Application Report

*实现 **Linux** 的实时通信*

Steffen Graf

商标

1 引言

不同的机器视觉库（例如 MVTec 的 HALCON）和神经网络在 Linux 操作系统上会作为应用程序运行。Linux 支持应用的轻松移植，并为不同的硬件（例如网络接口、大容量存储，甚至是摄像机）提供了通用软件接口。因此，当视觉传感器必须捕获和处理图像，但无需将原始图像传输到视觉计算机时，很适合使用 Linux 平台。

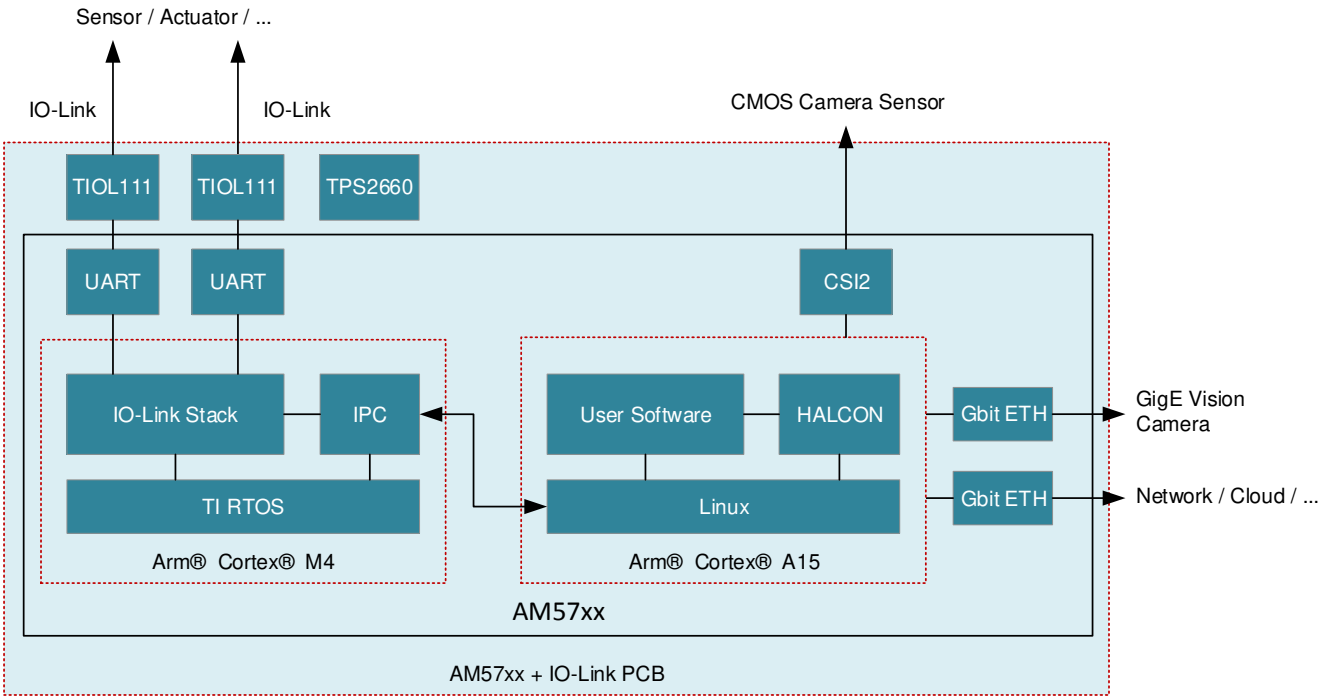

图 1-1. 具有 IO-Link 的视觉传感器简化方框图

这样的器件可能还需要使用 IO-Link 来控制外部执行器或传感器。例如，可将其用于测量与物体之间的距离或控制闪光强度和持续时间。如图 1-1 所示，可通过使用 IO-Link PHY（例如 TIOL111）和 SoC 的集成式 UART 来实现 IO-Link。本文档介绍在 Linux 上实现实时协议的不同方式以及获得的计时抖动。

