 总线上；2）将 AD/DA 等数据连接到简仪科技提供的 FirmDrive®收发引擎固件上，详见 FirmDrive®引擎及高层次接口；3）通过 Vivado 连接有关 FPGA 的引脚。

在顺利完成以上三步后，您就可以使用 FirmDriveCore C API 来打开、调用、数据传输、关闭您的硬件了。

为了方便您的设计工作和了解 FirmDrive®的工作方法，简仪科技提供了 PXIe-1010DK 开发套件。本章将以 PXIe-1010DK 为例来解释如何连接功能板（子板）和 PXIe-1010 总线控制器（母板）。

3.1 PXIe-1010DK 开发套件

PXIe-1010DK 开发套件由三部分组成：PXIe-1010 总线控制模块（母板）、DB-102 功能板（子板）和已经载入 FPGA 的 FirmDrive®驱动。

DB-102 功能板包含 2 通道 AD、2 通道 DA、8 通道 DI、8 通道 DO 及 2 通道 Counter。实现 DB-102 上述全部功能的 FirmDrive®驱动设计资料将被作为参考设计提供。简仪在开发套件中提供了所有的功能子板的硬件设计图、所有的 AD、DA 的固件 IP 源程序以及这些固件 IP 与 FirmDrive®的收发引擎的连接图。

DB-102 功能版固件整体结构如下图 3-1 DB-102 固件结构图所示，图示中隐藏了了时钟、复位及少量的功能信号线，以使得重要的数据总线更加清晰。本文其他章节中详细地介绍了实现这一涉及的方案及涉及的关键技术，此处概略地介绍整体设计：

- XDMA 是沟通 PCIe 总线和 FirmDrive®内信号的桥梁，具有 AXI-Lite 及 Memory Mapped 形式的 AXI 两个主要接口。XDMA 通过借助 AXI Interconnect 模块扇出多个 AXI-Lite 接口，并利用 AXI-Lite 总线实现对收发引擎、功能模块等的控制与状态获取。XDMA 使用 Memory Mapped 形式的 AXI 总线访问板载 DDR 内存，而板载 DDR 内存是 XDMA 与固件其他模块交换数据的媒介。

- RxEngine、TxEngine 是控制数据采样的核心模块，在使用 FirmDrive®构建的固件中起到控制数据采集、传输、存储的作用。RxEngine、TxEngine 是沟通功能组件（如 AIO、DIO 采样模块，C 模块等）与板载 DDR 内存的桥梁，它们使用多个 AXI-Stream 接口完成数据的传输。

- RxEngine 用于将具有输入功能的模块采集的数据写入 DDR 内存，以供 XDMA 获取；TxEngine 从 DDR 内存读取由 XDMA 存入的数据，并传输给输出功能的模块。

- RxEngine、TxEngine 总是与 AXI DataMover 配合使用，AXI DataMover 用于实现 Memory Mapped 形式的 AXI 总线数据与 AXI-Stream 总线数据的双向转换。高速数据在 AXI DataMover 与板载内存之间使用 Memory Mapped 形式的 AXI 总线传输，而在固件内的其他位置使用 AXI-Stream 总线传输。

- MIG 模块用于控制板载 DDR 内存，并利用 AXI SmartConnect 模块实现多个 Memory Mapped 形式的 AXI 总线接口，使得 XDMA 与 TxEngine、RxEngine 都可以通过 AXI 接口访问同一物理内存。

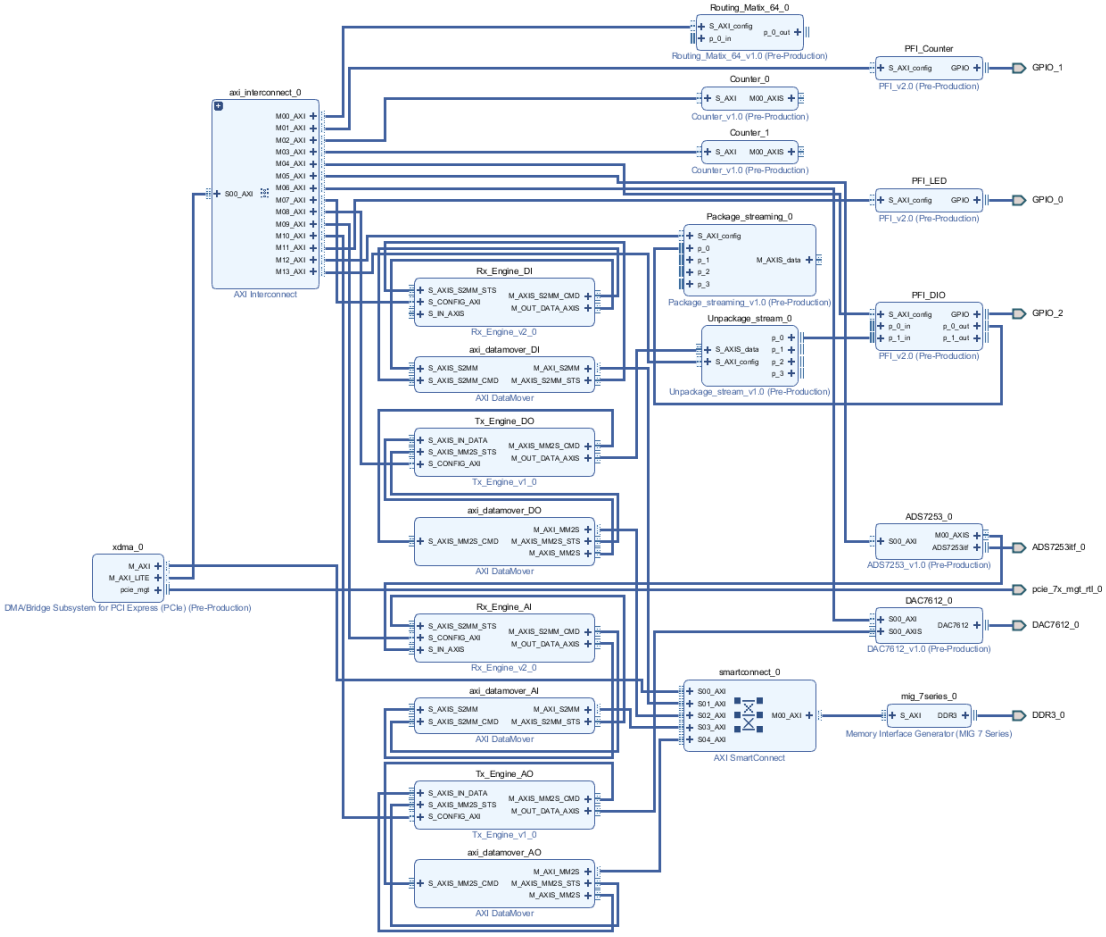


图 3-1 DB-102 固件结构图

图 3-1 DB-102 固件结构图的右侧显示了适用于 DB-102 功能板的固件外部接口，即板载 FPGA 的物理 IO 引脚。这些引脚可以进行以下分类：

- 数字信号接口：连接外部数字信号的接口，这些信号可以是 DIO 信号、Counter 信号或是参考时钟、同步信号，这些信号只需将相应的 IP 模块接口定义为外部接口，并进行基础的约束即可完成物理连接。

- 外设接口：驱动外设芯片的物理接口，如连接 DB-102 功能板的 ADC、DAC 芯片的 SPI 接口，这一类接口需要根据针这些外设的 IP 设计需要定义。在一般情况下针对外设芯片的 IP 模块及接口需要使用者进行设计，DB-102 的参考设计中包含了适配 DB-102 板载 ADC、DAC 芯片的 IP 和接口。

- PXIe-1010 总线控制模块基础功能接口：主要包括 PCIe 总线接口、板载 DDR 内存接口等，这些接口由 PCIe 技术规范、简仪科技 FirmDrive®定义，只需直接调用即可。除上述必要的接口外，FirmDrive®还提供了 PXIe、PXI 系统用所需的背板接口，如 Trig0-7、PXI_CLK_100、PXI_SYNC_100。

FirmDrive®提供的固件模块均可以实现多实例化，通过为这些模块分配不同的AXI-Lite 地址可以实现独立地控制各个模块。下图 3-2 DB-102 固件模块地址分配展示了在 DB-102 固件整体设计中的各个模块的地址分配形况。除了 AXI-Lite 地址外，板载 DDR 内存的地址范围同样需要分配给相应的模块，由于 FirmDrive®的 TxEngine、RxEngine 模块具备软件可配置的内存管理能力，DDR 内存的读写能够在 TxEngine、RxEngine 模块的控制下有序进行，因而所有访问板载 DDR 内存的设备均可分配全部 DDR 内存空间，图 3-2 DB-102 固件模块地址分配的示例中即采用这样的方案。

Diagram | Address Editor

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
xdma_0					
M_AXI (64 address bits : 16E)					
mig_7series_0	S_AXI	memaddr	0x0000_0000_0000_0000	1G	0x0000_0000_3FFF_FFFF
M_AXI_LITE (32 address bits : 4G)					
ADS7253_0	S00_AXI	S00_AXI_reg	0x0000_2000	4K	0x0000_2FFF
Counter_0	S_AXI	S_AXI_reg	0x0000_7000	4K	0x0000_7FFF
Counter_1	S_AXI	S_AXI_reg	0x0000_8000	4K	0x0000_8FFF
DAC7612_0	S00_AXI	S00_AXI_reg	0x0000_1000	4K	0x0000_1FFF
PFI_LED	S_AXI_config	S_AXI_config_reg	0x0000_4000	4K	0x0000_4FFF
PFI_DIO	S_AXI_config	S_AXI_config_reg	0x0000_3000	4K	0x0000_3FFF
PFI_Counter	S_AXI_config	S_AXI_config_reg	0x0000_5000	4K	0x0000_5FFF
Package_streaming_0	S_AXI_config	S_AXI_config_reg	0x0000_E000	4K	0x0000_EFFF
Routing_Matix_64_0	S_AXI_config	S_AXI_config_reg	0x0000_6000	4K	0x0000_6FFF
Rx_Engine_AI	S_CONFIG_AXI	reg0	0x0000_A000	4K	0x0000_AFFF
Rx_Engine_DI	S_CONFIG_AXI	reg0	0x0000_B000	4K	0x0000_BFFF
Tx_Engine_AO	S_CONFIG_AXI	reg0	0x0000_C000	4K	0x0000_CFFF
Tx_Engine_DO	S_CONFIG_AXI	reg0	0x0000_D000	4K	0x0000_DFFF
Unpackage_stream_0	S_AXI_config	Reg	0x0000_F000	4K	0x0000_FFFF
axi_datamover_DO					
Data_MM2S (32 address bits : 4G)					
mig_7series_0	S_AXI	memaddr	0x0000_0000	1G	0x3FFF_FFFF
axi_datamover_AI					
Data_S2MM (32 address bits : 4G)					
mig_7series_0	S_AXI	memaddr	0x0000_0000	1G	0x3FFF_FFFF
axi_datamover_AO					
Data_MM2S (32 address bits : 4G)					
mig_7series_0	S_AXI	memaddr	0x0000_0000	1G	0x3FFF_FFFF
axi_datamover_DI					
Data_S2MM (32 address bits : 4G)					
mig_7series_0	S_AXI	memaddr	0x0000_0000	1G	0x3FFF_FFFF

图 3-2 DB-102 固件模块地址分配

除 FirmDrive®提供的固件 IP 和驱动接口外，实现 AI、AO 功能需要根据子板的 ADC、DAC 芯片设计接口 IP，并实现相应的接口；而 DI、DO、Counter 功能无需单独编制固件 IP，只需要连接简仪提供的相应功能固件 IP 即可。匹配 DB-102 板卡 ADC、DAC 芯片的固件 IP 与软件接口将随其他设计资料一并提供。

3.2 AI 的固件制作和连接

DB-102 板载 ADC 芯片型号为 ADS7253。使用 FirmDrive®访问 ADC 器件需要对应的接口 IP，开发套件中使用 IP ADS7253 访问 ADC 芯片，封装后的 IP 见图 3-3。固件 IP 的源程序详见开发套件。

图 3-3 开发套件 ADC 接口 IP

IP ADS7253 采用典型的上行功能 IP 接口配置：发送数据的 AXI-Stream Master 总线接口、传输配置信息的 AXI-Lite Slave 接口和连接 ADC 芯片数字接口的物理接口 ADS7253itf。接口定义如下：

- S00_AXI：AXI-Lite Slave 总线数据接口，双工传输，用于接入 XDMA 的 AXI-Lite 总线，传输配置信息；
- s00_axi_aclk：AXI-Lite Slave 总线时钟接口；
- s00_axi_aresetn：AXI-Lite Slave 总线复位接口；
- m00_axis_aclk：AXI-Stream Master 总线时钟接口；
- m00_axis_aresetn：AXI-Stream Master 总线复位接口；
- M00_AXIS：AXI-Stream Master 总线数据接口，上行单工传输，用于向 RxEngine 发送数据；
- ADS7253itf：连接 ADC 的物理接口，通过 FPGA 物理 IO 与 ADC 芯片各通信引脚相连。

AXI-Stream Master 总线接口包含 M00_AXIS、m00_axis_aclk、m00_axis_aresetn 三组信号。M00_AXIS 信号需要连接至 RxEngine 的 S_IN_AXIS 接口（即 RxEngine 的 AXI-Stream 数据输入接口）。m00_axis_aclk、m00_axis_aresetn 分别为总线时钟、总线复位接口，在使用中需要与 RxEngine 的 s_in_axis_aclk、s_in_axis_aresetn（即 RxEngine 的 AXI-Stream 时钟、复位接口）使用同一时钟、复位信号。相关信号连接见图 3-4。

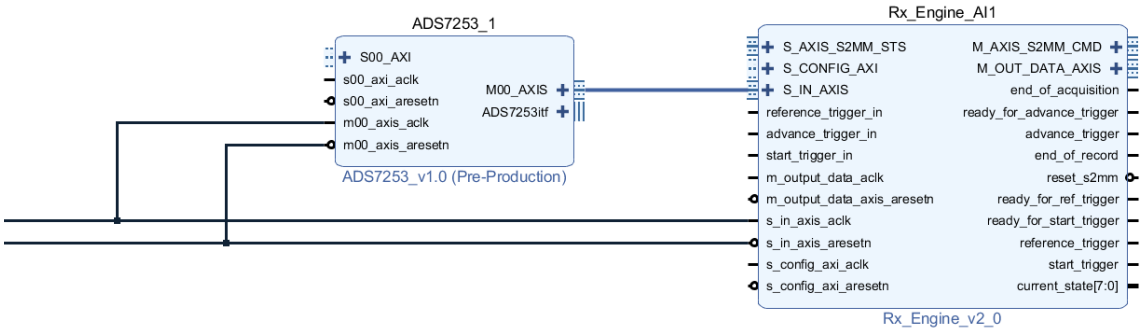


图 3-4 ADC 接口 IP 连接 Rx_Engine

AXI-Lite Slave 总线接口包含 S00_AXI、s00_axi_aclk、s00_axi_aresetn 三组信号。S00_AXI 接口需要连接至 AXI Interconnect 实例的 M**_AXI 接口，m00_axis_aclk、m00_axis_aresetn 接口需要与 AXI Interconnect 的 M**_ACLK、M**_ARESETN 使用同一时钟、复位信号。相关信号连接见图 3-5。

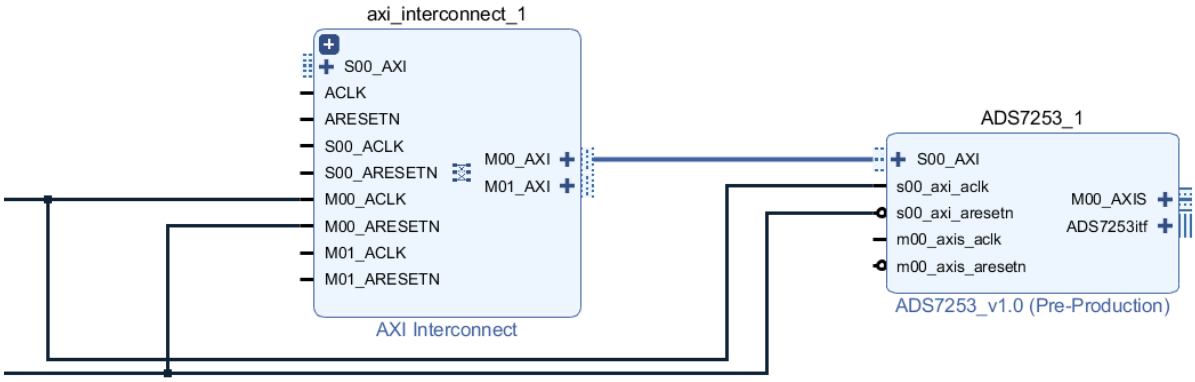


图 3-5 ADC 接口 IP 连接 AXI-Lite 总线

ADS7253itf 接口包含 ADC_DIN、ADC_CS、ADC_CLK、ADC_DOUT_A、ADC_DOUT_B 共 5 个 1 位信号线，与 ADC 接口管脚一一对应。ADS7253 接口仅需配置为外部接口（External），并在约束中映射至 FPGA 相应物理引脚即可，如下图 3-6。

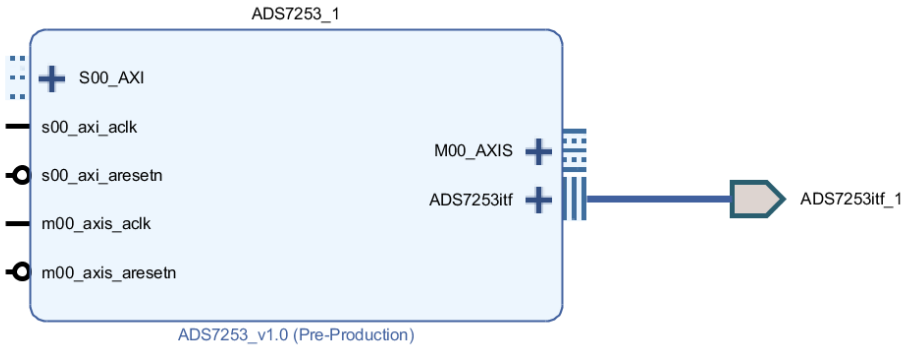


图 3-6 ADC 接口 IP 接口物理连接

IP ADS7253 使用 VIVADO “Create and Package New IP”工具创建 AXI-Stream Master 与 AXI-Lite Slave 接口的 Verilog 模板，进而在模板基础上通过“重构接口”、“重新定义寄存器驱动”、“加入用户逻辑”实现目标功能。

- 重构接口：重构接口是为了更简单地实现逻辑功能。“Create and Package New IP”工具会针对每一个接口生成独立的 Verilog 模板文件，重构接口首先将 AXI-Stream Master 接口定义并入 AXI-Lite Slave 接口模板文件，而后在接口中加入 ADC 物理接口的定义。
- 重新定义寄存器驱动：对于 AXI-Lite Slave 接口，模板创建的逻辑代码会默认由 AXI-Lite Slave 接口驱动寄存器（写寄存器值），若用户需要些这些寄存器值，则需要自覆盖原有驱动逻辑。
- 加入用户逻辑：在模板文件中加入用户逻辑，主要包括驱动 ADC 芯片和驱动 AXI-Stream Master 接口内容。

下图 3-7 AI 数据链路图示展示了 DB-102 子板 ADC 数据在固件中的传递关系：

- ADS7253 模块通过 ADS7253itf 接口连接 ADC 芯片，并获取 AD 转换数据；
- ADS7253 模块通过 AXI-Stream Master 接口发送数据给 RxEngine 模块；
- RxEngine 模块通过自身的 AXI-Stream Master 接口发送数据到 AXI DataMover 模块；
- 数据在 AXI DataMover 由 AXI-Stream 形式转变为 MemoryMapped 形式，并经由 AXI DataMover 的 AXI 接口发送给 AXI SmartConnect；
- AXI SmartConnect 将数据传递给 MIG 模块，最终由 MIG 模块将数据送入板载 DDR 内存。

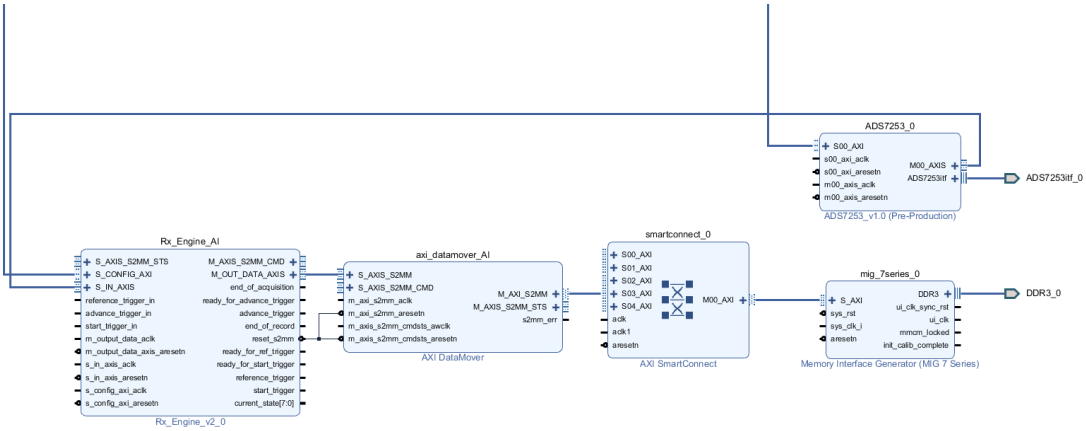


图 3-7 AI 数据链路图示

除了传送数据的 AXI-Stream 接口外，RxEngine 与 AXI DataMover 之间还存在两组 AXI-Stream 接口，这些接口用于在 RxEngine 与 AXI DataMover 之间传递控制和状态信息，使用者不必更多关注这些内容，仅需参照 RxEngine 相关章节内容进行配置即可。

作为 PXIe 数据通路的 XDMA 组件通过连接 AXI SmartConnect 访问 MIG 模块，从而实现 PXIe 总线从板载 DDR 内存中获取数据。ADS7253 模块和 RxEngine 模块还具有 AXI-Lite

Slave 接口，这些接口需要连接至 XDMA 的 AXI-Lite 总线系统。详情参见 FirmDrive®引擎相关资料。

3.3 AO 的固件编写和连接

DB-102 板载 DAC 芯片型号为 DAC7612。使用 FirmDrive®访问 ADC 器件需要对应的接口 IP，开发套件中使用 IP DAC7612 访问 DAC 芯片，封装后的 IP 见图 3-8。固件 IP 的源程序详见开发套件。

图 3-8 开发套件 DAC 接口 IP

IP DAC7612 采用典型的上行功能 IP 接口配置：接收数据的 AXI-Stream Master 总线接口、传输配置信息的 AXI-Lite Slave 接口和连接 ADC 芯片数字接口的物理接口 DAC7612。接口定义如下：

- S00_AXI：AXI-Lite Slave 总线数据接口，双工传输，用于接入 XDMA 的 AXI-Lite 总线，传输配置信息；
- S00_AXIS：AXI-Stream Slave 总线数据接口，下行单工传输，用于接收 TxEngine 传来的数据；
- s00_axi_aclk：AXI-Lite Slave 总线时钟接口；
- s00_axi_aresetn：AXI-Lite Slave 总线复位接口；
- s00_axis_aclk：AXI-Stream Slave 总线时钟接口；
- s00_axis_aresetn：AXI-Stream Slave 总线复位接口；
- DAC7612：连接 ADC 的物理接口，通过 FPGA 物理 IO 与 ADC 芯片各通信引脚相连。

AXI-Stream Slave 总线接口包含 S00_AXIS、s00_axis_aclk、s00_axis_aresetn 三组信号。M00_AXIS 信号需要连接至 TxEngine 的 M_OUT_DATA_AXIS 接口（即 TxEngine 的 AXI-Stream 数据输出接口）。s00_axis_aclk、s00_axis_aresetn 分别为总线时钟、总线复位接

口，在使用中需要与 TxEngine 的 m_axis_out_aclk、m_axis_out_aresetn（即 TxEngine 的 AXI-Stream 时钟、复位接口）使用同一时钟、复位信号。相关信号连接见图 3-9。

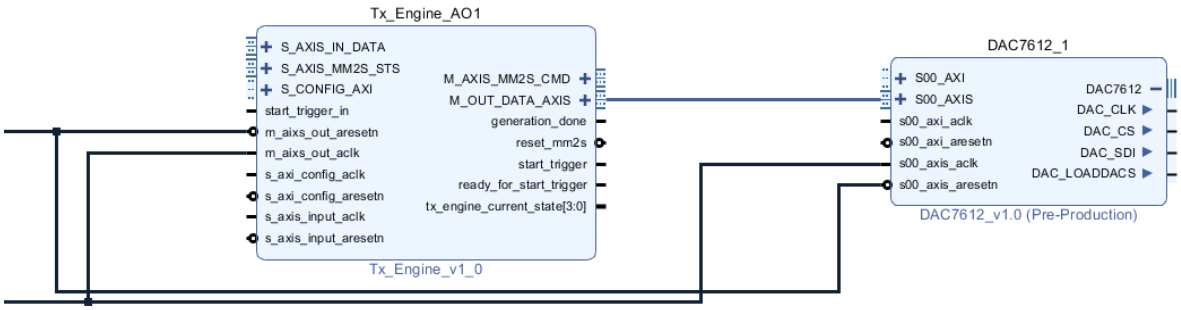


图 3-9 DAC 接口 IP 连接 Tx_Engine

AXI-Lite Slave 总线接口包含 S00_AXI、s00_axi_aclk、s00_axi_aresetn 三组信号。S00_AXI 接口需要连接至 AXI Interconnect 实例的 M**_AXI 接口，m00_axis_aclk、m00_axis_aresetn 接口需要与 AXI Interconnect 的 M**_ACLK、M**_ARESETN 使用同一时钟、复位信号。相关信号连接见图 3-10。

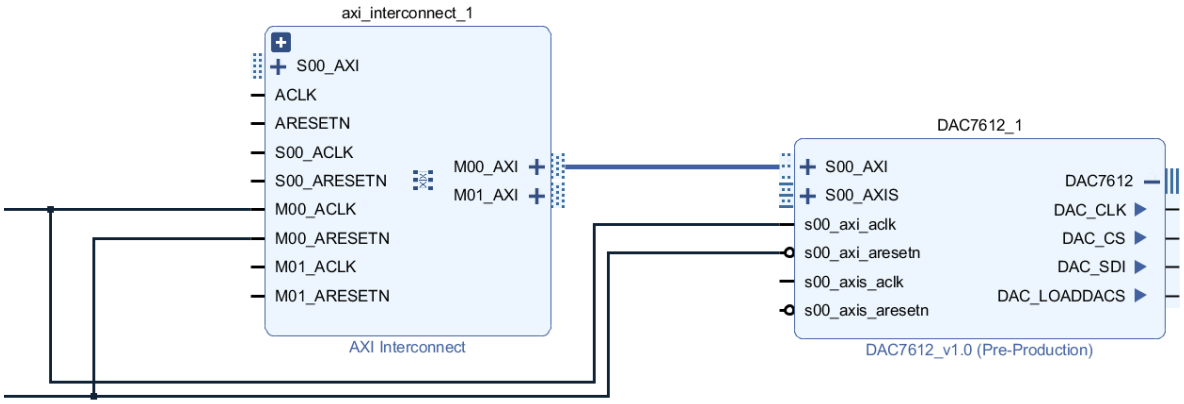


图 3-10 DAC 接口 IP 连接 AXI-Lite 总线

DAC7612 接口包含 DAC_CLK、DAC_CS、DAC_SDI、DAC_LOADDACS 共 4 个 1 位信号线，与 DAC 接口管脚一一对应。DAC7612 接口仅需配置为外部接口（External），并在约束中映射至 FPGA 相应物理引脚即可，如下图 3-11。

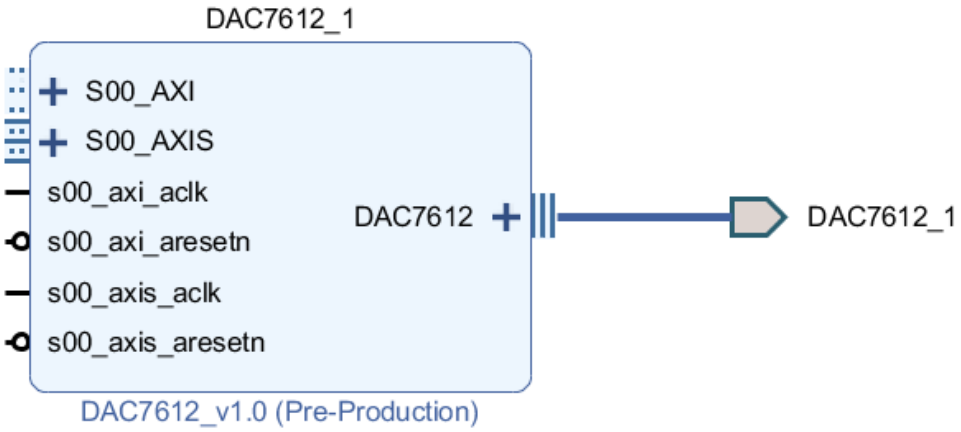


图 3-11 DAC 接口 IP 物理连接

IP DAC7612 使用 VIVADO “Create and Package New IP”工具创建 AXI-Stream Slave 与 AXI-Lite Slave 接口的 Verilog 模板，进而在模板基础上通过“重构接口”、“重新定义寄存器驱动”、“加入用户逻辑”实现目标功能。

- 重构接口：重构接口是为了更简单地实现逻辑功能。“Create and Package New IP”工具会针对每一个接口生成独立的 Verilog 模板文件，重构接口首先将 AXI-Stream Slave 接口定义并入 AXI-Lite Slave 接口模板文件，而后在接口中加入 ADC 物理接口的定义。

- 重新定义寄存器驱动：对于 AXI-Lite Slave 接口，模板创建的逻辑代码会默认由 AXI-Lite Slave 接口驱动寄存器（写寄存器值），若用户需要些这些寄存器值，则需要自覆盖原有驱动逻辑。

- 加入用户逻辑：在模板文件中加入用户逻辑，主要实在 AXI-Lite Slave 接口的驱动下控制 DAC 芯片、在 AXI-Stream Slave 接口驱动下向 DAC 传输数据。

下图 3-12 AO 数据链路图展示了 DB-102 子板 DAC 数据在固件中的传递关系：

- 在 TxEngine 的间接控制下，MIG 模块从板载 DDR 内存读出数据，并传递给 AXI SmartConnect；

- AXI SmartConnect 将数据传递给 AXI DataMover；

- 数据在 AXI DataMover 由 MemoryMapped 形式转变为 AXI-Stream 形式，并经由 AXI DataMover 的 AXI-Stream 接口发送给 TxEngine；

- TxEngine 模块通过自身的 AXI-Stream Master 接口发送数据到 DAC7612 模块；

- DAC7612 模块通过 DAC7612 接口连接 DAC 芯片，并传输待转换的 DA 数据。

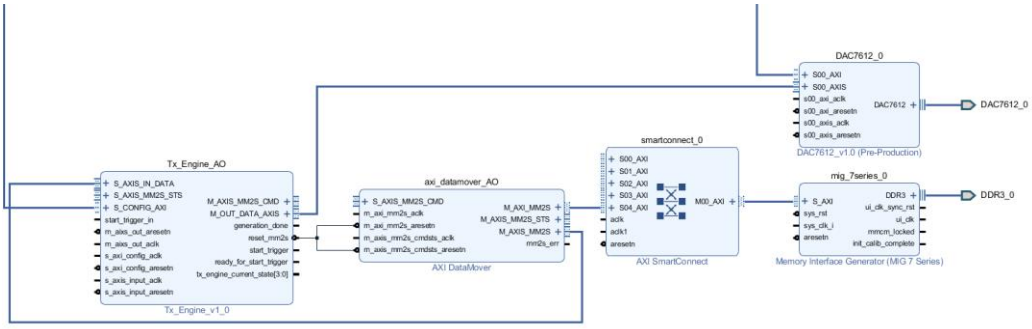


图 3-12 AO 数据链路图

除了传送数据的 AXI-Stream 接口外，TxEngine 与 AXI DataMover 之间还存在两组 AXI-Stream 接口，这些接口用于在 TxEngine 与 AXI DataMover 之间传递控制和状态信息，使用者不必更多关注这些内容，仅需参照 RxEngine 相关章节内容进行配置即可。

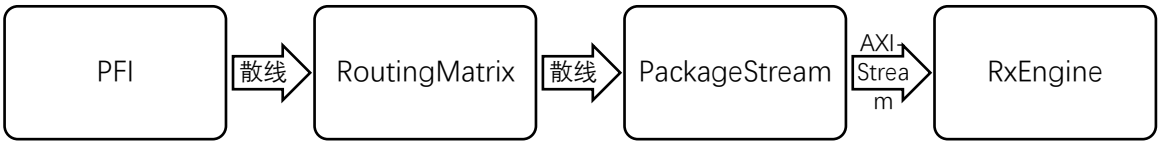
作为 PXIe 数据通路的 XDMA 组件通过连接 AXI SmartConnect 访问 MIG 模块，从而实现 PXIe 总线向板载 DDR 内存中写入数据，而这些数据将经由上述通路被最终发送到 DAC 芯片。DAC7612 模块和 TxEngine 模块还具有 AXI-Lite Slave 接口，这些接口需要连接至 XDMA 的 AXI-Lite 总线系统。详情参见 FirmDrive®引擎相关资料。

3.4 DI 的固件编写和连接

FirmDrive®包含通用数字测量组件，仅需调用、重组这些固件即可实现 DI 功能，所用组件包括 PFI、RoutingMatrix、PackageStream。

在设计 DI 系统时仅需要将外部输入信号连接至 PFI 的物理端口，通过 PFI 的 p_0_out 接口即可获取这些信号，并在 FPGA 内任意使用。在 FirmDrive®中可以简单地以下面的方式使用数字功能组件实现 DI 信号采集：

- PFI：软件配置方向的外部 IO 信号路由，在 DI 应用中需配置为输入模式。
- RoutingMatrix：FPGA 内的路由工具，在 DI 应用中能够重组 PFI 传入的数字信号。
- PackageStream：AXI-Stream 封装工具，可以在时钟驱动下将一组数字信号散线封装为 AXI-Stream 总线数据，以传输给 RxEngine。



DB-102 配套的固件为降低复杂程度没有使用 RoutingMatrix，采用直接将 PFI 数字信号连接至 PackageStream 的方式。DI 数字连接如下图 3-13。

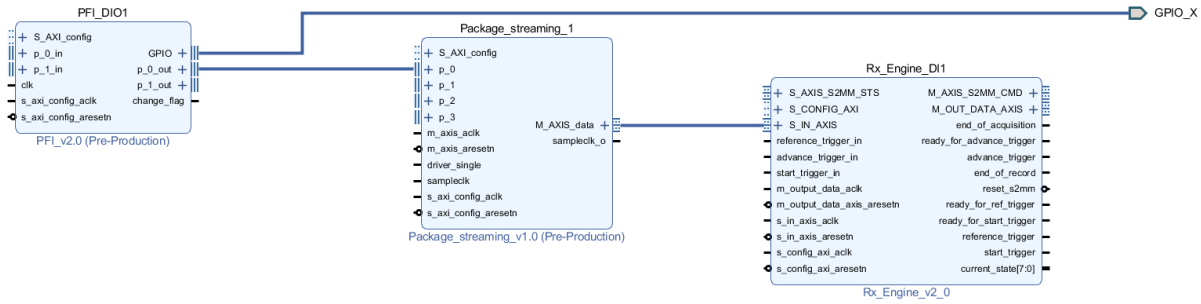


图 3-13 DI 关键信号连接

- PFI 接口 GPIO 为方向软件可控的信号路由接口，用于路由物理连接，仅需配置为外部接口（External），并在约束中映射至 FPGA 相应物理引脚即可。在 DI 应用中 PFI 被配置为输入模式，信号由物理引脚（GPIO 接口）向 PFI 的 p_*_out 接口传输。在输入模式下，PFI 的 p_*_out 接口用于输出外部路由进来的数字信号。

- PFI 完成信号路由不需要时钟驱动，因而 PFI 实例仅需接入 AXI-Lite Slave 的时钟 s_axi_comfig_aclk 和复位 s_axi_comfig_aresetn，以支持 AXI-Lite Slave 接口。PFI 的 AXI-Lite Slave 总线需要接入 XDMA 的 AXI-Lite 总线，其数据接口为 S_AXI_config，应与 AXI Interconnect 实例的 M**_AXI 接口相连。

- PackageStream 由 p_*接口输入信号，该接口不需要时钟驱动，可以直接连接在 PFI 的 p_*_out 接口。p_*接口的信号封装后经由 AXI-Stream Master 传输给 TxEngine，因而 PackageStream AXI-Stream Master 总线的 M_AXIS_data 接口需要连接 RxEngine 的 S_IN_AXIS 接口，PackageStream 的 m_axis_aclk、m_axis_aresetn 接口需要与 RxEngine 的 s_in_axis_aclk、s_in_axis_aresetn 接口连接同一时钟、复位信号。

3.5 DO 的固件编写和连接

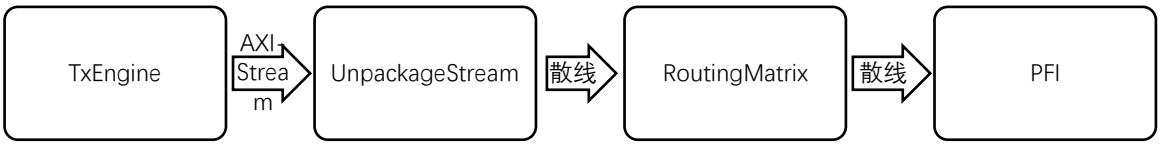
FirmDrive®包含通用数字测量组件，仅需调用、重组这些固件即可实现 DO 功能，所用组件包括 PFI、RoutingMatrix、UnpackageStream。

在设计 DO 系统时仅需要将外部输入信号连接至 PFI 的物理端口，并通过对 PFI 的 p_i_out 接口施加逻辑即可向外发送这些数据。在 FirmDrive®中可以简单地以下面的方式使用数字功能组件实现 DO 信号输出：

- PFI：软件配置方向的外部 IO 信号路由，在 DO 应用中需配置为输出模式。

- RoutingMatrix：FPGA 内的路由工具，在 DO 应用中能够重组 PackageStream 解码的数字信号。

- UnpackageStream：AXI-Stream 解码工具，可以在时钟驱动下将 AXI-Stream 总线数据转换为一组数字散线，以传输给路由组件。



DB-102 配套的固件为降低复杂程度没有使用 RoutingMatrix，采用直接将 PackageStream 数字信号连接至 PFI 的方式。DO 数字连接如下图 3-14。

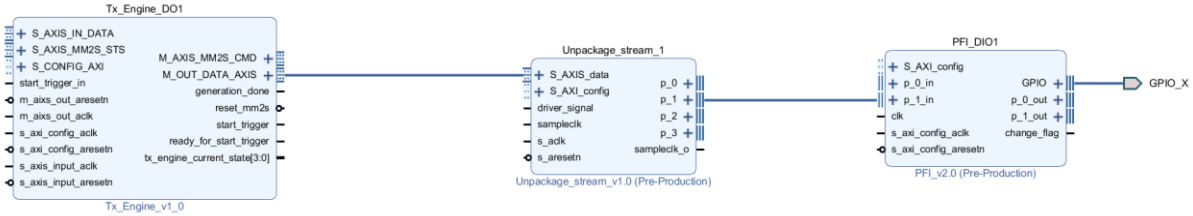


图 3-14 DO 关键信号连接

- PFI 接口 GPIO 为方向软件可控的信号路由接口，用于路由物理连接，仅需配置为外部接口（External），并在约束中映射至 FPGA 相应物理引脚即可。在 DO 应用中 PFI 被配置为输出模式，信号由 PFI 的 p_*_in 接口向物理引脚（GPIO 接口）传输。在输出模式下，PFI 的 p_*_in 接口用于输入需要向外输出的数字信号。

- PFI 完成信号路由不需要时钟驱动，因而 PFI 实例仅需接入 AXI-Lite Slave 的时钟 s_axi_comfig_aclk 和复位 s_axi_comfig_aresetn，以支持 AXI-Lite Slave 接口。PFI 的 AXI-Lite Slave 总线需要接入 XDMA 的 AXI-Lite 总线，其数据接口为 S_AXI_config，应与 AXI Interconnect 实例的 M**_AXI 接口相连。

- UnpackageStream 由 p_*接口输出信号，该接口不需要时钟驱动，可以直接连接在 PFI 的 p_*_in 接口。p_*接口的信号封装后经由 AXI-Stream Master 传输给 TxEngine，因而 UnpackageStream AXI-Stream Slave 总线的 S_AXIS_data 接口需要连接 TxEngine 的 M_OUT_DATA_AXIS 接口，UnpackageStream 的 s_aclk、s_aresetn 接口需要与 RxEngine 的 m_axis_out_aclk、m_axis_out_aresetn 接口连接同一时钟、复位信号。

3.6 Counter 的固件编写和连接

FirmDrive®包含通用数字测量组件，仅需调用、重组这些固件即可实现 Counter 功能，所用组件包括 Counter、PFI、RoutingMatrix。

- Counter：计数器功能主体，可以实现计数、多种参数测量、脉冲输出等功能。

- PFI：软件配置方向的外部 IO 信号路由，Counter 应用中用做双向外部信号路由，根据 Counter 状态配置输入输出模式。

- RoutingMatrix：FPGA 内的路由工具，在 Counter 应用中用于根据 Counter 工作需要路由被测信号、基准信号等信号。

每一个 Counter IP 实例可以支持 1 通道测量或 1 通道输出。实现 2 通道 Counter 需要在设计中加入 2 个 Counter 实例。Counter 的主要信号参见 Counter 章节。

- DB-102 配套的 FirmDrive®驱动为降低复杂程度，没有使用 AXI-Stream 高速传输 Counter 测量结果的方式，所以设计中不需要将 AXI-Stream Master 总线接入 RxEngine，可以空闲相关接口。
- Counter 的输出接口比较简单，所以将 Counter Cnt_Out 接口直接连接 PFI 实例的路由输入接口。
- Counter 的输入接口数量多，并且在不同的模式下需要实现不同的连接关系，因而使用 RoutingMatrix 实例管理信号路由。由于 RoutingMatrix 支持软件实时路由配置，因此使用上位机可以随时将 Counter 的信号连接分配为合理的状态。RoutingMatrix 的路由端点配置如下。
- AXI-Lite Slave 总线包含 S_AXI、axi_aclk、axi_aresetn 三组。S_AXI 接口需要连接至 AXI Interconnect 实例的 M**_AXI 接口，axis_aclk、axis_aresetn 接口需要与 AXI Interconnect 的 M**_ACLK、M**_ARESETN 使用同一时钟、复位信号。
- InternalClk 接口是 Counter 的主参考时钟接口，DB-102 中使用 FPGA 系统片内的主时钟。

4 FirmDrive®引擎及高层次接口

FirmDrive®架构除了提供 FirmDriveCore 固件和 C API 外，还为硬件设计师提供了 Vivado 环境下的一组专门用于测试测量更易于使用的通用固件 IP 接口和相对应的 C API。它们是：固件数据收发引擎（RxEngine、TxEngine）、可变功能接口固件 PFI、路由矩阵固件 IP、模拟触发固件 IP、模式触发固件 IP、和计数器固件 IP。这些固件 IP 都有相应的 C API。硬件设计师将所设计的硬件控制 IP 连接到 FirmDrive®这些通用 IP 模块，合并生成二进制文件后下载到 FPGA，然后就可以使用在 C 的环境里通过 C API 对所设计的硬件进行调试。

本节假设硬件设计师对 Xilinx 的 Vivado、XDMA、AXI4、AXIS 总线有基本的了解并会使用。为了便于理解，本章把于 FirmDrive®有关的 AXI 总线和 XDMA 做一个简要的叙述。

4.1 XDMA、AXI 总线等系统资源

FirmDrive®使用了一些 Xilinx 提供的资源：XDMA、AXI 总线等。AXI 总线是 FirmDrive®中的主要数据传递途径，所有的测量数据、控制数据在 FPGA 内均通过 AXI 总线实现传递。AXI 总线共有 AXI-Lite、AXI4、AXIS 三种类型，在 FirmDrive®内均有使用，如图 4-1 FirmDrive® FPGA 固件框架所示，在 Rx 引擎和 Tx 引擎的右侧，数据均通过 AXIS 进行传输，在 Rx 引擎和 Tx 引擎的左侧，均通过 AXI 总线与 Memory 进行数据交互。

