*Application Report*

# *Adding Real-Time Communication to Linux*

*Steffen Graf*

## Trademarks



## 1 Introduction

Different machine vision libraries (such as HALCON from MVTec) and neuronal networks are running as applications on Linux as an operating system. Linux offers easy portability of the application and provides common software interfaces for different hardware such as network interfaces, mass storage, or even cameras. This makes it an ideal platform for vision sensors that have to capture and process an image, but do not need to stream the raw image to a vision computer.

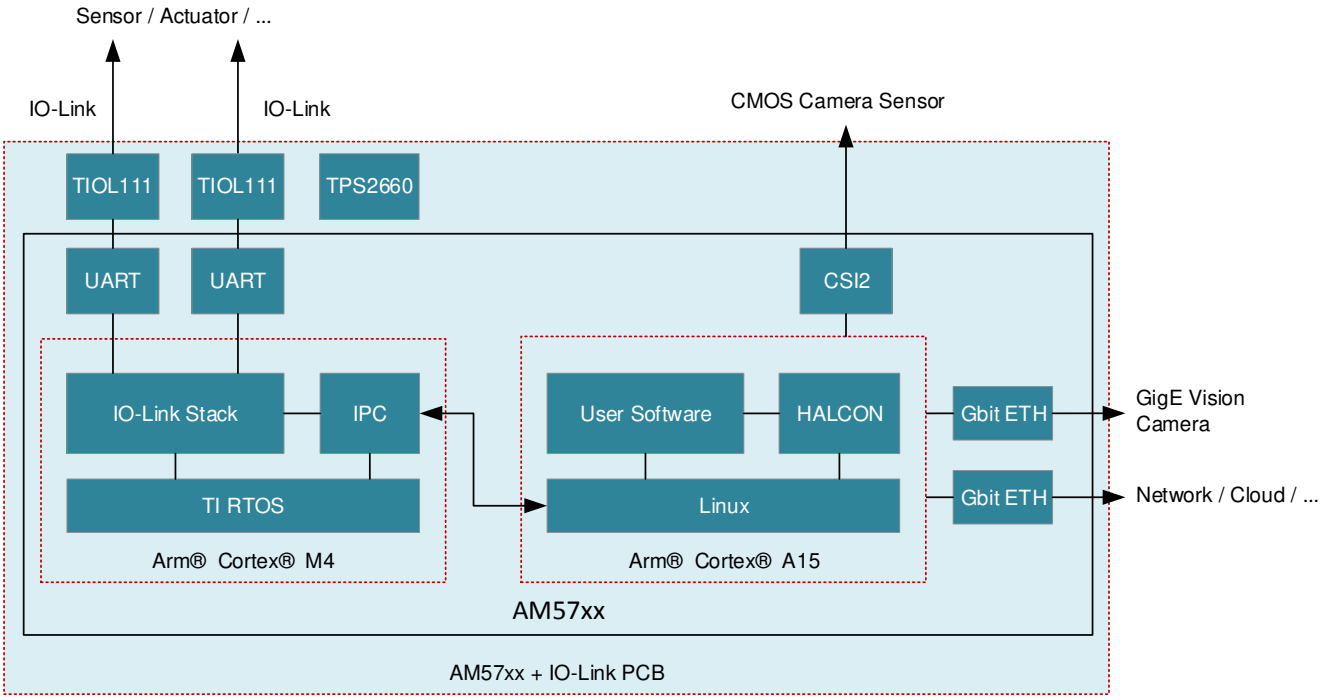




**Figure 1-1. Simplified Block Diagram Vision Sensor with IO-Link**

Such a device might also need to control external actuators or sensors using IO-Link. For example, you can use it to measure the distance to an object or to control the flash intensity and duration. As shown in Figure 1-1, implementing IO-Link can be done by using an IO-Link PHY, such as TIOL111, and an integrated UART of the SoC. This document describes different possibilities of how to implement a real-time protocol in Linux and the achieved timing jitter.

Handling real-time communication such as IO-Link on an embedded Linux system can be a challenge. From the Linux user space it is not possible to read and write a peripheral such as an UART with deterministic timings in the µs range. The Linux scheduler is not designed for such an application and has to handle other tasks

as well. The scheduler causes timing jitter, depending on the CPU load, which prevents implementing real-time communication.

Another approach is integrating into the Linux kernel. In the kernel space, high resolution timers can create precise timings that can only be blocked by another kernel timer or hardware peripherals. In that case, the jitter is better than a user space implementation, however it is still present. Also, moving a complex stack into kernel space is not always a good design practice.