在嵌入式 Linux 系统上处理诸如 IO-Link 之类的实时通信会是一项挑战。在 Linux 用户空间中，无法在 µs 范围内以确定性计时读写外设（如 UART）。Linux 调度程序并不适合此类应用，并且还必须处理其他任务。调度程序会导致计时抖动（具体情况取决于 CPU 负载），进而无法实现实时通信。

另一种方法是集成到 Linux 内核中。在内核空间中，高分辨率计时器可产生只能被另一个内核计时器或硬件外设阻止的精确计时。在这种情况下，抖动优于用户空间实现方式，但是其仍然存在。另外，将复杂的堆栈移入内核空间也并不总是一种良好设计做法。

第三种选择是使用单独的处理器内核来处理实时通信。除了两个 Arm® Cortex®-A15 内核之外，Sitara™ AM5728 SoC 还具有多个可用于该应用的 Arm Cortex-M4 内核。可将严格计时部分卸载，而 Arm Cortex-A15 和 M4 内核之间会建立非关键通信链路以用于交换数据。

2 实现

为了将 IO-Link 集成到 Linux，此处选择了第三个方案，因为它不会增加 Arm Cortex-A15 的 CPU 负载，并且计时不会因 Linux 上的 CPU 负载而发生干扰或改变。随后，该项目本身分为两个子项目，一个在 A15 内核上基于 Linux 运行，另一个作为 RTOS 项目在 Arm Cortex-M4 内核之一上运行。这两个子项目共享通用的命令定义，并通过处理器间通信 (IPC) 进行通信。

在 Linux 一端，编译到内核中的 Remoteproc 框架负责加载 Arm Cortex M4。必须将主内存配置为具有分割区，以使 Linux 不会接触到 Arm Cortex-M4 使用的内存，否则它们会相互干扰。此配置在 Linux dts 文件 *am57xx-beagle-x15-common.dtsi* 中完成，*ipu2_cma_pool* 部分针对第二个 IPU（这是 Arm Cortex-M4 子系统之一）进行此配置。

```
ipu2_cma_pool: ipu2_cma@95800000 {
compatible = "shared-dma-pool";
reg = <0x0 0x95800000 0x0 0x3800000>;
reusable;
status = "okay";
};
```

必须在 Arm Cortex-M4 项目的资源配置中完成相同的配置。此资源表会编译到项目中，并由 Remoteproc 驱动程序在启动时正确配置 MMU，从而向 IPU 提供所有必要的资源。

Arm Cortex-M4 在逻辑地址 0x0000 0000 处启动。该逻辑地址由 MMU 映射到分割区域中的相应物理 DDR 内存地址 (0x9580 0000)。Linux 驱动程序将二进制文件加载到该 DDR 位置，并且 Arm Cortex-M4 会因复位被释放。

Arm Cortex-M4 启动后，可在两个内核之间建立通信通道。这基本上就是一个客户端/服务器模型。Arm Cortex-M4 内核提供了一个服务器，可打开消息队列。在 Arm Cortex-A15 上的 Linux 中，可打开此消息队列，并传递数据。

Arm Cortex-M4 上的 RTOS 有两项任务，一项是处理 IO-Link 通信，另一项是等待从 Linux 接收到通信队列中的数据。一旦接收到数据，就可对数据进行处理并传递给 IO-Link 任务。

仅有少量数据被交换，因此所有内容均以 32 位字进行编码。Arm Cortex-M4 上的一个名为 void Server_handle_cmd(unsigned int *cmd) 的函数会处理从主机处理器接收到的命令（存储在 cmd 中）并将响应写入相同的位置。此后会发送回去。如果是发出读取 ISDU 请求，可能看起来像这样：

```
void Server_handle_cmd(unsigned int *cmd) {
switch ((*cmd) & App_CMD_MASK) {
...
case App_CMD_GET_ISDU_START:
MOD_Read_req((*cmd & 0x00f00000)>>20, (*cmd & 0x0000ffff), (*cmd & 0x000f0000)>>16);
isdu_read_state = 1;
break;
...
}
}
```