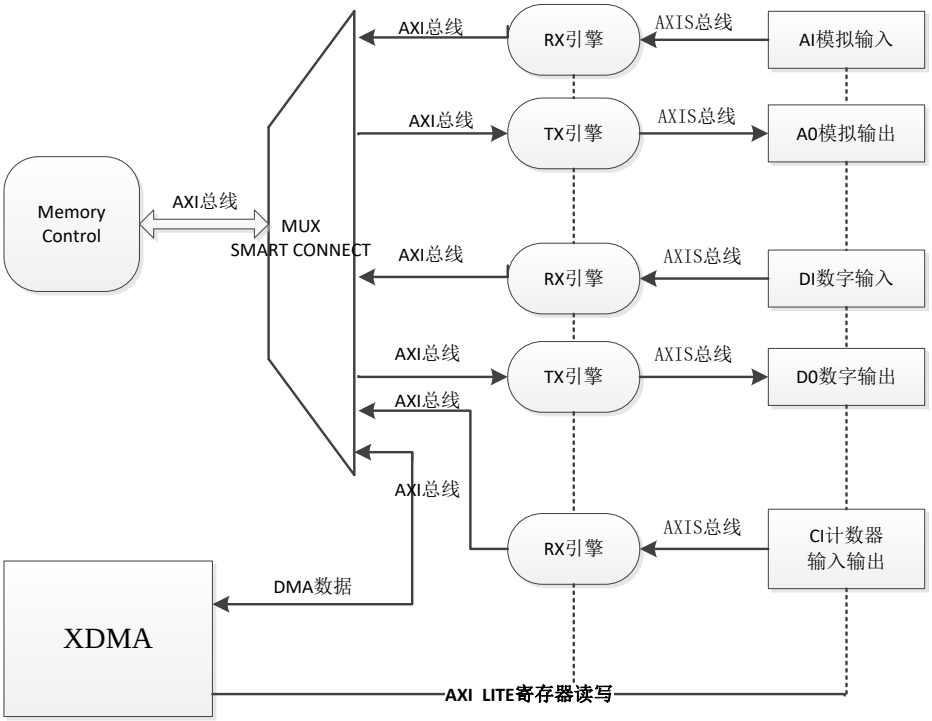


图 4-1 FirmDrive® FPGA 固件框架

4.1.1 AXI-Lite

AXI-Lite 是带有地址信息的低速 AXI 总线，在框架中用于传递寄存器配置信息。在 FirmDrive®中 XDMA 实例具有唯一的 AXI-Lite Master 接口，其他模块均为 AXI-Lite Slave 接口，通过 Interconnect 或者 Smart Connect 进行连接。

特别地，AXI-Lite 也用于传输低速的测量数据，如 PFI 模块被配置为软件控制时，AXI-Lite 总线用于传输静态的 DIO 数据。

用户在创建自己的 AXI-Lite slave 总线时，只需要按照简仪提供的模板添加用户寄存器即可，不需要关注总线传输的具体逻辑，同事注意 AXI-Lite slave 总线的时钟需要使用稳定可靠的时钟源，避免总线挂起导致计算机死机。

4.1.2 AXI4

AXI4 总线是带有地址信息的高速 AXI 数据总线，是 FirmDrive®中 XDMA 到收发引擎之间的主要测量数据传输途径，为了区别与 AXI-Lite 和 AXIS，通常又称为 AXI 总线。

XDMA 通过 AXI 总线，负责向 DDR 指定地址处读写数据，FirmDriveFramework 中的数据收发引擎 RxEngine 与 TxEngine 通过 AXI 总线接口读写 DDR 上指定地址处的数据。XDMA 与 DDR、RxEngine 和 TxEngine 与 DDR 之间均通过 Smart Connect 使用 AXI 总线进行连接，XDMA 与数据收发引擎 RxEngine 与 TxEngine 之间的数据交换是通过 DDR 存储间接完成的，这一过程总是需要地址信息，因而 AXI 总线是围绕 DDR 存储使用。

用户在使用 FirmDrive®框架时，只需要了解 AXI4 总线即可，不需要编写或者修改与 AXI4 相关的总线逻辑。

4.1.3 AXIS

AXIS(AXI-Stream)是没有地址信息的 AXI 总线，具有较高的速度和吞吐量。不携带地址信息，因而结构简单、使用方便，是功能 IP 传输测量数据的主要接口，比如 A/D 、D/A 模块均使用 AXIS 传输数据。

AXI-Stream 是实现收发引擎与功能模块数据传输的唯一形式，FirmDrive®中子板接口模块到收发引擎之间的测量数据主要使用 AXI-Stream 接口进行传输。因此用户需要清楚的知道 AXIS 总线数据的格式和规则要求。

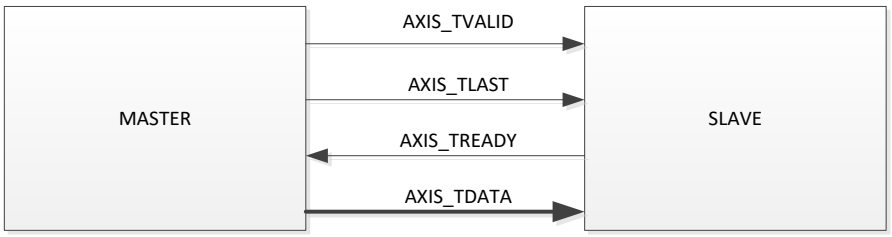


图 4-2 FirmDrive®**中标准** AXIS **总线**

如图 4-2 FirmDrive®中标准 AXIS 总线所示，在 FirmDrive®中，AXIS 总线一般包含 AXIS_TDATA、AXIS_TVALID 、AXIS_TREADY、AXIS_TLAST，其中 AXIS_TVALID 和 AXIS_TREADY 是 mater 与 slave 接口的握手信号，数据传输时 TREADY 和 TVALID 没有先后顺序的要求，当 TREADY 和 TVALID 同时为高电平时，当前 AXIS_TDATA 传输完成。AXIS_TLAST 是 AXIS 总线一个 packet 结束的标志位，在最后一个 TVALID 有效时拉高。

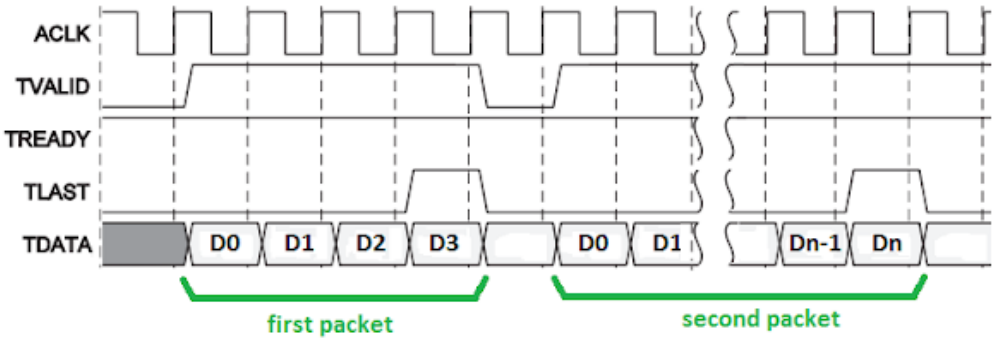


图 4-3 FirmDrive®中标准 AXIS 总线传输示例

如图 4-3 FirmDrive®中标准 AXIS 总线传输示例所示，slave 设备的 TREADY 一直为高，D0-D3 数据对应的 TVALID 均为高，则 D0-D3 都是有效数据，每 4 个 TVALID 对应 1 个 TLAST，表示该数据一个 packet 有 4 个有效数据。

4.1.4 XDMA

XDMA 全称 Xilinx DMA/Bridge Subsystem for PCI Express® (PCIe™)，是 FirmDrive®中提供上位机与下位机接口通讯、DMA 读写的 IP，支持 PCIe 1.0、2.0、3.0，具备可由用户选择的 AXI-Stream、AXI4 memory-mapped 接口和用于寄存器读写的 AXI-Lite 接口。

FirmDrive®框架中，要求 XDMA 选用 AXI4 memory-mapped 模式接口，如图 4-4 XDMA 接口(memory-mapped 模式)所示，memory-mapped 模式下，M_AXI 总线用于实现对 DDR 等缓存的读写操作。

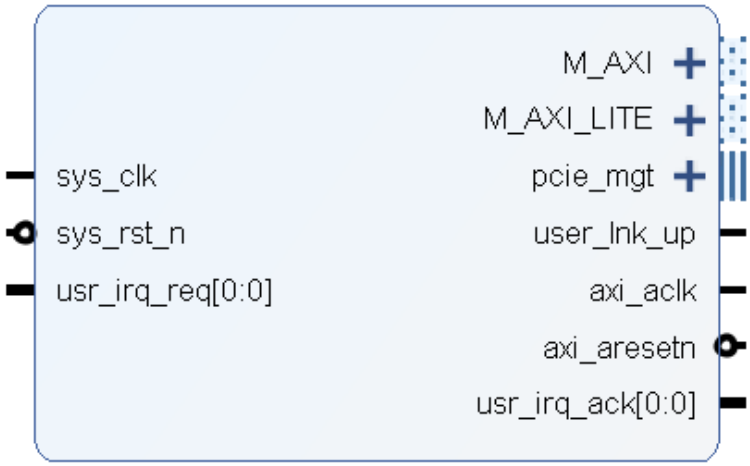


图 4-4 XDMA 接口(memory-mapped 模式)

- sys_clk: pcie 系统输入时钟，使用 Run Block Automation 功能可以自动生成该引脚的约束，用户不需要进行额外的连接或者约束；
- sys_rst_n: 与 sys_clk 同步的复位信号，使用 Run Block Automation 功能可以自动生成该引脚的约束；

- usr_irq_req[0:0]:用户中断输入信号，不连接；
- M_AXI: Master AXI 总线，用于连接板上存储器 DDR，通常通过 Smart Connect 进行桥接；
- M_AXI_Lite: Master AXI-Lite 总线，用于对 UserIP 进行寄存器读写；
- pcie_mgt: Pcie 规范的输入输出物理引脚，使用 Run Block Automation 的功能可以自动生成约束，与 pcie 规范相关，无需自己约束；
- user_ink_up: 上位机与 FPGA 的 pcie 连接成功标志，连接成功后输出高点平，可用于输出控制 LED 状态，表示 pcie 连接状态
- axi_aclk: 用户时钟，根据 XDMA 的不同速度等级配置而变化，用于 XDMA 的 M_AXI 和 M_AXI_Lite 以及其它输出信号的时钟；
- axi_aresetn:与 axi_aclk 同步的复位信号；
- usr_irq_ack[0:0]: 用户中断输出，不连接；

如图 4-5 XDMA 配置所示，固件开发人员在使用 XDMA 时，需要设定符合 FirmDrive®支持的 Vendor ID(VID)和 Device ID(PID)，用于上位机驱动的正确识别，详细的支持列表请参考最新驱动发布说明。

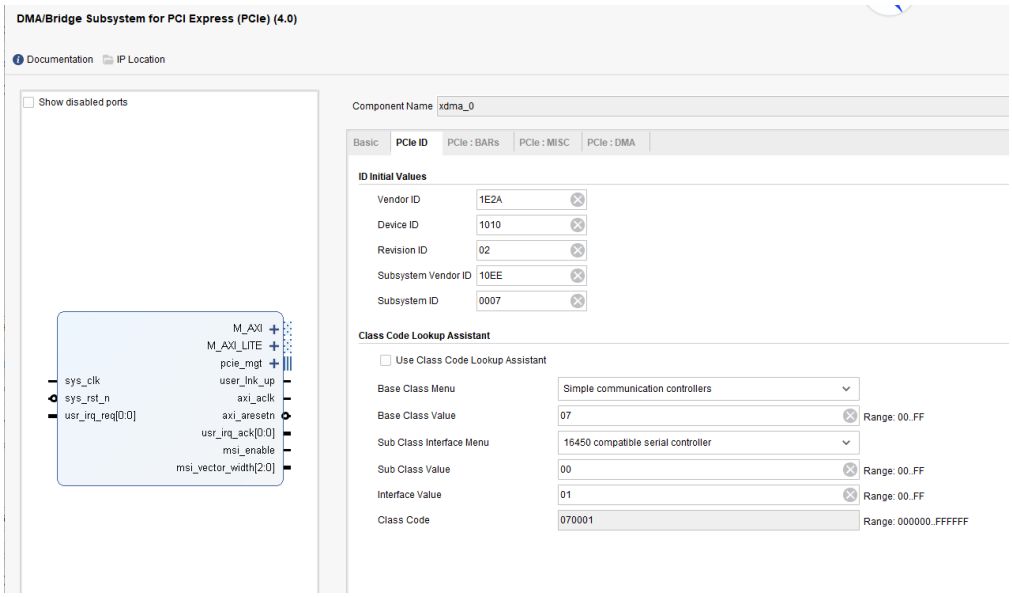


图 4-5 XDMA 配置

4.1.5 AXI DataMover

如图 4-1 FirmDrive® FPGA 固件框架所示，FirmDrive®中的 RxEngine 和 TxEngine 需要进行 AXIS 总线到 AXI 总线的转换，这里需要使用到 Xilinx 的 IP AXI DataMover 进行配合。下面对 DataMover 进行简单的说明，更多信息请参阅 IP 文档。

AXI DataMover 是 Xilinx 提供的一个 AXIS 和 AXI4 总线之间的互联 IP，用于 AXI memory-mapped 和 AXI-Stream 之间的高速数据传输。AXI DataMover 具备

MM2S(memory-mapped to Stream)、S2MM(Stream to memory-mapped)两个接口，如图 4-6 AXI DataMover 接口配置所示，可以根据需要进行配置。

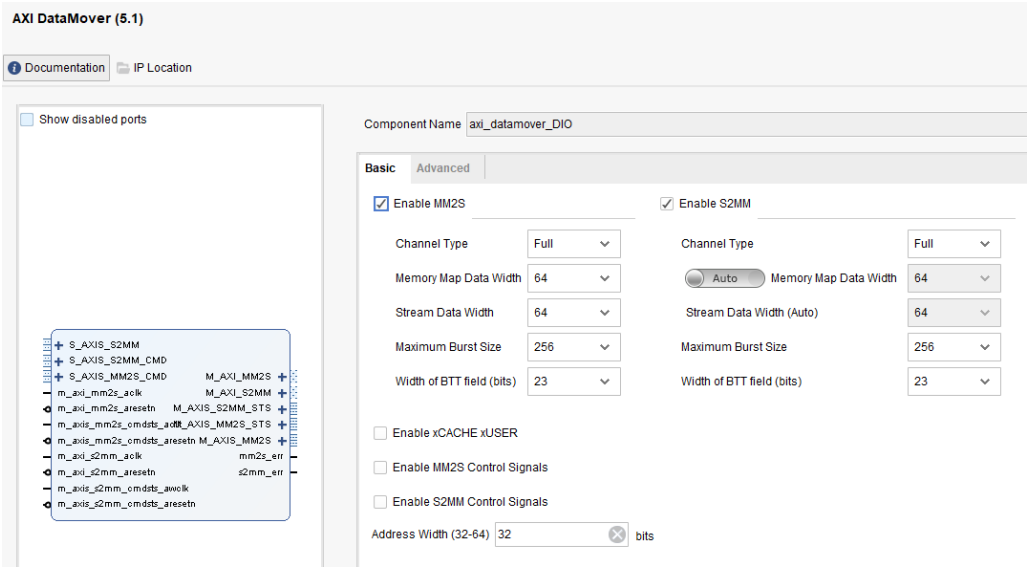


图 4-6 AXI DataMover 接口配置

如图 4-7 S2MM 接口所示，AXI DataMover 将不含地址信息的数据 S_AXIS_S2MM 转换成带地址信息的数据 M_AXI_S2MM。

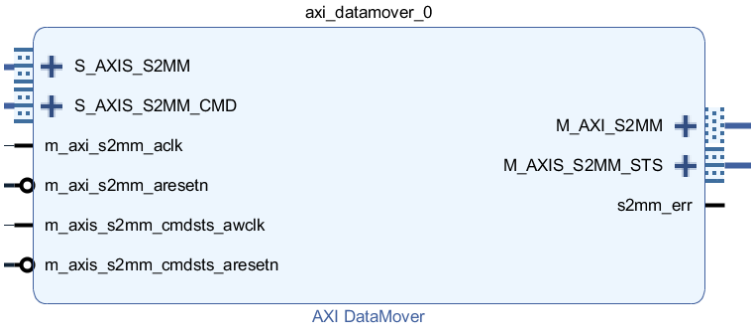


图 4-7 S2MM 接口

- S_AXIS_S2MM: 连接至 RxEngine 的 M_OUT_DATA_AXIS，位宽需要设置相同；
- S_AXIS_S2MM_CMD: 与 RxEnigne 的 M_OUT_DATA_AXIS 连接，用于 RxEnigne 控制 AXI DataMover；
- m_axi_s2mm_aclk: S2MM 接口的工作时钟，来自与 M_AXI_S2MM 连接的 memory 的工作时钟；
- m_axi_s2mm_aresetn: 与 m_axi_s2mm_aclk 同步的低电平有效复位信号，需要连接到 RxEngine 的 rst_s2mm；
- m_axis_s2mm_cmdsts_awclk: 输入 S_AXIS_S2MM_CMD 的工作时钟，一般连接到 m_axi_s2mm_aclk，选择同时钟源；

- m_axis_s2mm_cmdsts_aresetn: 与 m_axis_s2mm_cmdsts_awclk 同步的复位信号，需要连接到 RxEngine 的 rst_s2mm；
- M_AXI_S2MM: Master AXI 数据总线，通过 Smart Connect 连接至 DDR 控制器的 AXI 总线接口；
- M_AXIS_S2MM_STS: S2MM 的状态信息，连接至 RxEngine 的 S_AXIS_S2MM_STS 接口；
- s2mm_err: 不连接。

如图 4-8 MM2S 接口所示，DataMover 将含有地址信息的数据 M_AXI_MM2S 转成不含地址信息的数据 M_AXIS_MM2S。

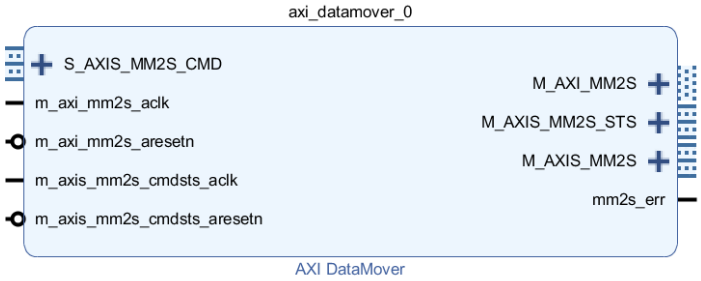


图 4-8 MM2S 接口

- S_AXIS_MM2S_CMD:连接至 TxEngine 的 M_AXIS_MM2S_CMD；
- m_axi_mm2s_aclk: MM2S 的接口工作时钟，来自与 M_AXI_MM2S 相连接的 memory 的时钟；
- m_axi_mm2s_aresetn: 低电平有效复位信号，连接至 TxEngine 的 reset_mm2s 接口；
- m_axis_mm2s_cmdsts_aclk: S_AXIS_MM2S_CMD 工作时钟，一般连接至 m_axi_mm2s_aclk；
- m_axis_mm2s_cmdsts_aresetn: 低电平有效复位信号，连接至 TxEngine 的 reset_mm2s 接口；
- M_AXI_MM2S: Master AXI 总线，通过 Smart Connect 连接至 memory 控制器的 AXI 总线接口；
- M_AXIS_MM2S_STS: 连接至 TxEngine 的 S_AXIS_MM2S_STS 接口；
- M_AXIS_MM2S: 连接至 TxEngine 的 S_AXIS_IN_DATA 接口，总线位宽设置需相同；
- mm2s_err: 不需连接。

如图 4-9 S2MM 和 MM2S 全部使能所示，既可以单独使能 S2MM 或 MM2S 接口，也可以同时使能两个接口，接口含义与连接方式同单独使用时相同。

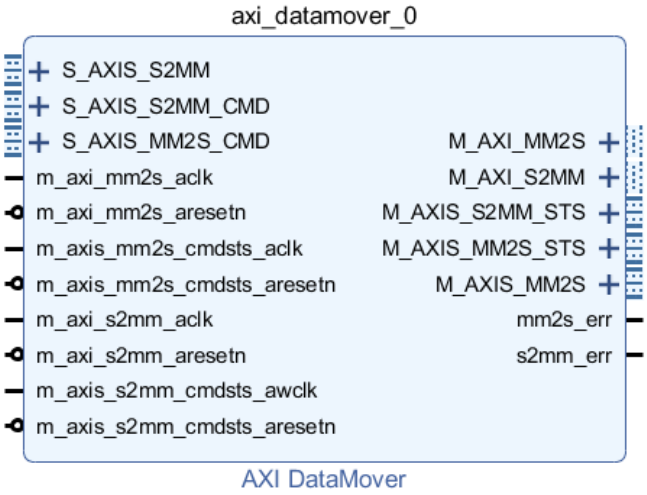


图 4-9 S2MM 和 MM2S 全部使能

如图 4-10 AXI DataMover 的高级配置所示，AXI DataMover 需要在高级配置里进行如图所示的勾选，在 S2MM 配置下，allow unaligned transfer、enable indeterminate BTT mode，在 MM2S 模式下，allow unaligned transfer，enable store forward。

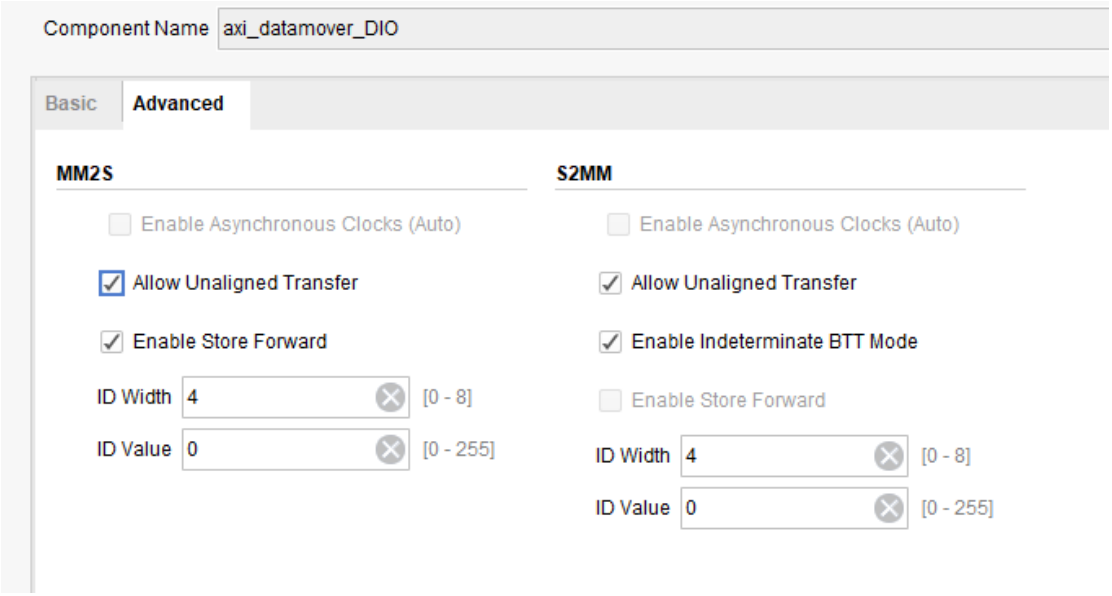


图 4-10 AXI DataMover 的高级配置

4.1.6 AXI Interconnect

AXI Interconnect IP 用于连接 m 个 AXI memory-mapped master 设备与 n 个 AXI memory-mapped slave 设备，从而进行 memory-mapped 数据传输，接口兼容 AXI-Lite，即 AXI Interconnet 既可连接数据亦可连接控制设备。可以在 IP 的配置选项中指定 master 接口和 slave 接口的数量，每个接口都对应了独立的时钟、复位接口。Master 和 slave 接口的数量没有限制，由用户指定。

如图 4-11 1-to-N AXI Interconnect 用例所示，AXI Interconnect 用于 1 个 memory-mapped master 设备连接 6 个 memory-mapped slave 设备，每个 master 和 slave 接口均

可连接独立的工作时钟。用户必须保证连接的时钟和总线内部的时钟信息一致，否则会产生编译错误。在 FirmDrive®框架中，AXI Interconnect 用于 XDMA 的 M_AXI_LITE 接口连接其它 IP 的 AXI-Lite slave 接口，进行寄存器的读写。

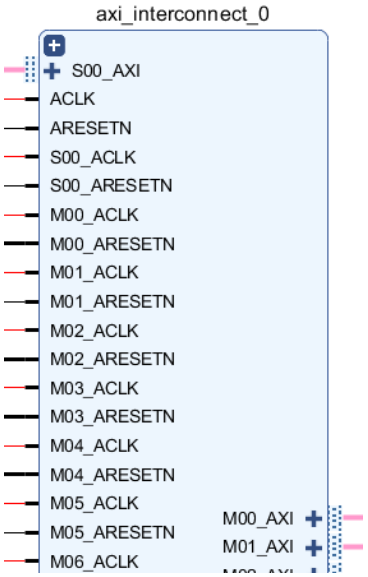


图 4-11 1-to-N AXI Interconnect 用例

- S00_AXI: memory-mapped slave 接口，连接到 XDMA 的 M_AXI_LITE 接口，位宽保持默认 32 位；
- ACLK: S00_AXI 的工作时钟，即 XDMA 的用户时钟 axi_aclk；
- ARESETN：与 ACLK 同步的低电平有效复位信号，即 XDMA 的用户复位信号 axi_aresetn；
- S00_ACLK: S00_AXI 的工作时钟，即 XDMA 的用户时钟 axi_aclk；
- S00_ARESETN：与 ACLK 同步的低电平有效复位信号，即 XDMA 的用户复位信号 axi_aresetn；
- M00_ACLK: M00_AXI 的工作时钟，即与该接口相连接的 IP AXI-Lite 接口工作时钟，通常为了保持总线时钟的稳定性，该时钟连接至 XDMA 的用户时钟 axi_aclk，此时要求用户 IP 如果具备多个时钟域，需要进行正确的跨时钟域转换；或者将用户 IP 的工作时钟连接到该引脚；
- M00_ARESETN：与 M00_ACLK 同步的低电平有效复位时钟；
- M00_AXI 到 M00_AXI5：连接到六个 memory-mapped 设备，与用户 IP 的 AXI-Lite Slave 接口连接。

4.1.7 AXI SmartConnect

AXI SmartConnect IP 用于连接 m 个 AXI memory-mapped master 设备与 n 个 AXI memory-mapped slave 设备，m 与 n 最大为 16，进行 memory-mapped 数据传输。接口兼容 AXI-Lite，即 AXI Interconnet 即可连接数据亦可连接控制设备。在 FirmDrive®框架中 AXI SmartConnect 用于 AXI DataMover 与 DDR 控制器 AXI 总线之间

的连接，进行 AXI memory-mapped 数据传输。每一个设备可以有自己的单独时钟，也可以和其它设备共享时钟。Slave 设备也可以和 Master 设备共享时钟。用户无需指定时钟和设备的配对，SmartConnect 根据总线自带的时钟信息完成配对。

如图 4-12 N-to 1 AXI SmartConnect 连接示例所示，有 6 个 memory-mapped slave 接口连接至 1 个 memory-mapped master 接口，并使用两个两个输入时钟和一个低电平有效复位信号。

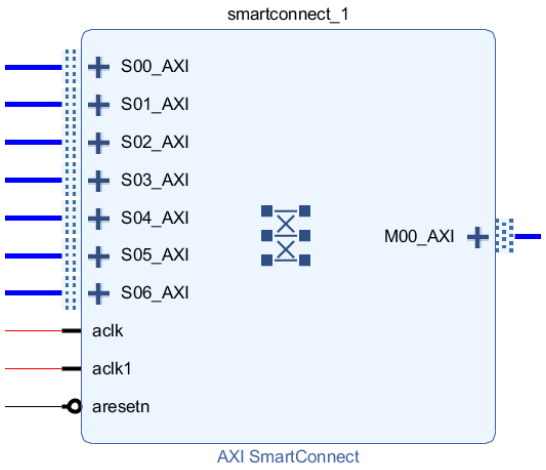


图 4-12 N-to 1 AXI SmartConnect 连接示例

- S00_AXI: memory-mapped master 接口，在 FirmDrive®框架中连接至 AXI DataMover 的 M_AXI_S2MM 或者 M_AXI_MM2S 接口；
- aclk: 输入时钟一，可以分配给任何一个或一组设备；
- aclk1: 输入时钟二，可以分配给任何一个或一组设备；；
- aresetn: 与 aclk 或者 aclk1 同步的低电平有效复位时钟；
- M00_AXI: memory-mapped Master 接口，用于连接至 DDR 的 AXI 总线。

AXI SmartConnect 与 AXI Interconnect 相比区别如下：

- AXI SmartConnect 最多支持 16 个 master 和 16 个 slave 接口。当设备数超出此限制时，就必须使用 AXI Interconnect。FirmDrive®框架推荐使用 AXI Interconnect 进行 XDMA 的 AXI-Lite 与其它模块的 AXI-Lite 之间的连接，是因为用户 AXI-Lite 接口容易遇到超出设备上限的场景；
- AXI SmartConnect 相同接口配置下占用资源更少；
- AXI SmartConnect 时钟域连接简单，只需要将 master 和 slave 的不同工作时钟连接至 IP 的始终接口即可，IP 内部会自动进行分配，复位信号只需要连接一个即可，IP 内部自动进行同步；

4.1.8 自定义模块的 AXI 接口

对于用户自定义的功能 IP，对外接口需要一个 Slave AXI-Lite 接口用于接收 XDMA 的配置信息，一个 AXIS 接口或者其它输出信号，用于输入输出数据。使用 AXIS 作为数据的输入输出是因为 AXIS 不需要地址信息，数据均为 Streaming 的方式流动，只需要 Master 和 Slave 做好配合即可，方便实现数据的高速传输。

如图 4-13 ADS7253 IP 接口示例所示，S00_AXI 是 AXI-Lite 总线，用于进行寄存器读写，s00_axi_aclk 为其工作时钟，M00_AXIS 是 AXIS 的数据传输接口，m00_axis_aclk 是其工作时钟，ADS7253itf 为该 IP 的其它对外接口。

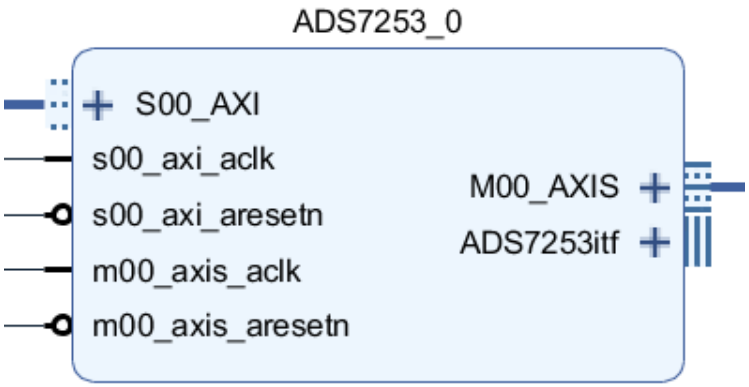


图 4-13 ADS7253 IP 接口示例

4.2 FirmDrive®中的地址配置

Block Design 的 Address Editor 可以让用户使用 GUI 的方式设置 AXI、AXI-Lite 总线所对应的地址。如图 4-14 Address Editor 地址分配示例所示，在 FirmDrive®固件架构中的地址分配主要分成三个部分：

- XDMA 的 AXI-Lite 总线，用于对 UserIP 的寄存器进行读写。
- XDMA 的 AXI 总线，通过 SmartConnect 连接至 DDR，即图中的 mig_7series_0。
- AXI DataMover 的 S2MM 和 MM2S AXI 总线，由于 XDMA 的 AXI 总线和 DataMover 的 AXI 总线均通过 AXI SmartConnect 连接到 mig_7series_0，因此对应的地址空间可以重叠。

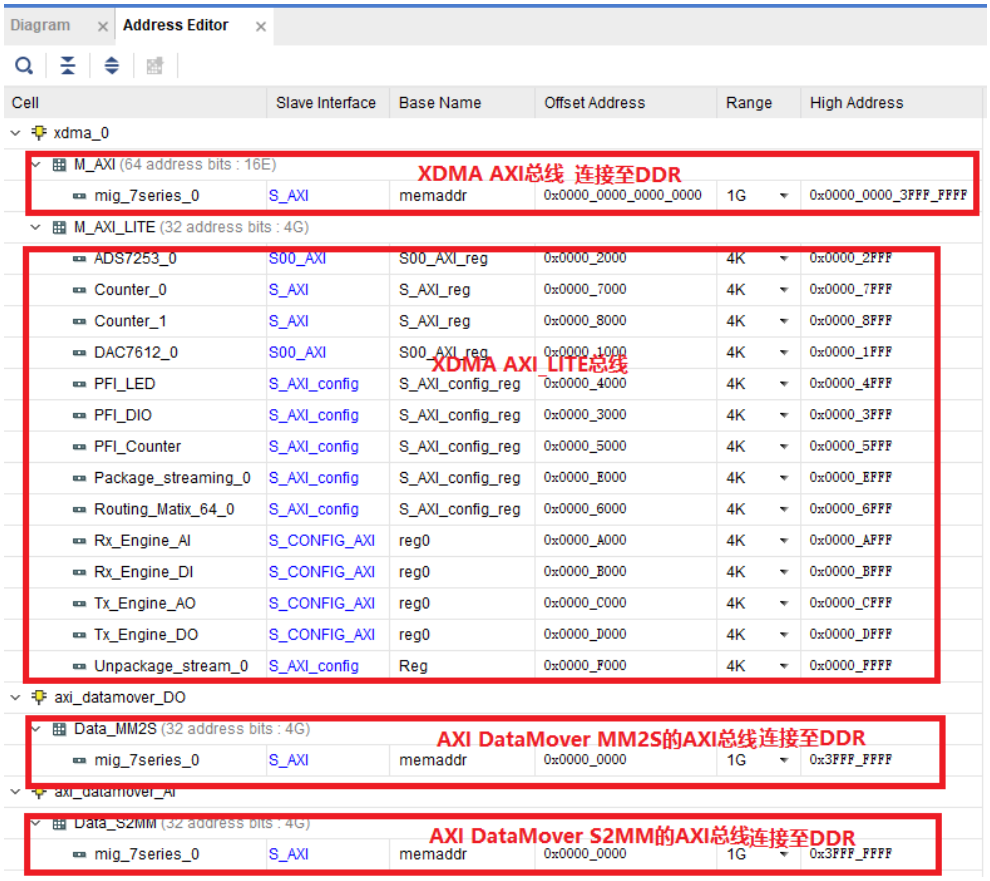


图 4-14 Address Editor 地址分配示例

4.2.1 FirmDrive®的 AXI-Lite 寄存器地址

寄存器地址即 AXI-Lite 接口寄存器的地址，这些地址是全局的，上位机通过唯一的地址信息访问 AXI-Lite 接口寄存器。

a)寄存器的位宽统一为 32bit；

b)寄存器地址的最小分配长度为 4k；

c)不同的 AXI-Lite 模块、接口需分配不同的寄存器地址，不能有重叠；

d)GA(PXI 规范中用于识别槽位号的物理引脚)引脚对应的 AXI GPIO 模块基地址需要设置为 0x0000_0000，用以驱动正确获取 PXI 板卡的卡槽号，因此基地址 0x0000_0000 为系统保留地址，不得分配给其它 IP；

e)其它 UserIP 的起始基地址为 0x0000_1000；

f)AXI-Lite 控制器级联时需要为每一个次级的模块单独分配地址，不论与 AXI-Lite 总线怎样连接，所有的模块都需分配地址，且均为全局属性。

4.2.2 FirmDrive®的 Memory 地址

如图 4-14 Address Editor 地址分配示例所示，XDMA、RX 引擎、TX 引擎连接的 DataMover 均通过 AXI 总线访问 DDR 内存。

a)不同模块、不同接口的内存地址可以相同；

b)不同的模块、接口间分配相同的内存地址才能访问相同的内存空间，以共享数据；

c)可以为所有模块分配全部内存地址，这是一种简单易行的办法；

4.3 FirmDrive®的时钟系统

4.3.1 XDMA 用户时钟

XDMA 的用户时钟 axi_aclk 频率会随着 XDMA 不同的配置而发生改变，当 XDMA 配置为 GEN2X8 时，XDMA 提供给用户的时钟为固定的 250Mhz，该时钟作为 XDMA AXI-Lite 和 M_AXI 接口的时钟；

4.3.2 DDR 时钟

DDR 时钟包括 MIG(VIVADO 中用于生成 DDR 控制器的插件)的系统输入时钟 sys_clk、DDR 的工作时钟、MIG 控制器输出的用户时钟 ui_clk。可以由客户根据需要的数据传输速率进行配置，但是不同的配置对设计时序有不同的要求。简仪科技建议使用 200MHz 的系统输入时钟， 800MHZ 的 DDR 工作时钟，200MHZ 的用户时钟，硬件支持的数据位宽为 8bit，DDR 双沿传输的极限吞吐率为 1.6GB/s。用户可以将 DDR 的工作时钟进行下调，从而在满足吞吐率要求的前提下，降低实现时序收敛的难度。

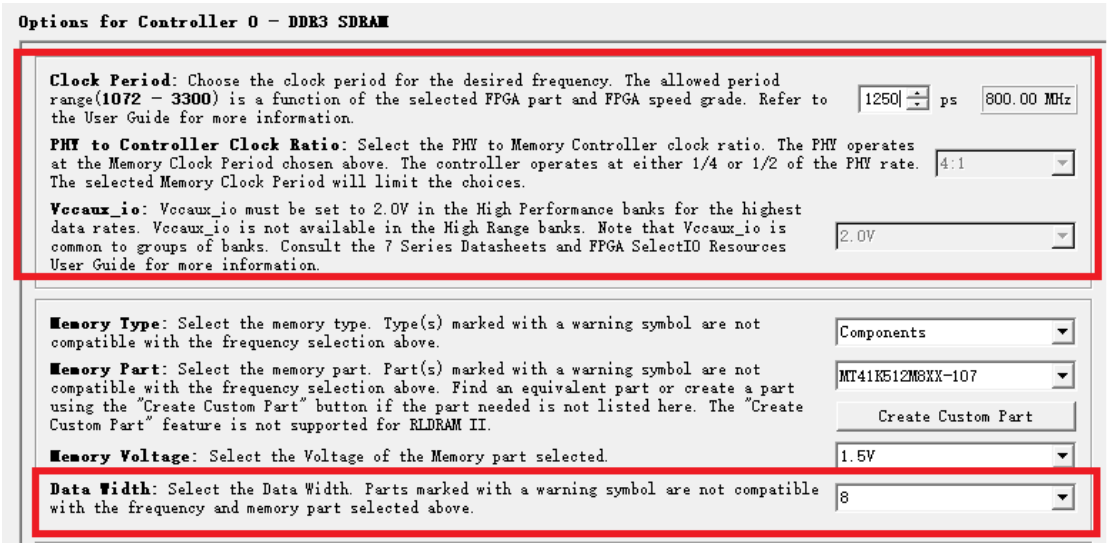


图 4-15 MIG 典型工作时钟配置

MIG 的输出用户时钟 ui_clk 是 AXI MM 总线的主要时钟，DDR 的输出输出数据均应工作在 ui_clk 时钟下。：

a)AXI Smart Connect 需要接入该时钟；

b)AXI DataMover 模块的全部时钟均使用 DDR 的 ui_clk 时钟；

4.3.3 功能模块的时钟

功能模块的时钟可以根据需要配置，其内部数字逻辑电路可以与 AXI 总线使用同一时钟，也可以使用独立的时钟。功能模块一般同时具备 AXI-Lite 和 AXIS 接口，两接口可以使用相同的时钟，也可以使用不同的时钟。

从任一连接外部设备的功能模块起，直至收发引擎，各 AXIS 线路必须使用同一时钟，避免过多的跨时钟域转换。

功能模块的 Slave AXI-Lite 接口时钟必须和配对连接的 Master AXI-Lite 接口使用同一时钟。

4.4 FirmDrive®固件的系统框图

如图 4-1 FirmDrive® FPGA 固件框架所示，FirmDrive®固件主要包括模拟输入(AI)、数字输入(DI)、模拟输出(AO)、数字输出(DO)、计数器输入(CI)、计数器输出(CO)6 部分(可根据需要进行裁剪)数据通路，收发引擎(RxEngine、 TxEngine)负责将从板上缓存(DDR)读出、写入指定地址的数据，使用 Smart Connect 作为 Mux(多路选择器)对 DDR 进行数据读写。XDMA 负责与上位机进行通信，通过 AXI 总线读写 Memory 的数据，通过 AXI-Lite 总线读写各个 IP 模块的寄存器。

4.4.1 AI 模拟输入

如图 4-16 模拟上行数据通路所示，模拟输入上行通路包括将模拟信号转化成数字信号的 A/D 模块、将数据转存到 DDR 的 RxEngine 模块，还有 DDR 模块和 XDMA 模块。另外若使用触发功能，需要 AnalogTrigger 模块(模拟触发)、PFI(数字触发)、RoutingMatrix 模块的配合，将触发信号连接到 RxEngine 的触发输入端口。

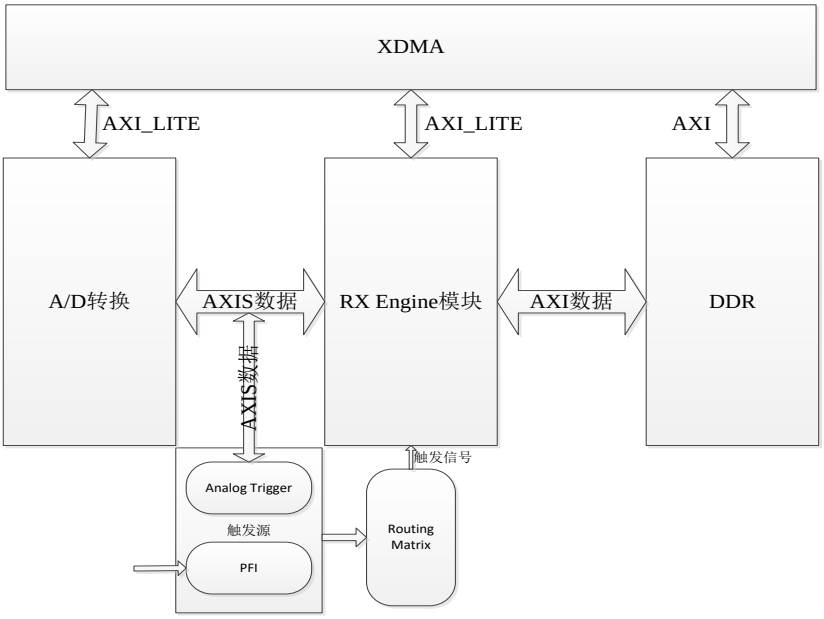


图 4-16 模拟上行数据通路

- AI 模块的功能是负责配置 A/D 芯片，同时将 A/D 芯片转换成的数字信号经过处理转换，通过 AXI Stream 的数据接口输出。不同的芯片对应的逻辑也不同，但是都应具备 AXI-Lite 的寄存器配置接口，AXI Stream 的数据接口或其他输出信号接口，AXI4 Lite 可以实现将上位机配置下来的静态信息通过一定的协议（比如 SPI）配置到 A/D 芯片的寄存器中。该模块将数据接口上的数据经过一定的格式转换，将每个通道的数据通过 AXI Stream 的数据格式传送到下一个模块。另外，建议 AI 模块自带校准功能，可以对输入信号的幅度和偏置进行微调。下图是 AD4249 A/D 芯片的 IP 模块对外的接口，如图 4-17 A/D IP 接口示例所示，S_AXI4_config 负责寄存器的读写、M_AXIS_ch0 是通道 0 的 stream 数据接口、M_AXIS_ch1 是通道 1 的 stream 数据接口，其余为时钟、A/D 芯片与 FPGA 的硬件物理接口。对于多通道的 A/D 模块，如果是扫描式工作，输出 M_AXIS 接口数据需要使用 tlast 信号作为当前使能通道最后一个通道数据的标记，用于 RxEnigne

对串行数据进行正确的筛选，具体参考 Rx_SetSerialModeChnCount()函数介绍。对于多通道并行工作的 A/D 通道，输出 M_AXIS 可以分别输出或者合并后输出，连接到 RxEngine 后，通过在上位机对 RxEngine 进行设置实现通道数据的筛选，具体参考 Rx_SetParallelModeByteMask()函数介绍。

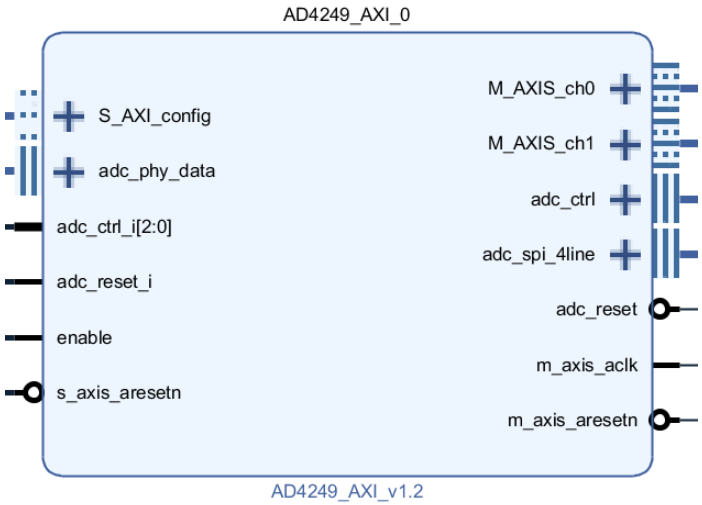


图 4-17 A/D IP 接口示例

- RxEngine 模块负责将 A/D 采集来的数据流（AXI Stream 格式）附加地址信息后转换成 AXI 总线的数据，存储到上位机指定的 DDR 地址空间内的一段环形缓冲区中。如图 4-28 RxEngine IP 接口所示，上位机通过 S_CONFIG_AXI(AXI-Lite 总线)负责将起始存储地址和环形缓冲区大小等参数设置到 RxEngine。RxEngine 与 AXI DataMover 配合，负责将 S_IN_AXIS 输入的数据转换成 AXI 总线的数据，附加上地址信息，按地址存储到 DDR 环形缓冲区中。上位机则通过 XDMA 从 DDR 指定环形缓冲区内将数据读走。

- AnalogTrigger 接收 A/D 输出的 AXI Stream 数据，进行上位机指定的比较后，输出高电平，通过 routing matrix 模块连接到其它 RxEngine 或者 TxEngine 进行触发。详细介绍参考 4.10。

- PFI 模块负责 FPGA 的数字 IO 端口连接与配置，比如某板卡预留了 32 路用户数字 IO 端口，那么就可以使用 PFI 对这些端口进行配置和连接。上位机通过 IP 的 AXI-Lite 总线配置 IO 端口的输入输出方向，查询输入状态。PFI 可配置为上位机或 FPGA 控制，上位机控制时可由上位机配置其输出电平，可实现数字滤波功能，当作为 Trigger 时，需将其配置为输入模式、FPGA 控制。

- RoutingMatrix 负责动态的连接不同的数字信号，功能类似于矩阵开关。模块最大支持 64 入 64 出，可以连通触发源与需要触发的输入端口，该模块同样支持 AXI Lite 总线，可以由上位机灵活配置，可任意接通输入输出两侧可能的触发源和所有被触发信号。

4.4.2 DI 数字输入

如图 4-18 DI 输入数据流向所示，数字输入上行通路包括 PFI、RoutingMatrix、PackageStream、RxEngine、PatternTrigger、DDR 模块和 XDMA 模块。PFI 负责接收外部

数字 IO 信号，转成电平信号送到 FPGA，RoutingMatrix 将 PFI 进来的电平信号重新排序组合后输出，PackageStream 负责将 routing 后的数字信号包装成 AXIStream 总线输出，可以上位机指定输入有效数据位宽后由 RxEngine 模块转存到 DDR 中。上位机通过 XDMA 读取 DDR 中的 DI 数据。其中 PatternTrigger 可实现 DI 的触发输出，当输入电平与上位机设定的电平符合时输入触发信号，触发信号连接到 routing matrix，再由 routing matrix 负责连接至 RxEngine。

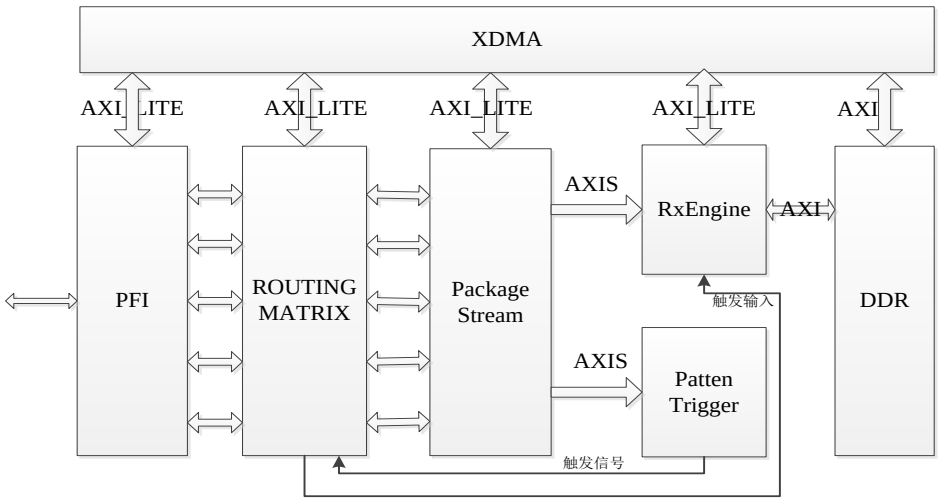


图 4-18 DI 输入数据流向

图 4-19 DI 示例连接时 DI 的部分连接图。

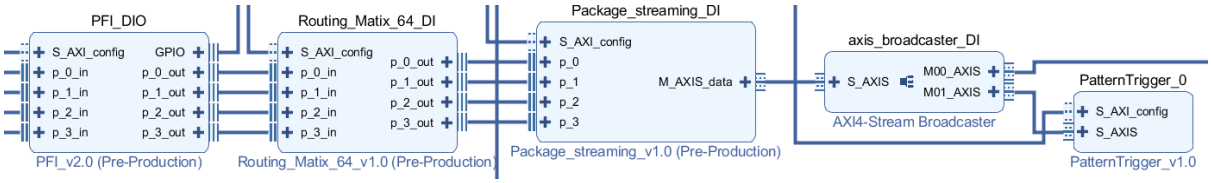


图 4-19 DI 示例连接

- PFI 模块是负责连接外部接口，可以配置其输入输出方向，可上位机查询输入状态，可配置为上位机控制或 FPGA 控制，上位机控制时可由上位机配置其输出电平，可实现数字滤波功能，当作为 DI 输入时，需将其配置为 FPGA 控制、输入模式，该模块支持数字滤波，可以根据用户需求配置是否支持数字滤波功能。该模块最大可支持 32 路数字输入。

- RoutingMatrix 负责将数字输入的信号进行整理排序，将 PFI 输入进来的若干路信号连接到 RoutingMatrix 的输入端，由上位机配置输入与输出的匹配关系，由此可以调整 DI 信号的顺序后输出。

- PackageStream 负责将整理顺序后的 DI 信号加上 valid 信号包装成 AXI Stream 信号输出，该模块支持 Driver 信号输入，可以实现在 Driver 信号的驱动下使能 DI 数据的有效信号，驱动信号支持在上升沿、下降沿、双沿 3 中模式下驱动一个有效 DI 数据，3 种模式可以由上位机配置。该模块最大支持将 32 路信号包装成 Stream 数据流。

- RxEngine 与 AI 中作用相同，负责将输入的 AXIS 数据进行筛选、拼接后转换成带有地址信息的 AXI 总线格式数据，写入到 DDR 的指定环形缓冲区，并记录必要的信息，用于上位机从对应的地址读出数据。

- PatternTrigger 用来分析输入 DI 的 AXI Stream 的数据流，输出 1bit 的指示信号。当 AXI Stream 数据流中出现与上位机设定的信号符合的数据时，输出高电平脉冲，可以连接到 Trigger 的 RoutingMatrix，作为触发源之一。

4.4.3 CI 计数器输入

如图 4-20 CI 数据流向所示, Counter 接收来自 Routing Matrix 的信号进行计数，将计数值以 AXIS 格式的数据发送至 RxEngine，然后由 RxEngine 对数据进行筛选拼接后，存储至 DDR 指定地址空间内.