A third option is using a separate processor core that handles the real-time communication. Besides the two Arm® Cortex®-A15 cores, the Sitara™ AM5728 SoC has several Arm Cortex-M4 cores that can be used for this application. The timing critical part can be offloaded and a non-critical communication link is established between the Arm Cortex-A15 and M4 core to exchange data.

## 2 Implementation

To integrate IO-Link to Linux, the third option is chosen, since it does not increase the CPU load of the Arm Cortex-A15 and the timings do not interfere or change depending on the CPU load on Linux. The project itself is then separated into two projects, one running under Linux on the A15 core and the other one as an RTOS project running on one of the Arm Cortex-M4 cores. Both projects share a common definition for the commands and communicate via inter processor communication (IPC).

On the Linux side, the remoteproc framework compiled into the kernel handles loading the Arm Cortex M4. The main memory has to be configured to have carveouts, so that the Linux does not touch the memory used by the Arm Cortex-M4, otherwise they interfere each other. This configuration is done in the Linux dts file *am57xx-beagle-x15-common.dtsi*, the section *ipu2_cma_pool* configures this for the second IPU, which is one of the Arm Cortex-M4 subsystems.

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

The same configuration has to be done in the resource configuration for the Arm Cortex-M4 project. This resource table is compiled into the project and is used by the remoteproc driver to configure the MMU properly at startup to provide all the necessary resources to the IPU.

The Arm Cortex-M4 starts up at logical address 0x0000 0000, which is mapped by the MMU to the corresponding physical DDR memory address (0x9580 0000) in the carveout region. The binary file is loaded by the Linux driver to that DDR location and the Arm Cortex-M4 is released from reset.

After the Arm Cortex-M4 has started, the communication channel between both cores can be established. This is basically a client server model. The Arm Cortex-M4 core provides a server, opening a message queue. From the Linux on the Arm Cortex-A15, this message queue can be opened and data can be passed through.

The RTOS on the Arm Cortex-M4 has two tasks, one is handling the IO-Link communication, the other one is waiting until data is received in the communication queue from Linux. As soon as data is received, it can be handled and passed over to the IO-Link task.

As only a small amount of data is exchanged, everything is encoded in 32 bit words. A function on the Arm Cortex-M4 called void Server_handle_cmd(unsigned int *cmd) handles the received command (stored in cmd) from the host processor and writes the response at the same location. This is sent back afterwards. For issuing a read ISDU request, this can look like this:

```
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;
...
```

```
}
}
```

The excerpt above parses the received command and passes the data on to the stack. The upper 8 bit of the 32 bit wide command encode the application command, here App_CMD_GET_ISDU_START, which requests to read an ISDU. The remaining bits of cmd include the port, the index as well as the subindex. These parameters are extracted and passed to the read function of the IO-Link stack by calling MOD_Read_req. In addition to that an internal flag isdu_read_state is set, that is used by other application commands to communicate the status of the read request. The status includes if the request is already processed by stack and if the data has already been passed over to the Linux host or not.

The Linux host has to encode all commands in 32 bit word and pass them into the communication queue. A function to start reading an ISDU on a specified port can then look like this:

```
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);
}
```

After MessageQ_get returns, the response from the Arm Cortex-M4 is stored in msg and can be processed further. For example, after a read command here the value that has been read back can be stored.

In addition to a function for starting the read request, also a function checking the state and reading back the buffer is implemented.

After starting the read request, the host processor has to check wether the read command is finished and then get the data from the read buffer. A complete read function to read string data, like the product name is implemented as shown below.

```
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';
}
```

At this level of abstraction, simple functions like get_port_state, isdu_read_char and isdu_write_char can be used in a user application to control IO-Link from the Linux user space.

The example implementation here uses these functions in a machine vision application. The user application makes use of the HALCON library, interfacing a GigE Vision camera on one of the Ethernet ports of a Sitara AM5728 processor. The same application controls a stacklight to signal if an object is recognized correctly or not and send the results via Ethernet to a server for monitoring.

## 3 Test Results

As discussed before, there are different possibilities to implement IO-Link or any other real-time protocol on Linux. Either user or kernel space can be used directly on the Arm Cortex-A15 core running Linux. It can also be implemented on a separate core independent from Linux. However IO-Link is timing critical and has to be handled regularly with a tolerance of -0% +10% of the cycle time. To get a feeling which implementation can achieve the required timing some tests are done before implementing IO-Link.