上面的代码片段会分析收到的命令，并将数据传递到堆栈。32 位宽命令的高 8 位对应用命令进行编码，此处的命令为 App_CMD_GET_ISDU_START，用于请求读取 ISDU。命令的其余位包括端口、索引以及子索引。通过调用 MOD_Read_req，可提取这些参数并将它们传递给 IO-Link 堆栈的读取函数。除了设置内部标志 isdu_read_state 之外，还可由其他应用程序命令将其用于传输读取请求的状态。此处的状态包括该请求是否已由堆栈处理以及数据是否已传递到 Linux 主机。

Linux 主机必须将所有命令编码为 32 位字并将它们传递到通信队列中。一个开始在指定端口上读取 ISDU 的函数如下所示：

```
void isdu_read_start(int port, int index, int subindex){
App_Msg *msg;
/* allocate message */
```

```
msg = (App_Msg *)MessageQ_alloc(Module.heapId, Module.msgSize);
/* fill in message payload */
msg->cmd = App_CMD_GET_ISDU_START | port<<20 | subindex<<16 | index;
/* send message */
MessageQ_put(Module.slaveQue, (MessageQ_Msg) msg);
/* wait for return message */
MessageQ_get(Module.hostQue, (MessageQ_Msg *) &msg, MessageQ_FOREVER);
/* free memory */
MessageQ_free((MessageQ_Msg)msg);
}
```

MessageQ_get 返回后，来自 Arm Cortex-M4 的响应将存储在 msg 中，并接受进一步处理。例如，在此处执行读取命令后，可以存储已回读的值。

除了用于启动读取请求的函数之外，还实现了一个用于检查状态并回读缓冲区的函数。

启动读取请求后，主机处理器必须检查读取命令是否已完成，然后从读取缓冲区中获取数据。如下所示的例子中实现了一个完整的读取函数，用于读取字符串数据，例如产品名称。

```
void isdu_read_char(int port, int index, int subindex, char *buf, int *len){
isdu_read_start(port, index, subindex);
while(isdu_read_state() != READ_COMPLETED);
isdu_read_data(buf, len);
buf[*len] = '\0';
}
```

在此抽象级上，可在用户应用中使用诸如 get_port_state、isdu_read_char 和 isdu_write_char 之类的简单函数来从 Linux 用户空间控制 IO-Link。

此处的实现示例在机器视觉应用中使用这些函数。用户应用会利用 HALCON 库，将 GigE Vision 摄像机连接到 Sitara AM5728 处理器的以太网端口之一。同样的应用控制着一个塔灯以发出信号，指示是否正确识别了某个物体，并将结果通过以太网发送到服务器进行监控。

3 测试结果

如前文所述，在 Linux 上实现 O-Link 或任何其他实时协议存在多种可能性。用户空间或内核空间都可以直接用在运行 Linux 的 Arm Cortex-A15 内核上。此外，也可以在独立于 Linux 的单独内核上实现。但是，IO-Link 有严格的计时要求，必须定期处理，其容差为周期时间的 -0% +10%。为了解哪种实现方式可以达到计时要求，在实现 IO-Link 之前需要先进行一些测试。

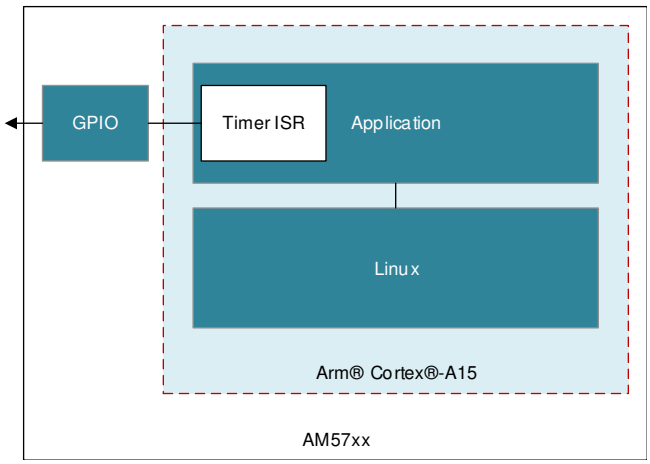

图 3-1. 用户空间中的计时器

为了实现 IO-Link，必须配置一个计时器并由计时器触发通信。为了评估计时精度，需要设置计时器，并且处理程序会切换 GPIO 以查看对外设的访问精度。可通过逻辑分析仪观察计时抖动。