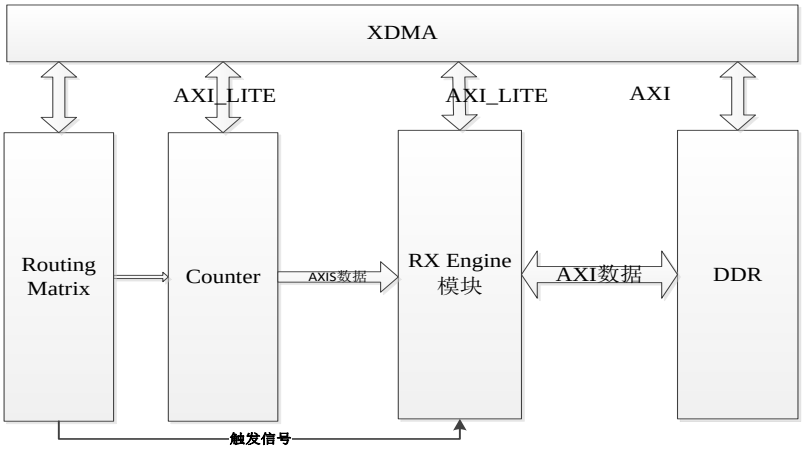


图 4-20 CI 数据流向

图 4-21 CI 部分连接示例.

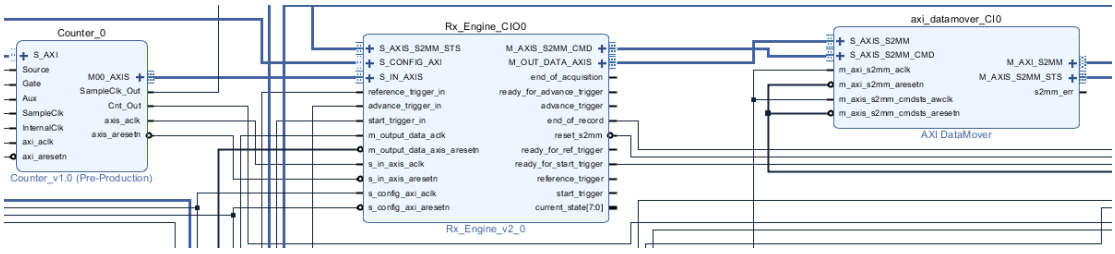


图 4-21 CI 部分连接示例

- Routing Matrix 负责连接 PFI 或者其它信号作为 Counter 输入，再由 Counter 进行计数.

- Couter 负责进行增减计数，实现通用数据采集的 counter 功能。Counter 可以将计数结果作为输入数据传送给用户、或者输出传输到硬件外设，工作时钟可以是内部时钟也可以是外部时钟，通过 C API 函数进行选择。

- RxEngine 作用通其它 AI DI 中描述.

- DDR 负责缓存数据,XDMA 负责上位机与 FPGA 固件之间的通讯.

4.4.4 AO 模拟输出

如图 4-22 AO 数据流向所示，模拟输出下行通路：AO、TxEngine、DDR、XDMA 即可构成最简单的模拟输出下行通路，若支持触发功能，还需增加 RoutingMatrix ，为 TxEngine 模块提供触发输入。

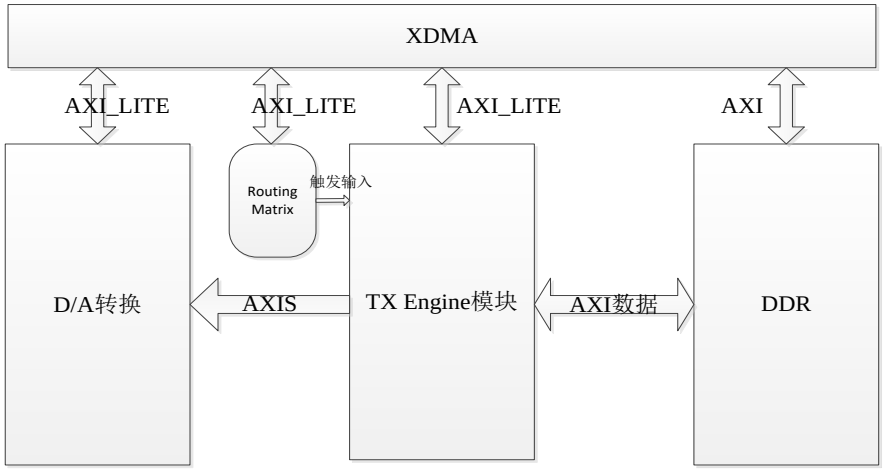


图 4-22 AO 数据流向

- AO 模块的功能是负责配置 D/A 芯片，不同的芯片需要不同的 IP 模块。AO 接收来自 TxEngine 的 AXI Stream 数据输入，并将数据转换成 D/A 芯片需要格式的数据，再转换成模拟信号发送出去。该模块支持 AXI Lite 配置接口，可以实现将上位机配置下来的静态信息通过一定的协议（比如 SPI）配置到 D/A 芯片的寄存器中。另外，建议 AO 模块自带校准功能，可以对输出信号的幅度和偏置进行微调。

- TxEngine 模块负责从 AXI 接口的 DDR 指定地址范围的缓冲区中读出数据，转换成 AXI Stream 格式的数据，发送到下一个模块。TxEngine 支持上位机配置数据环形缓冲区的起始地址和缓冲区大小，上位机通过 XDMA 的 AXI 总线将数据写入该缓冲区，再由 TxEngine 读出传送到 D/A 模块。TxEngine 支持外触发和软件触发 2 种触发方式，外触发支持上升沿、下降沿、高电平、低电平触发，支持有限点发送、连续发送和循环发送 3 种发送方式，TxEngine 的输出数据总线 AXIS 可以上位机指定输出数据的

- RoutingMatrix 与上行链路中作用相同，为 TxEngine 提供触发源。

4.4.5 DO 数字输出

如图 4-23 DO 数据流向所示，数字输出下行链路包括：XDMA、DDR、TxEngine、UnpackageStream、RoutingMatrix、PFI。上位机将需要发送的数字信号通过 XDMA 写入到 DDR 的对应内存块中，TxEngine 负责将指定地址的 DDR 中的将数据读出。根据上位机配置，将数据以 AXI Stream 总线格式输出到下一级。UnpackageStream 模块负责将 AXI Stream 的数据流拆分成若干个分立的信号线，连接到下游的 RoutingMatrix 上进行信号整合后通过 PFI 发送到外部数字输出引脚上，驱动数字 IO 端口输出对应的高低电平。

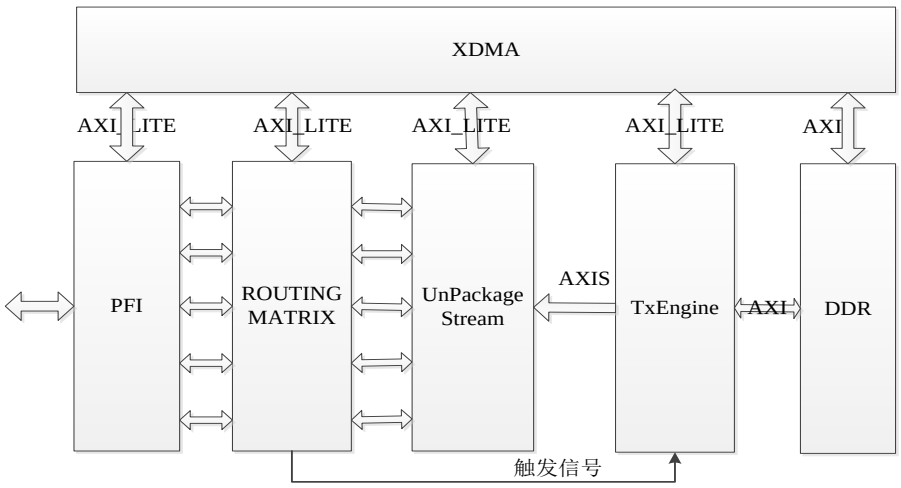


图 4-23 DO 数据流向

图 4-24 DO 连接示例，unpackage stream 接收来自 TxEngine 的数据，拆分后由 routing matrix 连接到 PFI 模块驱动 FPGA 相应的数字 IO 端口。

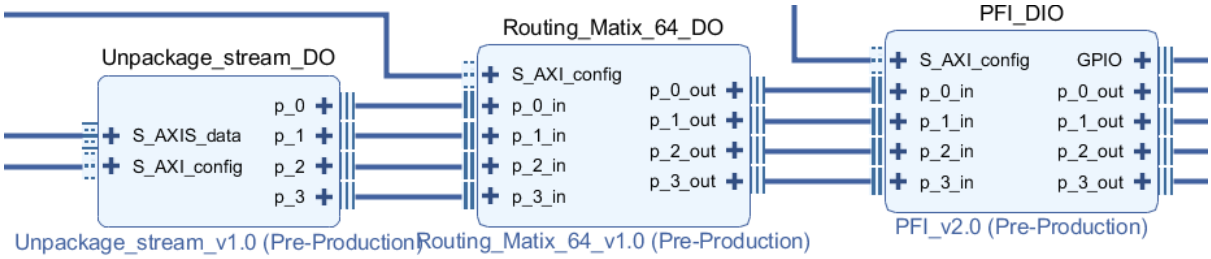


图 4-24 DO 连接示例

- TxEngine 模块作用同模拟下行部分介绍。
- UnpackageStream，输入接口是 AXI-Stream，数据位宽是 32 位，输出接口是若干散线，最大支持 32 路散线，当应用中的 AXI Stream 中的数据位宽比标准的 32 位宽时，比如 128 位，可以先经过一个 AXIS-DataWidth-Broadcast 将 AXI Stream 分割成 4 个 32 位的 Stream，然后实例化多个 UnpackageStream 将 Stream 差分成 128 个散线。
- RoutingMatrix 在数字输出下行通路中的作用与数字输入上行通路中的 RoutingMatrix 左右一致，均是对数字信号进行排序整合。
- PFI 负责将待发送的数字信号发送到对应物理引脚上。在下行通路中，PFI 需设置为 FPGA 控制，输出模式。

4.4.6 CO 计数器输出

如图 4-25 CO 数据流向所示, CO 较为简单，按照上位机配置, Counter 接收来自 Routing Matrix 的信号进行增减计数，计数值满足要求后输出高电平再通过 Routing Matrix 连接至 PFI 或者其它信号线.

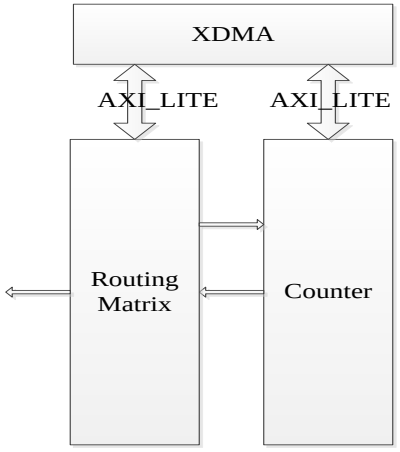


图 4-25 CO 数据流向

4.5 FirmDrive®的缓冲区

FirmDrive®的缓冲区是 FirmDrive®架构的重要部分。每一个 RxEngine、TxEngine 都有一个对应的缓冲区。缓冲区的大小是由用户在建立引擎时设置的。用户在使用缓冲区是必须注意以下几个方面。

PXIe-1010 最大 DDR 存储容量是 512M。所以最大缓冲区必须小于 512M。

由于可以有多个收引擎、发引擎，用户必须保证这些缓冲区在同一时刻不会相互干扰。

缓冲区中的数据是由上位机轮询后来调用的。通常每个缓冲区都有自己的轮询机制。上位机轮询的速率必须足够的快以免缓冲区溢出。通常的经验是轮询的间隔时间必须小于缓冲区代持时间的一半。简仪科技建议 30%。例如一个缓冲区可以缓冲 1s 的数据，则轮询的间隔时间可以设定在 300ms。无论如何，用户自己来保证有足够的余量来避免 DDR 缓冲区的溢出。

多个缓冲区的大小可以根据数据的速度来设置，数据慢的，缓冲区小；数据快的，缓冲区大一些。最佳情况是所有缓冲区都可以缓冲同样时间的数据。

4.6 RxEngine 固件数据采集引擎

RxEngine(Recieve Engine)是通用的数据采集 IP，负责数据采集的状态控制、输入数据有效位筛选、输入输出位宽转换、数据存储地址控制、multi record 处理。RxEngine 通过与 AXI DataMover(Xilinx 功能 IP，负责将 AXIS 总线转换至 AXI)配合将数据存储至 FPGA 板上指定的缓存位置。RxEngine IP 与其对应上位机 C 代码配合使用，可以实现通用的数据的采集流程控制和数据获取、采集状态检查等工作，是数据上行链路的核心。

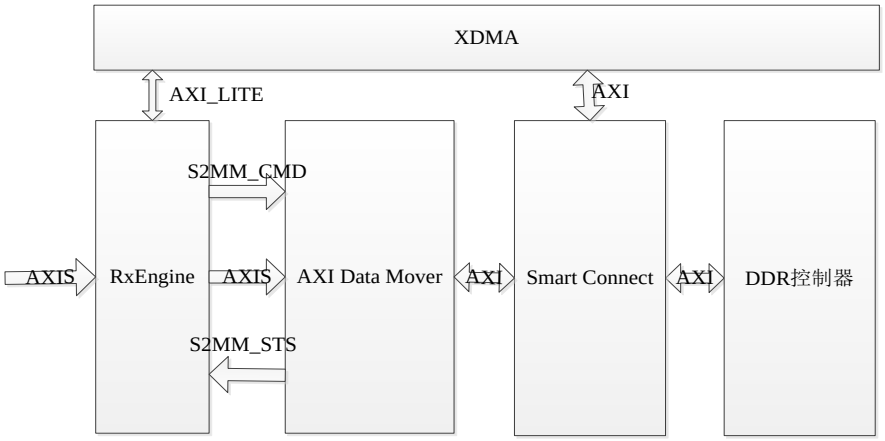


图 4-26 RxEngine 典型使用方式

如图 4-26 RxEngine 典型使用方式所示，Rx 接收来自数据源的 AXIS 总线数据，与 DataMover 的 S2MM 接口配合，将数据通过 Smart Connect 进行转接，传输到 DDR 控制器的 AXI 接口，写入 DDR 指定地址处。

Rx 引擎负责数据采集过程的控制，通用的数据采集有 8 个状态，如图 4-27 通用数据采集状态图所示：IDLE、 WAIT_FOR_START 、PRESAMPLE、 WAIT_FOR_REFTRIGGER 、POST_SAMPLE、RECORD_COMPLETE、WAIT_FOR_ADVANCE_TRIGGER、DONE.

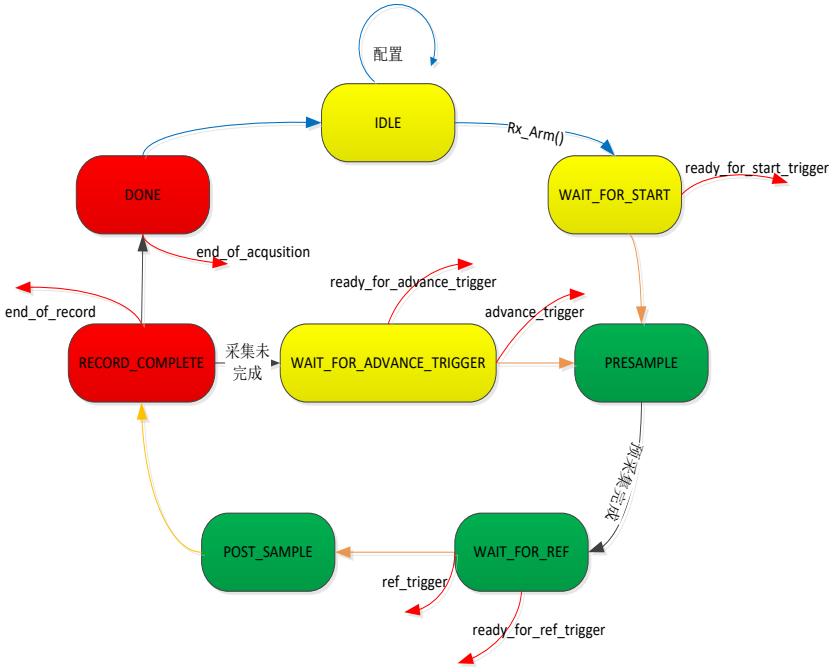


图 4-27 通用数据采集状态图

状态之间的转换条件可能来自三个源：状态本身、外部触发、上位机命令。

箭头颜色	状态跳转驱动源		
	软件	硬件	状态机内部

蓝色	YES		
黑色			YES
橙色	YES	YES	
绿色	YES		YES
红色	输出信号		

状态图颜色	含义
黄色	不进行数据采集
红色	停止采集
浅绿色	采集中

表 4-1 通用数据采集状态跳转与 event 输出条件

其中各个状态的定义如下：

- IDLE--当前状态不进行数据采集，所有的参数都可以在当前状态由上位机进行配置，只有当上位机调用了 Rx_Arm()函数，将配置信息提交后，状态跳转至 WAIT_FOR_START_TRIGGER，等待 Start_Trigger 到来进行预采集。
- WAIT_FOR_START--提交配置信息后，Rx 引擎进入当前状态. 当前状态下，ready_for_start_trigger 输出一直为高。如果 start_trigger_mode 设置为立即触发，则 Rx 引擎进入当前状态后，立即跳转到 PRSAMPLE 状态；如果触发模式设置为软件触发，则等待上位机发送软件触发命令后，跳转至 PRESAMPLE 状态；如果设置为硬件触发，则等待来自 start_trigger 的硬件触发(上升沿、下降沿、高电平、低电平等触发模式)，如表 4-1 通用数据采集状态跳转与 event 输出条件所示，触发到来后，采集跳转至 PRESAMPLE 状态。
- PRSAMPLE--当前状态根据 IDLE 状态配置好的参数、采集量进行数据的预采集，当指定数量数据采集完成后，跳转至 WAIT_FOR_REFTRIGGER 状态。
- WAIT_FOR_REF--Rx 引擎保持当前状态直到 reference_trigger 到来跳转到 POST_SAMPLE 状态，在当前状态下 ready_for_reftrigger 输出始终保持为高电平。当前触发模式同 start_trigger 相同，既可以配置成软件触发，也可以配置成

硬件触发，接收硬件接口 reference_trigger 的输入，检测到所需触发条件后，跳转到 POST_SAMPLE 状态进行下一阶段采集。

- POST_SAMPLE--如果在 IDLE 状态将 Rx 引擎配置为有限点采集，则当采集数据量满足设置要求后，Rx 引擎跳转至 RECORD_COMPLETE 状态；如果在 IDLE 状态将 Rx 引擎配置成无限点采集，则在 POST_SAMPLE 状态一直进行数据采集写入申请的环形缓冲区，直到上位机发送停止命令到 Rx 引擎，然后采集结束，状态跳转至 RECORD_COMPLETE。
- RECORD_COMPLETE--完成 ref trigger sample 采集后，状态跳转至该状态，同时输出 end of record 事件，判断是否有多余的 multi record 需要完成，如果没有，整个采集完成，跳转到 DONE，如果有，跳转到 WAIT_FOR_ADVANCE_TRIGGER 状态。
- WAIT_FOR_ADVANCE_TRIGGER--等待 advance trigge，同时向外输出 ready for advance trigger 事件,接收到 advance trigger 或 software advance trigger 等后，跳转至 PRESAMPLE 状态，同时输出 advance trigger 事件。
- DONE--生成 end_of_acquisition 事件向外输出，进入当前状态后，等待上位机程序完成数据的获取，然后由软件发出复位信号，跳转至 IDLE 状态。

4.6.1 RxEngine 固件 IP 接口

如图 4-28 RxEngine IP 接口所示：

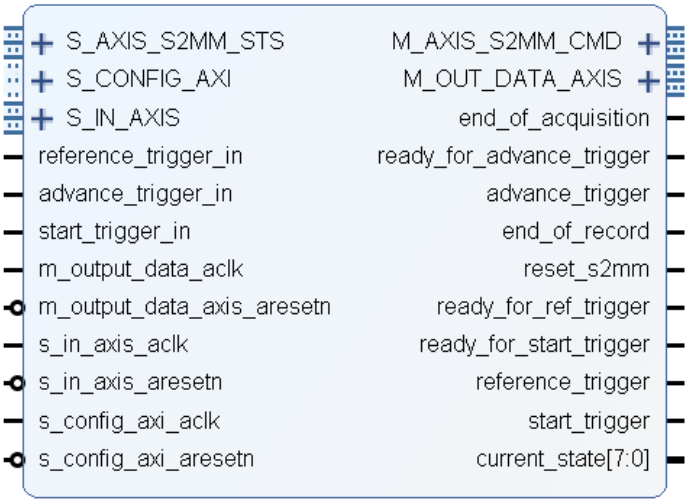


图 4-28 RxEngine IP 接口

- S_AXIS_S2MM_STS: AXI DataMover S2MM 接口的 AXIS 总线状态信息，如图 4-10 AXI DataMover 的高级配置所示，与 RxEngine 配合使用时，AXI DataMover 的 S2MM 接口需要选中 enable indeterminate BTT；
- S_CONFIG_AXI: Slave AXI 数据总线，用于接收 XDMA 的配置信息，进行寄存器读写；

- S_IN_AXIS: Slave AXIS 数据输入，来自 A/D 等需要存储至 DDR 的数据流。输入数据位宽可以在实例化 IP 时进行配置，选择范围 8/16/32/64/128 位, RxEngine 工作在串行数据输入模式下时，S_IN_AXIS 必须提供正确的 tlast 信号用以区分一个完整的 sample 的结束，即最后一个 tvalid 需要对应一个 tlast 信号；RxEngine 工作在并行数据输入模式下时每个 tvalid 都对应一个 tlast;
- reference_trigger_in: reference trigger 触发输入，RxEngine 所有触发信号的 li 触发模式可选软件触发、立即触发、上升沿触发、下降沿触发、高电平触发和低电平触发，促使采集从 PRESAMPLE 跳转到 POSTSAMPLE 状态，来自 RoutingMatrix;
- advance_trigger_in: advance_trigger 输入，此时状态机从 WAIT_FOR_ADVANCE_TRIGGER 跳转至 PRESAMPLE 状态；
- start_trigger_in: start trigger 触发输入，使得采集从 WAIT_FOR_START_TRIGGER 跳转至 PRESAMPLE 状态；
- m_output_data_aclk: M_OUT_DATA_AXIS 输出数据工作时钟；
- m_out_data_aresetn: 与 m_output_data_aclk 同步的复位时钟；
- s_in_axis_aclk: S_IN_AXIS 输入数据工作时钟，也是 RxEngine 内部逻辑主时钟；
- s_in_axis_aresetn: 与 s_in_axis_aclk 同步的复位时钟；
- s_config_axi_aclk: S_CONFIG_AXI 的工作时钟；
- s_config_axi_aresetn: 与 s_config_axi_aclk 同步的复位时钟；
- M_AXIS_S2MM_CMD: 向 DataMover 的 S2MM 发送控制命令，用户只需要将该接口连接至 DataMover 的 S_AXIS_S2MM_CMD 即可；
- M_OUT_DATA_AXIS: RxEngine 输出 AXIS 数据总线，位宽由用户进行指定，不小于输入数据的位宽；
- end_of_acquisition: 采集完成 event，采集状态跳转至 DONE 时向外发出;
- ready_for_advance_trigger: 当前采集状态处于 WAIT_FOR_ADVANCE_TRIGGER 时，ready_for_advance_trigger 保持高电平；
- advance_trigger: 用于触发用于 WAIT_FOR_ADVANCE_TRIGGER 到 PRESAMPLE 状态；
- end_of_record: 单次 record 结束 event，当采集状态跳转至 RECORD_COPLETE 时，end_of_record 向外发送高电平有效脉冲；
- reset_s2mm: 低电平有效的 S2MM 复位信号，连接至 AXI DataMover S2MM 接口工作时钟域的复位端口；

- ready_for_ref_trigger: 采集状态在 WAIT_FOR_REFTRIGGER 时，ready_for_ref_trigger 维持高电平；
- ready_for_start_trigger: 采集状态在 WAIT_FOR_START_TRIGGER 状态时等待，该信号保持高电平；
- refrence_trigger: refrence_trigger 输出信号，RxEngine 跳转至 POST_SAMPLE 时向外输出若干周期高电平；
- start_trigger: 当 RxEngine 跳转至 PRESAMPLE 时向输出外输出若干周期高电平；
- current_state: 向外输出 RxEngine 当前状态；

Rx_Engine IP 总共有三个时钟域，s_in_axis_aclk 对应输入数据的时钟域，m_output_data_aclk 对应输出数据的时钟域，同时 m_output_data_aclk 也是 DataMover 的工作时钟域，s_config_axi_aclk 是寄存器配置信息的时钟域。

如图 4-29 Rx Engine 可配置参数所示，在 IP 配置项中，可以配置 RxEngine 的工作模式，选择 AXIMM 或者 AXIS 模式，用于配合 XDMA 的 AXIMM 和 AXS 模式。输入 AXIS 的数据位宽根据数据源的数据位宽进行选择，可以由上位机指定输入数据的有效位，实现输入数据的筛选，比如输入数据物理位宽为 32 位,可以上位机配置只选中低 16 位有效。RxEngine 输出 AXIS 数据的位宽可以根据 DDR 控制器的输入数据位宽进行设置，比如 DDR 控制器的 AXI 数据位宽设置为 64，则 RxEngine 的 Output Data Width 也可以配置为 64。RxEngine 可以根据数据源的数据模式配置为并行或者串行模式，比如对于扫描式多通道 A/D，RxEngine 需要工作在串行模式，每个通道数据依次通过 AXIS 总线输出，因此需要使用 AXIS_TLAST 对数通道进行标记，RxEngine 才能知道哪一个是第一个通道的数据，从而正确的划分出不同通道的数据。另外对于并行工作的多通道 A/D，每个通道的数据位宽是 16bit，那么可以将数据拼接成 32 位 AXIS 数据送至 RxEngine，RxEngine 通过上位机指定的输入数据有效位，实现通道数据的动态选取。RxEngine 支持 multi record 即重触发，在使用 IP 时需要配置 multi record 的最大 buffer 个数，单位为 K 个，需要实现高频率有效触发时需要将该参数配置到合理值。

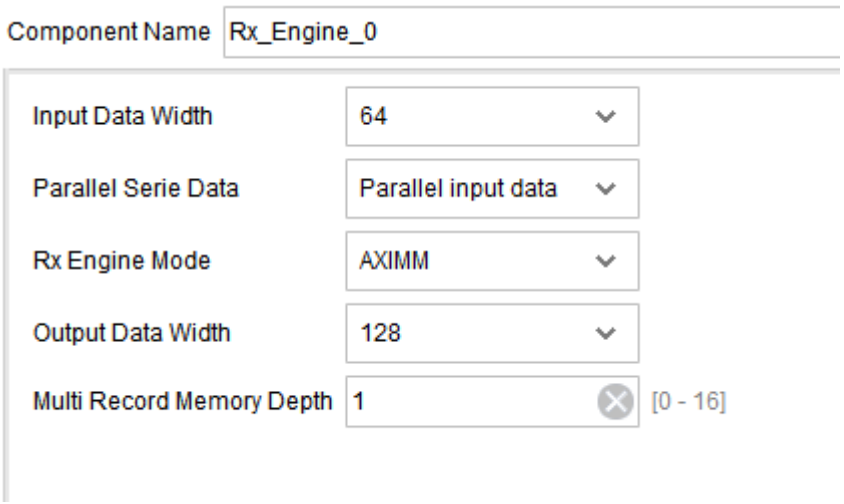


图 4-29 Rx Engine **可配置参数**

因为 RxEngine 需要与 DataMover 的 S2MM 接口配合使用，这里对 DataMover 的接口和配置进行说明。如图 4-30 DataMover 接口及配置选项所示，与 RxEngine 配合使用，需要使能 S2MM 接口，同时选择使能 Indeterminate 传输，为了使得传输更加灵活，同时启用 unaligned transfer，支持地址不对齐的数据传输。

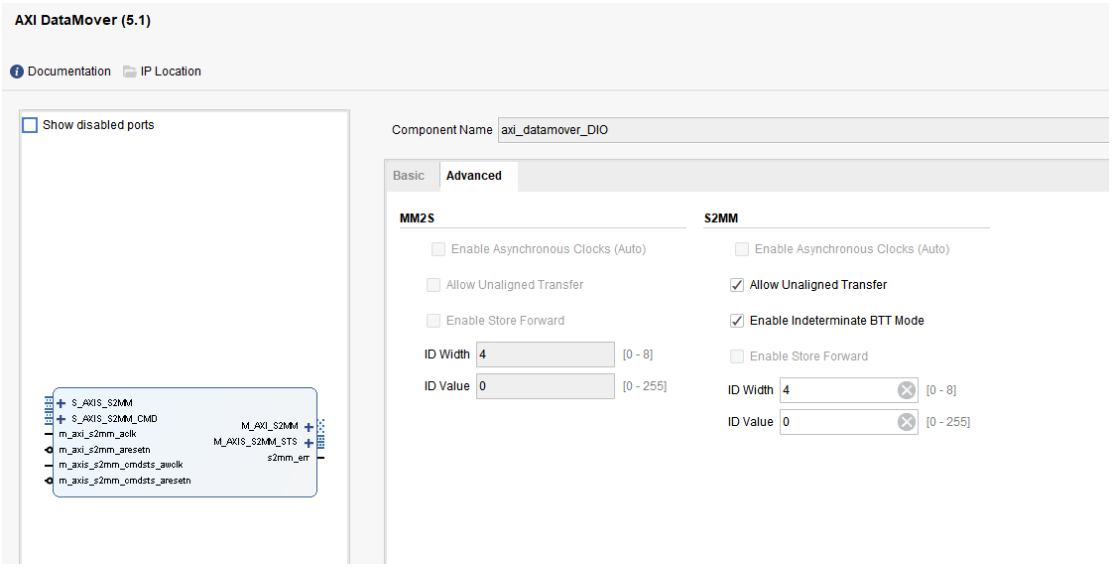


图 4-30 DataMover 接口及配置选项

如所示，数据源数据通过 S_IN_AXIS 接口进入 RxEngine，再由 RxEngine 进行数据的筛选、拼接后输出到 AXI DataMover，RxEngine 控制 DataMover 将 AXIS 接口数据转换成 AXI 数据，附加上地址信息，传递到带 AXI 总线的存储模块。

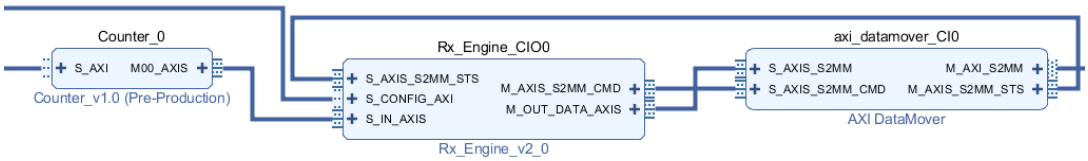


图 4-31 RxEngine 典型连接(未包含输入 trigger 和输出 event)

如图 4-32 RxEngine trigger 输入输出、event 输出与 Routing Matrix 典型连接图所示，RxEngine 的 trigger 输入信号来自 Routing Matrix，用户可以将任意的 Routing Matrix 输入信号作为 RxEngine trigger 的输入。同时 RxEngine 的 trigger 输出和 event 输出也可以连接到 Routing Matrix 的输入端，作为其它模块的指示信号或者触发信号。

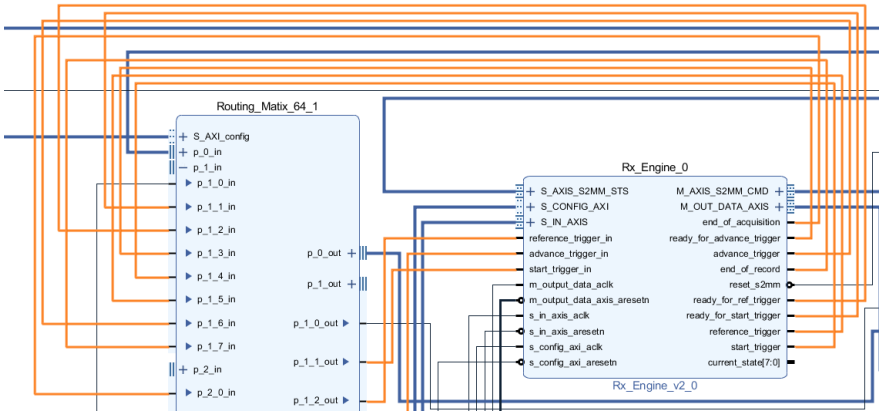


图 4-32 RxEngine trigger 输入输出、event 输出与 Routing Matrix 典型连接图

4.6.2 RxEngine C API

RxEngine IP 有对应的 C API 函数进行交互，用于配置数据采集，读取采集数据。C API 常用的工作模式可以分为：单次有限点采集、单次连续采集、multi record 采集，通过调用不同函数和调用流程，可以实现对应的采集过程。

C API 函数如下：

4.6.2.1 Rx_Init()

```
/// ********************************************************************************
/// <summary>
/// 初始化RX ENGINE结构体，配置RX IP对应的基地址
/// </summary>
/// <param name="hDev">设备结构体</param>
/// <param name="RxEngineBaseAddress">RX模块基地址</param>
/// <returns>成功：RX传输结构体指针；失败：NULL</returns>
/// <created>yasing,2018/1/29</created>
/// <changed>yasing,2018/1/29</changed>
/// ********************************************************************************
EXTERN_C HRX_ENGINE Rx_Init(FirmDriveDevice hDev, UINT32 RxEngineBaseAddress);
```

4.6.2.2 Rx_Close()

```
/// ********************************************************************************
/// <summary>
/// 释放RX ENGINE等资源
/// </summary>
/// <param name="hDev">Rx_Engine handle</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>yasing,2018/1/30</created>
/// <changed>yasing,2018/1/30</changed>
/// ********************************************************************************
EXTERN_C int Rx_Close(HRX_ENGINE hEngine);
```

4.6.2.3 Rx_EnableHWTrigger()

```
/// ********************************************************************************
/// <summary>
/// 触发可以选用软件触发和硬件触发，该函数使能硬件触发,单次采集需要设置start trigger(用于跳转至预采集点数)和ref trigger,
/// 使用multi record功能时有三个外部触发，start trigger、ref trigger和 advance trigger
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <param name="TrigerPort">使能硬件触发：NONE_TRIGGER_PORT:禁用所有硬件触发，
/// PRE_TRIGGER_PORT- start trigger enable，REF_TRIGGER_PORT-reftrigger,ALL_TRIGGER_PORT-enable all the HW trigger</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>yasing,2018/1/10</created>
/// <changed>yasing,2018/1/18</changed>
/// ********************************************************************************
EXTERN_C int Rx_EnableHWTrigger(HRX_ENGINE hEngine, TRIGGER_PORT TrigerPort);
```

其中trigger port为枚举类型，定义为：

```
typedef enum

{

    NONE_TRIGGER_PORT = 0,

    PRE_TRIGGER_PORT = 1 << 0,

    REF_TRIGGER_PORT = 1 << 1,

    ADVANCE_TRIGGER_PORT = 1 << 2,

    ALL_TRIGGER_PORT = 1 << 3,

}TRIGGER_PORT;
```

4.6.2.4 Rx_ConfigTrigger()

```
/// ********************************************************************************
/// <summary>
/// 配置触发模式
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <param name="TriggerMap">PRE_TRIGGER预采集触发；POST_TRIGGER触发；ADVANCE_TRIGGER步进触发，用于multi record</param>
/// <param name="TrigerMode">rising edge上升沿，falling下降沿，high高电平，low低电平，immediate立刻触发</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>Yasing,2017/12/25</created>
/// <changed>yasing,2018/1/12</changed>
/// ********************************************************************************
EXTERN_C int Rx_ConfigTrigger(HRX_ENGINE hEngine, TRIGGERMAP TriggerMap, TRIGER_MODE TrigerMode);
```

其中TRIGGERMAP枚举定位为：

```
typedef enum
{
    PRE_TRIGGER,// presample trigger
    REF_TRIGGER,// refsample trigger
    ADVANCE_TRIGGER,// advace trigger
}TRIGGERMAP;
```

TRIGGERMODE 枚举定义为：

```
typedef enum
{
    NONE_TRIGGER = (0x1 << 0),//software trigger
    RISING_EDGE_TRIGGER = (0x1 << 1),//rising edge trigger
    FALLING_DEGE_TRIGGER = (0x1 << 2),//falling edge trigger
    HIGH_LEVEL_TRIGGER = (0x1 << 3),//high level trigger
    LOW_LEVEL_TRIGGER = (0x1 << 4),//low level trigger
```

```
    IMMEDIATE_TRIGGER = (0x1 << 5)//immediate trigger

}TRIGER_MODE;//trigger mode
```

4.6.2.5 Rx_SetAutoTrigger()

```
/// ********************************************************************************
/// <summary>
/// 当处于WAIT_FOR_REF_TRIGGER状态时，等待若干个clock后，如果既没有软件触发也没有硬件触发到来，则自动产生一个ref trigger，采集进入POST_SAMPLE状态
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <param name="tickCount">产生Aoto trigger需要等待的周期数</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>yasing,2018/6/11</created>
/// <changed>yasing,2018/7/2</changed>
/// ********************************************************************************
EXTERN_C int Rx_SetAutoTrigger(HRX_ENGINE hEngine, UINT32 tickCount);
```

4.6.2.6 Rx_SetHoldOffTime()

```
/// ********************************************************************************
/// <summary>
/// 用于multi record应用场景下，两个record之间 ref trigger间隔的最小时间长度，在HOLD_OFF这段时间内，不允许接收新的ref trigger信号
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <param name="tickCount">hold off time user defined tick count, user shuold know the clock frequency</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>yasing,2018/6/12</created>
/// <changed>yasing,2018/6/24</changed>
/// ********************************************************************************
EXTERN_C int Rx_SetHoldOffTime(HRX_ENGINE hEngine, UINT32 tickCount);
```

图 4-33 hold off time 所示为有预采集点数的 MultiRecord 应用，以收到一次有效的 Reference Trigger 后，在预设的 Hold Off Time 时间段内，不接受触发信号。

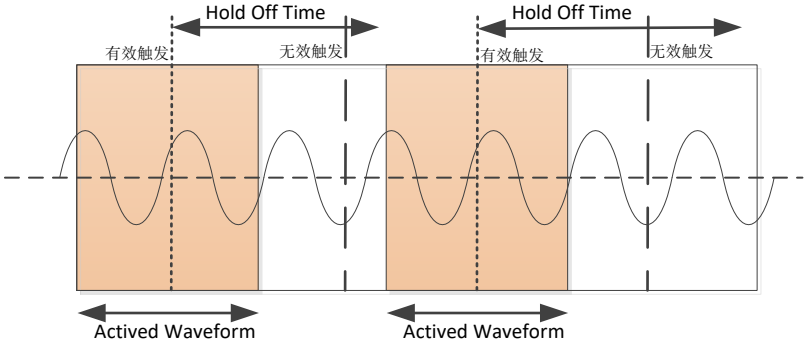


图 4-33 hold off time 示意图

4.6.2.7 Rx_SetSerialModeChnCount()

```
/// ************************************************************************************
/// <summary>
/// 对于串行输入数据，需要配置通道数，从而决定一个sample所占的字节数，此时默认输入的数据bytemask为全部有效，比如输入位宽32，则32位数据全部有效；
/// 如果有特殊要求，可以在该函数调用结束后，继续调用Rx_SetParallelModeByteMask()进行特殊配置
/// 如果是并行输入，可以不配置也可以配置通道数为1
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <param name="channelNum">串行输入数据通道数量</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>yasing,2018/7/2</created>
/// <changed>yasing,2018/7/2</changed>
/// ************************************************************************************
EXTERN_C int Rx_SetSerialModeChnCount(HRX_ENGINE hEngine, UINT32 numOfChannel);
```