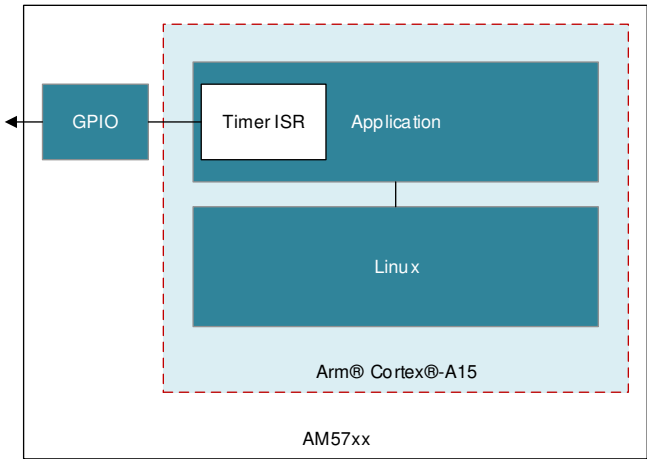


**Figure 3-1. Timer in User Space**

For implementing IO-Link, a timer has to be configured and triggers the communication. To evaluate the timing accuracy, a timer is set up and the handler is switching a GPIO to see how accurately it is possible to access a peripheral. With a logic analyzer, the timing jitter can be observed.

As shown in Figure 3-1 in the user space environment as part of the application, a 100 µs timer is created using timer_create and timer_settime. This timer fires a SIGALRM event when expired. This event is registered to a handling function that toggles an GPIO. Figure 3-2 shows the histogram of the actual timing. The average value of this timer is fine with 101 µs, however, the standard deviation with 59 µs and extreme values of minimal 58 µs and maximum 4.9 ms are too high, more than 2% of the samples are out of the 90 µs–110 µs window displayed in the plot. This violates the specification.

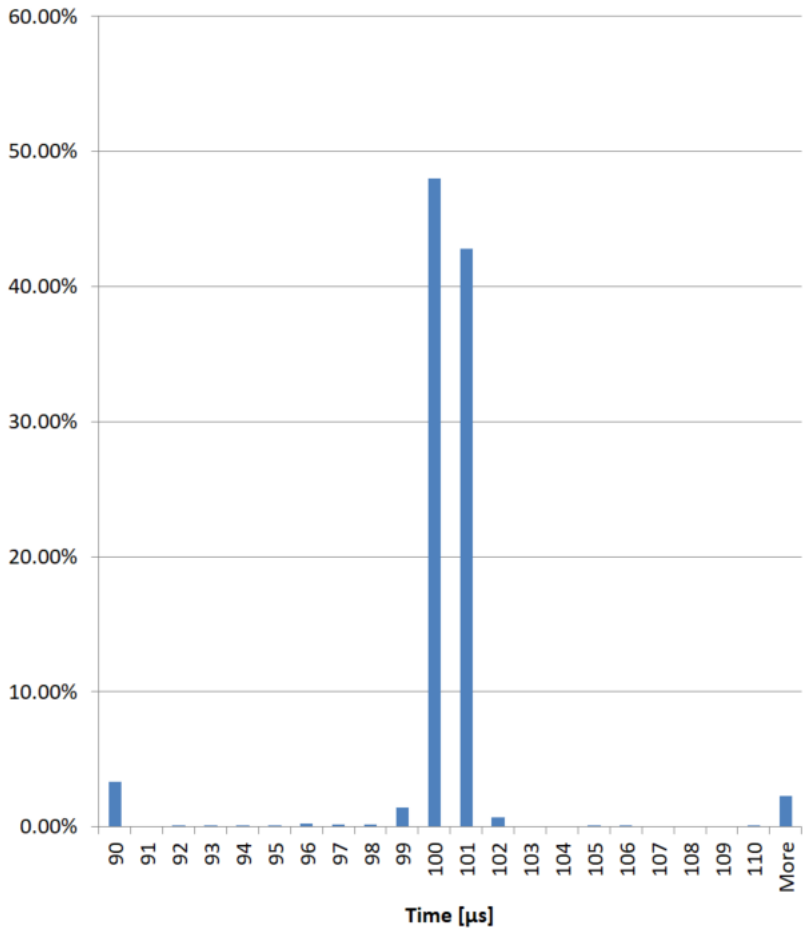


**Figure 3-2. Timing Jitter with User Space Implementation**

For testing in kernel space, a module is created instantiating a high resolution timer, setting it to 100 µs and restarting it every time it fires. By using hrtimer_init and hrtimer_start, it is simple to get a cyclic function calling. As shown in Figure 3-3 the timer is now directly handled in the kernel, it has a much higher priority than in user space and less timing violations.