如图 3-1 所示，在应用所包含的用户空间环境中，使用 timer_create 和 timer_settime 创建了一个 100µs 计时器。此计时器在到期时会触发一个 SIGALRM 事件。此事件会记录至一个触发 GPIO 的处理函数。图 3-2 所示为

实际计时的直方图。此计时器的平均值 101µs 是令人满意的，但标准偏差 59µs、极端最小值 58µs 以及最大值 4.9ms 则过高，超过 2% 的样本不在图中所示 90µs – 110µs 窗口范围之内。这样的表现不符合规范要求。

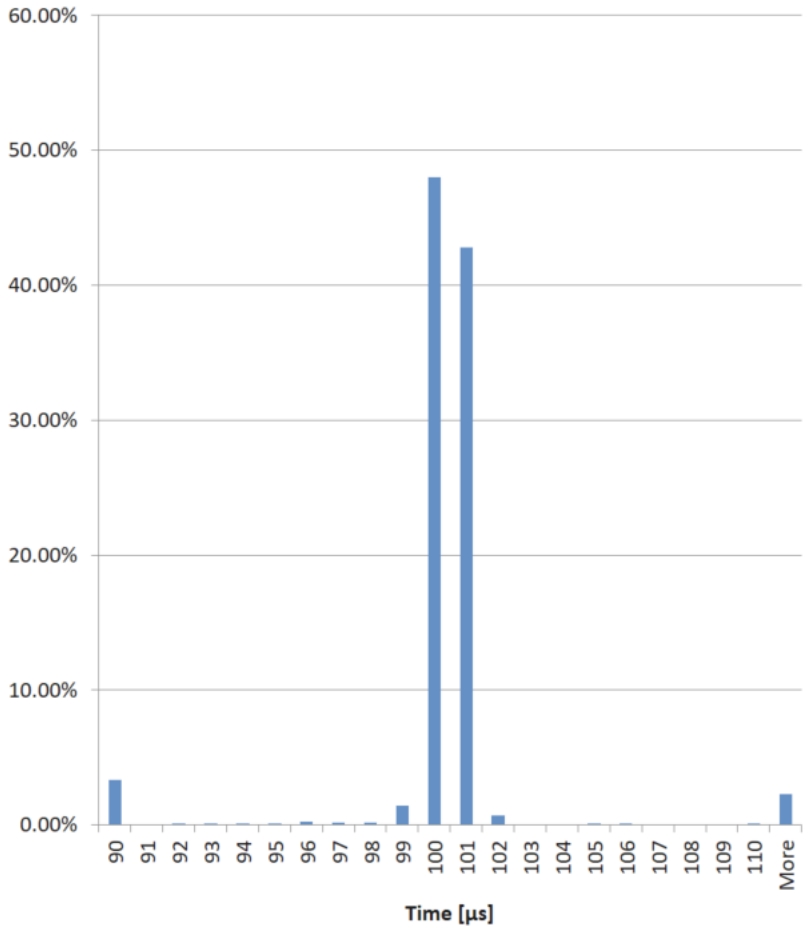

图 3-2. 采用用户空间实现方式时的计时抖动

为了在内核空间中进行测试，需要创建一个模块来实例化一个高分辨率计时器，并将这个计时器设置为 100µs，并在每次触发时重新开始计时。使用 hrtimer_init 和 hrtimer_start，则易于进行循环函数调用。如图 3-3 所示，该计时器现在直接在内核中接受处理，它的优先级比在用户空间实现方式中高得多，并且计时冲突更少。