Rx 引擎可以工作在串行或者并行数据输入模式，该函数用于配置串行工作模式下的通道数量，即多少个有效数据后有一个 tlast 信号。如图 4-34 串行输入所示，通道数量为 3，每 3 个 tvalid 信号对应一个 tlast，表征一次完整数据的结束：

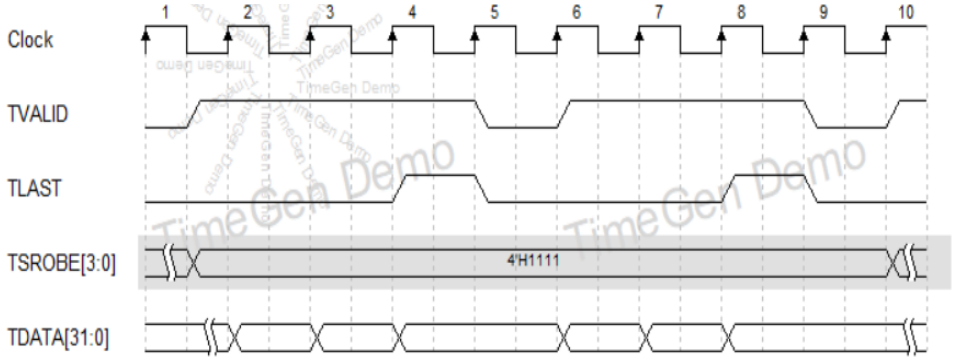


图 4-34 串行输入

图 4-35 4 通道串行输入数据所示，每 4 个 tvalid 的最后一个 tvalid 对应一个 tlast：

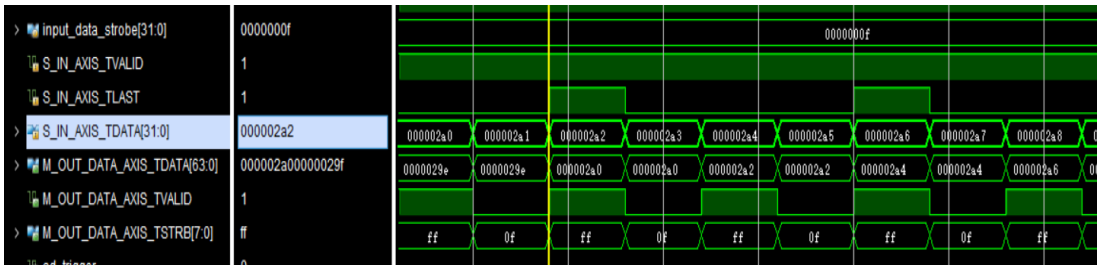


图 4-35 4 通道串行输入数据

4.6.2.8 Rx_SetParallelModeByteMask()

```
/// ********************************************************************************
/// <summary>
/// 配置并行数据输入时的bytes位选择,表征输入数据的哪一个byte有效，类似于strobe信号的作用，不支持奇数字节选取
/// 比如输入数据in_data位宽64位，如果strobe=0x3,表示输入数据低2个byte有效,in_data[15:0]有效;
/// 如果strobe=0xf,表示低4个byte有效, in_data[31:0]有效; 如果strobe=0x30,表示第3、4byte有效，in_data[47:32]有效
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <param name="strobe">输入数据有效位选择，对应的byte有效</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>Yasing,2017/12/25</created>
/// <changed>yasing,2018/7/9</changed>
/// ********************************************************************************
EXTERN_C int Rx_SetParallelModeByteMask(HRX_ENGINE hEngine, UINT32 strobe);
```

配置并行输入情况下 Rx 引擎输入数据的数据有效位，比如：输入数据 32 位，4 个 byte，strobe 为 32'b0011 表示低 16 位有效，则输出只对低 16 位进行拼接，如图 4-36 Rx 引擎输出数据与输入数据之间的关系所示

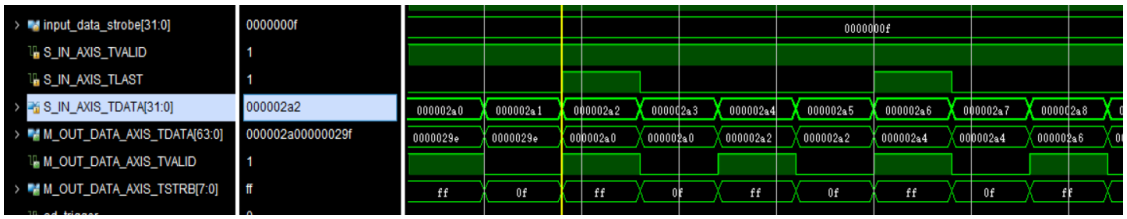


图 4-36 Rx 引擎输出数据与输入数据之间的关系

4.6.2.9 Rx_ConfigBuffer()

```
/// ************************************************************************************
/// <summary>
/// 配置FPGA板上环形缓冲区大小、起始偏移量，RxEngine采集来的数据传输至这里指定的环形缓冲区内
/// 对于multiricord 应用，有两个环形缓冲区概念，单个record缓冲区是一个，整个N个缓冲区又是一个环形缓冲区,
/// 当不启用multirecord时， multiRecordBufNum 个数为1，启用时，multiRecordBufNum的数量目前规定最多不能超过用户编写固件时设定的最大值
/// 需要指出的是multirecord的个数并不受限于buf的个数，即numOfBuf=512，record个数可以是100000，只要上位机及时将数据读走避免溢出即可
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <param name="startAddr">FPGA缓存起始地址，设置时需要根据FPGA设计进行，比如当DDR起始地址位0x40000,大小512M，StartAddr应大于等于0x40000,上限为512M</param>
/// <param name="bufSize">FPGA 板上用于单个record的memory size,单位是bytes，设置的cmd_single_transfer_bytes必须是该值的整数倍</param>
/// <param name="numOfBuf">N个record缓冲区组成的record缓冲区大小，应该是singleBufSize的整数倍</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>Yasing,2017/12/25</created>
/// <changed>yasing,2018/7/9</changed>
/// ************************************************************************************
EXTERN_C int Rx_ConfigBuffer(HRX_ENGINE hEngine, UINT32 startAddr, UINT32 bufSize, UINT32 numOfBuf);
```

4.6.2.10 Rx_ConfigAquisition()

```
/// ********************************************************************************
/// <summary>
/// 配置采集点数、预触发采集点数，配置Record的个数。
/// 这里使用sample的概念，一个sample作为一个完整的数据包
/// 比如 串行数据输入 输入位宽32位（strobe=4'b1111,串行输入时认为数据全部有效）输入通道为3，那么1个sample是4*3 bytes
/// 并行输入数据时 输入位宽32位 strobe表示哪一位byte有效，比如strobe=4'b0101，那么1个sample是2字节
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <param name="PreTriggerSamples">预采集点数</param>
/// <param name="RefTriggerSamples">设置采集点数，RefTriggerSamples=-1为连续采集，正整数为有限点采集</param>
/// <param name="RecordNum">默认值应该为1，表示单次采集</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>Yasing,2017/12/25</created>
/// <changed>yasing,2018/6/11</changed>
/// ********************************************************************************
EXTERN_C int Rx_ConfigAquisition(HRX_ENGINE hEngine, UINT32 PreTriggerSamples,
    UINT64 RefTriggerSamples, UINT64 RecordNum);
```

4.6.2.11 Rx_Arm()

```
/// ********************************************************************************
/// <summary>
/// commit配置信息到FPGA，RxEngine从IDLE状态跳转至WAIT_FOR_START_TRIGGER状态
/// Arm前先进行配置信息合理性的检查，不合理的需要给出错误
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>yasing,2018/1/5</created>
/// <changed>yasing,2018/1/12</changed>
/// ********************************************************************************
EXTERN_C int Rx_Arm(HRX_ENGINE hEngine);
```

4.6.2.12 Rx_Send_SW_Trigger()

```
/// ********************************************************************************
/// <summary>
/// 发送软件触发，RxEngine完成相应的状态跳转
/// </summary>
/// <param name="hDev">Rx_Engine handle</param>
/// <param name="TrigerMap">PRE_TRIGGER预采集触发；POST_TRIGGER触发</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>Yasing,2017/12/25</created>
/// <changed>yasing,2018/1/8</changed>
/// ********************************************************************************
EXTERN_C int Rx_Send_SW_Trigger(HRX_ENGINE hEngine, TRIGGERMAP TrigerMap);
```

对应的trigger map：

```
typedef enum

{

    PRE_TRIGGER,// presample trigger

    REF_TRIGGER,// refsample trigger

    ADVANCE_TRIGGER,// advace trigger

}TRIGGERMAP;
```

4.6.2.13 Rx_CheckStatus()

```
/// ********************************************************************************
/// <summary>
/// 检查Rx Engine当前状态，获取缓冲区剩余可读数据，上位机已完成读取数据，是否发生溢出，用于单个record的检查
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <param name="pnSamplsRemaining">当前剩余可读数据量,单位sample，一个sammple对应的bytes数量为hEngine->ElementSize</param>
/// <param name="pnSamplsTransfered">从获取ref trigger到现在已经得到的数据量,包括预采集点数</param>
/// <param name="pfOverflow">是否溢出，当写数据指针追上读指针时认为数据溢出</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>Yasing,2018/1/4</created>
/// <changed>yasing,2018/1/18</changed>
/// ********************************************************************************
EXTERN_C  int  Rx_CheckStatus(HRX_ENGINE  hEngine,  UINT64  *pnSamplsRemaining,  UINT64 *pnSamplsTransfered, BOOL *pfOverflow);
```

4.6.2.14 Rx_Read()

```
/// ********************************************************************************
/// <summary>
/// 读取Rx Engine指定环形缓冲区内的数据，上位机进行读指针维护，支持任意长度的数据读取
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <param name="ID">选取DMA通道</param>
/// <param name="pUserBuf">用户缓存</param>
/// <param name="nBytesToRead">读取的数据长度，sample,sample对应的bytes为
hEngine->ElementSize</param>
/// <param name="nTimeout">超时时间单位ms</param>
/// <param name="pnActualBytesRead">实际读取数据长度,sample</param>
/// <param name="pnDataRemaining">剩余需要读取的数据量,sample</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Yasing,2018/1/4</created>
/// <changed>yasing,2018/1/12</changed>
/// ********************************************************************************
EXTERN_C int Rx_Read(HRX_ENGINE hEngine, UINT32 ID, PVOID pUserBuf, UINT32
nSamplsToRead, UINT32 nTimeout,UINT32 *pnActualSamplsRead, UINT32 *pnSamplesRemaining);
```

4.6.2.15 Rx_MultiRecord_Check_Status()

```
/// ********************************************************************************
/// <summary>
/// 当使用multi records模式时，读取已经完成的record的memory data，查询已经完成了多少个
record，是否读的跟不上写的速度导致了溢出
/// 这里是将N个Record的的memory当作一个大的circular buffer进行操作,从而可以指定任意数量
的record number
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <param name="pnRecordsRemaining">已经采集完成，没有进行读取的record数量，PC计算得到
</param>
/// <param name="pnRecordsCompleted">已经读取完成的的record数量，PC负责更新</param>
/// <param name="pfOverflow">circular buffer是否溢出</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>yasing,2018/6/11</created>
/// <changed>yasing,2018/6/11</changed>
/// ********************************************************************************
EXTERN_C int Rx_MultiRecord_Check_Status(HRX_ENGINE hEngine, UINT64 *pnRecordsRemaining,
UINT64 *pnRecordsCompleted, BOOL *pfOverflow);
```

4.6.2.16 Rx_MultiRecord_Read()

```
// ********************************************************************************
/// <summary>
/// 每个record都按照一个整体进行读取,单次读取的samaple数量是Rx_ConfigAquisition()指定的单次record采集的sample总数
/// 如果一个sample对应2个字节，设置单次record采集0x1000个sample，那么读取一个record的数据就是读取0x1000*2个字节
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <param name="ID">选取DMA通道</param>
/// <param name="pUserBuf">用户缓存区</param>
/// <param name="numRecords">一次读取几个record</param>
/// <param name="nTimeout">超时时间，ms</param>
/// <param name="pnActualRecordsRead">实际读取完成的record数量,个</param>
/// <param name="pnRecordsRemaing">剩余未完成读取的record数量，个</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>yasing,2018/7/2</created>
/// <changed>yasing,2018/8/13</changed>
// ********************************************************************************
EXTERN_C int Rx_MultiRecord_Read(HRX_ENGINE hEngine, UINT32 ID, PVOID pUserBuf, UINT32 numRecords, UINT32 nTimeout,UINT32 *pnActualRecordsRead, UINT64 *pnRecordsRemaing);
```

4.6.2.17 Rx_Stop()

```
/// ********************************************************************************
/// <summary>
/// 关闭RX/RX engine，结束采集，状态机恢复IDLE
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <returns>成功：0；  失败：错误码</returns>
/// <created>Yasing,2018/1/4</created>
/// <changed>yasing,2018/1/12</changed>
/// ********************************************************************************
EXTERN_C int Rx_Stop(HRX_ENGINE hEngine);
```

4.6.2.18 Rx_Clear()

```
/// ********************************************************************************
/// <summary>
/// 复位Rx_Engine、DataMover S2MM模块,清空所有寄存器信息，释放内存
/// </summary>
/// <param name="hEngine">Rx_Engine handle</param>
/// <returns></returns>
/// <created>yasing,2018/1/10</created>
/// <changed>yasing,2018/1/10</changed>
/// ********************************************************************************
EXTERN_C int Rx_Clear(HRX_ENGINE hEngine);
```

4.6.2.19 Rx_GetErrorCodeInformation()

```
/// ********************************************************************************
/// <summary>
/// 根据error code获取最近错误文本信息
/// </summary>
/// <param name="errorCode"></param>
/// <param name="information"></param>
/// <returns></returns>
/// <created>yasing,2018/7/17</created>
/// <changed>yasing,2018/7/17</changed>
/// ********************************************************************************
EXTERN_C int Rx_GetErrorCodeInformation(int errorCode, char * information);
```

4.6.3 RxEngine API 使用方法

在介绍 API 使用之前，先对使用过程中容易混淆的地方进行说明：

sample: 1 个 sample 指代一个完整的数据包，不一定是 1 个 byte，需要根据 Rx_SetSerialModeChnCount()的 numOfChannel 和 Rx_SetParallelModeByteMask()的二进制数 strobe 中 1 的个数 m 确定，sample 对应字节数 s=numOfChannel*m 分为两种情况：

1. 当 RxEngine 工作在串行输入数据模式下，numOfChannel 表示串行输入的通道，S_IN_AXIS 接口数据 numOfChannel 个 tvalid 对应 1 个 tlast，如图 4-34 串行输入所示，numOfChannel 为 3。此时 S_IN_AXIS 接口数据应该全部有效，无需进行筛选，strobe 需要根据 S_IN_AXIS 接口数据位宽确定，比如输入数据位宽是 32，那么 strobe 应该为 32`b1111，一个 sample 对应的字节数 s=3*4=12(bytes)；

2. 当 RxEngine 工作在并行输入数据模式下，无需调用 Rx_SetSerialModeChnCount()函数，numOfChannel 默认为 1，通过 strobe 来对输入数据进行筛选，比如输入数据位宽是 32 位，strobe=32`b0011，表示输入数据的低 16 位有效，高 16 位无效，只存储有效数据，sample 代表的字节数 s=1*2(bytes)；

Buffer: 凡是带有 buffer 关键字的变量，单位都是 bytes，需要注意 buffer 和 sample 之间的关系，避免内存出错，比如 Rx_Read 中 pUserBuf 的长度(单位 byte)必须大于等于 nSamples2Read 对应的字节数。

根据采集点数的多少采集过程分为有限点采集和连续采集，根据采集模式又可以分为 multi record 采集和非 multi record 采集，下面分别介绍使用方法：

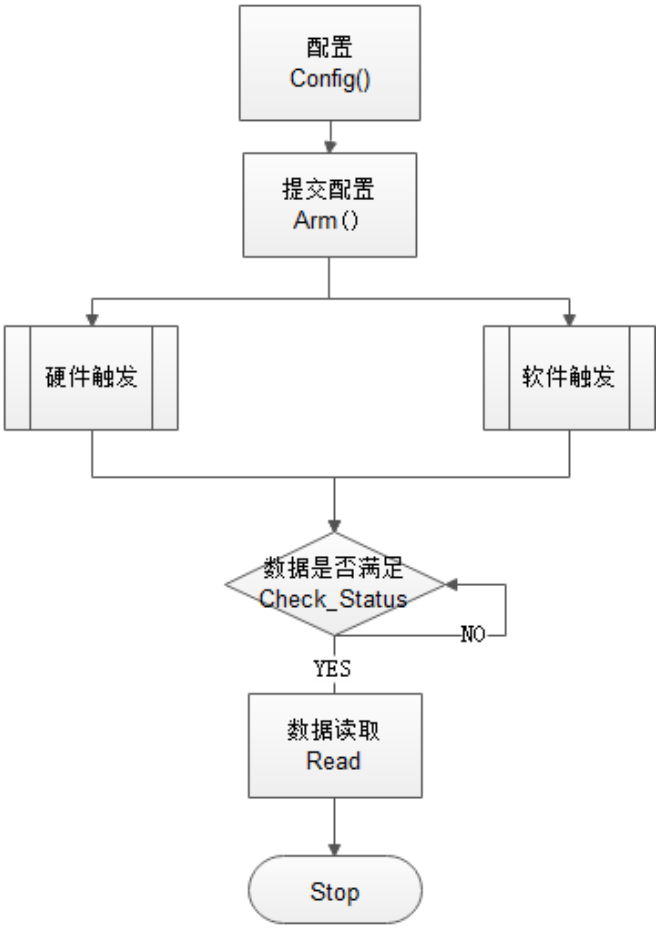


图 4-37 RxEngine **有限点采集流程**

图 4-37 RxEngine 有限点采集流程，提交配置信息后，等待有效触发信号的到来，开始进行数据采集，检查数据采集满足读取要求后，进行数据的读取，读取完所有数据后，采集过程结束。

示例代码如下：

```
HRX_ENGINE hEngine = NULL;
UINT64 samples = 0x1000;
UINT32 samples_read = 0x1000;
UINT64 samples_remaning = 0;
UINT64 samples_transferd = 0;
UINT64 actual_sample_read = 0;
UINT64 overflow = FALSE;
short userBuf = malloc(samples * sizeof(short));//申请用户
hEngine = Rx_Init(hdev, RX_ENGINE_BAR_BASE_ADDRESS);//初始化RX
Rx_SetParallelModeByteMask(hEngine, 0x3);//设置RX输入数据有效位
Rx_ConfigAquisition(hEngine, 0, samples, 1);//配置采集sample数
Rx_ConfigBuffer(hEngine, 0, samples * sizeof(short), 1);//配置buffer数量,单位bytes，注意单个sample对应的bytes数
Rx_Arm(hEngine);//提交配置信息
while (samples_remaning<samples_read)
{
    Rx_CheckStatus(hEngine, &samples_remaning, &samples_transferd, overflow);//检查可读数据，如果是制定了多个record，则换成Rx_MultiRecord_Check_Status（）
}
Rx_Read(hEngine, 0, userBuf, samples_read, 1000, &actual_sample_read, &samples_remaning);//读取点数，如果是多个record，则调用 Rx_MultiRecord_Read()
Rx_Close(hEngine);
```

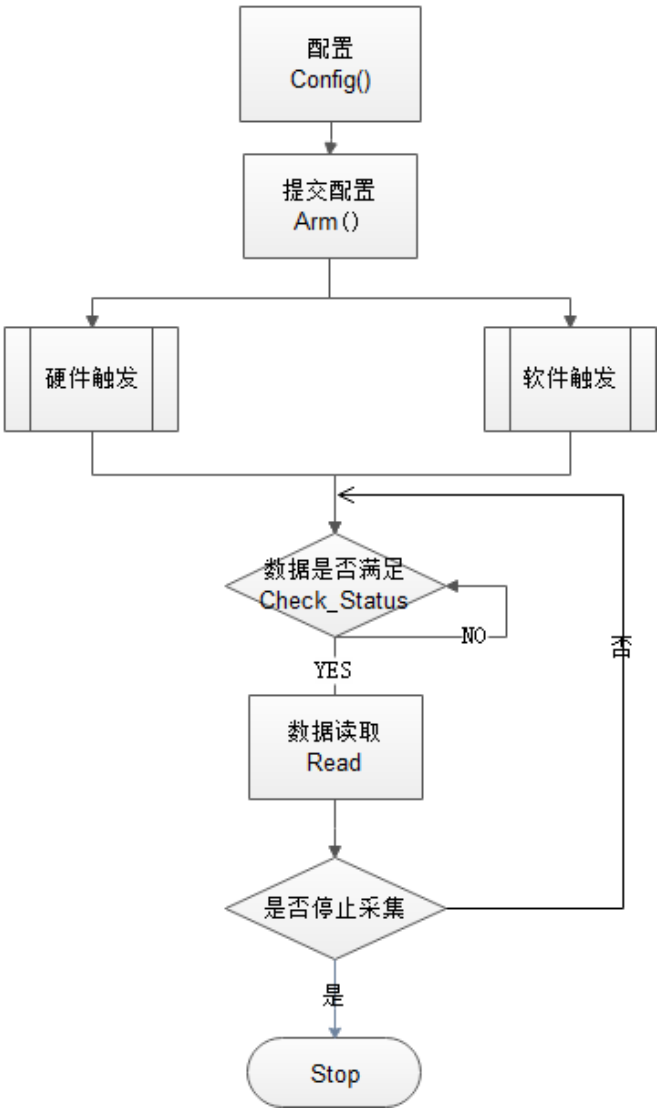


图 4-38 RxEngine 连续采集流程

图 4-38 RxEngine 连续采集流程，提交配置信息后，等待触发信号到来，触发完成后 RxEnigne 开始向指定的环形缓冲区中写入数据，上位机 C 代码调用 Rx_CheckStatus()或者 Rx_MultiRecord_Check_Status()函数，可读的数据满足要求后开始进行数据到读取。

示例代码如下：

```
HRX_ENGINE hEngine = NULL;
UINT64 samples = -1;
UINT32 samples_read = 0x1000;
UINT64 samples_remaning = 0;
UINT64 samples_transferd = 0;
UINT64 actual_sample_read = 0;
UINT64 overflow = FALSE;
short userBuf = malloc(samples * sizeof(short));//申请用户
hEngine = Rx_Init(hdev, RX_ENGINE_BAR_BASE_ADDRESS);//初始化RX
Rx_SetParallelModeByteMask(hEngine, 0x3);//设置RX输入数据有效位
Rx_ConfigAquisition(hEngine, 0, samples, 1);//配置采集sample数
Rx_ConfigBuffer(hEngine, 0, 1024 * 1024 * 100, 1);//配置buffer数量,单位bytes,
Rx_Arm(hEngine);//提交配置信息
while (1)
{
    Rx_CheckStatus(hEngine, &samples_remaning, &samples_transferd, overflow);//检查可读
数据，如果是制定了多个record，则换成Rx_MultiRecord_Check_Status（）
    Rx_Read(hEngine, 0, userBuf, samples_read, 1000, &actual_sample_read,
&samples_remaning);//读取点数，如果是多个record，则调用 Rx_MultiRecord_Read()
}
Rx_Close(hEngine);
```

4.7 TxEngine 固件发送引擎

TxEngine(简称 Tx 引擎)主要负责从板上缓存(BRAM 或者 DDR 等)指定地址处获取数据，然后发送至 D/A 等模块。与 Rx 引擎类似，TxEngine 需要和 Xilinx 的通用 IP AXI DataMover 配合使用，将 AXI 总线数据转换成 AXIS 类型数据，通过 AXIS Broadcaster 模块对数据进行划分，最后传递给 D/A 等模块。TxEnigne 的典型使用方式以及数据流向如图 4-39 TxEngine 典型使用方式所示，TxEngine 的 AXIS 数据输出可以通过 axis broadcaster 进行分割后传递到对应 D/A 等模块。

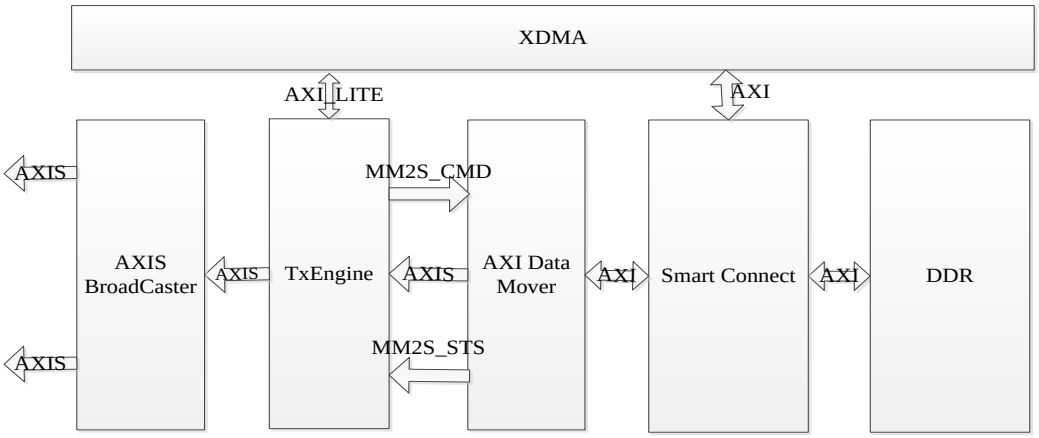


图 4-39 TxEngine 典型使用方式

4.7.1 TxEngine 固件 IP 接口

如图 4-40 TxEngine 接口所示：

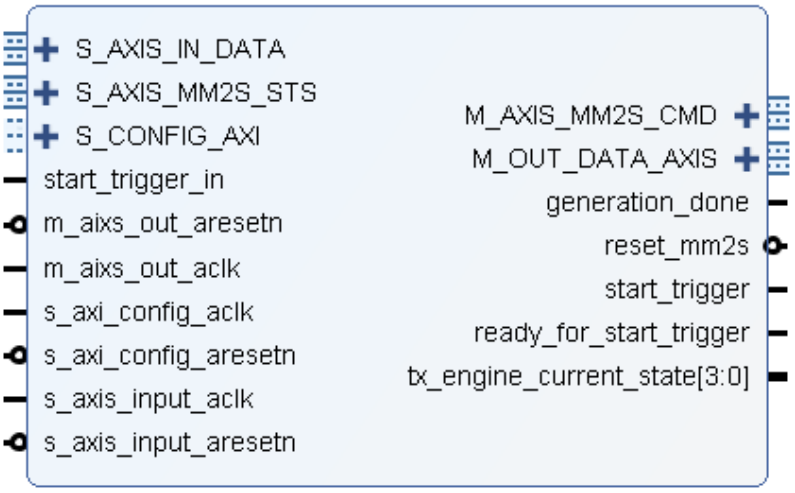


图 4-40 TxEngine 接口

- S_AXIS_IN_DATA: 从 AXI DataMover mm2s 接口输出的数据，位宽由用户指定，可选范围 8/16/32/64/128。TxEngine 控制 DataMover 读出 AXI 总线的 memory 指定地址的数据，转换成 AXIS 格式数据传输至 TxEngine 的 S_AXIS_IN_DATA 接口;
- S_AXIS_MM2S_STS: AXI DataMover MM2S 接口的状态信息，用户只需要连接至 DataMover 的 M_AXIS_MM2S_STS 接口即可；
- S_CONFIG_AXI: Slave AXI-Lite 总线，接收来自 XDMA 的寄存器配置信息，进行寄存器读写；
- start_trigger_in: 输入 trigger 信号，用于 TxEngine 状态从 WAIT_FOR_START_TRIGGER 跳转至 GENERATION 状态;
- m_axis_out_aresetn: 与 m_axis_out_aclk 同步的低电平异步复位信号；
- m_axis_out_aclk: M_OUT_DATA_AXIS 的工作时钟，一般使用 D/A 等数据接收端的工作时钟；
- s_axi_config_aclk: S_CONFIG_AXI 的工作时钟，来自 XDMA 的 AXI 总线用户时钟；
- a_axi_config_aresetn: 与 s_axi_config_aclk 同步的复位时钟；
- s_axis_input_aclk: S_AXIS_IN_DATA 的工作时钟，来自 DDR 的 ui_clk；
- s_axis_input_aresetn: 与 s_axis_input_aclk 同步的复位时钟；
- M_AXIS_MM2S_CMD: TxEngine 向 DataMover 的 MM2S 端口发送控制命令，用户只需要连接至 AXI DataMover MM2S 的 S_AXIS_MM2S_CMD 端口即可；
- M_OUT_DATA_AXIS: 输出 AXIS 数据，位宽由用户指定，可以在上位机指定输出数据的 byte 有效位，实现通道选择；

- generation_done: TxEngine 的数据生成结束时，generation_done 发出若干周期高电平有效脉冲；
- reset_mm2s: 低电平有效复位端口，用于复位 DataMover 的 MM2S 接口；
- start_trigger: 当 TxEngine 状态跳转到 GENERATION 状态时，start_trigger 向外发送若干周期高电平有效信号；
- ready_for_start_trigger: TxEngine 状态处于 WAIT_FOR_START_TRIGGER 时，该信号输出高电平；
- tx_engine_current_state: 输出 TxEngine 当前状态。

如图 4-41 TxEngine 配置选项所示，M_AXIS_OUT_TDATA_WIDTH 表示输出数据的宽度，S_AXIS_IN_TDATA_WIDTH 表示输入数据的位宽，输出数据的位宽不能大于输入数据的位宽。TX_ENGINE_MODE 有 AXIMM 和 AXIS 两种模式，分别对应于 XDMA 的 AXIMM 和 AXIS 两种工作模式。

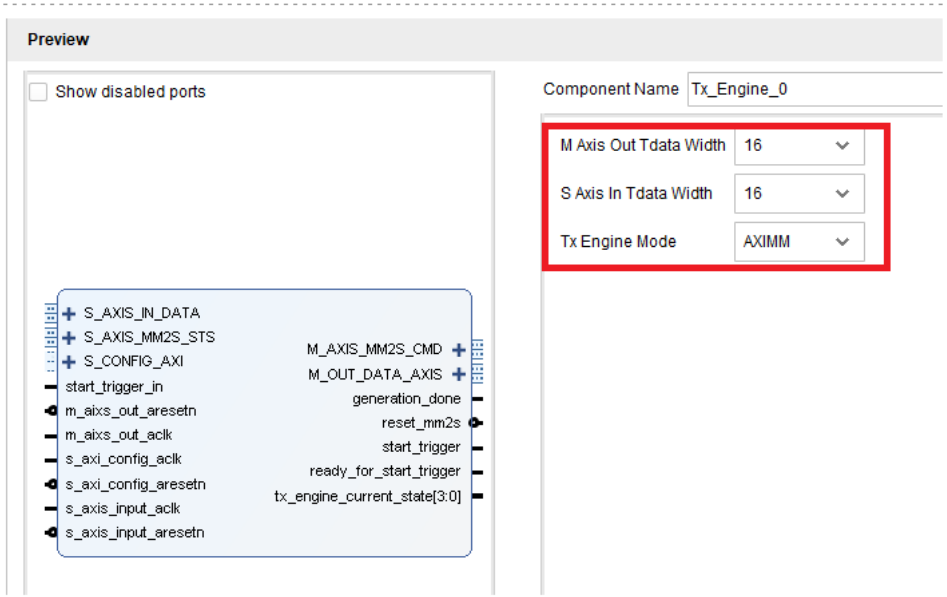


图 4-41 TxEngine 配置选项

TxEngine 在与 DataMover 配合使用时，需要设置 DataMover 如图 4-42 DataMover 配置所示，允许地址不对齐的数据传输，其它选为默认选项。

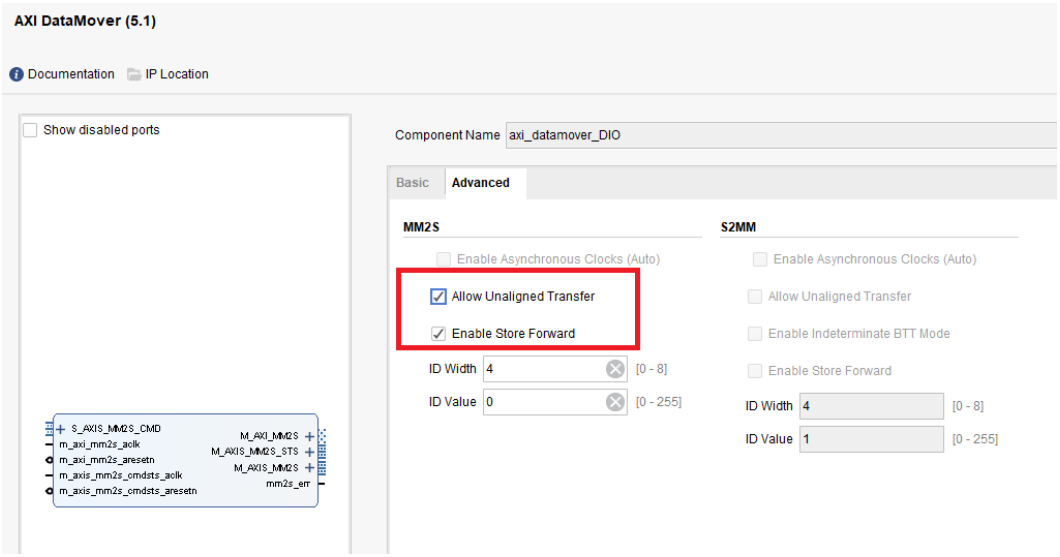


图 4-42 DataMover 配置

Tx 引擎负责 FirmDrive 数据采集框架的下行数据传输状态，如图 4-43 TxEngine 状态跳转所示目前下行数据传输共四个状态：IDLE、 WAIT_FOR_START 、GENARATION、DONE。

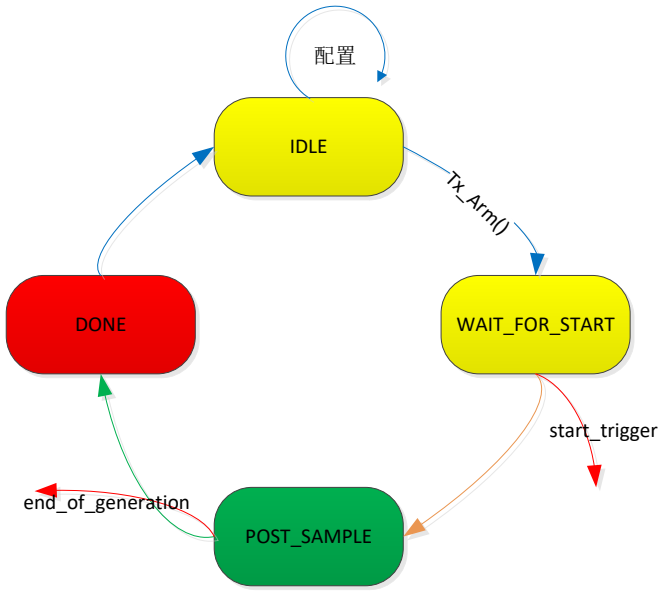


图 4-43 TxEngine 状态跳转

状态之间的转换条件可能来自三个触发源：状态本身、外部触发、上位机命令，如表 4-2 状态跳转图中不同颜色的含义所示。

TxEngine 各个状态定义如下：

- Idle—当前状态不生成数据波形，上位机在该状态对 Tx 引擎进行参数配置，当调用 Tx_Arm()函数将配置信息提交后，Tx 引擎由当前状态跳转至 WAIT_FOR_START 状态。

- WAIT_FOR_START—Tx 引擎在当前状态等待软件或者硬件触发到来，ready_for_start_trigger 在当前状态保持高电平，不进行数据采集。如果 start_trigegr 触发模式设置为立即触发，则 Tx 引擎进入该状态后立即跳转至 GENERATION 状态；如果触发模式配置为软件触发或者硬件触发，则 Tx 引擎保持在该状态，直到有效触发事件到来，然后跳转至 GENERATION 状态。
- GENERATION—Tx 引擎在当前状态根据在 IDLE 状态配置的模式，从配置的环形缓冲区读取数据生成设定波形，发送至 D/A 等。当 Tx 引擎配置为有限点生成时，待完成指定数量数据传输后，跳转至 DOEN 状态；当配置为无限点数据生成后，直到上位机发送停止命令到 Tx 引擎，从当前状态跳转至 DONE 状态，结束数据生成。
- DONE—Tx 引擎生成 End_Of_Recoard 事件，状态机在当前状态保持，等待上位机指令跳转至 IDLE 状态。

箭头颜色	状态跳转驱动源		
	软件	硬件	状态机内部
蓝色	YES		
黑色			YES
橙色	YES	YES	
绿色	YES		YES
红色	输出信号		

状态图颜色	含义
黄色	不进行数据生成
红色	停止生成
浅绿色	数据发送中

表 4-2 状态跳转图中不同颜色的含义

4.7.2 TxEngine C API

TxEngine C API 函数提供 TxEngine 的配置和数据写入，供用户调用实现数据的发送。

C API 函数接口如下：

4.7.2.1 Tx_Init()

```
/// ********************************************************************************
/// <summary>
/// 初始化Tx Engine结构体，包括分配内存的操作
/// </summary>
/// <param name="hDev">输入设备结构体</param>
/// <param name="TxEngineBaseAddress">RX模块基地址</param>
/// <returns>成功：RX传输结构体指针；失败：NULL</returns>
/// <created>yasing,2018/1/29</created>
/// <changed>yasing,2018/2/2</changed>
/// ********************************************************************************
EXTERN_C HTX_ENGINE Tx_Init(FirmDriveDevice hDev, UINT32 TxEngineBaseAddress);
```

4.7.2.2 Tx_Close()

```
/// ********************************************************************************
/// <summary>
/// 释放TX结构体资源
/// </summary>
/// <param name="hDev"></param>
/// <returns></returns>
/// <created>yasing,2018/1/30</created>
/// <changed>yasing,2018/2/2</changed>
/// ********************************************************************************
EXTERN_C int Tx_Close(HTX_ENGINE hEngine);
```

4.7.2.3 Tx_Clear()

```
/// ********************************************************************************
/// <summary>
/// 复位Tx_Engine,清空所有配置信息
/// </summary>
/// <param name="hEngine"></param>
/// <returns>成功：0；  失败：错误码</returns>
/// <created>yasing,2018/1/10</created>
/// <changed>yasing,2018/1/10</changed>
/// ********************************************************************************
EXTERN_C int Tx_Clear(HTX_ENGINE hEngine);
```

4.7.2.4 Tx_EnableHWTrigger()

```
/// ********************************************************************************
/// <summary>
/// 使能 start trigger 的硬件触发
/// </summary>
/// <param name="hEngine"></param>
/// <returns>0：成功；失败：错误码</returns>
/// <created>yasing,2018/2/2</created>
/// <changed>yasing,2018/2/2</changed>
/// ********************************************************************************
EXTERN_C int Tx_EnableHWTrigger(HTX_ENGINE hEngine);
```

4.7.2.5 Tx_ConfigTrigger()

```
/// ********************************************************************************
/// <summary>
/// 软件触发控制状态跳转，也可以由硬件进行触发，在TxEngine中外部触发源只有一个--start trigger
/// </summary>
/// <param name="hEngine">Tx_Engine结构体</param>
/// <param name="TrigerMode">触发模式none软件触发，rising edge上升沿，falling下降沿，high高电平，low低电平，immediate立刻触发</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>Yasing,2017/12/25</created>
/// <changed>yasing,2018/2/2</changed>
/// ********************************************************************************
EXTERN_C int Tx_ConfigTrigger(HTX_ENGINE hEngine, TRIGER_MODE TrigerMode);
```

Trigger mode定义如下：

```
typedef enum

{

    NONE_TRIGGER = (0x1 << 0),//software trigger

    RISING_EDGE_TRIGGER = (0x1 << 1),//rising edge trigger

    FALLING_DEGE_TRIGGER = (0x1 << 2),//falling edge trigger

    HIGH_LEVEL_TRIGGER = (0x1 << 3),//high level trigger

    LOW_LEVEL_TRIGGER = (0x1 << 4),//low level trigger

    IMMEDIATE_TRIGGER = (0x1 << 5)//immediate trigger

}TRIGER_MODE;//trigger mode
```

4.7.2.6 Tx_ConfigOutputStrobe()

```
/// ********************************************************************************
/// <summary>
/// 配置数据输出的有效byte位，比如输出固定64位时，如果outputStrobe=32'b11,则输出只有对应低16位是有效数据，其它位输出0
/// 用于进行通道选择
/// </summary>
/// <param name="hEngine">Tx_Engine结构体</param>
/// <param name="outputStrobe">输出有效位选择</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>yasing,2018/6/21</created>
/// <changed>yasing,2018/6/21</changed>
/// ********************************************************************************
EXTERN_C int Tx_ConfigOutputStrobe(HTX_ENGINE hEngine, UINT32 outputStrobe);
```

4.7.2.7 Tx_ConfigBuffer()

```
/// ********************************************************************************
/// <summary>
/// 配置FPGA板上缓冲区大小、缓冲区起始偏移量，Rx Enigne根据这里设定的地址进行数据读取
/// </summary>
/// <param name="hEngine">Tx_Engine结构体</param>
/// <param name="StartAddr">FPGA缓存起始地址，设置时需要根据FPGA设计进行，比如当DDR起始地址位0x40000,大小512M，StartAddr应大于等于0x40000,上限为512M</param>
/// <param name="Size">FPGA 板上用于Tx Engine的缓冲区大小,应小于等于FPGA板上存储资源大小</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>Yasing,2017/12/25</created>
/// <changed>yasing,2018/1/12</changed>
/// ********************************************************************************
EXTERN_C int Tx_ConfigBuffer(HTX_ENGINE hEngine, UINT32 StartAddr, UINT32 Size);
```

4.7.2.8 Tx_ConfigGeneration()

```
/// ********************************************************************************
/// <summary>
/// 配置发送点数、发送模式，可以使用循环发送模式、有限点发送、连续发送
/// </summary>
/// <param name="hEngine">Tx_Engine结构体</param>
/// <param name="TotalSamples">TotalSamples>0时为有限点发送或者循环发送，TotalSamples小于0为连续发送</param>
/// <param name="AllowRegeneration">设置是否允许循环发送数据</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>Yasing,2017/12/25</created>
/// <changed>yasing,2018/2/2</changed>
/// ********************************************************************************
EXTERN_C int Tx_ConfigGeneration(HTX_ENGINE hEngine, UINT64 TotalSamples, BOOL ContinueSent, BOOL AllowRegeneration);
```

4.7.2.9 Tx_Arm()

```
/// ********************************************************************************
/// <summary>
/// commit配置信息到FPGA，从IDLE状态跳转至WAIT_FOR_START_TRIGGER状态
/// </summary>
/// <param name="hEngine"></param>
/// <returns></returns>
/// <created>yasing,2018/1/5</created>
/// <changed>yasing,2018/1/12</changed>
/// ********************************************************************************
EXTERN_C int Tx_Arm(HTX_ENGINE hEngine);
```

4.7.2.10 Tx_Send_SW_Trigger()

```
/// ********************************************************************************
/// <summary>
/// 发送软件触发,TX开始数据采集 start trigger
/// </summary>
/// <param name="hEngine">Tx_Engine结构体</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>Yasing,2017/12/25</created>
/// <changed>yasing,2018/2/2</changed>
/// ********************************************************************************
EXTERN_C int Tx_Send_SW_Trigger(HTX_ENGINE hEngine);
```

4.7.2.11 Tx_CheckStatus()

```
/// ********************************************************************************
/// <summary>
/// 检查Tx Engine当前状态，获取当前剩余可写入的buffer长度、FPGA已经完成的数据传输量
/// </summary>
/// <param name="hEngine">设备handle</param>
/// <param name="pnBufferAvalliable">当前剩余可用buffer大小,单位是sample，单个sample对应的bytes数为hEngine->ElementSize</param>
/// <param name="pnSamplsTransfered">已经完成的数据传输量,单位是sample</param>
/// <returns>成功：0；失败：错误码</returns>
/// <created>Yasing,2018/1/4</created>
/// <changed>yasing,2018/2/2</changed>
/// ********************************************************************************
EXTERN_C int Tx_CheckStatus(HTX_ENGINE hEngine, UINT32 *pnBufferAvalliable, UINT64 *pnSamplsTransfered);
```