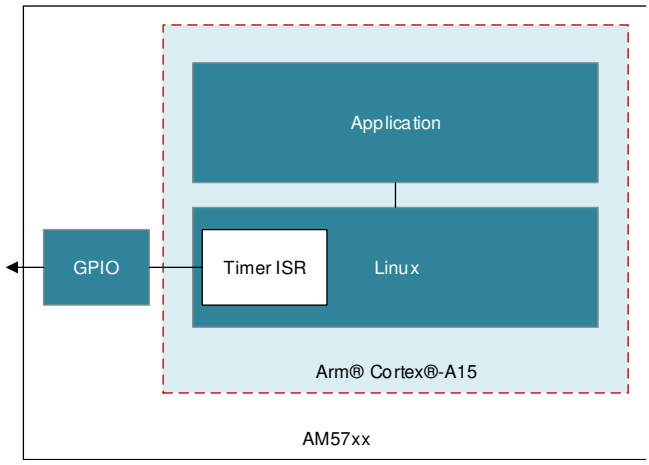


**Figure 3-3. Timer in Kernel Space**

Figure 3-4 shows the resulting histogram. The timing is much better than in user space. The average is 100 µs and the standard deviation is less than 1 µs. However, the extreme values are still too high with 80 µs and 122 µs, even though this is less than 1% of all samples captured.

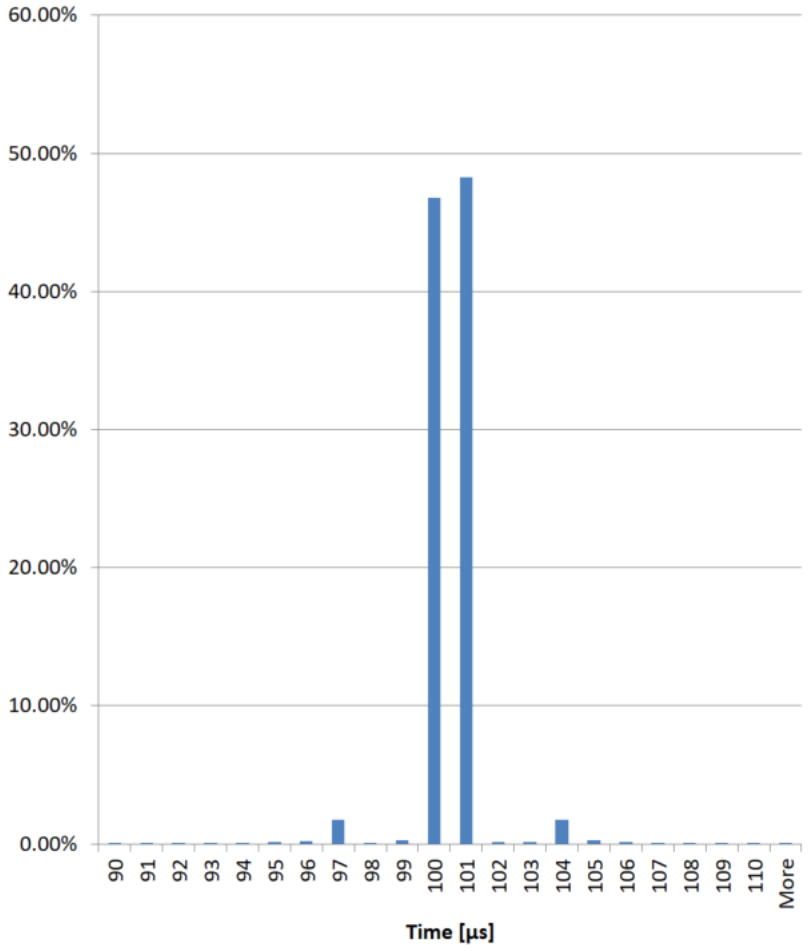


**Figure 3-4. Timing Jitter with Kernel Space Implementation**

A third test is done as shown in Figure 3-5 using on of the Cortex M4 inside the SoC doing the same functions. Here, TI RTOS is running and a timer can be initiated by using the Timer_create function. This timer is configured to periodically call a function. Since this is running on a separate core, it is not interfered by what is going on at the Arm Cortex-A15.