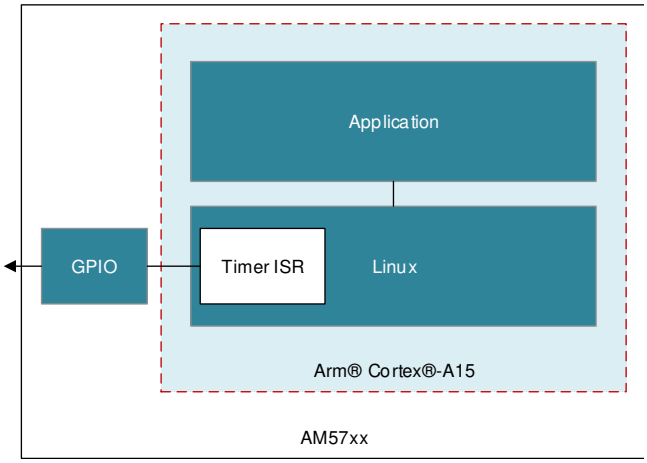

图 3-3. 内核空间中的计时器

图 3-4 所示为生成的直方图。计时比在用户空间实现方式中要好得多。平均值为 100µs，标准偏差小于 1µs。但是，80µs 和 122µs 的极值仍然过高（虽然小于所捕获的全部样本的 1%）。

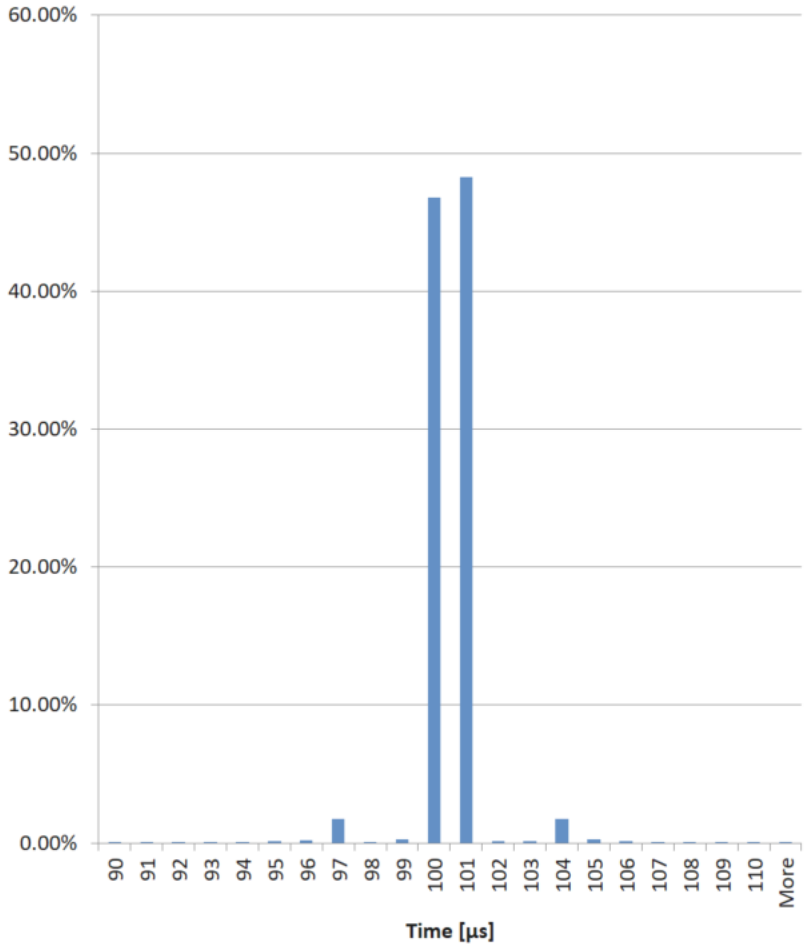

图 3-4. 采用内核空间实现方式时的计时抖动

如图 3-5 所示，使用 SoC 内部具有相同功能的 Cortex M4 之一进行了第三项测试。此处运行 TI RTOS，并可使用 Timer_create 函数来启动计时器。此计时器配置为定期调用某个函数。由于它在单独的内核上运行，因此不会受到 Arm Cortex-A15 正在进行的操作的干扰。