4.7.2.12 Tx_Write()

```
/// ********************************************************************************
/// <summary>
/// 写入数据到Tx Engine，上位机进行写指针维护，支持任意长度的数据写入
/// </summary>
/// <param name="hEngine">设备handle</param>
/// <param name="ID">选取DMA通道</param>
/// <param name="pUserBuf">用户地址</param>
/// <param name="nSamplsToWrite">写的数据长度,单位sample</param>
/// <param name="nTimeout">超时时间，ms</param>
/// <param name="pnActualSamplsWritten">实际读取数据长度,单位为sample</param>
/// <param name="pnBufferAvailable">剩余可用buffer大小，单位bytes</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Yasing,2018/1/4</created>
/// <changed>yasing,2018/4/19</changed>
/// ********************************************************************************
EXTERN_C int Tx_Write(HTX_ENGINE hEngine, UINT32 ID, PVOID pUserBuf, UINT32 nSamplsToWrite, UINT32 nTimeout,UINT32 *pnActualSamplsWritten, UINT32 *pnBufferAvailable);
```

4.7.2.13 Tx_Stop()

```
/// ************************************************************************************
/// <summary>
/// 关闭TX/RX engine，状态机恢复IDLE
/// </summary>
/// <param name="hEngine">设备handle</param>
/// <returns>成功：0；  失败：错误码</returns>
/// <created>Yasing,2018/1/4</created>
/// <changed>yasing,2018/1/12</changed>
/// ************************************************************************************
EXTERN_C int Tx_Stop(HTX_ENGINE hEngine);
```

4.7.2.14 Tx_Clear()

```
/// ************************************************************************************
/// <summary>
/// 释放TX结构体资源
/// </summary>
/// <param name="hDev"></param>
/// <returns></returns>
/// <created>yasing,2018/1/30</created>
/// <changed>yasing,2018/2/2</changed>
/// ************************************************************************************
EXTERN_C int Tx_Close(HTX_ENGINE hEngine);
```

4.7.2.15 Tx_GetErrorCodeInformation()

```
/// ************************************************************************************
/// <summary>
/// 获取错误码对应的错误信息
/// </summary>
/// <param name="errorCode">错误码</param>
/// <param name="information">错误文本信息</param>
/// <returns>成功 0；错误：错误码</returns>
/// <created>yasing,2018/7/17</created>
/// <changed>yasing,2018/7/17</changed>
/// ************************************************************************************
EXTERN_C int Tx_GetErrorCodeInformation(int errorCode, char * information);
```

4.7.3 TxEngnine API 使用方法

在介绍 API 使用之前，先对使用过程中容易混淆的地方进行说明：

sample: 1 个 sample 指代一个完整的数据包，不一定是 1 个 byte，需要根据 Tx_ConfigOutputStrobe()函数中 strobe 参数确定，TxEngine 不用区分串行、并行工作

模式，sample 所占字节数 s 等于 strobe 二进制数中 1 的个数，比如 TxEngine 输出数据 M_OUT_DATA_AXIS 位宽 64，strobe=32`b00001111，则 s=4(bytes)；

buffer: 含有 buffer 关键字的变量单位均为 byte，要注意 sample 和 byte 的对应关系，避免内存出错；

Tx_ConfigBuffer()、 Tx_ConfigGeneration()、Tx_Write()要相互配合进行调用，Tx_ConfigBuffer()中的 bufferSize 必须大于等于 Tx_Write()中的 nSamples2Write 所占的字节数。当 Tx_ConfigGeneration()中的 AllowRegeneration 为 TRUE 时，bufferSize 必须等于的 nSamples2Write 所占的字节数。

如图 4-44 Tx 基本调用流程所示，Tx 配置完成后需要先向 memory 缓存中写入一定数量的数据，然后再提交配置信息，从而避免开始时刻，memory 缓存中没有数据供 TxEngine 读取。

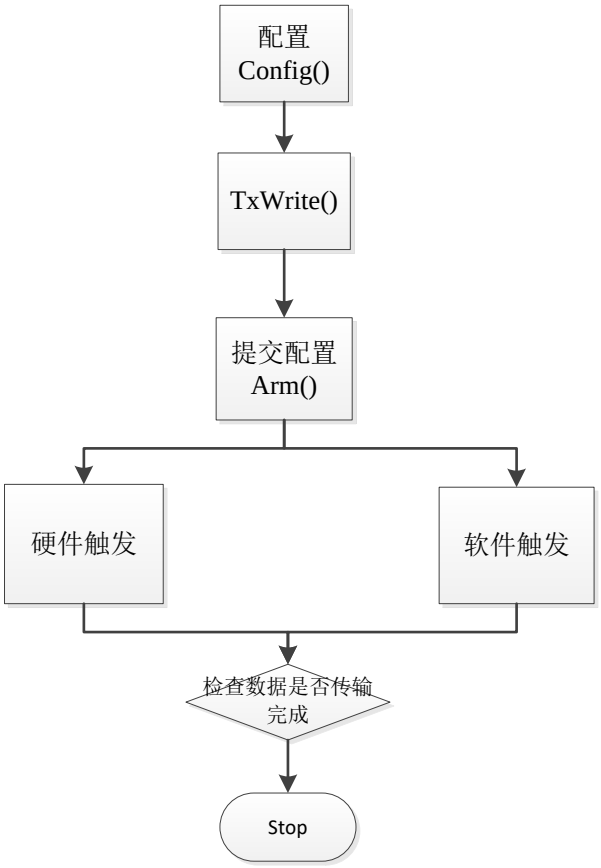


图 4-44 Tx 基本调用流程

TxEngine 有循环发送、有限点发送、连续发送三种工作模式，下面分别进行介绍：

循环发送示例如下，这里要求配置的 buffer 长度要和循环发送的 bytes 数相等，否则 TxEngine 会读取到无效数据，导致波形中参杂了无效的数据。

```
HTX_ENGINE hEngine = NULL;
UINT64 sample = 0x1000;
short userBuf = malloc(sample * sizeof(short));
UINT32 buffer_size = (UINT32)sample * sizeof(short);
Tx_Init(hDev, RX_ENGINE_BAR_BASE_ADDRESS);//初始化TxEngine
Tx_ConfigOutputStrobe(hEngine, 0x3);// output strobe信号
Tx_ConfigBuffer(hEngine, 0x0, buffer_size);//循环发送时，buffer长度应当与sample对应的字节数相等
Tx_Write(hEngine, 0, userBuf, sample, 1000, NULL, NULL);//先将数据写入环形缓冲区
Tx_Arm(hEngine);//提交配置信息
Sleep(1000);
Tx_Stop(hEngine);//关闭数据传输
Tx_Close(hEngine);//释放引擎
```

有限点发送同上，只是 buffer 不要求必须和 sample 对应的字节数相等。连续发送模式下只是需要不断检查可写入的数据量，然后写入指定数量的数据。

4.8 可变功能接口 PFI

PFI 是可编程的数字输入输出 IP，用于连接 FPGA 的外部数字 IO 与 FPGA 内部的逻辑信号。

4.8.1 PFI 固件 IP 接口

如图 4-45 PFI IP 接口所示:

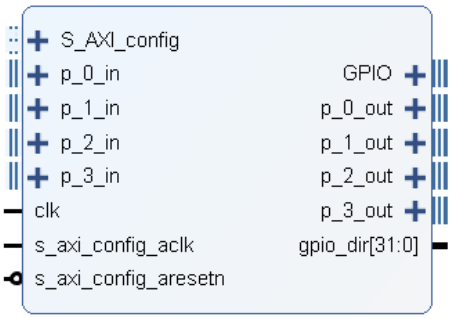


图 4-45 PFI IP 接口

- S_AXI_config: AXI-Lite 接口，用于接收 XDMA M_AXI_LITE 总线的寄存器配置、读写；
- p_0_in-p_3_in: 接收用户指定宽度的每组 8 位带宽的信号线，本例以 4 组 32bit 为例。总数可以由用户进行配置，最大 4 组 32bit 信号线，最少 1 组 8bit 的信号线；
- clk：该 IP 模块的数字 IO 工作时钟；
- s_axi_config_ackl：AXI-Lite 的工作时钟;

- s_axi_config_aresetn：与 s_axi-config_ackl 同步的 Reset 信号，用于复位该接口的 AXI-Lite 总线；
- GPIO：用于连接板卡的通用数字引脚（IO）GPIO 的位宽时由用户在此 IP 的配置里指定，最多 32 位，最少 8 位本案以 32 位为例。；
- p_0_out-p_3_out: 输出用户指定的 n 组 8bit 带宽的数字输出信号，本例以 4 组 32bit 为例。总数可以由用户进行配置，最大 4 组 32bit 信号线，最少 1 组 8bit 的信号线；
- gpio_dir[31：0]：32 路控制信号，用来控制 32 位宽的 GPIO 数字引脚的输入输出方向。

PFI 模块的功能如下：

1. 输出模式
   - 在 FPGA 控制模式下，可以将 FPGA 的逻辑电平输入(本例中 p_0_in-p_3_in，每个引脚包含 8 个 bit，所以共支持 32 路逻辑电平输入) 送到对应的 32 路物理引脚上
   - 上位机控制模式下，将上位机配置的逻辑电平送到物理引脚上
2. 在输入模式
   - 可以将 32 路物理引脚的电平送至 FPGA 的逻辑输出引脚上(本例中 p_0_out-p_3_out)供 FPGA 内部逻辑使用，同时上位机可通过 AXI-Lite 接口查询电平状态。
3. 支持 AXI-Lite 接口，可以由上位机进行配置其中可配置的功能：
   - 可配置 PFI 的输入 输出方向
   - 在输出模式下可配置时软件输出还是硬件输出，当配置为软件输出时，由上位机配置输出电平；当配置为硬件输出时，由逻辑引线 p_0_in-p_3_in 控制输出电平
   - 可以查询输入模式下的输入电平
   - 可配置是否进行数字滤波

对应的 C API 函数如下。

4.8.2 PFI C API

4.8.2.1 PFI_Config()

```
// ********************************************************************************
/// <summary>
/// 配置PFI的输入输出模式、输出模式下是软件控制输出电平还是FPGA引线控制输出电平
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">该模块的基地址，当多实例化时，配置PFI要填写相应的基地址
</param>
/// <param name="softCtrlEnable">是否由上位机控制：0-FPGA控制 1-上位机控制。
/// 该参数是位选择的，每个bit对应一个数字IO口，即32位对应32路的配置选择，比如设置通道
0,3,4,为上位机控制，那么该值应该配置为0x19</param>
/// <param name="dirCtrl">PFI的方向：1-输入 0-输出，位选择，使用方法同参数
softCtrlEnable</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaohui Hu,2018/1/19</created>
/// <changed>yasing,2018/8/22</changed>
// ********************************************************************************
EXTERN_C int PFI_Config(FirmDriveDevice hDev, unsigned int baseAddress,unsigned int
softCtrlEnable,unsigned int dirCtrl);
```

4.8.2.2 PFI_SoftOutput()

```
// ********************************************************************************
/// <summary>
/// 上位机控制PFI的输出电平
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">该模块的基地址</param>
/// <param name="outputLevel">输出电平0-低电平 1-高电平
/// 该参数是位选择的，每个bit对应一个数字IO口，即32位对应32路的配置选择，比如设置通道
0,3,4,为上位机控制输出高电平，那么该值应该配置为0x19</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaohui Hu,2018/1/19</created>
/// <changed>yasing,2018/8/22</changed>
// ********************************************************************************
EXTERN_C int PFI_SoftOutput(FirmDriveDevice hDev, unsigned int baseAddress, unsigned int
outputLevel);
```

4.8.2.3 PFI_Query()

```
// ********************************************************************************
/// <summary>
/// 查询PFI GPIO的输入电平
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">该模块的基地址</param>
/// <param name="inputLevel">输入信号的高低电平
/// 该参数是位选择的，每个bit对应一个数字IO口，即32位对应32路的配置选择，比如需要查询通道0,3,4的高低电平状态，那么读取上来该值应该为0x19</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaohui Hu,2018/1/19</created>
/// <changed>yasing,2018/8/22</changed>
// ********************************************************************************
EXTERN_C int PFI_Query(FirmDriveDevice hDev, unsigned int baseAddress, unsigned int *inputLevel);
```

4.8.2.4 PFI_DigitalFilterSel()

```
/// ********************************************************************************
/// <summary>
/// 配置滤波器
/// </summary>
/// <param name="hDev">设备句柄</param>
/// <param name="baseAddress">该模块的基地址</param>
/// <param name="channelSel">选择要配置的通道：0-31
/// 该参数是位选择的，每个bit对应一个数字IO口，比如设置通道0,3,4进行数字滤波去除抖动，那么该值应该配置为0x19</param>
/// <param name="tclkCnt">设置数字滤波要滤掉tclkcnt一下的抖动</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaohui Hu,2018/4/19</created>
/// <changed>Xiaohui Hu,2018/4/19</changed>
/// ********************************************************************************
EXTERN_C int PFI_ConfigFilter(FirmDriveDevice hDev, unsigned int baseAddress, unsigned int channelSel,unsigned int tclkCnt);
```

4.9 路由矩阵（RoutingMatrix）

RoutingMatrix 负责连通 FPGA 内部逻辑的任意两个输入与输出信号线。

4.9.1 路由矩阵固件 IP 接口

如图 4-46 RoutingMatrix 接口所示:

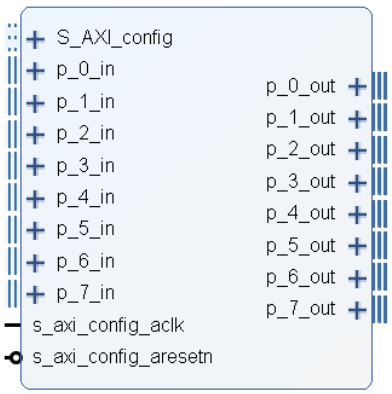


图 4-46 RoutingMatrix 接口

- S_AXI_confi: AXI-Lite 总线，用于接收 XDMA 的 AXI-Lite 总线的配置信息，进行寄存器读写；
- p_0_in-p_7_in: 每组 8bit 的逻辑信号输入信号线，用户可以指定总的输入信号位数，最小 1bit，最大 64bit，每 8 个一组。
- s_axi_config_aclk: S_AXI_config 总线的工作时钟；
- s_axi_config_aresetn: 与 s_axi_config_aclk 同步的复位时钟；
- p_0_out-p_7_out:每组 8bit 的逻辑信号输出信号线，用户可以指定总的输出信号位数，最小 1bit，最大 64bit，每 8 个一组。

RoutingMatrix 负责连通任意两个输入与输出，即输出可以选择任意输入，一个输入可以对应多个输出，但一个输出只能对应一个输入，最大支持 64 入 64 出。S_AXI_config 接口进行寄存器的读写，接收配置信息，输入输出各 8 组，每组有 8 个输入、8 个输出。

为了灵活的选择 RxEngine、TxEngine 的触发输入输出或者其它模块的输入信号，可以将这些模块的输入、输出分别连接到 RoutingMatrix 的输出、输入上。

p_0_in 的 8 个 bit 从低到高位依次对应编号 0-7，以此类推，p_7_in 的 8 个 bit 从低到高位依次对应编号 56-63。p_0_out 的 8 个 bit 从低到高位依次对应编号 0-7，以此类推，p_7_out 的 8 个 bit 从低到高位依次对应编号 56-63。

C API 函数接口如下：

4.9.2 Routing Matrix C API

4.9.2.1 Routing_Matrix_Config()

```
// ********************************************************************************
/// <summary>
/// 配置Routing_Matrix的输入输出之间的连接
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">该IP模块对应的基地址，在固件开发时由用户指定</param>
/// <param name="outputRoutingChannel">设置的输出信号通道编号：RoutingMatrix最多64位输入，最少1位输入，取值范围0-63</param>
/// <param name="inputRoutingSel">选择输入信号通道编号：RoutingMatrix最多64位输出，最少1位输出，取值范围0-63</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaohui Hu,2018/1/4</created>
/// <changed>yasing,2018/8/22</changed>
// ********************************************************************************
EXTERN_C int Routing_Matrix_Config(FirmDriveDevice hDev, unsigned int baseAddress,int outputRoutingChannel,int inputRoutingSel);
```

4.10 模拟触发（AnalogTrigger）

模拟触发模块负责分析模拟输入信号，符合设定要求后向外输出触发信号。

4.10.1 模拟触发固件 IP 接口连接

如图 4-47 AnalogTrigger 接口所示:

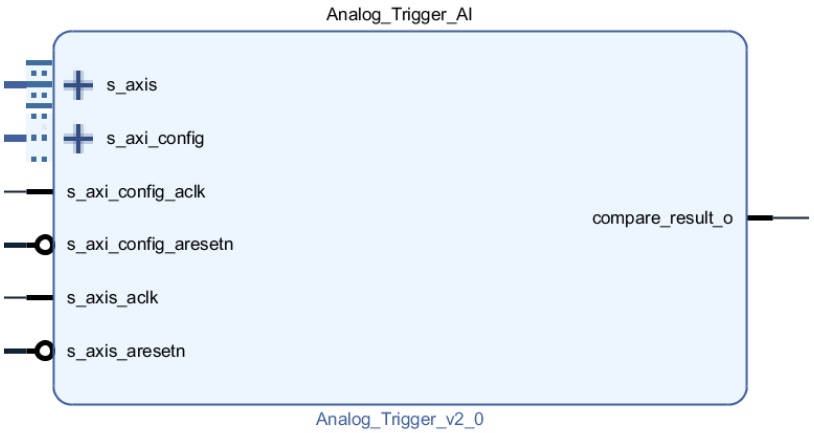


图 4-47 AnalogTrigger 接口

- s_axis：Slave AXIS 输入数据总线，AnalogTrigger 的输入也可以配置成串行输入模式、或者并行输入模式，串行模式下，多个 tvalid 信号对应一个 tlast 信号，表示一个完整 sample 的结束，配置触发时，统一使用首个通道的数据进行比较；并行模式下默认输入的每个 tvalid 都对应了一个 tlast 信号；

- s_axi_config：Slave AXI_LTE 总线，接收来自 XDMA 的配置信息，进行寄存器读写；
- s_axi_config：s_axi_config 总线的工作时钟；
- s_axi_arestn: 与 s_axi_config 同步的低电平有效复位信号；
- s_axis_aclk：s_axis 输入数据总线的工作时钟；
- s_axis_arestn: 与 s_axis_aclk 同步的复位低电平有效复位信号；
- compare_result_0: Analog_Trigger 模块完成上位机指定的比较后输出高电平有效信号，用于连接到 RoutingMatrix 或者 PFI 与其它信号线进行动态连接。

该模块可支持 2 种触发模式：滞回比较器和窗口比较器：

- 滞回比较器：当信号大于阈值上限时输出高电平，小于阈值下限时输出低电平。该模式下可有效防抖动
- 窗口比较器：当信号大于阈值上限或小于阈值下限时输出高电平，当信号小于阈值上限并且大于阈值下限时输出低电平。

C API 函数如下：

4.10.2 AnalogTrigger C API

```
// ********************************************************************************
/// <summary>
/// 模拟触发参数配置
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress"></param>
/// <param name="highThreshold">阈值上限，归一化值,其中 1 对应满量程的正值</param>
/// <param name="lowThreshold">阈值下限，归一化值，其中 -1 对应满量程的负值</param>
/// <param name="modeSel">模式选择：HYSTERESIS=0  滞回比较器  WINDOW=1 窗口比较器
</param>
/// <param name="mask">掩码：并行数据时可以选择哪些byte有效，比如32’b11表示低16bit有
效，使用低16bit进行比较；
/// 串行数据时使用串行数据的第一个通道进行比较，mask为0</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaohui Hu,2018/1/4</created>
/// <changed>yasing,2018/8/22</changed>
// ********************************************************************************
EXTERN_C int AnalogTrigger_Config(FirmDriveDevice hDev, unsigned int baseAddress,double
highThreshold,double lowThreshold,ANALOGTRIGGER_MODE modeSel,unsigned int mask);
```

ANALOGTRIGGER_MODE 结构体定义：

```
typedef enum
{
    HYSTERESIS,//滞回比较器
    WINDOW,//窗口比较器
}ANALOGTRIGGER_MODE;
```

4.11 模式触发（PatternTrigger）

PattenTrigger 负责将输入数据与上位机指定值进行比较，如果相等就输出高电平，不相等则输出低电平。

4.11.1 模式触发 IP 接口

如图 4-48 PatternTrigger IP 接口所示：

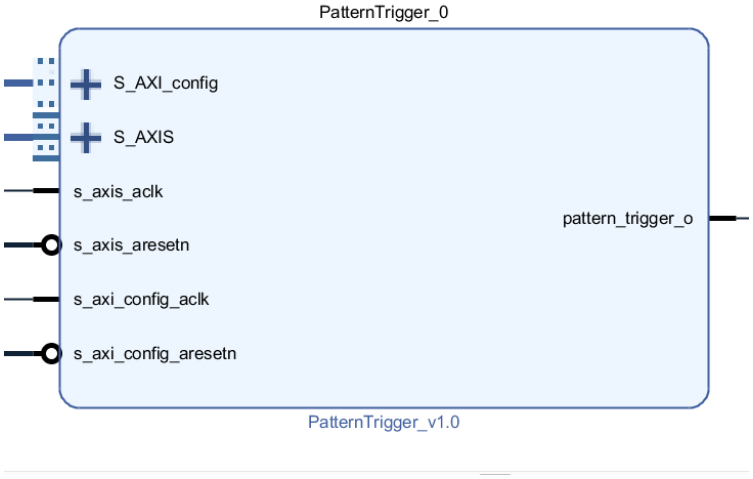


图 4-48 PatternTrigger IP 接口

- S_AXI_config: Slave AXI-Lite 总线，接收来自 XDMA 的寄存器配置信息；
- S_AXIS: Slave AXIS 数据输入，输入数据位宽由用户指定；
- s_axis_aclk: S_AXIS 接口输入数据的工作时钟；
- s_axis_arestn: 与 s_axis_aclk 同步的复位时钟；
- s_axi_config_aclk: S_AXI_config 接口的工作时钟；
- s_axi_config_aresetn: 与 s_axi_config_aclk 同步的复位时钟；
- patten_trigger_0: 输出比较结果，高电平有效。

C API 函数接口如下：

4.11.2 PatternTrigger C API

4.11.2.1 PatternTrigger_Config()

```
// ********************************************************************************
/// <summary>
/// 配置Pattern Trigger的匹配触发的值和bit位选择
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">该模块的基地址，由固件开发人员指定</param>
/// <param name="value">patten trigger需要匹配的值</param>
/// <param name="valueMask">使能需要做匹配的bit位，如果只需要匹配低8位，该值设置为
0xFF</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaohui Hu,2018/3/26</created>
/// <changed>yasing,2018/8/22</changed>
// ********************************************************************************
EXTERN_C int PatternTrigger_Config(FirmDriveDevice hDev, unsigned int baseAddress,int
value,int valueMask);
```

4.12 计数器（Counter）

Counter 负责进行增减计数，实现通用数据采集的计数器功能。Counter 可以将计数结果作传送给 RxEngine、Routing Matrix、或者输出传输到硬件外设，工作时钟可以是内部时钟也可以是外部时钟。

4.12.1 计数器的固件 IP 接口

如图 4-49 Counter 计数器接口所示：

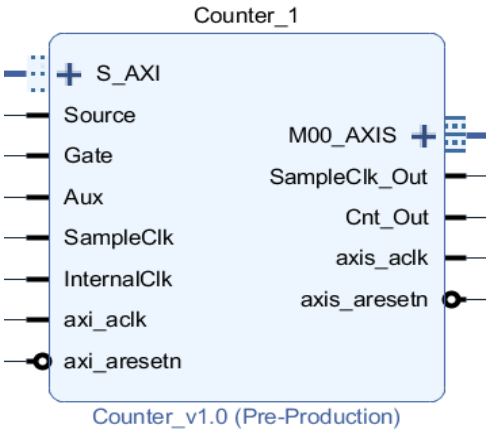


图 4-49 Counter 计数器接口

- S_AXI: Slave AXI-Lite 总线，用于接收 XDMA 的配置信息，进行寄存器的读写。
- Source: 要进行计数的信号源，来自 RoutingMatrix 或者 PFI 模块；
- Gate：输入门限信号，来自 RoutingMatrix 或者 PFI 模块，可以在 Gate 的上升沿启动计数，直到下一次 Gate 信号的上升沿到来；

- Aux: 辅助输入信号，来自 RoutingMatrix 或者 PFI 模块，可以在 Aux 信号的上升沿启动计数，直到下一次 Aux 信号的上升沿到来；
- SampleClk: 计数器工作的外部时钟；
- InternalClk: 固件提供的固定内部时钟，可以使用 clk_wizard(VIVADO 内部信号)生成；
- axi_aclk: S_AXI 的工作主时钟；
- axi_aresetn: 与 axi_aclk 同步的低电平有效复位信号；
- M00_AXIS: 计数器的 AXIS 数据输出，连接至 RxEngine 用于实时获取当前计数值；
- SampleClk_Out: 计数器的当前工作时钟输出；
- Cnt_Out: 计数器输出信号，当满足上位机指定的计数值后，输出高电平；
- axis_aclk: M00_AXIS 的工作时钟，提供给下游 RxEngine 输入数据接口使用；
- axis_aresetn: 与 axis_aclk 同步的低电平有效复位信号。

4.12.2 计数器 C API

4.12.2.1 Counter_SetInternalClkDivNumber()

```
// ********************************************************************************
/// <summary>
/// 设置InternalClk的分频数，Counter输入输出同时有效
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="divNumber">内部时钟分频数</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/6/20</created>
/// <changed>yasing,2018/8/22</changed>
// ********************************************************************************
EXTERN_C    int     Counter_SetInternalClkDivNumber(FirmDriveDevice hDev, unsigned int baseAddress, unsigned int divNumber);
```

4.12.2.2 CounterIn_SetSampleClkDivNumber()

```
/// ************************************************************************************
/// <summary>
/// 设置SampleClk的分频数,当SampleClk源为内部时钟时有效
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="divNumber">分频数</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/6/20</created>
/// <changed>Xiaojiao Xie,2018/6/20</changed>
/// ************************************************************************************
EXTERN_C    int      CounterIn_SetSampleClkDivNumber(FirmDriveDevice  hDev,  unsigned  int
baseAddress, unsigned int divNumber);
```

4.12.2.3 CounterIn_SetSampleClkSource()

```
/// ************************************************************************************
/// <summary>
/// 选择counter的工作时钟，可选外部或者板上固定内部时钟
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="sampleClkSource">分频数</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/6/20</created>
/// <changed>Xiaojiao Xie,2018/6/20</changed>
/// ************************************************************************************
EXTERN_C    int      CounterIn_SetSampleClkSource(FirmDriveDevice  hDev,  unsigned  int
baseAddress, SampleClkSource sampleClkSource);

typedef enum
{
    InternalSampleClk = 0,//内部时钟
    ExternalSampleClk = 1//外部时钟
}
SampleClkSource;
```

4.12.2.4 CounterIn_SetDirection()

```
/// ********************************************************************************
/// <summary>
/// 设置计数方向，增计数或者减计数
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="direction">计数方向,Up/Down/External，为External时高电平加计数，低电平减计数</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/4/9</created>
/// <changed>Xiaojiao Xie,2018/4/9</changed>
/// ********************************************************************************
EXTERN_C   int    CounterIn_SetDirection(FirmDriveDevice hDev,unsigned int baseAddress, CounterInDirection direction);
```

4.12.2.5 CounterIn_SetInitialCount()

```
/// ********************************************************************************
/// <summary>
/// 设置初始计数值
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="initialCount">初始计数值</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/4/9</created>
/// <changed>Xiaojiao Xie,2018/4/9</changed>
/// ********************************************************************************
EXTERN_C    int       CounterIn_SetInitialCount(FirmDriveDevice  hDev,  unsigned  int baseAddress,unsigned int initialCount);
```

4.12.2.6 CounterIn_SetCounterType()

```
/// ********************************************************************************
/// <summary>
/// 设置计数类型，分为边沿计数、测量频率和周期（带平均）、方波测量、双边沿间隔等
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="counterType">计数类型</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/4/9</created>
/// <changed>Xiaojiao Xie,2018/4/9</changed>
/// ********************************************************************************
EXTERN_C  int   CounterIn_SetCounterType(FirmDriveDevice hDev, unsigned int baseAddress, CounterInType counterType);
```

4.12.2.7 CounterIn_EnableSampleClk()

```
/// ********************************************************************************
/// <summary>
/// 是否使能采样时钟
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="enableSampleClk">采样时钟使能，CounterInType为FrequencyOrPeriodMeasure
时，enableSampleClk必须为EnableExplicitClk</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/4/9</created>
/// <changed>Xiaojiao Xie,2018/4/9</changed>
/// ********************************************************************************
EXTERN_C  int  CounterIn_EnableSampleClk(FirmDriveDevice hDev, unsigned int baseAddress,
EnableClk enableSampleClk);
```

4.12.2.8 CounterIn_Start()

```
/// ********************************************************************************
/// <summary>
/// 开始计数
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/4/9</created>
/// <changed>Xiaojiao Xie,2018/4/9</changed>
/// ********************************************************************************
EXTERN_C  int   CounterIn_Start(FirmDriveDevice hDev,unsigned int baseAddress );
```

4.12.2.9 CounterIn_Stop()

```
/// ********************************************************************************
/// <summary>
/// 停止计数
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/4/9</created>
/// <changed>Xiaojiao Xie,2018/4/9</changed>
/// ********************************************************************************
EXTERN_C  int   CounterIn_Stop(FirmDriveDevice hDev, unsigned int baseAddress);
```

4.12.2.10 CounterIn_ReadCounter()

```
/// ********************************************************************************
/// <summary>
/// 读计数值
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="counter1">计数值，
///CounterType为EdgeCounter时，返回计数值；
///CounterType为FrequencyOrPeriodMeasure时，返回脉冲数；
///CounterType为PulseMeasure时，返回低电平的脉冲数；
///CounterType为EdgeSeparation时，返回第一个信号上升沿与第二个信号上升沿之间的脉冲数；
///</param>
/// <param name="counter2">计数值，
///CounterType为EdgeCounter时，无效；
///CounterType为FrequencyOrPeriodMeasure时，返回周期数；
///CounterType为PulseMeasure时，返回高电平的脉冲数；
///CounterType为EdgeSeparation时，返回第二个信号上升沿与第一个信号第二次上升沿之间的脉冲数；
///</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/4/9</created>
/// <changed>Xiaojiao Xie,2018/4/9</changed>
/// ********************************************************************************
EXTERN_C	int		CounterIn_ReadCounter(FirmDriveDevice	hDev,	unsigned	int	baseAddress,unsigned int	* counter1, unsigned int	* counter2 );
```

4.12.2.11 CounterOut_SetInitialDelay()

```
/// ********************************************************************************
/// <summary>
/// 设置计数器初始延迟，延迟指定周期数
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="initialDelay">初始延迟，单位clk，当前工作时钟周期</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/5/25</created>
/// <changed>Xiaojiao Xie,2018/5/25</changed>
/// ********************************************************************************
EXTERN_C	int	CounterOut_SetInitialDelay(FirmDriveDevice hDev, unsigned int baseAddress unsigned int initialDelay);
```

4.12.2.12 CounterOut_SetPulseCounts()

```
/// ********************************************************************************
/// <summary>
/// 设置脉冲计数时的脉冲计量个数
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="pulseCounts">脉冲循环次数，pulseCounts=-1时，循环次数无限大</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/4/9</created>
/// <changed>Xiaojiao Xie,2018/4/9</changed>
/// ********************************************************************************
EXTERN_C  int  CounterOut_SetPulseCounts(FirmDriveDevice hDev, unsigned int baseAddress,
int pulseCounts);
```

4.12.2.13 CounterOut_SetHighLowTicks()

```
/// ********************************************************************************
/// <summary>
/// 设置输出波形的高低tick数，一个tick表示一个当前工作的时钟周期
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="highTicks">高电平保持的source周期个数</param>
/// <param name="lowTicks">低电平保持的source周期个数</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/5/25</created>
/// <changed>Xiaojiao Xie,2018/5/25</changed>
/// ********************************************************************************
EXTERN_C  int  CounterOut_SetHighLowTicks(FirmDriveDevice hDev, unsigned int baseAddress
unsigned int highTicks, unsigned int lowTicks);
```

4.12.2.14 CounterOut_SetIdleState()

```
/// ********************************************************************************
/// <summary>
/// 设置开始输出时信号的电平
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="idleState">初始状态:LowLevel/HighLevel</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/5/25</created>
/// <changed>Xiaojiao Xie,2018/5/25</changed>
/// ********************************************************************************
EXTERN_C  int  CounterOut_SetIdleState(FirmDriveDevice hDev, unsigned int baseAddress,
CounterOutIdleState idleState);
```

4.12.2.15 CounterOut_SetStep()

```
/// ************************************************************************************
/// <summary>
/// 设置计数器步进
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="step">步进</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/5/25</created>
/// <changed>Xiaojiao Xie,2018/5/25</changed>
/// ************************************************************************************
EXTERN_C    int    CounterOut_SetStep(FirmDriveDevice  hDev,  unsigned  int  baseAddress,
unsigned int step);
```

4.12.2.16 CounterOut_Start()

```
/// ************************************************************************************
/// <summary>
/// 开始输出
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/5/25</created>
/// <changed>Xiaojiao Xie,2018/5/25</changed>
/// ************************************************************************************
EXTERN_C  int  CounterOut_Start(FirmDriveDevice hDev, unsigned int baseAddress);
```

4.12.2.17 CounterOut_Stop()

```
/// ************************************************************************************
/// <summary>
/// 停止输出
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <returns>成功：0；错误：错误码</returns>
/// <created>Xiaojiao Xie,2018/5/25</created>
/// <changed>Xiaojiao Xie,2018/5/25</changed>
/// ************************************************************************************
EXTERN_C  int  CounterOut_Stop(FirmDriveDevice hDev, unsigned int baseAddress);
```

4.13 Streaming 数据拼接

FirmDrive®提供的 Package_streaming 将输入端口的分散原始数据拼接成指定位宽的 streaming 数据总线。用户可以在此 IP 中指定位宽。一个典型的应用时将 DIO 的不同通道（不同 Bits）拼接成指定位宽的 AXIS 数据流。

4.13.1 Packag_streaming 固件 IP 接口

如图 4-50 Package_streaming 所示：

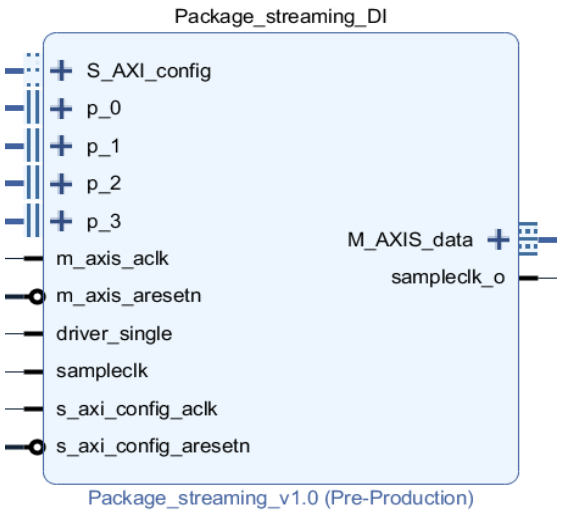


图 4-50 Package_streaming IP 接口

- S_AXI_config: Slave AXI-Lite 数据总线，接收 XDMA 的寄存器配置信息，读写寄存器；
- p_0-p_3: 4 组 8 位的输入信号线，共 32 位，输入源来自 RoutingMatrix；
- m_axis_aclk: p_0-p_3 输入信号，M_AXIS_data 均工作在该时钟下；
- m_axis_aresetn: 与 m_axis_aclk 同步的复位信号；
- driver_single: IP 根据该信号设定的触发模式，包括上升沿触发、下降沿触发、高电平触发、低电平触发、立即触发输出一次 M_AXIS_data 的 tvalid 有效数据；
- sampleclk: 同步时钟，当需要进行板卡间时钟同步时，sampleclk 替代 m_axis_aclk 作为 M_AXIS_data 的基准时钟；
- s_axi_config_aclk: S_AXI_config 的工作时钟；
- s_axi_config_aresetn: s_axi_config_aclk 的同步低电平有效复位时钟；
- M_AXIS_data: 输出 Master AXIS 总线数据；
- sample_clk_o: m_axis_aclk 或者 sampleclk 进行上位机指定分频后的时钟输出。

C API 函数如下：

4.13.2 Package_streaming C API

4.13.2.1 PackageStream_Config()

```
/// ************************************************************************************
/// <summary>
/// 对PackageStream进行设置，配置drive_single信号的触发模式、输出sample_clk_o的分频系数
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="modeSel">模式选择：     POS_EDGE = 0,      上升沿
///                                      NEG_EDGE = 1,      下降沿
///                                      DDR_EDGE = 2,      双沿
///                                      HIGH_LEVEL_MODE = 3,高电平
///                                      LOW_LEVEL_MODE = 4, 低电平
///                                      ALWAYS_VALID = 5,    一直有效，即时钟驱动
下，每个clk都有有效数据
///                                      DIV_VALID = 6      分频，即时钟驱动下，每
DIV_VALID个clk有一个有效数据
/// <param name="divNum">分频系数</param>
/// <param name="tlastSet">TLast有效情况：0xFFFFFFFF表示不加TLast，其余表示每tlastSet多
个有效数据加一个tlast标志</param>
/// <returns></returns>
/// <created>Xiaohui Hu,2018/4/16</created>
/// <changed>Xiaohui Hu,2018/4/17</changed>
/// ************************************************************************************
EXTERN_C     int     PackageStream_Config(FirmDriveDevice     hDev,     unsigned     int
baseAddress,PACKAGE_MODE modeSel,int divNum,int tlastSet);
```

4.14 Streaming 数据分割

Unpackage_stream 负责将输入的 AXIS 总线数据分割成若干组 8bit 的信号线进行输出(总数由用户设置，最少时一组，1bit，最多时 4 组 32bit)。一个典型的应用时在高速 DIO 时将 AXIS 数据流分割。

4.14.1 Unpackage_stream 固件 IP 接口

如图 4-51 Unpackage_stream IP 接口所示：

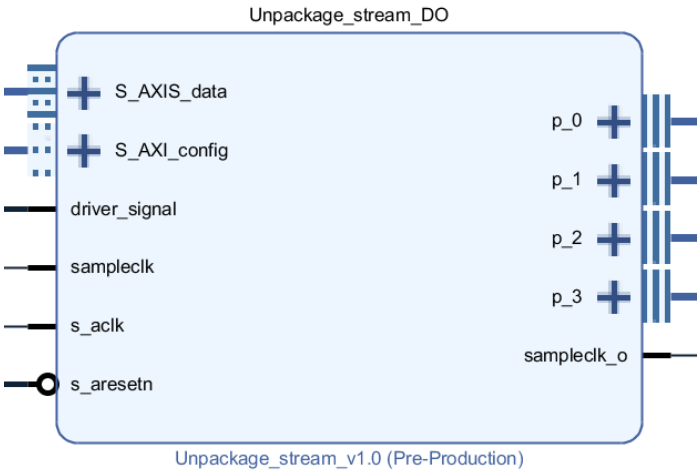


图 4-51 Unpackage_stream IP 接口

- S_AXIS_data: AXIS 输入数据，位宽由用户指定，范围为 8、16、32；
- S_AXI_config: Slave AXI-Lite 总线，用于接收 XDMA 的配置信息，进行寄存器读写；
- drive_signal:IP 根据该信号设定的触发模式，包括上升沿触发、下降沿触发、高电平触发、低电平触发、立即触发拆分一次 M_AXIS_data 的有效数据；
- sample_clk:同步时钟，当需要进行板卡间时钟同步时，sampleclk 替代 s_aclk 作为 S_AXIS_data 的基准时钟；
- s_aclk：S_AXI_config 的工作时钟；
- s_aresetn: 与 s_aclk 同步的低电平有效复位信号；
- p_0-p_3:每组 8bit 的输出信号线，总数由固件开发人员指定，最少 1bit，最多 32bit。本例中为 32bit，4 组输出信号；
- sampleclk_o: s_aclk 或者 sampleclk 进行上位机指定分频后的时钟输出。

C API 函数如下：

4.14.2 Unpackage_stream C API

4.14.2.1 UnPackageStream_Config()

```
/// ********************************************************************************
/// <summary>
/// 将输入的stream信号根据配置进行数据的分割，将多位数据分割为多个一位数据
/// </summary>
/// <param name="hDev">板卡句柄</param>
/// <param name="baseAddress">基地址</param>
/// <param name="modeSel">模式选择：     POS_EDGE = 0,      上升沿
///NEG_EDGE = 1,        下降沿
///DDR_EDGE = 2,        双沿
/// HIGH_LEVEL_MODE = 3,高电平
/// LOW_LEVEL_MODE = 4, 低电平
/// ALWAYS_VALID = 5,    一直有效，即时钟驱动下，每个clk都有有效数据
/// DIV_VALID = 6      分频，即时钟驱动下，每DIV_VALID个clk有一个有效数据
/// <param name="divNum">分频系数</param>
/// <returns></returns>
/// <created>Xiaohui Hu,2018/4/16</created>
/// <changed>Xiaohui Hu,2018/4/17</changed>
/// ********************************************************************************
EXTERN_C int UnPackageStream_Config(FirmDriveDevice hDev, unsigned int baseAddress,
UNPACKAGE_MODE modeSel,int divNum);
```

5 使用 FirmDrive®构造常用测量 C API

数据采集和模块仪器有以下六大类的应用：

模拟输入（Analog Input，AI）

模拟输出（Analog Output，AO）

数字输入（Digital Input，DI）

数字输出（Digital Output，DO）

计数器输入（Counter Input，CI）

计数器输出（Counter Output，CO）

除了以上的六类应用外，还需要一些辅助的功能比如说缓冲区管理、报错等。简仪科技在锐视测控平台的 C#驱动里详细定义了这六大类和使用这六大类需要的其它辅助工具。FirmDrive®引擎对这六类功能和其它辅助功能都做了非常好的支持。本章介绍如何使用 FirmDrive® 模块来构造以上六类的 C API 和如何生成 C 驱动库

为了便于硬件设计师能够更好地理解和掌握使用 FirmDrive®技术，简仪特地设计了 PXIe-1010DK 开发套件。以下的文档都是以 PXIe-1010DK 为例的。DB102 功能板的名称。

5.1 构造模拟输入（AI）

5.1.1 AI C API 详细参数

5.1.1.1 DB102API int DB102_AI_EnableChannel(JY_DeviceHandle hDevice, unsigned int channelCount, unsigned char channels);*

表 5-1 DB102_AI_EnableChannel

函数功能：	使能 AI 通道	
输入参数：	hDevice	板卡资源句柄
	channelCount	要使能的通道个数。
	channels	通道数组，定义要使能的通道号
返回值：	成功返回 0，失败返回错误码	

5.1.1.2 DB102API int DB102_AI_SetMode(JY_DeviceHandle hDevice, DB102_AI_SampleMode mode);

表 5-2 DB102_AI_SetMode

函数功能：	设置 AI 采集模式	
输入参数：	hDevice	板卡资源句柄
	mode	AI 采集模式（可选 Finite 或者 Continuous）
返回值：	成功返回 0，失败返回错误码	

5.1.1.3 DB102API int DB102_AI_SetSampleRate(JY_DeviceHandle hDevice, double sampleRate, double actualSampleRate);*

表 5-3 DB102_AI_SetSampleRate

函数功能：	设置 AI 采样率	
输入参数：	hDevice	板卡资源句柄
	sampleRate	需要设置的 AI 采样率
输出参数	actualSampleRate	实际设置的 AI 采样率

返回值：	成功返回 0，失败返回错误码

5.1.1.4 DB102API int DB102_AI_SetSamplesToAcquire(JY_DeviceHandle hDevice,unsigned int sampleToAcquire);

表 5-4 DB102_AI_SetSamplesToAcquire

函数功能：	设置 AI 有限采集点数	
输入参数：	hDevice	板卡资源句柄
	sampleToAcquire	有限采集模式下需要采集的点数
返回值：	成功返回 0，失败返回错误码	

5.1.1.5 DB102API int DB102_AI_Start(JY_DeviceHandle hDevice);

表 5-5 DB102_AI_Start

函数功能：	启动 AI 采集	
输入参数：	hDevice	板卡资源句柄
返回值：	成功返回 0，失败返回错误码	

5.1.1.6 DB102API int DB102_AI_CheckBufferStatus(JY_DeviceHandle hDevice, unsigned long* avaliableSamples, bool* overrun);

表 5-6 DB102_AI_CheckBufferStatus

函数功能：	获取缓冲区状态	
输入参数：	hDevice	板卡资源句柄
输出参数：	avaliableSamples	缓冲区当前可读取的点数
	overrun	缓冲区是否溢出
返回值：	成功返回 0，失败返回错误码	

5.1.1.7 DB102API int DB102_AI_ReadData(JY_DeviceHandle hDevice, double* dataBuffer, unsigned int dataLength, int timeOut, unsigned long* actualReadLength);

表 5-7 DB102_AI_ReadData

函数功能：	读取数据	
输入参数：	hDevice	板卡资源句柄
	dataBuffer	数据数组，用来存放读取到的数据
	dataLength	需要读取的数据长度
	timeOut	超时时间
输出参数：	actualReadLength	实际读取到的数据长度
返回值：	成功返回 0，失败返回错误码	