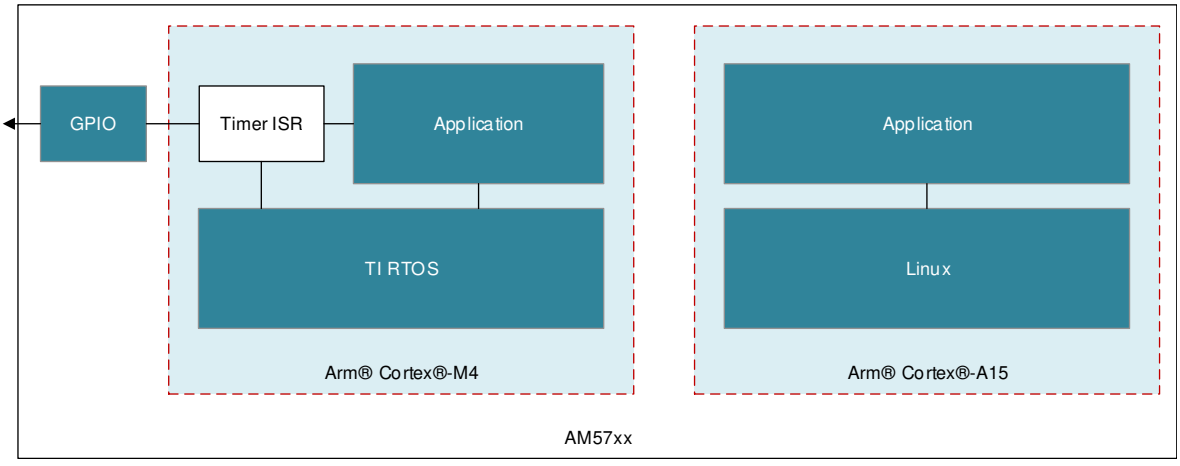

图 3-5. Arm Cortex M4 上的计时器

图 3-6 所示为生成的直方图。该图选用 0.01µs 的分辨率来展示计时器的精度，分辨率为 1µs 的直方图只会有一个区间 (bin)。101µs 的最小值和 102µs 的最大值为实现 IO-Link 提供了精确的计时基础。

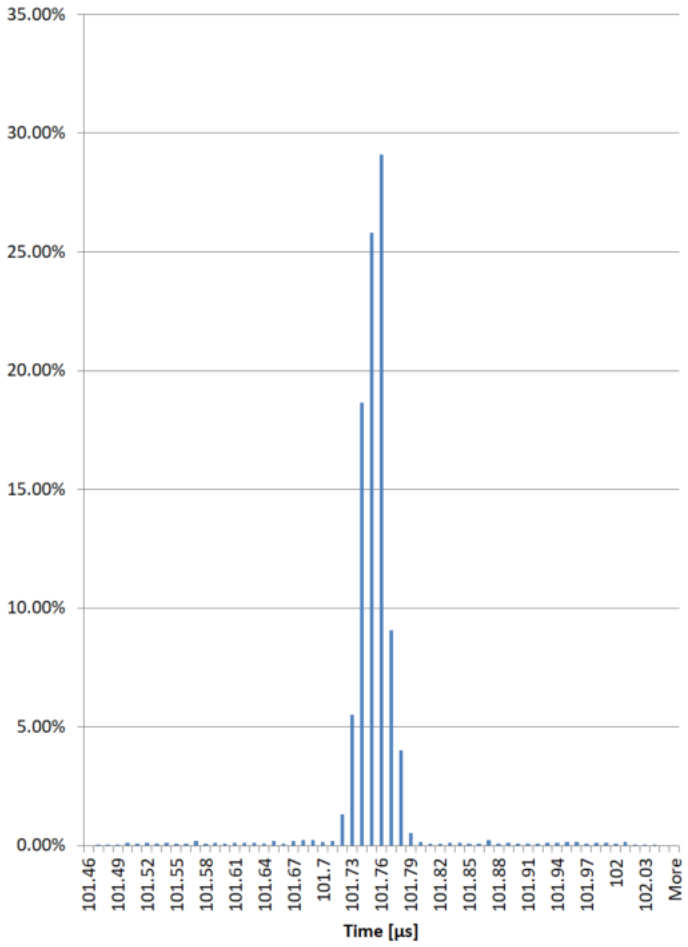

图 3-6. 在单独的 Arm Cortex-M4 上实现时的计时抖动

根据这项计时研究的结果可以得出结论：为了结合 Linux 来实现实时协议，需要一个可独立于 Linux 运行的额外 CPU 内核。否则，至少对于 IO-Link 而言，操作系统会导致计时抖动超过规格值。Sitara 器件内部的 IPU 包含两个 Arm Cortex-M4 子系统，每个子系统都有两个内核，适合用于实现此类实时协议。

图 3-7 中显示了 Sitara AM5728 处理器与 IO-Link 器件之间活动通信的 CQ 线路。此处使用的器件具有 COM3 数据速率（230400 波特率）和 1ms 的循环时间。屏幕截图显示了循环的过程数据交换，此处设置为零。通过测量，循环时间为 1.02ms，此值在允许的容差范围内。现在可以从 Linux 系统读取和写入 ISDU，如下所示：

```
a@BeagleBoard-X15:~$ sudo ./app_host IPU2
ISDU 16 18 bytes: Texas Instruments
ISDU 17 11 bytes: www.ti.com
ISDU 18 41 bytes: TIDA-01437 RGB Signal Light with IO Link
ISDU 19 2 bytes: 1
ISDU 20 11 bytes: TIDA-01437
ISDU 21 9 bytes: 00000000
ISDU 22 7 bytes: HW-V1.0
ISDU 23 20 bytes: FW-V1.0 - COM3 - 1ms
ISDU 24 32 bytes: ********************************
```