5.1.1.8 DB102API int DB102_AI_Stop(JY_DeviceHandle hDevice);

表 5-8 DB102_AI_Stop

函数功能：	停止 AI 采集	
输入参数：	hDevice	板卡资源句柄
返回值：	成功返回 0，失败返回错误码	

5.1.2 AI 基本工作流程

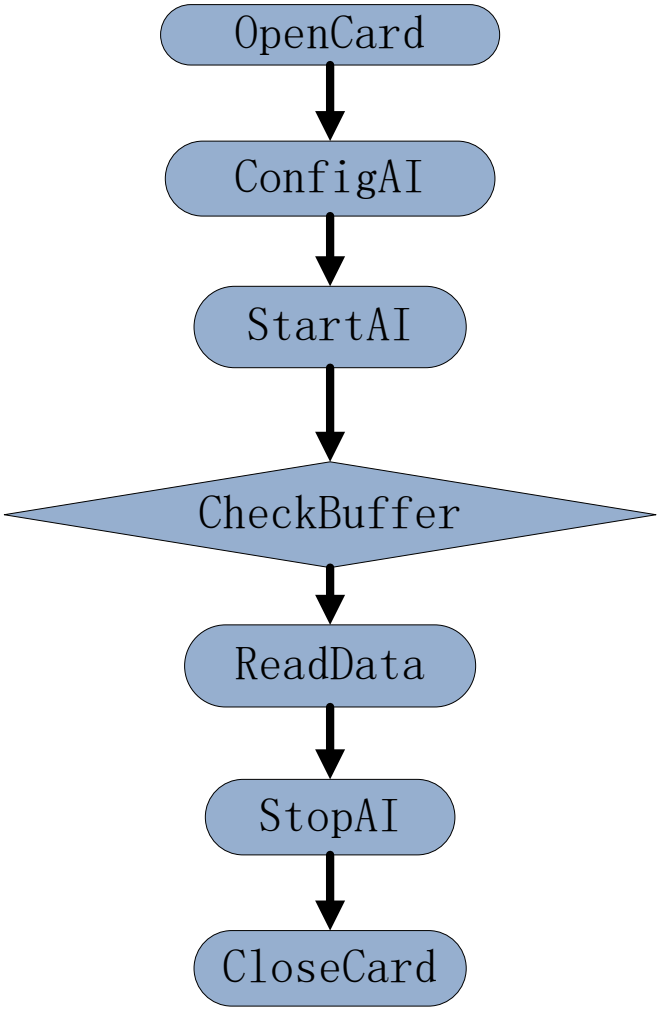


图 5-1 AI 基本工作流程

对于模拟输入大致流程如上图，

- 首先打开板卡，输入对应的 VID，PID，卡槽号可正常打开板卡，返回板卡句柄。
- 配置 AI：包括配置 AI 采集通道，配置采样率，配置采集模式等：
- 启动 AI 采集
- 查询缓冲区中的数据量
- 读取数据
- 停止 AI 采集
- 关闭板卡

下面会在有限点采集和连续采集 2 个典型应用中介绍详细的流程。

5.1.3 范例 1：有限点采集

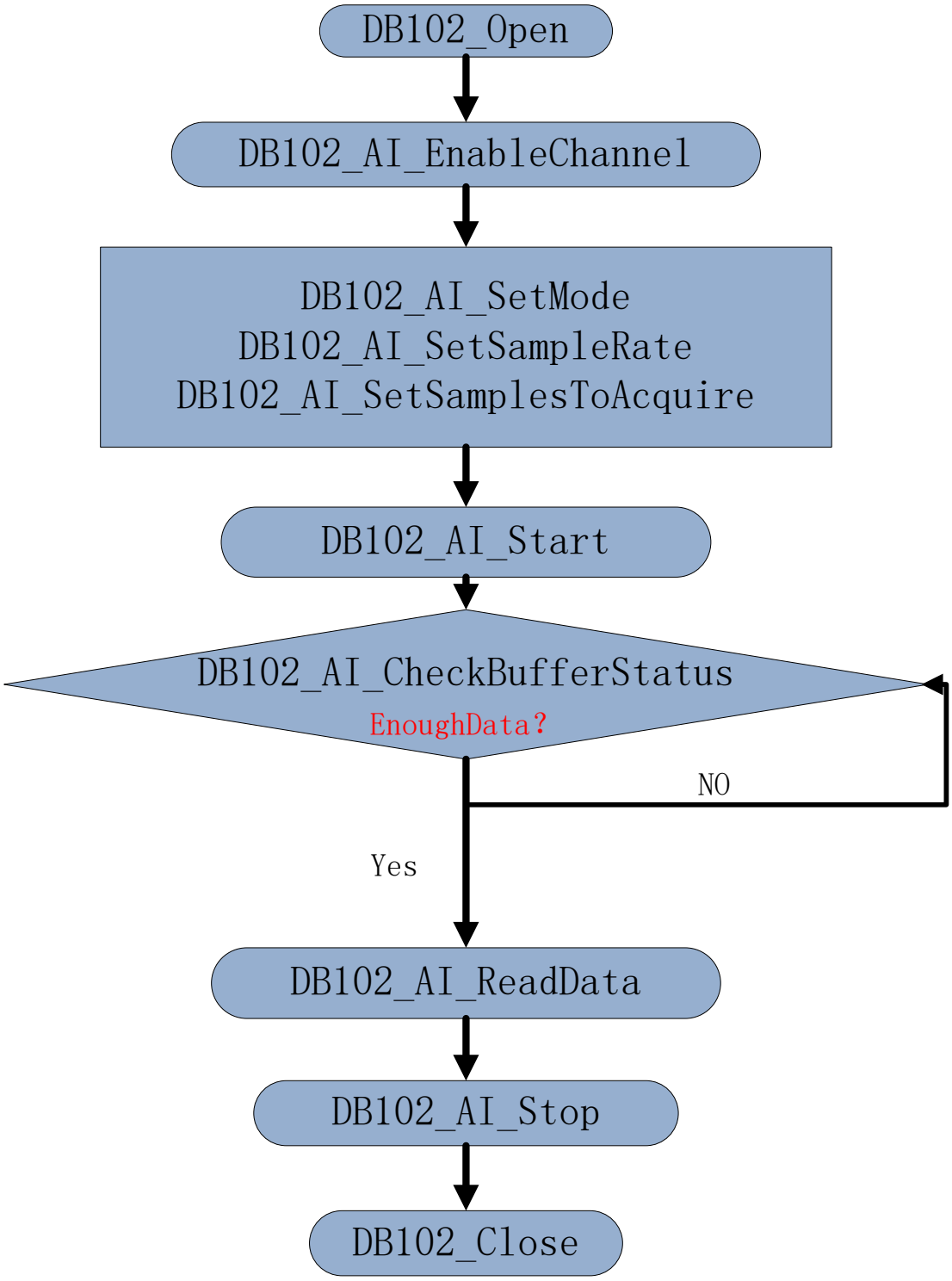


图 5-2 AI 有限采集

5.1.4 范例 2：连续采集

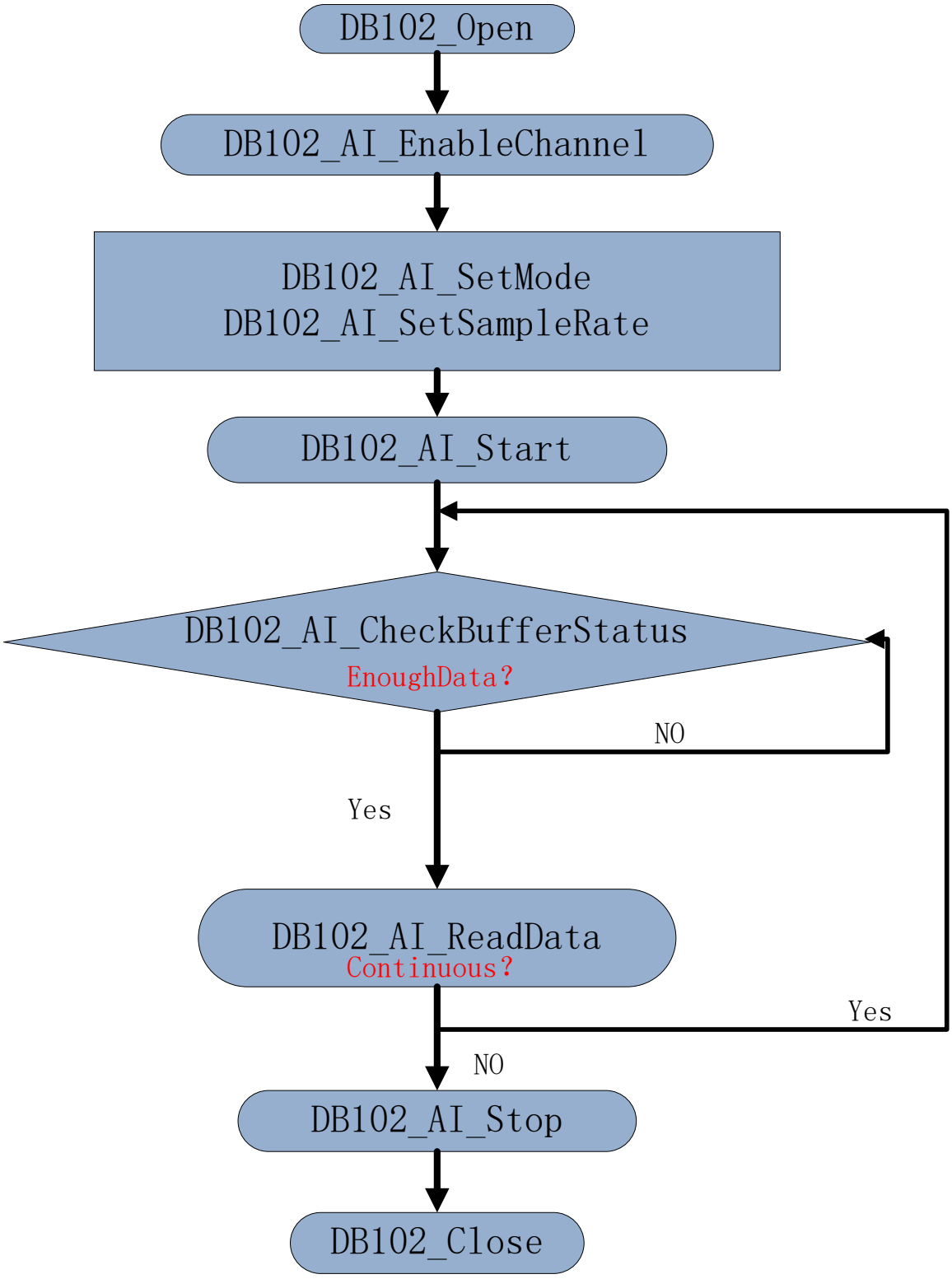


图 5-3 AI 连续采集

5.2 构造模拟输出（AO）

5.2.1 AO C API 详细参数

5.2.1.1 DB102API int DB102_AO_EnableChannel(JY_DeviceHandle hDevice, unsigned int channelCount, unsigned char channels);*

表 5-9 DB102_AO_EnableChannel

函数功能：	使能 AO 通道	
输入参数：	hDevice	板卡资源句柄
	channelCount	要使能的通道个数。
	channels	通道数组，定义要使能的通道号
返回值：	成功返回 0，失败返回错误码	

5.2.1.2 DB102API int DB102_AO_SetMode(JY_DeviceHandle hDevice, DB102_A0_UpdateMode mode);

表 5-10 DB102_AO_SetMode

函数功能：	设置 AO 输出模式	
输入参数：	hDevice	板卡资源句柄
	mode	AO 输出模式（可选 Finite 或者 ContinuousWrapping、ContinuousNoWrapping）
返回值：	成功返回 0，失败返回错误码	

5.2.1.3 DB102API int DB102_AO_SetUpdateRate(JY_DeviceHandle hDevice, double updateRate, double actualUpdateRate);*

表 5-11 DB102_AO_SetUpdateRate

函数功能：	设置 AO 更新速率	
输入参数：	hDevice	板卡资源句柄
	updateRate	需要设置的 AO 更新速率

输出参数	actualUpdateRate	实际设置的 AO 更新速率
返回值：	成功返回 0，失败返回错误码	

5.2.1.4 DB102API DB102_AO_SetSamplesToUpdate(JY_DeviceHandle hDevice, unsigned int samplesToUpdate);

表 5-12 DB102_AO_SetSamplesToUpdate

函数功能：	设置 AO 有限输出点数	
输入参数：	hDevice	板卡资源句柄
	samplesToUpdate	有限采集模式下需要输出的点数
返回值：	成功返回 0，失败返回错误码	

5.2.1.5 DB102API int DB102_AO_Start(JY_DeviceHandle hDevice);

表 5-13 DB102_AO_Start

函数功能：	启动 AO 输出	
输入参数：	hDevice	板卡资源句柄
返回值：	成功返回 0，失败返回错误码	

5.2.1.6 DB102API int DB102_AO_CheckBufferStatus(JY_DeviceHandle hDevice, unsigned long* avaliableSamples, bool* overrun);

表 5-14 DB102_AO_CheckBufferStatus

函数功能：	获取缓冲区状态	
输入参数：	hDevice	板卡资源句柄
输出参数：	avaliableSamples	缓冲区当前还可以写入的点数
	overrun	缓冲区是否溢出
返回值：	成功返回 0，失败返回错误码	

5.2.1.7 DB102API int DB102_AO_WriteData(JY_DeviceHandle hDevice, short* dataBuffer, unsigned int dataLength, int timeOut, unsigned long* actualWriteLength);

表 5-15 DB102_AO_WriteData

函数功能：	读取数据	
输入参数：	hDevice	板卡资源句柄
	dataBuffer	数据数组，用来存放读取到的数据
	dataLength	需要读取的数据长度
	timeOut	超时时间
输出参数：	actualReadLength	实际读取到的数据长度
返回值：	成功返回 0，失败返回错误码	

5.2.1.8 DB102API int DB102_AO_WaitUntilDone(DB102_DeviceHandle hDevice,bool* done,int timeOut);

表 5-16 DB102_AO_WaitUntilDone

函数功能：	AO 有限输出完成状态	
输入参数：	hDevice	板卡资源句柄
	timeOut	超时时间
输出参数	done	有限输出完成状态，true 为完成，false 为未完成
返回值：	成功返回 0，失败返回错误码	

5.2.1.9 DB102API int DB102_AO_Stop(JY_DeviceHandle hDevice);

表 5-17 DB102_AO_Stop

函数功能：	停止 AO 输出	
输入参数：	hDevice	板卡资源句柄
返回值：	成功返回 0，失败返回错误码	

5.2.2 AO 工作流程

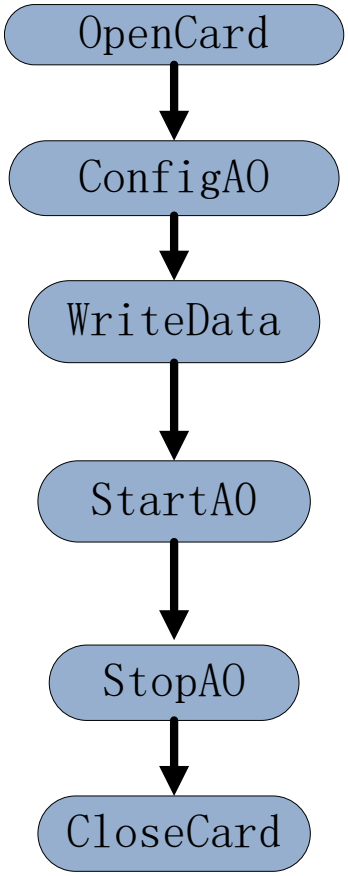


图 5-4 AO 工作流程

对于模拟输出大致流程如上图，

- 首先打开板卡，输入对应的 VID，PID，卡槽号可正常打开板卡，返回板卡句柄。
- 配置 AO：包括配置 AO 输出通道，配置更新率，配置输出模式等：
- 写入数据
- 启动 AO 输出
- 停止 AO 输出
- 关闭板卡

5.2.3 范例 1 有限输出

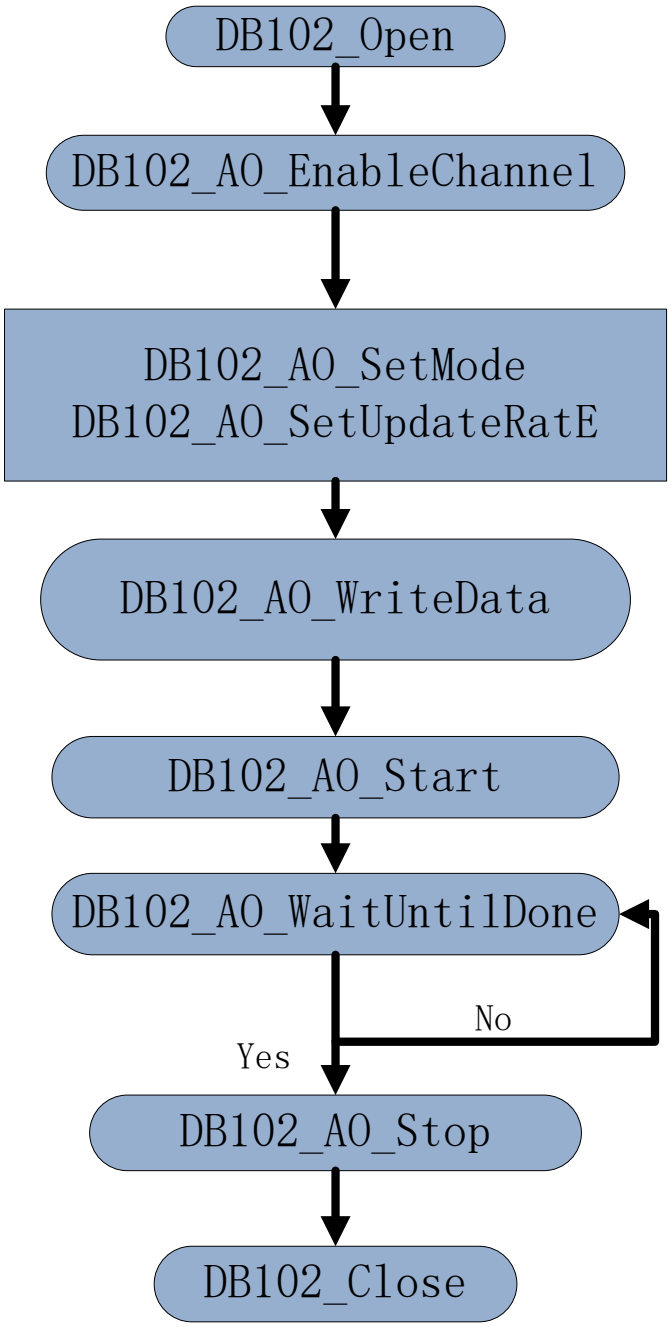


图 5-5 AO 有限输出

5.2.4 范例 2 循环输出

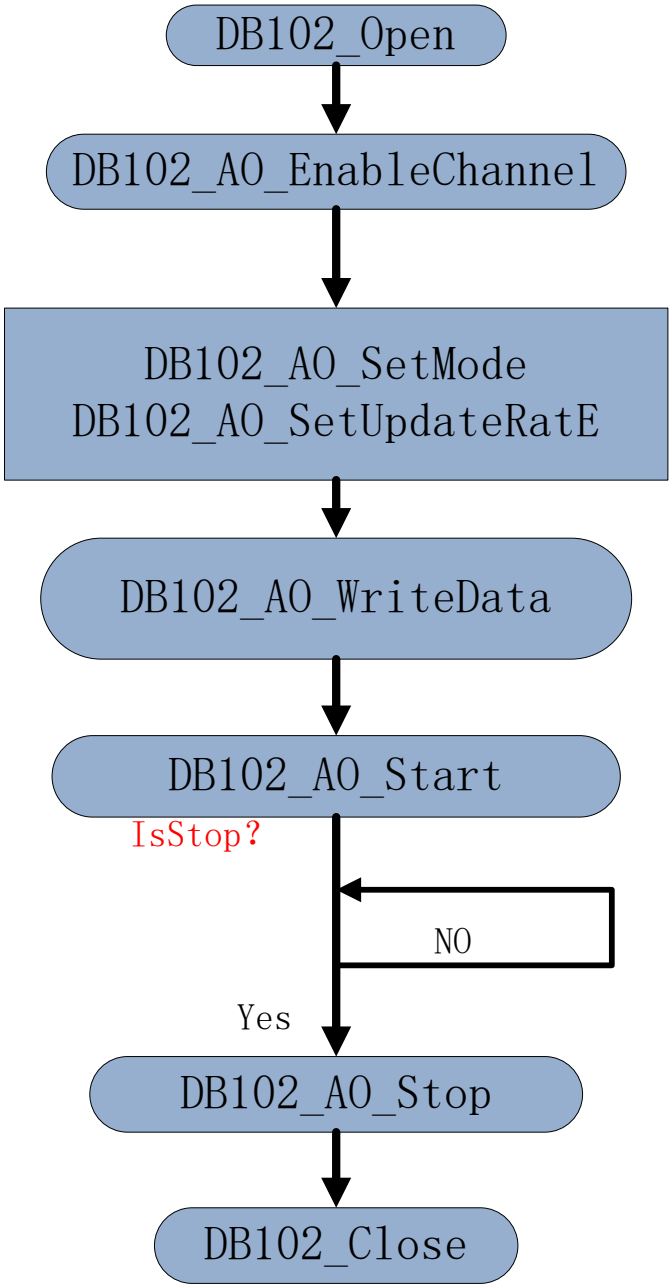


图 5-6 AO 循环输出

5.2.5 范例 3 连续输出

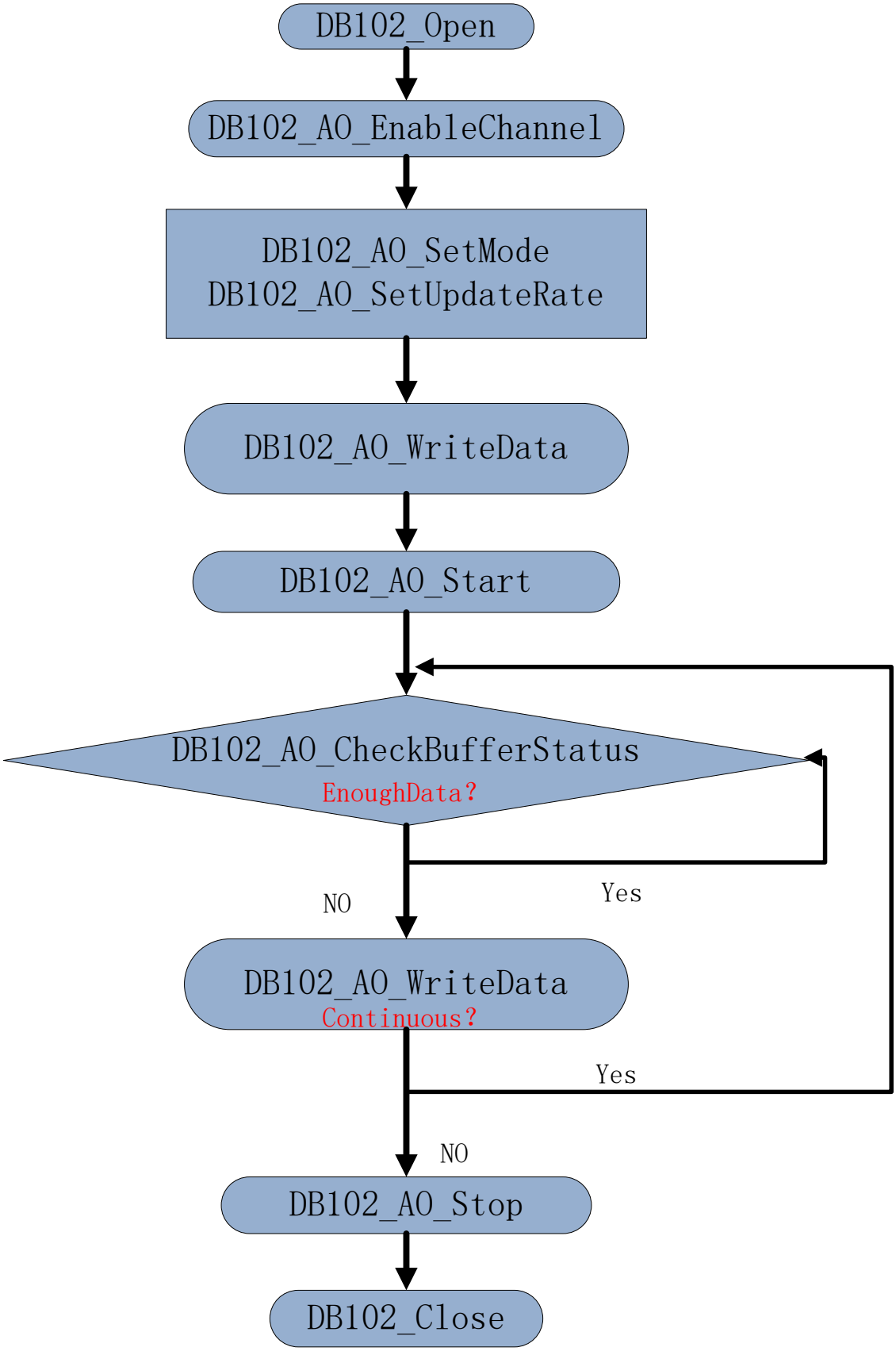


图 5-7 AO 连续输出

5.3 构造数字输入(DI)

5.3.1 DI C API 详细参数

5.3.1.1 DB102API int DB102_DI_AddLines(DB102_DeviceHandle hDevice,unsigned int lines,unsigned int lineCount);*

表 5-18 DB102_DI_AddLines

函数功能：	添加 DI 通道	
输入参数：	hDevice	板卡资源句柄
	lines	要添加的 DI 通道
	lineCount	要添加的 DI 通道长度
返回值：	成功返回 0，失败返回错误码	

5.3.1.2 DB102API int DB102_DI_SetMode(JY_DeviceHandle hDevice, DB102_DI_SampleMode mode);

表 5-19 DB102_DI_SetMode

函数功能：	设置 DI 采集模式	
输入参数：	hDevice	板卡资源句柄
	mode	DI 采集模式（可选 Finite 或者 Continuous）
返回值：	成功返回 0，失败返回错误码	

5.3.1.3 DB102API int DB102_DI_SetSampleRate(JY_DeviceHandle hDevice, double sampleRate, double actualSampleRate);*

表 5-20 DB102_DI_SetSampleRate

函数功能：	设置 DI 采样率	
输入参数：	hDevice	板卡资源句柄
	sampleRate	需要设置的 DI 采样率
输出参数	actualSampleRate	实际设置的 DI 采样率

返回值：	成功返回 0，失败返回错误码

5.3.1.4 DB102API int DB102_DI_SetSamplesToAcquire(JY_DeviceHandle hDevice,unsigned int sampleToAcquire);

表 5-21 DB102_DI_SetSamplesToAcquire

函数功能：	设置 AI 有限采集点数	
输入参数：	hDevice	板卡资源句柄
	sampleToAcquire	有限采集模式下需要采集的点数
返回值：	成功返回 0，失败返回错误码	

5.3.1.5 DB102API int DB102_DI_Start(JY_DeviceHandle hDevice);

表 5-22 DB102_DI_Start

函数功能：	启动 DI 采集	
输入参数：	hDevice	板卡资源句柄
返回值：	成功返回 0，失败返回错误码	

5.3.1.6 DB102API int DB102_DI_CheckBufferStatus(JY_DeviceHandle hDevice, unsigned long* avaliableSamples, bool* overrun);

表 5-23 DB102_DI_CheckBufferStatus

函数功能：	获取缓冲区状态	
输入参数：	hDevice	板卡资源句柄
输出参数：	avaliableSamples	缓冲区当前可读取的点数
	overrun	缓冲区是否溢出
返回值：	成功返回 0，失败返回错误码	

5.3.1.7 DB102API int DB102_DI_ReadData(JY_DeviceHandle hDevice, unsigned char* dataBuffer, unsigned int dataLength, int timeOut, unsigned long* actualReadLength);

表 5-24 DB102_DI_ReadData

函数功能：	读取数据	
输入参数：	hDevice	板卡资源句柄
	dataBuffer	数据数组，用来存放读取到的数据
	dataLength	需要读取的数据长度
	timeOut	超时时间
输出参数：	actualReadLength	实际读取到的数据长度
返回值：	成功返回 0，失败返回错误码	

5.3.1.8 DB102API int DB102_DI_Stop(JY_DeviceHandle hDevice);

表 5-25 DB102_DI_Stop

函数功能：	停止 DI 采集	
输入参数：	hDevice	板卡资源句柄
返回值：	成功返回 0，失败返回错误码	

5.3.2 DI 基本流程

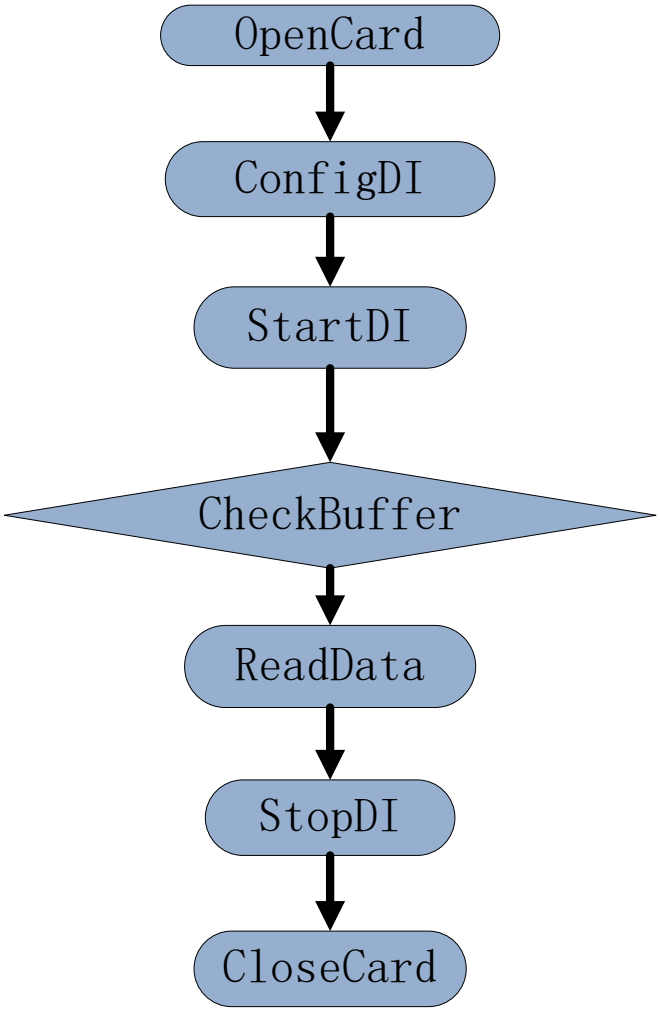


图 5-8 DI 基本流程

对于数字输入大致流程如上图，

- 首先打开板卡，输入对应的 VID，PID，卡槽号可正常打开板卡，返回板卡句柄。
- 配置 DI：包括配置 DI 采样率，配置采集模式等：
- 启动 DI 采集
- 查询缓冲区中的数据量
- 读取数据
- 停止 DI 采集

- 关闭板卡

5.3.3 DI 有限采集

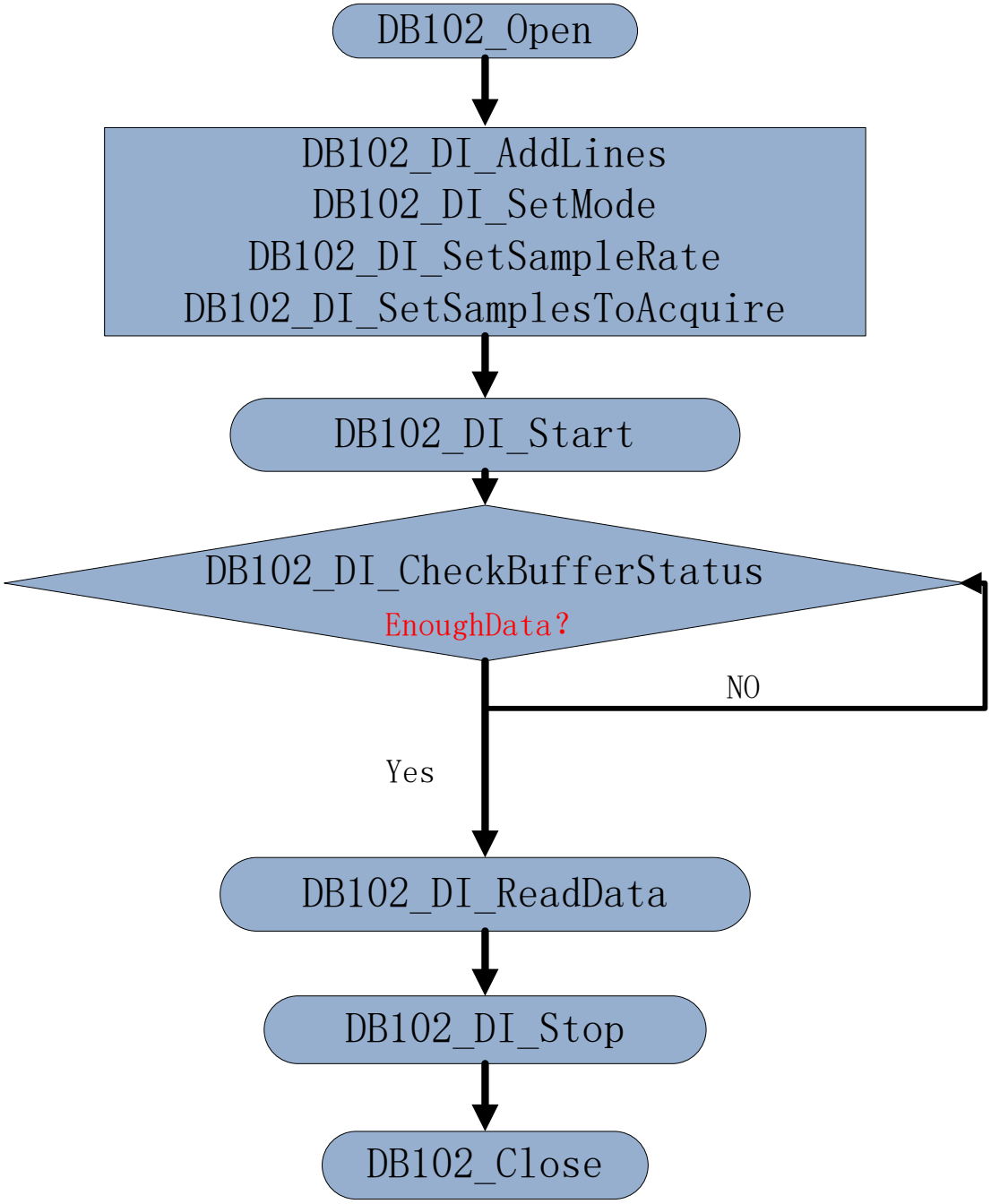


图 5-9 DI 有限采集

5.3.4 DI 连续采集

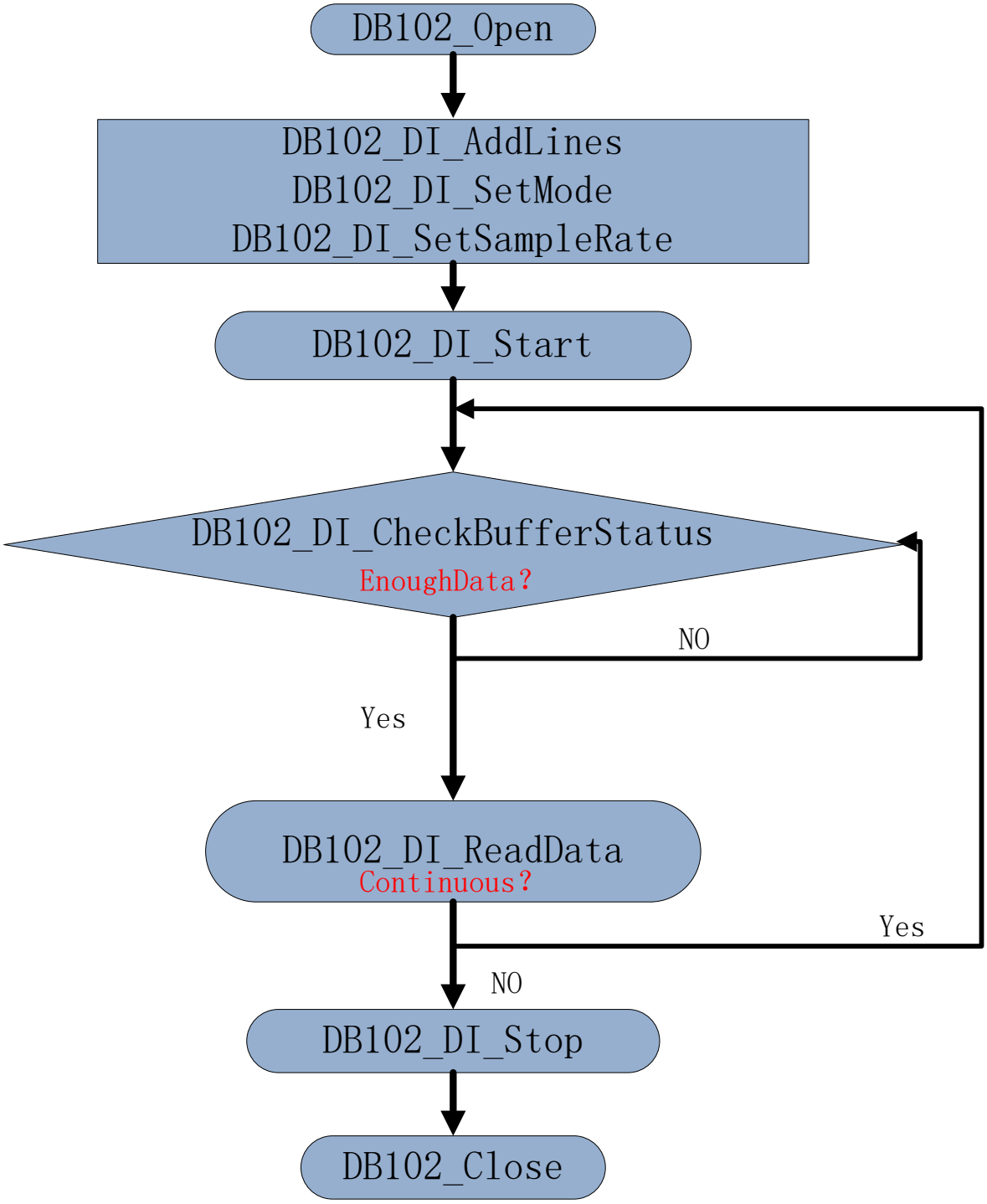


图 5-10 DI 连续采集

5.4 构造数字输出（DO）

5.4.1 DO C API 详细参数

5.4.1.1 DB102API int DB102_DO_AddLines(DB102_DeviceHandle hDevice,unsigned int lines,unsigned int lineCount);*

表 5-26 DB102_DO_AddLines

函数功能：	添加 DO 通道	
输入参数：	hDevice	板卡资源句柄
	lines	要添加的 DO 通道
	lineCount	要添加的 DO 通道长度
返回值：	成功返回 0，失败返回错误码	

5.4.1.2 DB102API int DB102_DO_SetMode(JY_DeviceHandle hDevice, DB102_DO_UpdateMode mode);

表 5-27 DB102_DO_SetMode

函数功能：	设置 DO 输出模式	
输入参数：	hDevice	板卡资源句柄
	mode	DO 输出模式（可选 Finite 或者 ContinuousWrapping、ContinuousNoWrapping）
返回值：	成功返回 0，失败返回错误码	

5.4.1.3 DB102API int DB102_DO_SetUpdateRate(JY_DeviceHandle hDevice, double updateRate, double actualUpdateRate);*

表 5-28 DB102_DO_SetUpdateRate

函数功能：	设置 DO 更新速率	
输入参数：	hDevice	板卡资源句柄
	updateRate	需要设置的 AO 更新速率

输出参数	actualUpdateRate	实际设置的 AO 更新速率
返回值：	成功返回 0，失败返回错误码	

5.4.1.4 DB102API DB102_DO_SetSamplesToUpdate(JY_DeviceHandle hDevice, unsigned int samplesToUpdate);

表 5-29 DB102_DO_SetSamplesToUpdate

函数功能：	设置 DO 有限输出点数	
输入参数：	hDevice	板卡资源句柄
	samplesToUpdate	有限采集模式下需要输出的点数
返回值：	成功返回 0，失败返回错误码	

5.4.1.5 DB102API int DB102_DO_Start(JY_DeviceHandle hDevice);

表 5-30 DB102_DO_Start

函数功能：	启动 DO 输出	
输入参数：	hDevice	板卡资源句柄
返回值：	成功返回 0，失败返回错误码	

5.4.1.6 DB102API int DB102_DO_CheckBufferStatus(JY_DeviceHandle hDevice, unsigned long* avaliableSamples, bool* overrun);

表 5-31 DB102_DO_CheckBufferStatus

函数功能：	获取缓冲区状态	
输入参数：	hDevice	板卡资源句柄
输出参数：	avaliableSamples	缓冲区当前还可以写入的点数
	overrun	缓冲区是否溢出
返回值：	成功返回 0，失败返回错误码	

5.4.1.7 DB102API int DB102_DO_WriteData(JY_DeviceHandle hDevice, unsigned char* dataBuffer, unsigned int dataLength, int timeOut, unsigned long* actualWriteLength);

表 5-32 DB102_DO_WriteData

函数功能：	读取数据	
输入参数：	hDevice	板卡资源句柄
	dataBuffer	数据数组，用来存放读取到的数据
	dataLength	需要读取的数据长度
	timeOut	超时时间
输出参数：	actualReadLength	实际读取到的数据长度
返回值：	成功返回 0，失败返回错误码	

5.4.1.8 DB102API int DB102_DO_WaitUntilDone(DB102_DeviceHandle hDevice,bool* done,int timeOut);

表 5-33 DB102_DO_WaitUntilDone

函数功能：	DO 有限输出完成状态	
输入参数：	hDevice	板卡资源句柄
	timeOut	超时时间
输出参数	done	有限输出完成状态，true 为完成，false 为未完成
返回值：	成功返回 0，失败返回错误码	

5.4.1.9 DB102API int DB102_DO_Stop(JY_DeviceHandle hDevice);

表 5-34 DB102_DO_Stop

函数功能：	停止 DO 输出	
输入参数：	hDevice	板卡资源句柄
返回值：	成功返回 0，失败返回错误码	

5.4.2 DO 基本流程

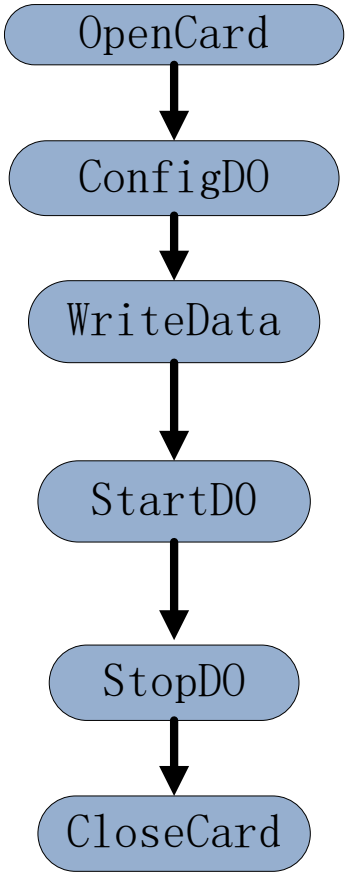


图 5-11 DO 基本流程

对于数字输出大致流程如上图，

- 首先打开板卡，输入对应的 VID，PID，卡槽号可正常打开板卡，返回板卡句柄。
- 配置 DO：包括配置更新率，配置输出模式等：
- 写入数据
- 启动 DO 输出
- 停止 DO 输出
- 关闭板卡

5.4.3 范例 1 有限点输出

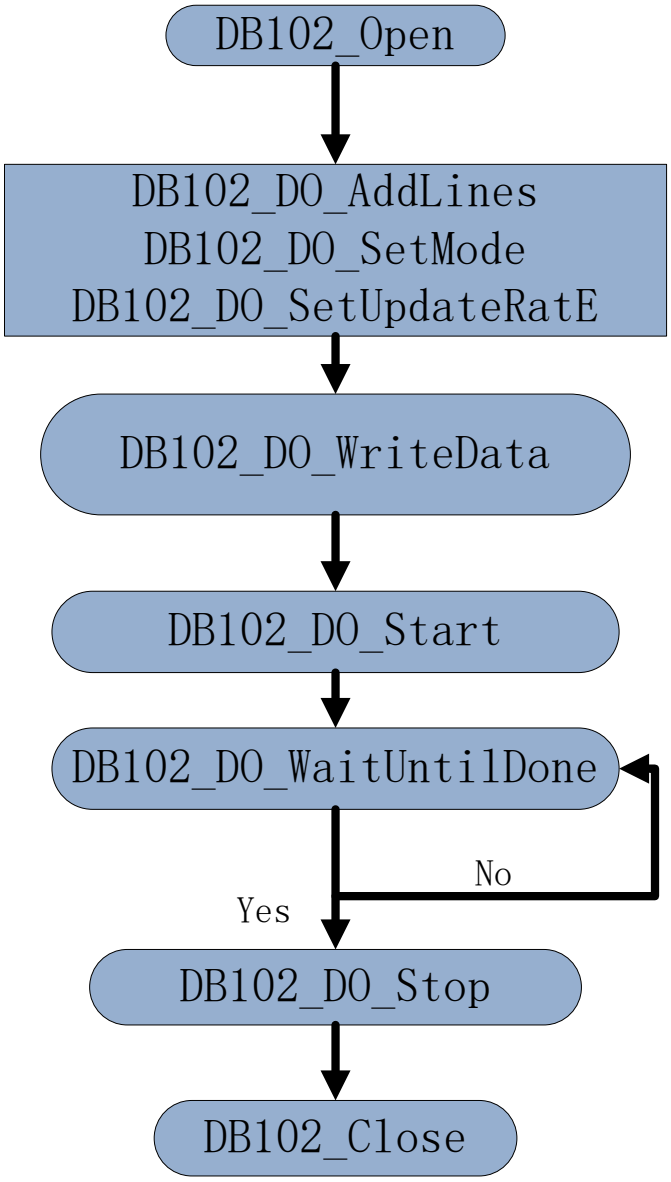


图 5-12 DO 有限输出

5.4.4 范例 2 循环输出

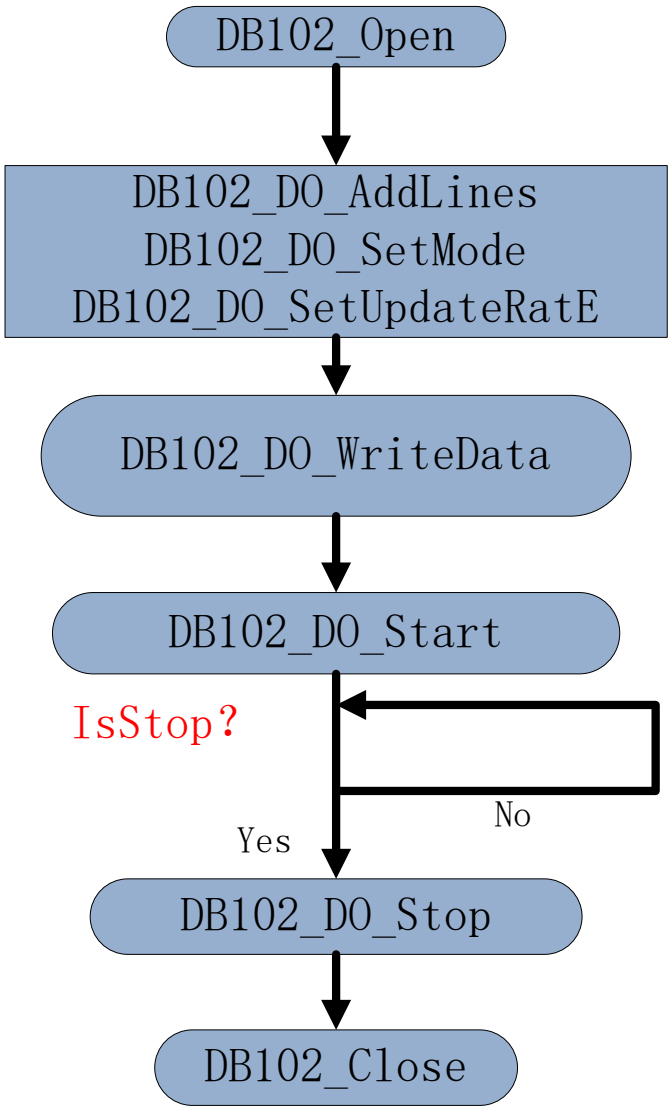


图 5-13 DO 循环输出

5.4.5 范例 3 连续输出

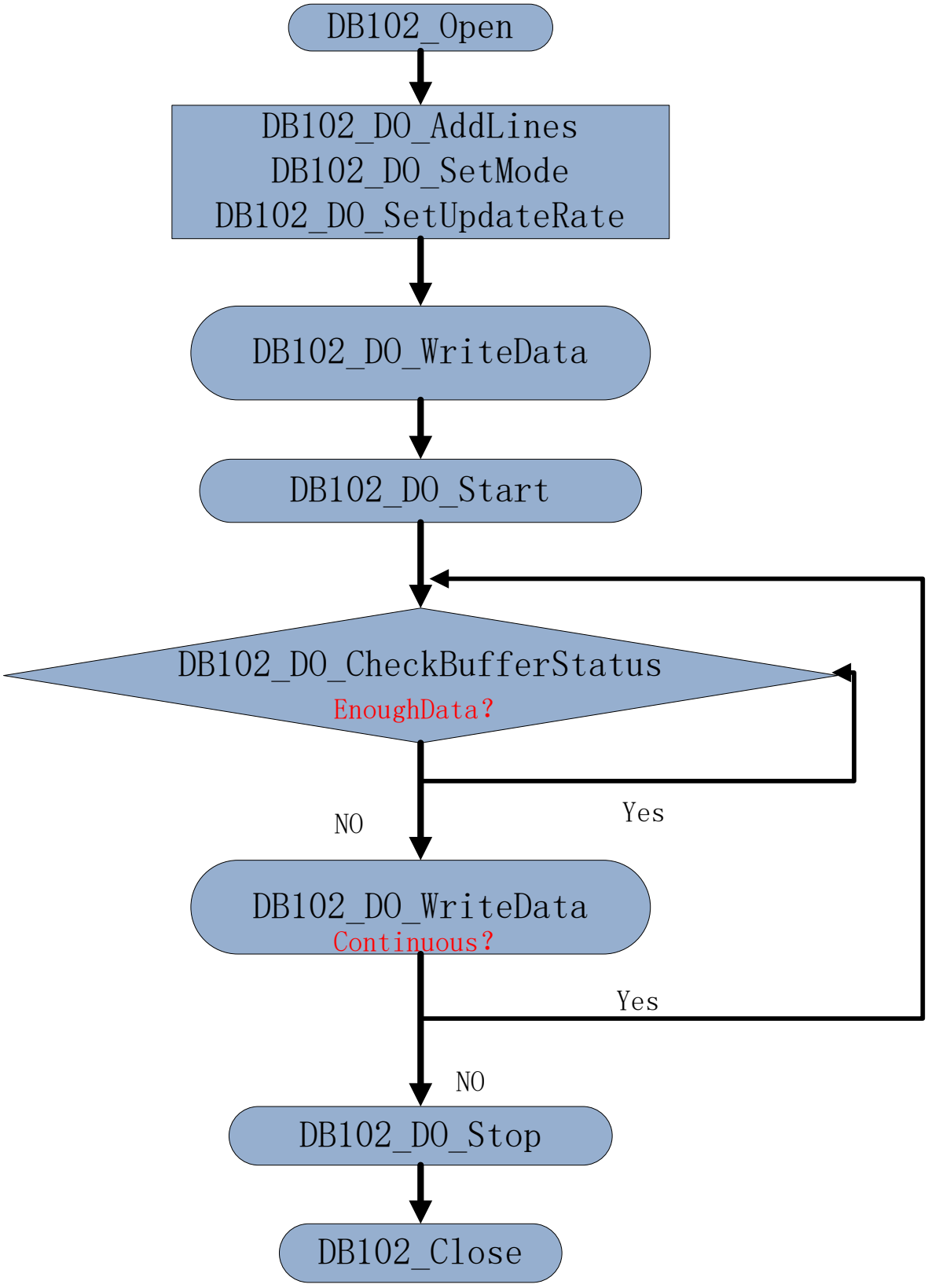


图 5-14 DO 连续输出

5.5 构造计数器入（CI）

5.5.1 CI C API 详细参数

5.5.1.1 DB102API int DB102_CI_EnableChannel(DB102_DeviceHandle hDevice,unsigned int channel);

表 5-35 DB102_CI_EnableChannel

函数功能：	添加 CI 通道	
输入参数：	hDevice	板卡资源句柄
	channel	要添加的 CI 通道
返回值：	成功返回 0，失败返回错误码	

5.5.1.2 DB102API int DB102_CI_SetCounterParament(DB102_DeviceHandle hDevice,unsigned int channel,int initCount,DB102_CountDirection countDirection,DB102_CI_MeasureType measureType);

表 5-36 DB102_CI_SetCounterParament

函数功能：	设置计数器计数参数	
输入参数：	hDevice	板卡资源句柄
	channel	CI 通道
	initCount	计数器的初始值
	countDirection	计数器计数方向
	measureType	计数器测量类型
返回值：	成功返回 0，失败返回错误码	

5.5.1.3 DB102API int DB102_CI_ReadSingleCountValue(DB102_DeviceHandle hDevice, unsigned int channel, int* countValue);

表 5-37 DB102_CI_ReadSingleCountValue

函数功能：	读取计数器计数值，用于边沿计数模式下	
输入参数：	hDevice	板卡资源句柄
	channel	CI 通道
输出参数	countValue	计数器计数值
返回值：	成功返回 0，失败返回错误码	

5.5.1.4 DB102API int DB102_CI_ReadSingleMeasureValue(DB102_DeviceHandle hDevice, unsigned int channel, double* measureValue);

表 5-38 DB102_CI_ReadSingleMeasureValue

函数功能：	读取计数值测量值，用于脉宽测量，周期测量和频率测量模式下	
输入参数：	hDevice	板卡资源句柄
	channel	CI 通道
输出参数	measureValue	计数器测量值
返回值：	成功返回 0，失败返回错误码	

5.5.1.5 DB102API int DB102_CI_Start(DB102_DeviceHandle hDevice, unsigned int channel);

表 5-39 DB102_CI_Start

函数功能：	CI 开始	
输入参数：	hDevice	板卡资源句柄
	channel	CI 通道
返回值：	成功返回 0，失败返回错误码	

5.5.1.6 DB102API int DB102_CI_Stop(DB102_DeviceHandle hDevice, unsigned int channel);

表 5-40 DB102_CI_Stop

函数功能：	CI 停止	
输入参数：	hDevice	板卡资源句柄
	channel	CI 通道
返回值：	成功返回 0，失败返回错误码	

5.5.2 CI 基本流程

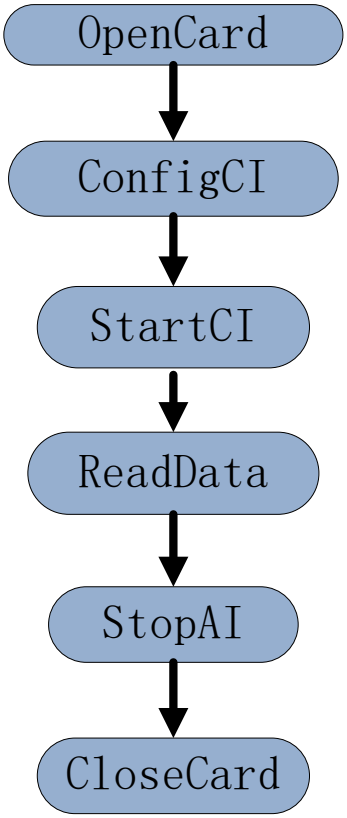


图 5-15 CI 基本流程

5.5.3 CI 计数

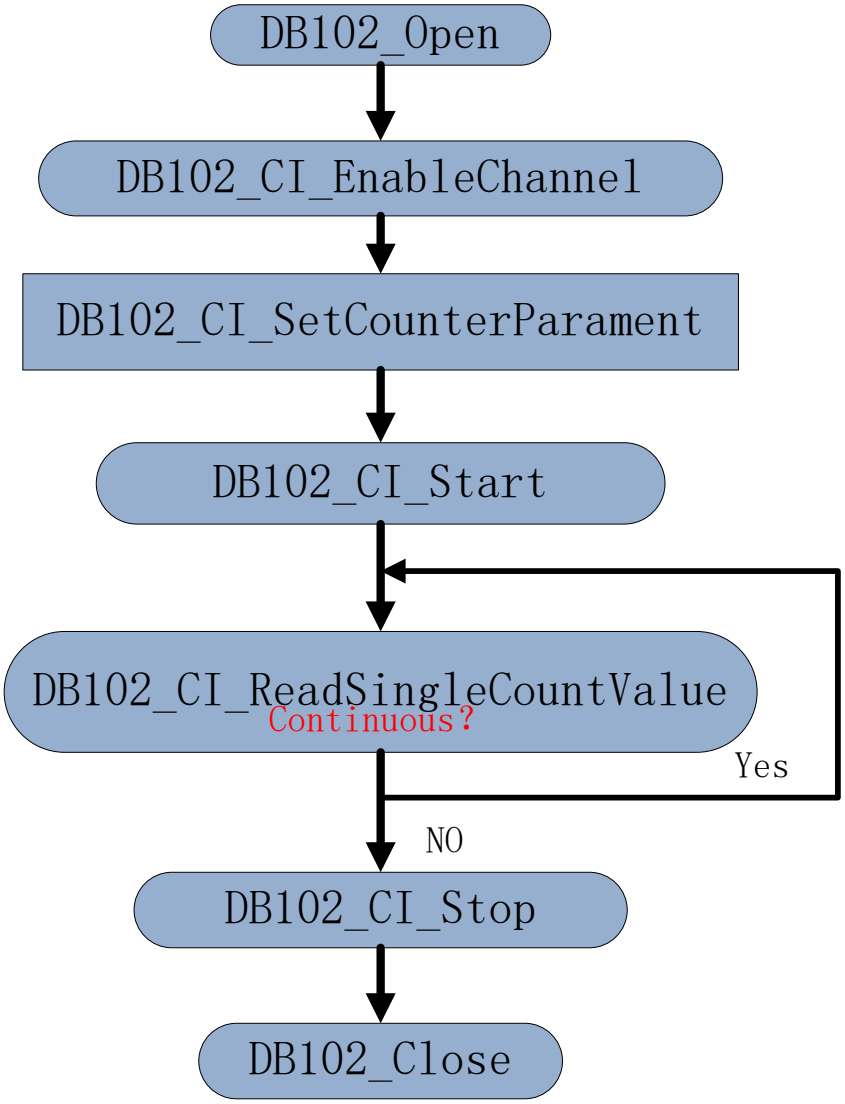


图 5-16 CI 计数

5.5.4 CI 测量

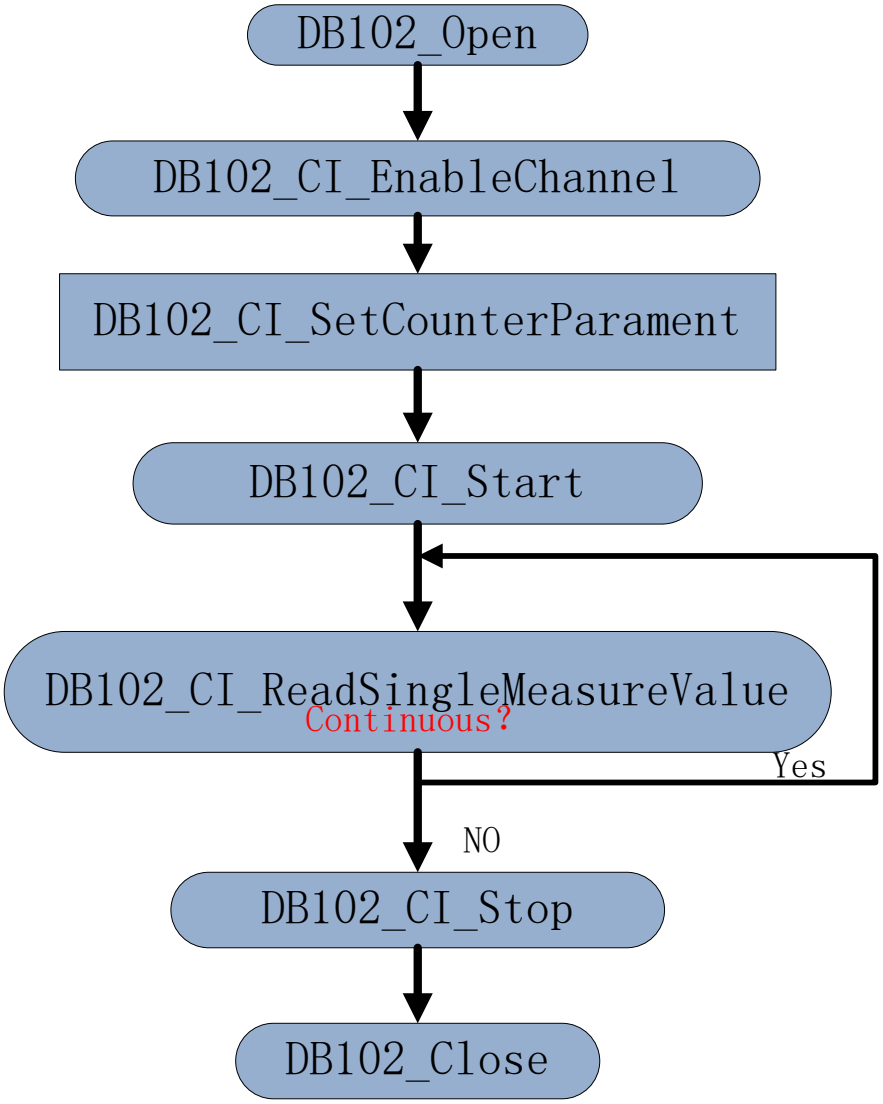


图 5-17 CI 测量

5.6 构造计数器输出（CO）

5.6.1 CO C API 详细参数

5.6.1.1 DB102API int DB102_CO_EnableChannel(DB102_DeviceHandle hDevice, unsigned int channel);

表 5-41 DB102_CO_EnableChannel

函数功能：	添加 CO 通道	
输入参数：	hDevice	板卡资源句柄
	channel	要添加的 CO 通道
返回值：	成功返回 0，失败返回错误码	

5.6.1.2 DB102API int DB102_CO_SetFrequency(DB102_DeviceHandle hDevice, unsigned int channel, double frequency, double duty);

表 5-42 DB102_CO_SetFrequency

函数功能：	设置 CO 频率	
输入参数：	hDevice	板卡资源句柄
	channel	CO 通道
	frequency	CO 输出的信号频率
	duty	CO 输出信号的占空比
返回值：	成功返回 0，失败返回错误码	

5.6.1.3 JYPXIe5510API int JYPXIe5510_CO_Start(JY_DeviceHandle hDevice, unsigned int channel);

表 5-43 JYPXIe5510_CO_Start

函数功能：	CO 开始	
输入参数：	hDevice	板卡资源句柄
	channel	CO 通道

返回值：	成功返回 0，失败返回错误码

5.6.1.4 JYPXIe5510API int JYPXIe5510_CO_Stop(JY_DeviceHandle hDevice, unsigned int channel);

表 5-44 JYPXIe5510_CO_Stop

函数功能：	CO 停止	
输入参数：	hDevice	板卡资源句柄
	channel	CO 通道
返回值：	成功返回 0，失败返回错误码	

5.6.2 CO 基本流程

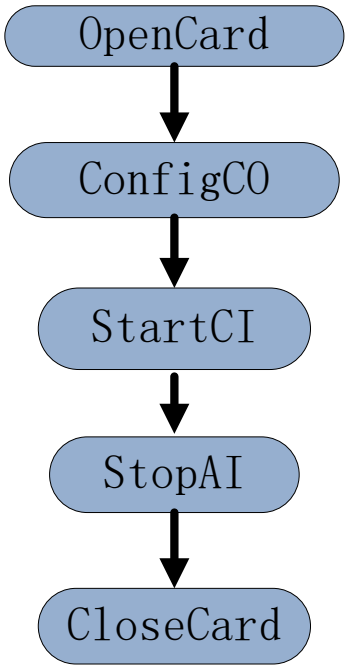


图 5-18 CO 基本流程

5.6.3 CO 输出范例

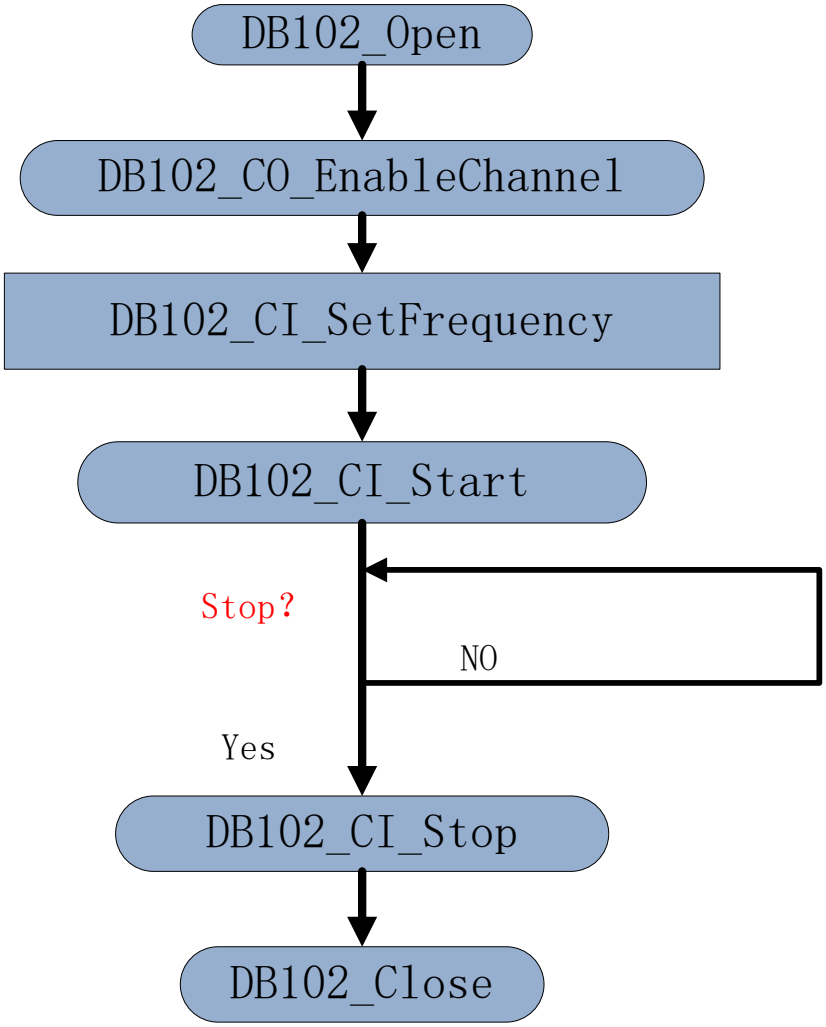


图 5-19 CO 输出范例

6 使用 FirmDrive®编制 C 驱动

6.1 安装 FirmDrive®驱动

FirmDrive®驱动是驱动程序的最底层，安装 FirmDrive®驱动后控制器可以识别装入的 FirmDrive®设备，同时 FirmDrive®驱动提供了两组文件 FirmDriveCore 和 FirmDriveFramework 来操作 FirmDrive®的固件和设备。所以，用户在开始 C 驱动工程开发前应先安装 FirmDrive®驱动。

选择与控制器所安装操作系统相适应的驱动：

a)32 位操作系统使用‘Win32’版本驱动；

b)64 位操作系统使用‘x64’版本驱动；

6.1.1 FirmDriveCore

FirmDriveCore 是 FirmDrive®驱动的最底层，FirmDriveCore 实现了最基本的设备的打开、关闭以及寄存器的读写等操作。

FirmDriveCore 提供的是动态链接库，其中包含了以下三类文件：

- FirmDriveCore.h，此文件是 FirmDriveCore 提供的操作接口的头文件；
- FirmDriveCore.lib，此文件提供了 FirmDriveCore 所有操作接口的入口；
- FirmDriveCore.dll，此文件提供了 FirmDriveCore 所有操作接口的最终实现。

  其中 FirmDriveCore.lib 和 FirmDriveCore.dll 需要区分 X86 和 X64 版本。

6.1.2 FirmDriveFramework

FirmDriveFramework 是 FirmDrive®驱动的框架，FirmDriveFramework 实现了和固件对应的 RxEngine、TxEngine、Routing_Matrix、PFI.h、AnalogTrigger、Counter 等 IP 的软件接口。

FirmDriveFramework 提供的是静态链接库，其中包含了以下两类文件：

- FirmDriveFramework.h，此文件包含了 FirmDriveFramework 中提供的所有的接口定义；
- FirmDriveFramework.lib，此文件包含了 FirmDriveFramework 所有接口的最终实现。

  其中 FirmDriveFramework.lib 需要区分 X86 和 X64 版本。

6.1.3 FirmDrive®驱动的数字签名

FirmDrive®驱动的数字签名是简仪购买的经过微软认证的基于 SHA-2 格式数字签名。在 Windows8 和 Windows 系统下自动支持了此格式的数字签名，但早期版本的 Windows 7 操作系统不支持 SHA-2 格式，安装驱动会出现“Windows 无法验证此设备所需的驱动程序

的数字签名”错误。通过更新操作系统或安装 KB3033929 补丁可以解决该问题。参见简仪科技文档《硬件驱动”数字签名验证不成功“的解决方法》。

6.2 C 驱动开发

安装好了 FirmDrive®驱动之后，用户就可以基于 FirmDrive®驱动来开发相应的 C 驱动。

6.2.1 创建 C 驱动工程

使用 Visual Studio 新建 C/C++工程，步骤如下：

1．在 Visual Studio 新建 C++工程，选择 Win32 项目，填写工程名，文件位置和解决方案名，如下图：

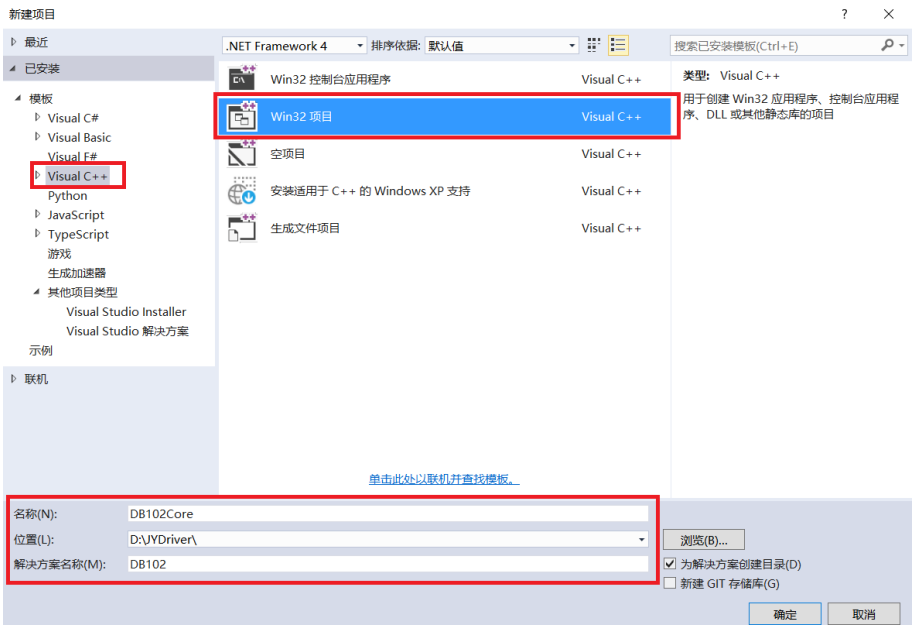


图 6-1 创建 C/C++ 工程

2．在应用程序设置界面，选择应用程序类型为 DLL，点击完成即可创建工程。

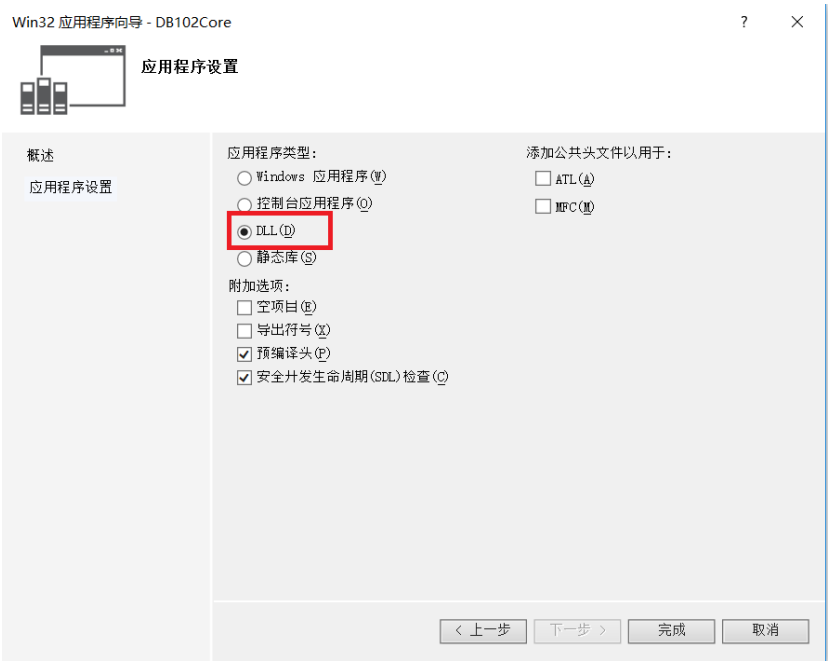


图 6-2 应用程序类型

3．创建好的工程列表如下图：

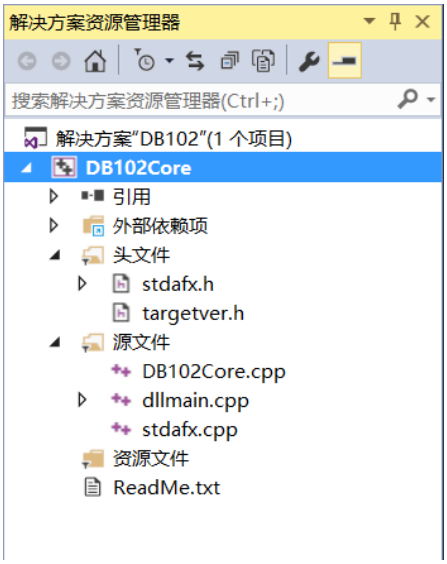


图 6-3 完整工程列表

通过以上三步就可以创建 C++/C 驱动工程。

6.2.2 驱动工程配置

驱动工程创建好之后，需要对工程做必要的配置：

1．在驱动工程中包含需要用到的头文件路径，如下图所示：在附件包含目录中添加需要包含的头文件目录，分别为 FirmDriveCore 中的头文件和 FirmDriveFramework 中的头文件。

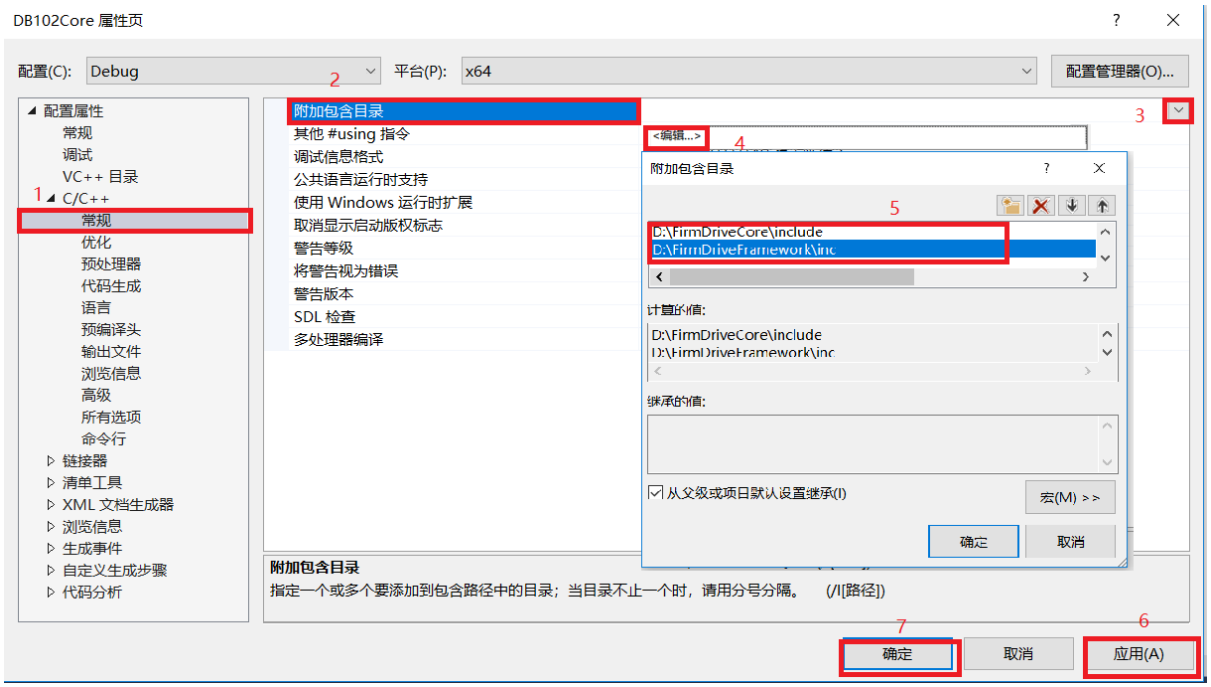


图 6-4 包含头文件目录

2．在工程中包含 FirmDriveCore.lib

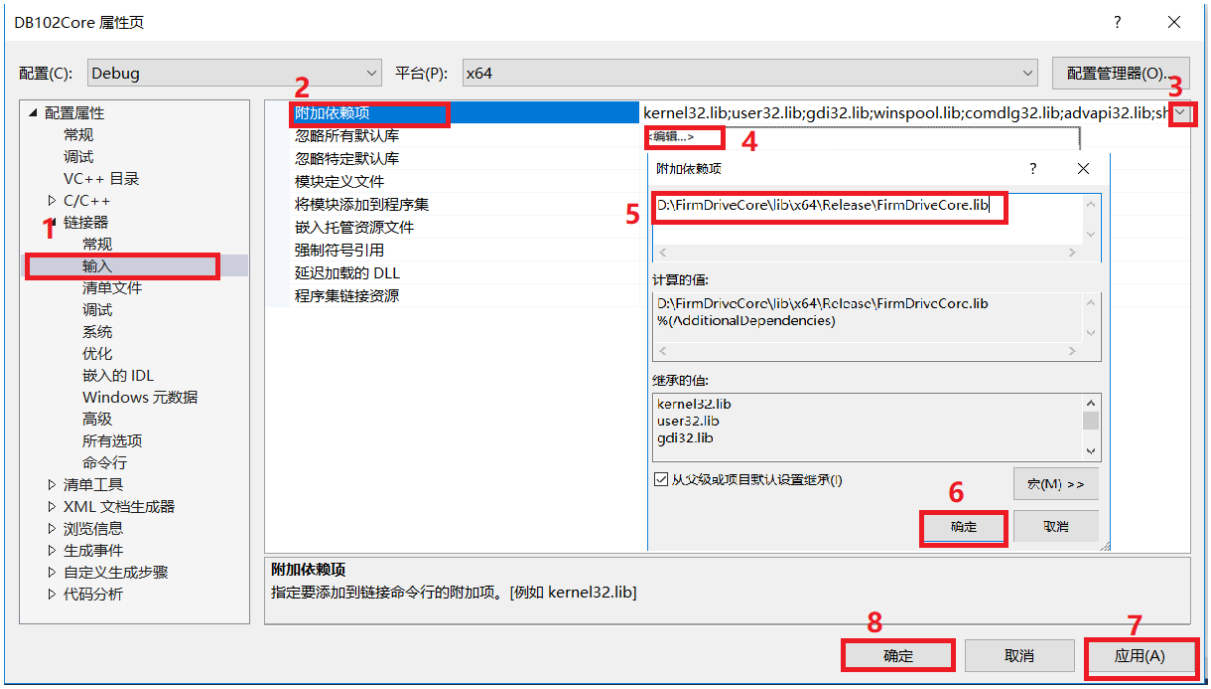


图 6-5 包含 FirmDriveCore.lib

3．在工程中包含 FirmDriveFramework.lib

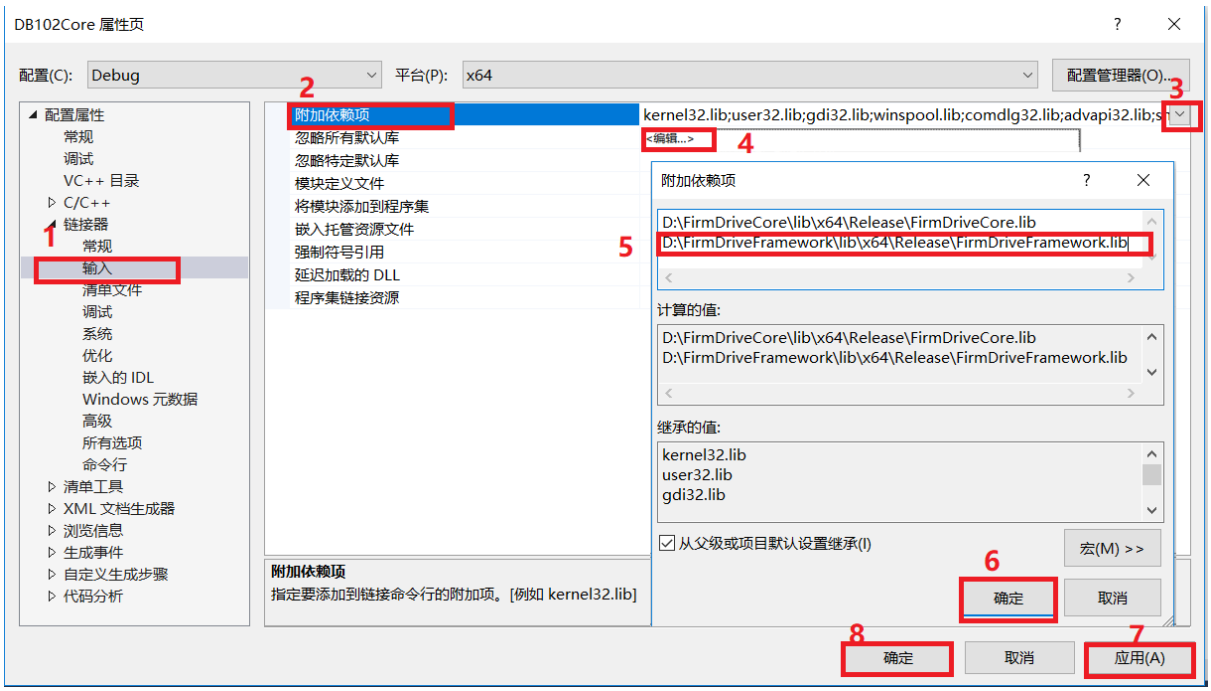


图 6-6 包含 FirmDriveFramework.lib

注意：FirmDriveCore.lib 和 FirmDriveFramework.lib 分为‘x86’、‘x64’两个版本，请根据工程配置使用匹配的库文件。

6.2.3 驱动文件开发

驱动工程配置好之后，接下来就是开发对应的驱动源文件，需要建立的驱动文件如下图：

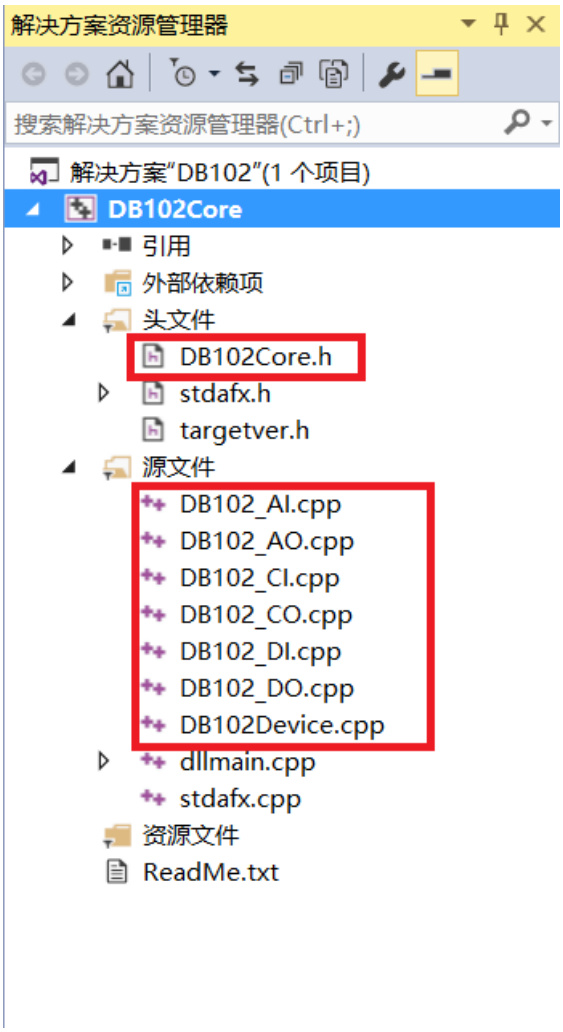

解决方案资源管理器
搜索解决方案资源管理器(Ctrl+;)
解决方案"DB102"(1 个项目)
DB102Core
引用
外部依赖项
头文件
DB102Core.h
stdafx.h
targetver.h
源文件
DB102_AI.cpp
DB102_AO.cpp
DB102_CI.cpp
DB102_CO.cpp
DB102_DI.cpp
DB102_DO.cpp
DB102Device.cpp
dllmain.cpp
stdafx.cpp
资源文件
ReadMe.txt

图 6-7 驱动文件

6.3 生成 C 驱动库

根据第五章定义的六大类功能 API 开发完成后，用户就可以在工程中编译代码生成完成

的 C 驱动接口库。以 DB102 为例，最终生成的 C 驱动接口库包含以下几部分：

1. 头文件：如 DB102Core.h，在头文件中定义了以上六大类的功能 API，以及定义的枚举、常量和错误代码等信息。
2. 动态库：如 DB102Core.dll，需要分 X86 和 X64 两个平台。
3. lib 库：如 DB102Core.lib，需要分 X86 和 X64 两个平台。

如果用户直接用 C 驱动来开发应用程序，则在应用工程中需要包含以上三个部分的文件。

如果用户用 C#来封装面向对象的接口，则需要引用头文件和动态库两部分文件。

注意：在开发 C 驱动时，需要将开发的 C 驱动工程名和头文件名定义为“型号+“Core””，例如，DB102 的工程名为 DB102Core，头文件名为 DB102Core.h。生成的动态库是 DB102Core.dll，lib 库是 DB102Core.lib。

这么做的原因是为了防止和 C#驱动的库名冲突。

7 使用 FirmDrive®编制 C#驱动

数据采集和模块仪器的应用程序可以直接建立在 FirmDrive®引擎接口上来使用硬件。但更好的方式是通过简仪科技定义的 C#驱动软件来调用设计的硬件。这是因为所有简仪科技的硬件都有几乎完全类似的 C#接口，所以编写应用的软件工程师可以完全脱离硬件的具体细节来使用需要的硬件。

简仪科技定义了数据采集和模块仪器的六类 C#接口及其辅助功能接口。我们仅以 DB102 的 AI 作为例子。AI 公共属性和方法的实现及其余 AO, DI,DO 等的实现的请直接参考简仪提供的源程序。

7.1 AI 接口说明

7.1.1 AI 公共属性

表 7-1 AI 公共属性

属性	类型	说明
Mode	enum	AI 的采集模式，支持 Finite 和 Continuous 两种
SampleRate	double	每通道采样率
SamplesToAcquire	int	Finite 模式下每通道采集的点数
Channels	List	用户配置的通道列表，只支持获取
AvaliableSamples	int	当前缓冲区中可读取的每通道数据点数，只支持获取

7.1.2 AI 公共方法

1) AddChannel 方法，此方法有两个参数的重载，描述如下：

表 7-2 AddChannel

void AddChannel (int channelID)		
添加一个 AI 通道		
参数	channelID	通道号

表 7-3 AddChannel

void AddChannel(int[] channelsID)		
添加多个 AI 通道		
参数：	channelsID	通道号数组

2) RemoveChannel 方法

表 7-4 RemoveChannel

void RemoveChannel(int channelID)		
删除一个 AI 通道		
参数：	channelID	通道号

3) Start 方法

表 7-5 Start

void Start()
启动 AI 采集

4) Stop 方法

表 7-6 Stop

void Start()
停止 AI 采集

5) ReadData 方法，此方法有两个重载方法，描述如下：

表 7-7 ReadData

void ReadData(ref double[] buf, int samplesPerChannel, int timeout)		
读取数据，数据为一维数组，数据格式为按通道交织存放		
参数：	buf	存放数据的一维数组
	samplesPerChannel	每通道读取的点数
	timeOut	超时时间

表 7-8 ReadData

void ReadData(ref double[,] buf, int samplesPerChannel, int timeout)		
读取数据，数据为二维数组，数据格式为一列代表一个通道的数据		
参数：	buf	存放数据的二维数组
	samplesPerChannel	每通道读取的点数
	timeOut	超时时间

7.2 C#驱动生成

用户对定义并实现完成好的 C#驱动进行编译，就会生成对应的 C#驱动库。以 DB102 为例，最终用户生成的 C#驱动库包含两部分：

1) C#驱动库：如 DB102.dll，库文件中实现了用户定义的公共属性和公共方法；

7.3 C#接口列表

以下是简仪科技已经定义好的 C#驱动下的接口，用户可以方便的使用这些定义好的接口来实现 C#的驱动。具体接口的说明可以参考 DB102 的 C#驱动来查看。

7.3.1 AI 接口列表

表 7-9 AI 接口列表

AddChannel()	AI	公共方法	添加通道
GetRecordPreviewData()	AI	公共方法	获取预览数据
GetRecordStatus()	AI	公共方法	获取预览状态
ReadData()	AI	公共方法	读取数据
ReadRawData()	AI	公共方法	读取原始数据
ReadSinglePoint()	AI	公共方法	读取单点
RemoveChannel()	AI	公共方法	移除通道
Start()	AI	公共方法	开始
Stop()	AI	公共方法	停止
WaitUntilDone()	AI	公共方法	完成状态
Advanced	AI	公共属性	高级属性
AvailableSamples	AI	公共属性	缓冲区每通道点数
BufLenInSamples	AI	公共属性	缓冲区大小
Channels	AI	公共属性	已添加的通道
ClockEdge	AI	公共属性	时钟边沿
ClockSource	AI	公共属性	时钟源
Mode	AI	公共属性	模式
Record	AI	公共属性	流盘
SampleRate	AI	公共属性	采样率
SamplesToAcquire	AI	公共属性	每通道采样点数
Trigger	AI	公共属性	触发
AIClockEdge	AI	公共枚举	AI 时钟边沿
Falling	AI	公共枚举	下降沿
Rising	AI	公共枚举	上升沿
AIClockSource	AI	公共枚举	AI 时钟源
External	AI	公共枚举	外部
Internal	AI	公共枚举	内部
AIMode	AI	公共枚举	AI 采样模式
Continuous	AI	公共枚举	连续
Finite	AI	公共枚举	有限
Record	AI	公共枚举	流盘
Single	AI	公共枚举	单点
RecordMode	AI	公共枚举	AI 流盘模式