此示例读取 ISDU 16 至 24，返回数据和字段长度。

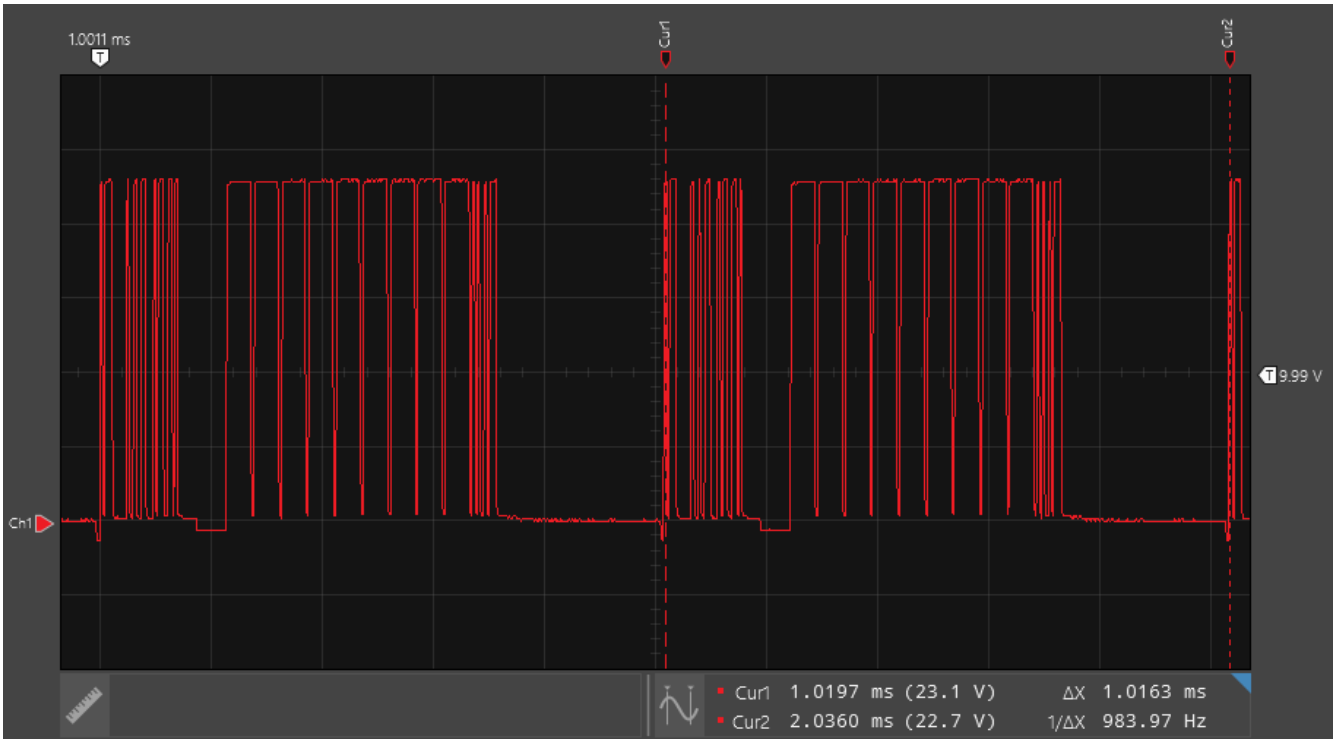

图 3-7. 与 TIDA-01437 塔灯之间的 IO-link 通信

通过将 IPU 内核用于有严格计时要求的协议，则可轻松将有严格计时要求的协议（例如 IO-Link）集成到 Linux 应用中。在执行与 Linux 应用或内核模块相同的实现方式时，通过移至单独的内核可实现更精确的计时。使用 IPC 框架，无需修改 Linux 内核即可轻松进行软件开发。Sitara 处理器提供集成的 Arm Cortex-M4 内核，这些内核易于使用并且可以很好地满足实时要求。

这种方法使得将 IO-Link 主站和器件集成到 Linux 成为可能，例如可以通过这种方式使得在 Linux 上运行 HALCON 视觉库的视觉传感器具有 IO-Link 接口。

4 备选器件建议

表 4-1. 备选器件建议

器件	优化参数
AM5728	2 个 Arm Cortex A15，2 个具有双核 Arm Cortex M4 的 IPU
AM5718	1 个 Arm Cortex A15，2 个具有双核 Arm Cortex M4 的 IPU
AM6548	4 个 Arm Cortex A53，2 个 Arm Cortex R5F
AM6528	2 个 Arm Cortex A53，2 个 Arm Cortex R5F

参考文献

- *《Processor SDK Linux 软件开发人员指南》中“基础组件”下的 3.7. IPC*
- *Linux 程序员手册*

德州仪器 (TI)，*《AM572x Sitara 处理器技术参考手册》*

德州仪器 (TI)，*《AM572x Sitara 处理器器件版本 2.0 数据手册》*

8 端口 IO-Link 主站参考设计

配备 IO-Link 接口的 RGB 信号灯参考设计

5 修订历史记录

注：以前版本的页码可能与当前版本的页码不同

Changes from Revision * (April 2019) to Revision A (May 2021) **Page**

- 更新了整个文档中的表格、图和交叉参考的编号格式。 1

重要声明和免责声明

TI 提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他安全、安保或其他要求。这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款 (https:www.ti.com/legal/termsofsale.html) 或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265

重要声明和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、某特定用途方面的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他功能安全、信息安全、监管或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的应用。严禁对这些资源进行其他复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。您应全额赔偿因在这些资源的使用中对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，TI 对此概不负责。

TI 提供的产品受 TI 的销售条款或 ti.com 上其他适用条款/TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

邮寄地址：Texas Instruments, Post Office Box 655303, Dallas, Texas 75265