Finite	AI	公共枚举	有限
Infinite	AI	公共枚举	无限
AIAnalogTriggerCondition	AI	公共枚举	AI 模拟触发条件
AboveLevel	AI	公共枚举	高于阈值
BelowLevel	AI	公共枚举	低于阈值
AIAnalogTriggerSource	AI	公共枚举	AI 模拟触发源
CH	AI	公共枚举	触发通道
AIDigitalTriggerSource	AI	公共枚举	AI 数字触发源
AITG	AI	公共枚举	触发源
AIDigitalTriggerEdge	AI	公共枚举	AI 数字触发边沿
Falling	AI	公共枚举	下降沿
Rising	AI	公共枚举	上升沿
AITerminal	AI	公共枚举	AI 接线方式
RSE	AI	公共枚举	参考单端
NRSE	AI	公共枚举	非参考单端
Differential	AI	公共枚举	差分
Pseudodifferential	AI	公共枚举	伪差分
AITriggerGate	AI	公共枚举	AI 门触发
HighActive	AI	公共枚举	高有效
LowActive	AI	公共枚举	低有效
AITriggerMode	AI	公共枚举	AI 触发模式
Start	AI	公共枚举	开始
Reference	AI	公共枚举	参考
Pause	AI	公共枚举	暂停
Coupling	AI	公共枚举	耦合方式
AC	AI	公共枚举	交流
DC	AI	公共枚举	直流
AITriggerType	AI	公共枚举	AI 触发类型
Analog	AI	公共枚举	模拟触发
Digital	AI	公共枚举	数字触发
Immediate	AI	公共枚举	立即触发
FileFormat	AI	公共枚举	文件格式
Bin	AI	公共枚举	二进制文件
AIRecord	AI	公共类	AI 流盘
FileFormat	AI	公共类	文件格式
FilePath	AI	公共类	文件路径
Length	AI	公共类	流盘长度
Mode	AI	公共类	流盘模式
AIAnalogTrigger	AI	公共类	AI 模拟触发
Condition	AI	公共类	触发条件
Level	AI	公共类	触发阈值
Source	AI	公共类	触发源
AIAdvanced	AI	公共类	AI 高级设置

SamplingInterval	AI	公共类	采样间隔
AITrigger	AI	公共类	AI 触发
Analog	AI	公共类	模拟触发
Delay	AI	公共类	延迟
Digital	AI	公共类	数字触犯
ReTriggerCount	AI	公共类	重触发次数
Type	AI	公共类	触发类型
Mode	AI	公共类	触发模式
PreTriggerSamples	AI	公共类	预触发点数
AIChannel	AI	公共类	AI 通道
ChannelID	AI	公共类	通道号
RangeHigh	AI	公共类	量程最大值
RangeLow	AI	公共类	量程最小值
Terminal	AI	公共类	接线方式
Coupling	AI	公共类	耦合方式
EnableIEPE	AI	公共类	IEPE 激励
AIDigitalTrigger	AI	公共类	AI 数字触发
Edge	AI	公共类	触发边沿
Source	AI	公共类	触发源
Gate	AI	公共类	门触发

7.3.2 AO 接口列表

表 7-10 AO 接口列表

AddChannel()	AO	公共方法	添加通道
RemoveChannel()	AO	公共方法	移除通道
Start()	AO	公共方法	开始
Stop()	AO	公共方法	停止
WaitUntilDone()	AO	公共方法	完成状态
WriteData()	AO	公共方法	写入数据
WriteRawData()	AO	公共方法	写入原始数据
WriteSinglePoint()	AO	公共方法	写入单点数据
AvaliableLenInSamples	AO	公共属性	缓冲区每通道点数
BufLenInSamples	AO	公共属性	缓冲区大小
Channels	AO	公共属性	已添加的通道
ClockEdge	AO	公共属性	时钟边沿
ClockSource	AO	公共属性	时钟源
Mode	AO	公共属性	输出模式
SamplesToUpdate	AO	公共属性	有限输出点数
Trigger	AO	公共属性	触发
UpdateRate	AO	公共属性	更新率

WrappingInterval	AO	公共属性	循环输出间隔
AOClockEdge	AO	枚举	AO 时钟边沿
Falling	AO	枚举	下降沿
Rising	AO	枚举	上升沿
AOClockSource	AO	枚举	AO 时钟源
External	AO	枚举	外部时钟
Internal	AO	枚举	内部时钟
AOMode	AO	枚举	AO 输出模式
ContinuousNoWrapping	AO	枚举	连续非循环
ContinuousWrapping	AO	枚举	连续循环
Finite	AO	枚举	有限
Single	AO	枚举	单点
AOTrigger	AO	枚举	AO 触发
Delay	AO	枚举	触发延迟
Digital	AO	枚举	数字触发
ReTriggerCount	AO	枚举	重触发次数
Type	AO	枚举	触发类型
AODigitalTriggerSource	AO	枚举	AO 数字触发源
AOTG	AO	枚举	触发源
AODigitalTriggerEdge	AO	枚举	AO 数字触发边沿
Falling	AO	枚举	下降沿
Rising	AO	枚举	上升沿
AOTriggerType	AO	枚举	AO 触发类型
Digital	AO	枚举	数字触发
Immediate	AO	枚举	立即触发
AOChannel	AO	公共类	AO 通道
ChannelID	AO	公共类	通道号
RangeHigh	AO	公共类	量程最大值
RangeLow	AO	公共类	量程最小值
AODigitalTrigger	AO	公共类	AO 数字触发
Edge	AO	公共类	触发边沿
Source	AO	公共类	触发源

7.3.3 DI 接口列表

表 7-11 DI 接口列表

AddChannel()	DI	公共方法	添加通道
ReadSinglePoint()	DI	公共方法	读取单点值
RemoveChannel()	DI	公共方法	移除通道
Start()	DI	公共方法	开始
Stop()	DI	公共方法	停止
Channels	DI	公共属性	已添加的通道
DIChannel	DI	公共类	DI 通道
BitNum	DI	公共类	位号
PortNum	DI	公共类	端口号
LineNum	DI	公共类	线号

7.3.4 DO 接口列表

表 7-12 DO 接口列表

AddChannel()	DO	公共方法	添加通道
WriteSinglePoint()	DO	公共方法	写入单点值
RemoveChannel	DO	公共方法	移除通道
Start()	DO	公共方法	开始
Stop()	DO	公共方法	停止
Channels	DO	公共属性	已添加的通道
DOChannel	DO		DO 通道
BitNum	DO	公共类	位号
PortNum	DO	公共类	端口号
LineNum	DO	公共类	线号

7.3.5 CI 接口列表

表 7-13 CI 接口列表

ReadCounter()	CI	公共方法	读取计数值
ReadMeasure()	CI	公共方法	读取测量值
Start()	CI	公共方法	开始
Stop()	CI	公共方法	停止
Counter	CI	公共属性	计数模式
Measure	CI	公共属性	测量模式
Mode	CI	公共属性	模式
CIClockEdge	CI	枚举	CI 时钟边沿
Falling	CI	枚举	下降沿

Rising	CI	枚举	上升沿
CIClockSource	CI	枚举	CI 时钟源
External	CI	枚举	外接信号
Internal	CI	枚举	内部信号
CIGateSource	CI	枚举	CI 门源
External	CI	枚举	外接信号
Internal	CI	枚举	内部信号
CIMode	CI	枚举	CI 模式
Counter	CI	枚举	计数模式
Measure	CI	枚举	测量模式
CIPolarity	CI	枚举	CI 有效电平
HighActive	CI	枚举	高有效
LowActive	CI	枚举	低有效
CountDirection	CI	枚举	计数方向
Down	CI	枚举	向下计数
External	CI	枚举	外部控制计数方向
Up	CI	枚举	向上计数
MeasureType	CI	枚举	测量类型
EdgeSeparationMSR	CI	枚举	边沿测量
SinglePeriodMSR	CI	枚举	单周期测量
SinglePulseWidthMSR	CI	枚举	单脉宽测量
Measure	CI	公共类	测量
ClockEdge	CI	公共类	时钟边沿
ClockSource	CI	公共类	时钟源
ExternalClockRate	CI	公共类	外部时钟频率
Type	CI	公共类	测量类型
Counter	CI	公共类	计数
ClockEdge	CI	公共类	时钟边沿
ClockSource	CI	公共类	时钟源
Direction	CI	公共类	计数方向
DirectionPolarity	CI	公共类	方向有效电平
GatePolarity	CI	公共类	门有效电平
GateSource	CI	公共类	门
InitialCount	CI	公共类	计数初始值
CountDirection	CI	公共类	计数方向

7.3.6 CO 接口列表

表 7-14 CO 接口列表

ApplyParam()	CO	公共方法	参数应用
Start()	CO	公共方法	开始
Stop()	CO	公共方法	停止

Clock	CO	公共属性	时钟
Gate	CO	公共属性	门
IdleState	CO	公共属性	空闲状态
Mode	CO	公共属性	输出模式
Pulse	CO	公共属性	脉冲
Frequence	CO	公共属性	频率
SampleToGeneration	CO	公共属性	有限输出脉冲数
COPolarity	CO	枚举	CO 有效电平
HighActive	CO	枚举	高有效
LowActive	CO	枚举	低有效
COClockSource	CO	枚举	CO 时钟源
External	CO	枚举	外部时钟
Internal	CO	枚举	内部时钟
COGateSource	CO	枚举	CO 门源
External	CO	枚举	外部门
Internal	CO	枚举	内部门
COMode	CO	枚举	CO 模式
ContGatedPulseGen	CO	枚举	连续脉冲生成
ContGatedPulseGenPWM	CO	枚举	连续 PWM 生成
MultipleGatedPulseGen	CO	枚举	多个脉冲生成
RetrigSinglePulseGen	CO	枚举	重触发单脉冲生成
SingleGatedPulseGen	CO	枚举	单脉冲生成
SingleTrigContPulseGen	CO	枚举	触发连续脉冲生成
SingleTrigContPulseGenPWM	CO	枚举	触发连续 PWM 生成
SingleTrigPulseGen	CO	枚举	触发单脉冲生成
COPulseType	CO	枚举	CO 脉冲类型
DutyCycleFrequency	CO	枚举	频率占空比方式
HighLowTick	CO	枚举	高低滴答方式
HighLowTime	CO	枚举	高低时间方式
COSignalEdge	CO	枚举	CO 信号有效边沿
Falling	CO	枚举	下降沿
Rising	CO	枚举	上升沿
COSignalLevel	CO	枚举	CO 信号有效电平
High	CO	枚举	高电平
Low	CO	枚举	低电平
COClock	CO	公共类	CO 时钟源
Edge	CO	公共类	时钟边沿
Source	CO	公共类	时钟源
COGate	CO	公共类	CO 门
Polarity	CO	公共类	门有效电平
Source	CO	公共类	门
COPulse	CO	公共类	CO 脉冲
Count	CO	公共类	脉冲数

DutyCycleFrequency	CO	公共类	占空比频率
InitialDelay	CO	公共类	初始延迟
Tick	CO	公共类	滴答
Time	CO	公共类	时间
Type	CO	公共类	输出类型
DutyCycleFrequency	CO	公共类	频率占空比
DutyCycle	CO	公共类	占空比
Frequency	CO	公共类	频率
Tick	CO	公共类	滴答
High	CO	公共类	高滴答数
Low	CO	公共类	低滴答数
Time	CO	公共类	时间
PulseHighTime	CO	公共类	高电平时间
PulseLowTime	CO	公共类	低电平时间

8 FirmDrive®跨平台

8.1 跨平台编程

C 语言本身的一种跨平台的编程语言，但涉及到具体的一些系统函数的使用时，Windows 和 Linux 平台是有所区别的。因此，在编写跨平台的程序时需要注意代码跨平台的一些注意事项，如下：

- 在 Windows 中，正斜杠和反斜杠都可以，但是在 Linux 中，只能是“/”，在 Windows 中，路径大小写无所谓，在 Linux 中严格区分大小写。
- char 类型变量需要明确指定是 signed 或者 unsigned，因为不同平台直接声明 char，会导致 signed 或者 unsigned 的不确定性。
- 在 Windows 中，wchar_t 占两个字节，Linux 中占四个字节，但是在 Linux 可以指定两个字节，这样也会造成一个问题，就是某些第三方库中 wchar_t 可能只指定四个字节的，这样就会导致不兼容。
- 与平台相关的调用尽量用宏隔离开来
- 在 Windows 下某些 C 标准库的头文件不用显式包含，但是在 linux 下需要显式包含。所以在.c 和.cpp 文件中尽量包含这个文件中需要的头文件。
- 尽量只使用 STL 较早出现的函数或类，较早出现的东西相对来说比较稳定，STL 的各个实现基本上都会有实现，这样跨平台的时候可以兼容多个平台。
- 尽量使用标准 C 和 C++的函数以及 STL，使用 C 语言中定义的类型。
- 头文件重复包含的问题，尽量用保卫宏去实现防止头文件的重复包含，很多代码在 Windows 下直接用#pragma once，这不能保证跨平台需要。
- 关于结构体对齐的问题，CPU 为了简化内存和 CPU 之间的处理以及加快 CPU 从内存中取数据的速度，往往都会做一定的对齐，即结构体的各个成员并不是紧凑存储的，往往在成员中间填充一些字节。所以，我们一般不推荐用结构体直接读取和写入数据，这样在不同系统或者计算机之间进行移植时，会出现错误的结果。
- 注意 BOM 的陷阱（字节顺序标记），如果你在 Windows 用记事本创建一个源文件，那么 Windows 会在文件最前面加上一个 BOM 标记，即所谓的字节顺序标记，这样的源码在 Windows 下没问题，但是在 Linux 下就编译不过，所以需要用其他的文本编辑器或者直接在 VS 里面创建源文件。Linux 下 gcc/g++不认带 BOM 标记的源文件。
- 注意调用函数时的形参类型和函数声明中参数列表的类型不匹配。这里特指有无 const 或者是否是引用参数。在 Windows 下的 cl 编译器没问题，linux 下 GCC/G++会报错。

C#语言本身就具有跨平台的特性，用 C#编写的绝大多数程序，都可以实现无缝的跨平台运行。目前，FirmDrive®的 C#驱动就可以直接无缝跨平台的调用。由于 C#驱动会调用 Native 的动态库，因此，FirmDrive®的 C/C++驱动在 Windows 和 Linux 平台使用相同的 API 接口，内部做了跨平台编译的兼容设计。

以下代码片段为跨平台兼容的头文件调用示例：

```
#ifdef WINDOWS_IMPL
#include <SetupAPI.h>
#include <INITGUID.H>
#include <WinIoCtl.h>
#include "crtdbg.h"
#include <strsafe.h>
#else
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <fcntl.h>
#include <unistd.h>
#include <libudev.h>
#include <errno.h>
#endif

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "xdma_public.h"


......
#ifdef WINDOWS_IMPL
    _itoa_s(ID, chID, 2, 10);
#else
    sprintf(chID, "%d", ID);
#endif
    strcat_s(DMAChannelPath, sizeof DMAChannelPath, chID);
    strcat_s(DevDMAPath, sizeof DevDMAPath, DMAChannelPath);
#ifdef WINDOWS_IMPL
    hDMA = CreateFile(DevDMAPath, ReadWriteOption, FILE_SHARE_READ | FILE_SHARE_WRITE,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);//FILE_SHARE_READ |
FILE_SHARE_WRITE ,调用device control时，需要设定FILE_SHARE_READ | FILE_SHARE_WRITE
#else
    pJXIDev->hUser = (HANDLE)open(DevDMAPath, O_RDWR);
#endif
......
```

8.2 FirmDrive®跨平台概述

FirmDrive®从内核驱动到 C#驱动的各个层次如图 7-1 所示。

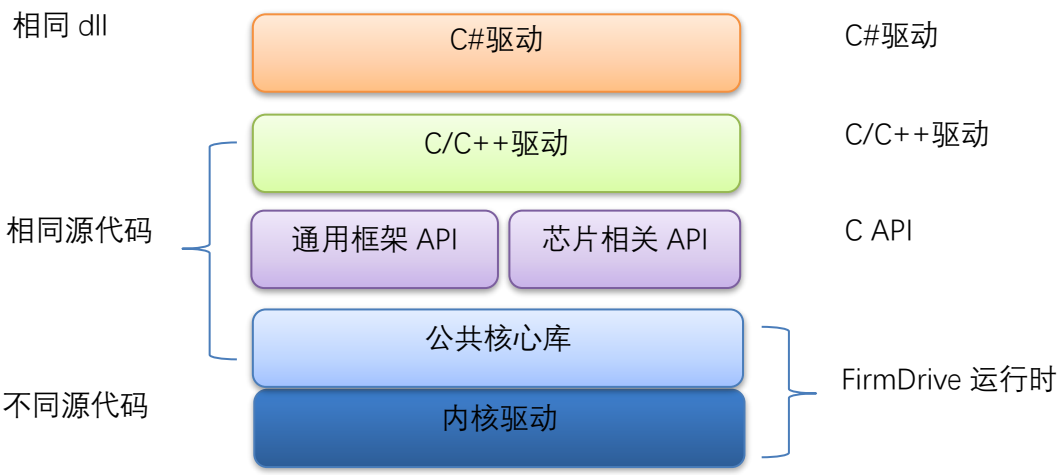


图 8-1 FirmDrive®驱动结构

- 内核驱动以外的各个层次的代码和库都有跨平台兼容的设计，在 Window 和 Linux 平台中都使用相同的接口进行操作。
- 为了代码版本维护的方便性，内核层以外的 C/C++相关层次的代码在 Windows 和 Linux 平台，使用同一份源代码，并在代码中做好跨平台兼容的设计，针对不同平台单独编译生成不同的库。
- 由于内核层的实现与具体的操作系统相关，各平台的内核驱动代码单独维护，无需使用相同的代码。
- C#的驱动不仅源代码相同，甚至可以不用针对不同平台各自生成 DLL，可以在 Windows 和 Linux 中使用任意一个平台编译出来的 DLL。

接下来，将分别对 FirmDrive®各个层次的内容进行跨平台的设计，开发和使用的介绍。

8.3 FirmDrive®跨平台运行时

8.3.1 运行时的组成

FirmDrive®运行时（Runtime）负责硬件的识别，管理以及寄存器和 DMA 的基本操作的接口，主要包括内核驱动（FirmDrive Kernel）和公共核心库（FirmDriveCore）。其中内核驱动针对不同平台有不同的源代码，并生成各自的内核驱动文件（Windows 为 FirmDrive.sys，Linux 为 FirmDrive.ko）。公共核心库（FirmDriveCore）使用相同的

源代码并做好跨平台编译的兼容，在不同平台分别编译生成各自的动态链接库（Windows 为 FirmDriveCore.dll，Linux 为 libFirmDriveCore.so）。

FirmDrive® Runtime 是简仪和聚星联合开发的，以内核驱动文件和动态库的安装包（Windows）或安装脚本（Linux）形式发布，对于使用 FirmDrive®架构的硬件，在 Windows 和 Linux 平台，驱动开发工程师只需要安装使用即可。

8.3.2 运行时的安装

- 在 Windows 平台，FirmDrive® Runtime 提供可执行的安装文件，直接执行安装文件就会自动安装 Runtime 相关的文件到系统中。
- 在 Linux 平台，FirmDrive®提供自动安装脚本文件，头文件和库文件，如下所示：

```
include/
install.sh
lib/
```

进入到 FirmDriveRuntime 目录，执行 sudo ./install.sh 命令，就会自动安装相关的文件到系统相应的目录中。

8.3.3 运行时的使用

FirmDrive®的内核驱动是在公共核心库使用的内核接口，开发工程师无需调用。公共核心库的具体接口说明见 2.2 节中的具体描述。在不同平台，使用公共核心库的具体描述如下：

- Windows 平台使用公共核心库

公共核心库是以动态链接库的形式提供，执行安装文件之后，动态链接库文件就自动安装到了系统目录和运行时的安装目录中。在要调用公共核心库的程序中，需要在 VisualStudio 的项目属性中进行相关的设置。

首先，设置 include 头文件的目录，在 VisualStudio 的项目上右键->属性->C/C++->附加包含目录，添加公共核心库安装目录下的 include 目录。然后，设置库目录，在 VisualStudio 的项目上右键->属性->链接器->常规->附加库目录，添加公共核心库（FirmDriveCore）安装目录下的 lib\$(PlatformShortName)\$(Configuration)目录，并在输入中添加附加依赖项 FirmDriveCore.lib。通过以上两步的设置，在代码中添加引用头文件的代码#include **“FirmDrive.h”**就可以成功引用 FirmDrive®公共核心库的所有 API 了。

- Linux 平台使用公共核心库

在 Linux 中执行 FirmDrive®运行时的安装脚本之后，libFirmDriveCore.so 就拷贝到了系统库目录中。在需要调用公共核心库的程序中需要在 Makefile 中添加编译选项“-

lFirmDriveCore”，在源文件中添加头文件引用#include “FirmDriveCore.h”就可以成功使用 FirmDriveCore 公共核心库了。

8.4 FirmDrive®跨平台 C API

8.4.1 C API 组成

图 7.1 中所示的 C API 包括通用框架 IP 和硬件相关 IP 的 API 实现。其中，通用框架 IP 的 API 是以静态库的形式提供，如下所示：

```
FirmDriveFramework
--include
--lib
```

FirmDriveFramework 中的 IP 是 FirmDrive®中比较通用的 IP，对应的 C API 是以静态库的形式，提供给驱动开发工程师，针对不同平台分别生成对应的静态库文件（Windows 平台为 FirmDriveFramework.lib，Linux 平台为 libFirmDriveFramework.a）

硬件相关 IP，由于是与具体硬件相关，这部分 C API 是以源代码或静态库的形式集成在 C/C++驱动的工程中，在开发对应 C API 时需要注意跨平台兼容的相关细节。

8.4.2 硬件相关 C API

每个硬件都有自己专有的 IP，这些 IP 需要由使用 FirmDrive®架构开发开发硬件的工程师结合自己实际硬件进行开发和设计，并且对应于这些 IP 需要单独开发控制的 C API 接口。这些 C API，是由开发 IP 的工程师直接开发，以源代码或静态库的形式直接包含在 C/C++的驱动项目中。在开发这些 API 接口时需要考虑到 7.1 节所描述的跨平台的一些注意事项。此外，只要是需要对 FPGA 中的 IP 模块进行配置等操作，就需要使用到 FirmDriveCore 的 API。参照 7.3.3 节的使用描述，对于 Windows 和 Linux 平台分别调用即可。

8.4.3 通用框架 C API

通用框架是 FirmDrive®提供的一些非常通用的 IP，对应 C API 是以静态库的形式提供，接下来将对通用框架 IP 的 API 的使用进行说明。

- Windows 平台使用 FirmDriveFramework C API

先将 FirmDriveFramework 整个目录拷贝至工程目录中，然后，设置 include 文件的目录：在 VisualStudio 项目属性->C/C++->附加包含目录，添加$(ProjectDir)FirmDriveFramwork\include。然后设置附加库目录，在 VisualStudio 的项目属性->链接器->常规->附加库目录，添加$(ProjectDir)\FirmDriveFramework\lib_win

\$(PlatformShortName)\$(Configuration)，并在输入中添加附加依赖项FirmDriveFramework.lib。最后，在需要使用的源文件中添加对需要引用的 IP 对应 C API 的头文件的引用即可，如#include “TxEngine.h”。

- Linux 平台使用 FirmDriveFramework C API

先将 FirmDriveFrame 拷贝到当前项目的目录中，然后在 Makefile 中添加编译选项-L FirmDriveFramework/lib_linux，-lFirmDriveFramework，-I FirmDriveFramework /include，在源文件中添加各个模块的头文件引用，就可以成功使用 FirmDriveFramework 通用框架的 API 了。

8.5 FirmDrive® 跨平台 C/C++驱动

8.5.1 C/C++驱动的组成

FirmDrive®的 C/C++驱动是以动态链接库的形式提供给 C#驱动直接调用的一组 API，包含了对硬件的所有功能进行配置和控制的所有 API 接口。

8.5.2 C/C++驱动的开发

C/C++驱动需要调用通用框架的 API 和硬件相关的 API 对硬件的各个功能模块进行配置和操作。硬件相关的 API 是由使用 FirmDrive®进行硬件开发的工程师开发，以源代码或静态库的形式引用到项目中，通用框架的 API 则是由 FirmDrive®以静态库的形式提供的，参照 7.4.3 节中通用框架 API 的使用方法，对通用框架的 IP 进行配置和操作。

通用框架 C API 及核心库本身具有跨平台的特性，因此，在 C/C++驱动也需要遵循 7.1 节中所描述跨平台编程的一些注意事项，编写可以跨平台的源代码。

最后 C/C++驱动在 Windows 平台发布的是 xxx.dll，在 Linux 平台发布的是 libxxx.so，其中 xxx 表示具体的板卡型号。

8.6 FirmDrive® 跨平台 C#驱动

C#语言本身是跨平台的编程语言，FirmDrive®的 C#驱动的设计是完全能够跨平台直接使用的。在 Windows 中 FirmDrive®的 C#驱动依赖于.NET Framework 4.0，在 Linux 中依赖于 mono。

由于 Linux 和 Windows 的 C/C++动态库名称有一定的差异，在 Linux 的底层驱动库（通常是由 C/C++所写）的扩展名是“.so”，这与 Windows 中的扩展名“.dll”不一样。示例如下：

- Windows 中调用底层驱动：

```
[DllImport("FirmDriveDAQ.dll")]
public static extern double InitDev(int boardNum);
```

- Linux 中调用底层驱动：

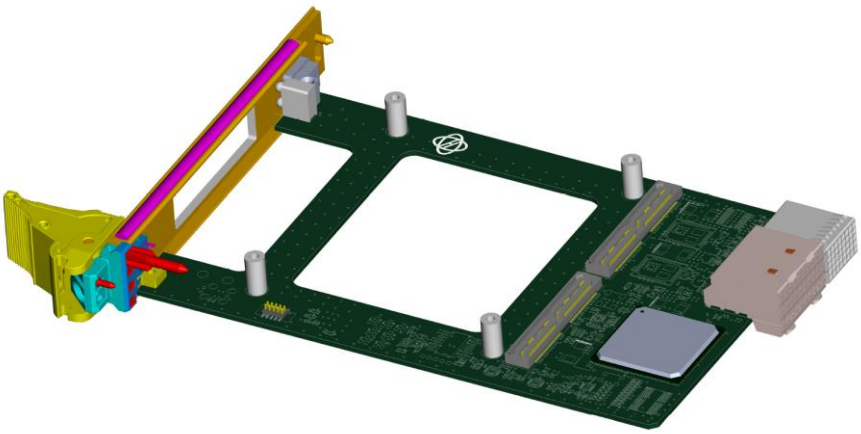
```
[DllImport("libFirmDriveDAQ.so")]
public static extern double InitDev(int boardNum);
```

在我们执行 Dllimport 导入底层驱动的 API 时需要指定底层驱动库的文件名常量。扩展名或库文件名不一样会导致在 Windows 中设定的底层驱动库名，在 Linux 中找不到的情况。虽然可以通过在 Dllimport 时写两套代码，运行时通过判断当前程序运行的平台来决定调用哪一套代码，但这样增加了额外的开发工作量，而且会在不同平台用不同的动态库文件。Mono 专门针对这种情况提供了解决的办法：通过在 mono 配置文件 config 中添加 DllMaps 来映射.dll 和.so 文件。DllMaps 可以通过在 mono/config 文件中新增映射的配置信息来实现 dll 到 so 的映射，也可以直接在应用程序或库文件的配置文件（exe.config）中新增映射的配置信息。示例如下：

```
<configuration>
  <dllmap dll="FirmDriveDAQ.dll" target="libFirmDriveDAQ.so"/>
</configuration>
```

以上配置实现从引用 FirmDriveDAQ.dll 到 libFirmDriveDAQ.so 的映射，这样在 Windows 平台写的代码在 Linux 中就不会因为库名或扩展名不同而无法找到了。这样，只要保证 Windows 和 Linux 中的 C/C++驱动的 API 接口完全一致，就可以无缝在 Linux 平台直接执行 Windows 中开发的 C# 驱动。

9 简仪 PXIe-1010 总线控制模块



9.1 技术特点

传统的 PXIe 硬件开发，需要硬件工程师充分了解 PXIe 总线的技术规范，将自己需要的电路功能与背板的 PCIe 总线、PXIe 同步触发总线连接。简仪 PXIe-1010 总线控制模块，总结了数据采集板卡的共同需求，在母板上预先设计了 FPGA、电源、DRAM、时钟等硬件模块，完成了与 PXIe 总线的互联，并通过两个高密度连接器提供与子板的接口。高密度连接器中提供了电源、时钟、直连到 FPGA 的高速 IO 接口。硬件工程师只需关注自己的硬件功能，将自己的硬件模块与连接器中的信号互联，即可获得高速的 PCIe 总线的数据传输及控制、PXIe 同步总线、以及大容量的 DRAM 存储等能力。这大大减轻了硬件工程师的工作。

简仪 PXIe-1010 在硬件上提供了以下特性：

- 背板接口
  - PCIe Gen2 x8 高速数据传输接口
  - PXIe 时钟信号: PXIe_CLK100, PXIe_SYNC100
  - PXIe 同步/时钟信号： PXIe_DSTAR_A/B/C
  - PXI 同步信号：PXI_TRIG[0:7]
- 512MB DDR3 大容量数据缓存
- 板载 Xilinx Kintex-7 XC7K160T FPGA，具有丰富的硬件乘法器和 RAM 资源，方便用户自定义复杂的实时信号处理算法
- 高速板间连接器
  - 36 路单端(或 18 对差分) x3 组高速 DIO
  - 电源：12V、3.3V
  - JTAG
  - I2C
  - 子母板间同步时钟
- 工作温度范围 0~55C°

9.2 子板设计

PXIe 1010 总线控制模块遵循 PXIe 3U 规范设计。其物理尺寸数据见图 9-1。

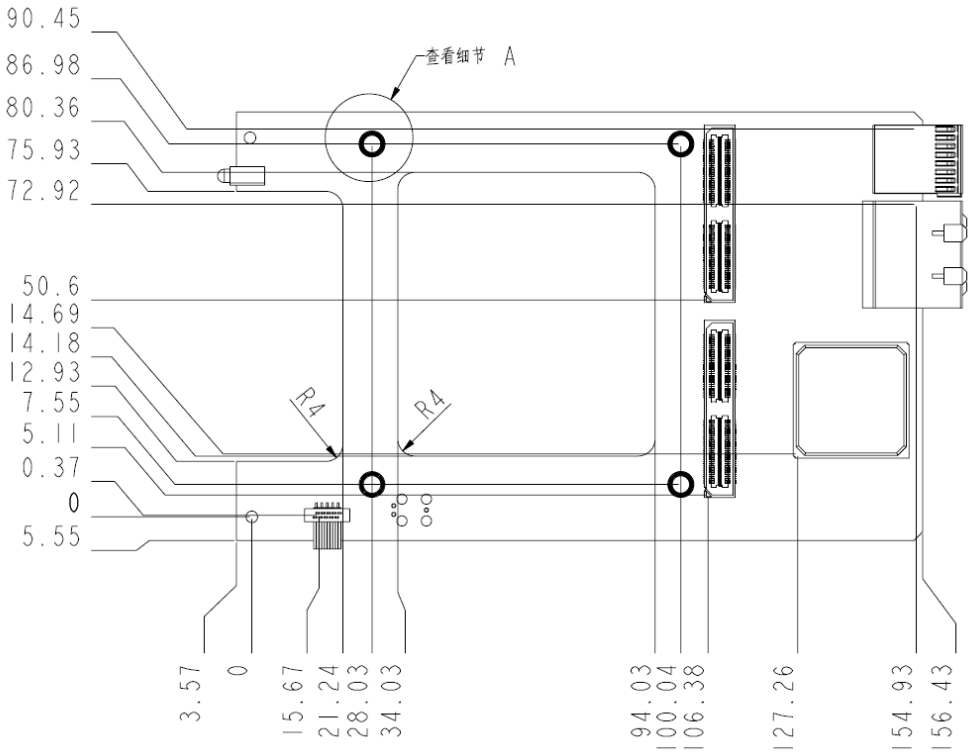


图 9-1 PXIe 1010 总线控制模块机械尺寸

子板可以通过堆叠的方式叠加在 PXIe-1010 总线控制模块上，如

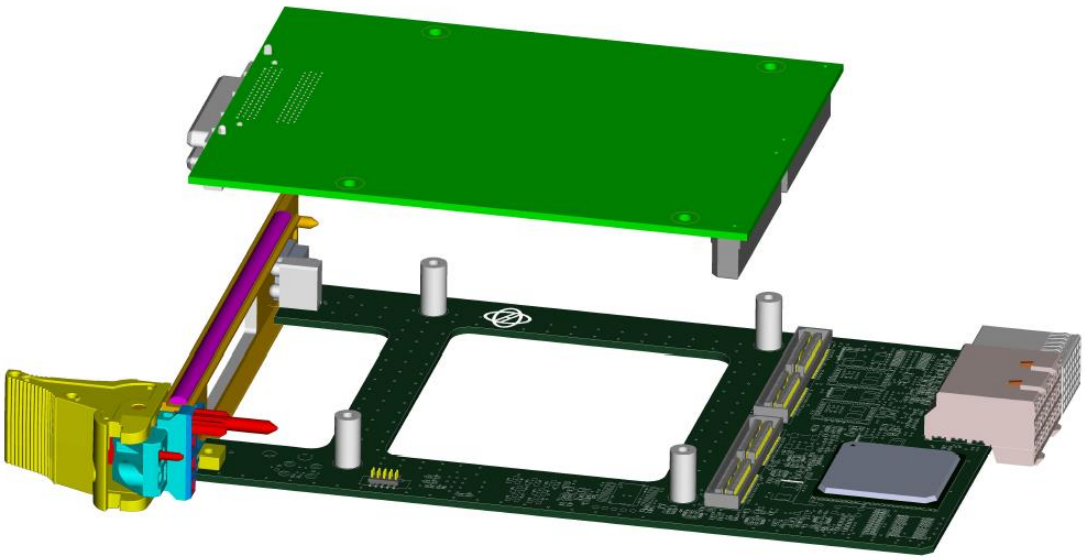


图 9-2 PXIe-1010 总线控制模块与子板的连接方式

使用者在设计子板时可参考图 9-3 的尺寸进行设计。

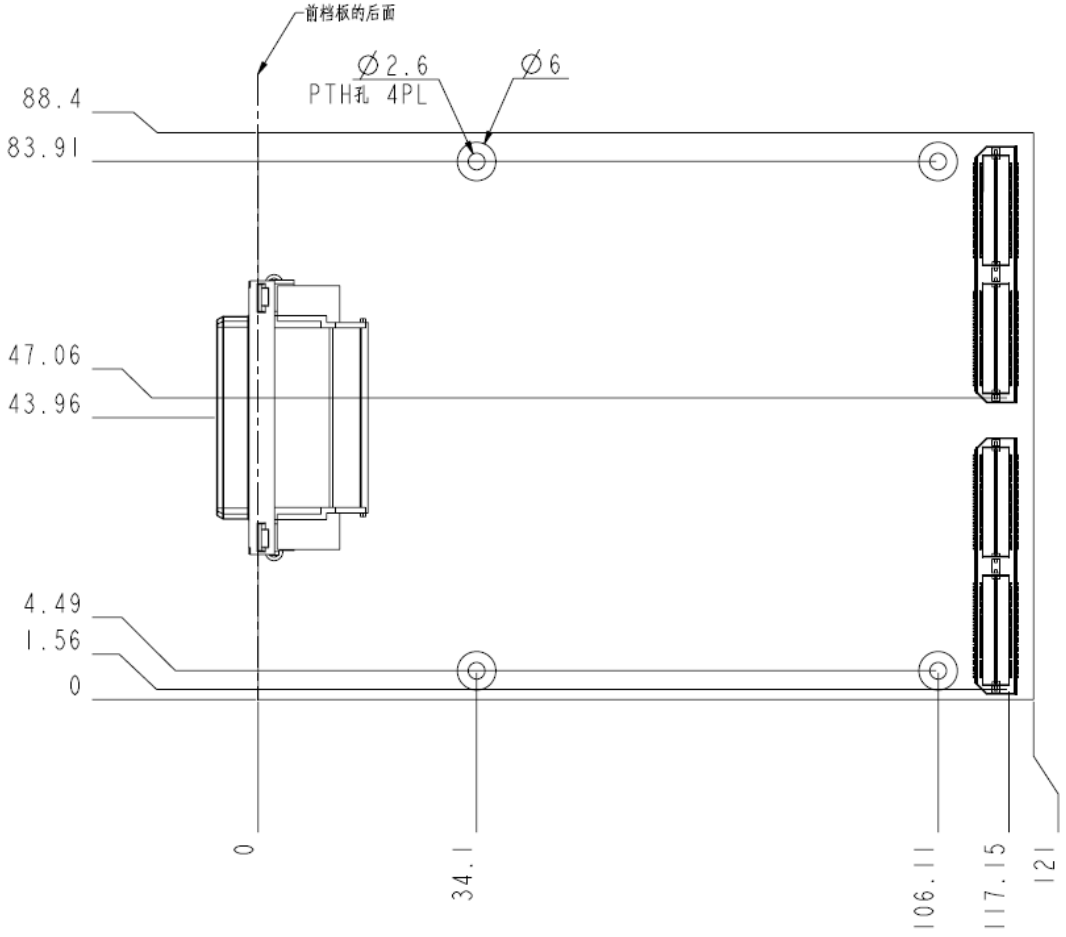


图 9-3 子卡参考设计尺寸

9.3 电气连接

PXIe-1010 总线控制模块与子板之间使用了两个高密度连接器。PXIe-1010 总线控制模块上的连接器是 Samtec 的 QSH-060-01-L-D-A-K。子板上推荐使用 QTH-060-03-C-D-A。每个连接器含 120 个引脚。接插件的传输性能可达到 5GHz（差分，-3dB 带宽）。最大电流可达每引脚 2A。

编号	功能	FPGA 引脚	编号	功能	FPGA 引脚
1	GND		2	IO_C_16N	H11
3	IO_C_17N	H8	4	IO_C_16P	H12
5	IO_C_17P	H9	6	GND	
7	GND		8	IO_C_14N	H13
9	IO_C_15N	F12	10	IO_C_14P	J13
11	IO_C_15P	G12	12	GND	
13	GND		14	IO_C_12N	D10
15	IO_C_13N	G9	16	IO_C_12P	E10
17	IO_C_13P	G10	18	GND	
19	GND		20	IO_C_10N	D11
21	IO_C_11N	F10	22	IO_C_10P	E11
23	IO_C_11P	G11	24	GND	
25	GND		26	IO_C_8N	G14
27	IO_C_9N	D8	28	IO_C_8P	H14
29	IO_C_9P	D9	30	GND	
31	GND		32	IO_C_6N	D13
33	IO_C_7N	A8	34	IO_C_6P	D14
35	IO_C_7P	A9	36	GND	
37	GND		38	IO_C_4N	B11
39	IO_C_5N	A10	40	IO_C_4P	B12
41	IO_C_5P	B10	42	GND	
43	GND		44	IO_C_2N	C13
45	IO_C_3N	A12	46	IO_C_2P	C14
47	IO_C_3P	A13	48	GND	
49	GND		50	IO_C_0N	A15
51	IO_C_1N	C11	52	IO_C_0P	B15
53	IO_C_1P	C12	54	GND	
55	GND		56	VDDIO_C	A11,B8,C15,D12,E9,G13,H10
57	VDDIO_C	A11,B8,C15,D12,E9,G13,H10	58	GND	
59	GND		60	12P0V	
61	3P3V		62	GND	
63	GND		64	VDDIO_B	B18,E19,F16,H20,J17,M18

65	VDDIO_B	B18,E19,F16,H20,J17,M18	66	GND	
67	GND		68	IO_B_16N	F15
69	IO_B_17N	B16	70	IO_B_16P	G15
71	IO_B_17P	C16	72	GND	
73	GND		74	IO_B_14N	K17
75	IO_B_15N	C18	76	IO_B_14P	K16
77	IO_B_15P	C17	78	GND	
79	GND		80	IO_B_12N	A19
81	IO_B_13N	A17	82	IO_B_12P	A18
83	IO_B_13P	B17	84	GND	
85	GND		86	IO_B_10N	D18
87	IO_B_11N	J19	88	IO_B_10P	E18
89	IO_B_11P	J18	90	GND	
91	GND		92	IO_B_8N	D16
93	IO_B_9N	F18	94	IO_B_8P	D15
95	IO_B_9P	G17	96	GND	
97	GND		98	IO_B_6N	E17
99	IO_B_7N	J20	100	IO_B_6P	F17
101	IO_B_7P	K20	102	GND	
103	GND		104	IO_B_4N	E20
105	IO_B_5N	F20	106	IO_B_4P	F19
107	IO_B_5P	G19	108	GND	
109	GND		110	IO_B_2N	H18
111	IO_B_3N	L18	112	IO_B_2P	H17
113	IO_B_3P	M17	114	GND	
115	GND		116	IO_B_0N	K18
117	IO_B_1N	J16	118	IO_B_0P	L17
119	IO_B_1P	J15	120	GND	

表 9-1 J1 的引脚定义

编号	功能	FPGA 引脚	编号	功能	FPGA 引脚
1	TDO		2	GND	
3	TDI		4	12P0V	
5	GND		6	GND	
7	TCK		8	12P0V	
9	TMS		10	GND	
11	GND		12	12P0V	
13	3P3V		14	SDA	
15	GND		16	SCL	
17	3P3V		18	GND	
19	GND		20	IO_A_16N	L24

21	3P3V		22	IO_A_16P	M24
23	IO_A_17N	K26	24	GND	
25	IO_A_17P	K25	26	IO_A_14N	N17
27	GND		28	IO_A_14P	P16
29	IO_A_15N	L25	30	GND	
31	IO_A_15P	M25	32	IO_A_12N	N23
33	GND		34	IO_A_12P	P23
35	IO_A_13N	M26	36	GND	
37	IO_A_13P	N26	38	IO_A_10N	P18
39	GND		40	IO_A_10P	R18
41	IO_A_11N	N22	42	GND	
43	IO_A_11P	N21	44	IO_A_8N	P21
45	GND		46	IO_A_8P	R21
47	IO_A_9N	P26	48	GND	
49	IO_A_9P	R26	50	IO_A_6N	R20
51	GND		52	IO_A_6P	T20
53	IO_A_7N	P25	54	GND	
55	IO_A_7P	R25	56	IO_A_4N	R23
57	GND		58	IO_A_4P	R22
59	IO_A_5N	M19	60	GND	
61	IO_A_5P	N18	62	IO_A_2N	T23
63	GND		64	IO_A_2P	T22
65	IO_A_3N	T17	66	GND	
67	IO_A_3P	U17	68	IO_A_0N	P20
69	GND		70	IO_A_0P	P19
71	IO_A_1N	T19	72	GND	
73	IO_A_1P	T18	74	VDDIO_A	N25,P22,R19,T16,T26,K24
75	GND		76	NC	
77	VDDIO_A	N25,P22,R19,T16,T26,K24	78	NC	
79	NC		80	GND	
81	GND		82	GND	
83	NC		84	NC	
85	GND		86	NC	
87	NC		88	GND	
89	NC		90	GND	
91	GND		92	NC	
93	GND		94	NC	
95	NC		96	GND	
97	NC		98	GND	
99	GND		100	NC	

101	GND		102	NC	
103	CLK1_C2M_N		104	GND	
105	CLK1_C2M_P		106	GND	
107	GND		108	CLK1_M2C_N	
109	GND		110	CLK1_M2C_P	
111	CLK0_C2M_N		112	GND	
113	CLK0_C2M_P		114	GND	
115	GND		116	CLK0_M2C_N	
117	GND		118	CLK0_M2C_P	
119	PG_C2M		120	GND	

表 9-2 J2 引脚定义

根据连接器上的信号类型，可以将引脚分为以下几类

- 电源
  - 12P0V: 12V，5A
  - 3P3V: 3.3V，5A
  - PG_C2M: 母板到子板的 Power Good
- 数字信号，3 组（A\B\C）,每组含
  - IO_A/B/C_xx: 18 对差分或 36 路单端
    - 带 MRCC 的引脚是连接到 FPGA 的全局时钟引脚的。
  - VDDIO_A/B/C，母板侧的信号接口电源，默认为 3.3V。如果板上的数字信号电平需要通过 level shifter 转换，可使用此电源。
- JTAG 接口
  - TDI、TDO、TCK、TMS
- I2C 接口
  - SCL、SDA
- 2 对母板到子板的同步时钟
  - CLK0_C2M: 实际连接的是 PXIe_CLK100
  - CLK0_C2M: 实际连接的是 PXIe_SYNC100
- 2 对子板到母板的同步时钟