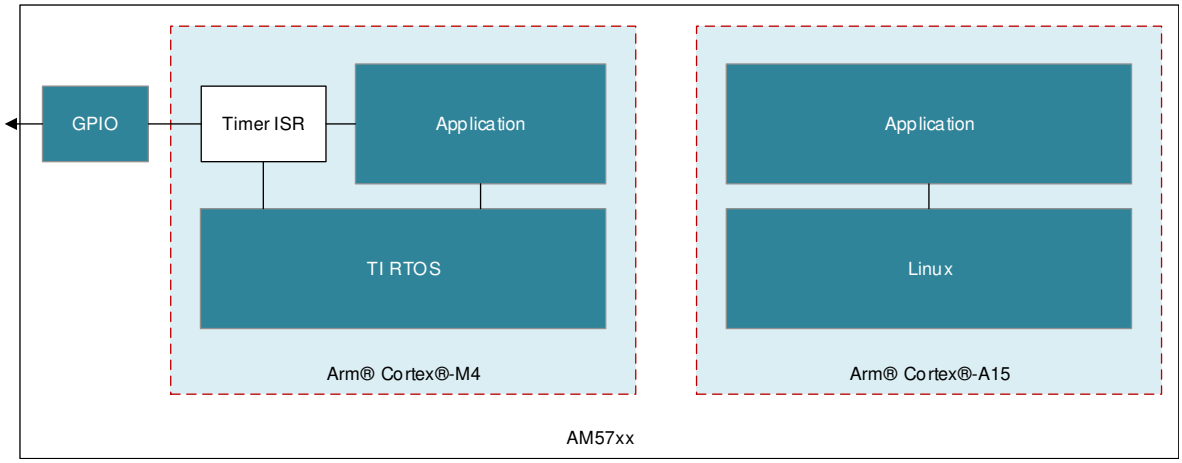


**Figure 3-5. Timer on Arm Cortex M4**

Figure 3-6 shows the resulting histogram. The plot is chosen to display a resolution of 0.01 µs to show how precise the timer can be, a histogram with 1 µs resolution would only have one bin. With a minimal value of 101 µs and a maximum of 102 µs, this gives a precise timing basis for implementing IO-Link.

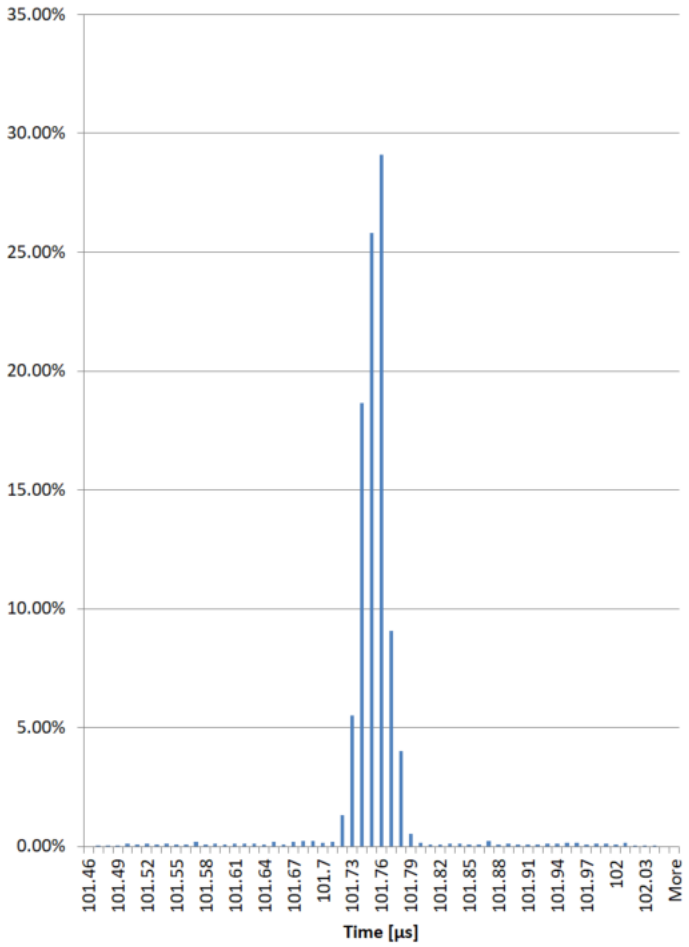


**Figure 3-6. Timing Jitter with Implementation on Separate Arm Cortex-M4**

As a result of this timing study, it can be said that implementing a real-time protocol in combination with Linux requires an additional CPU core that can be operated independent from Linux. Otherwise, the operating system causes timing jitter that exceeds the specification, at least for IO-Link. The IPU inside the Sitara devices consist of two Arm Cortex-M4 subsystems with two cores each and are a good way to implement such real time protocols.

In Figure 3-7, the CQ line of an active communication between an Sitara AM5728 processor and an IO-Link device is shown. The device used here uses COM3 datarate (230400 Baud) and a cycle time of 1 ms. The screenshot shows the cyclic exchange of process data, that is set to zero here. The cycle time can be measured with 1.02 ms and is withing the allowed tolerance. From the Linux system it is now possible to read and write ISDUs as shown here:

```
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: ********************************
```

This example reads ISDU 16 to 24, returns the data and the length of the fields.

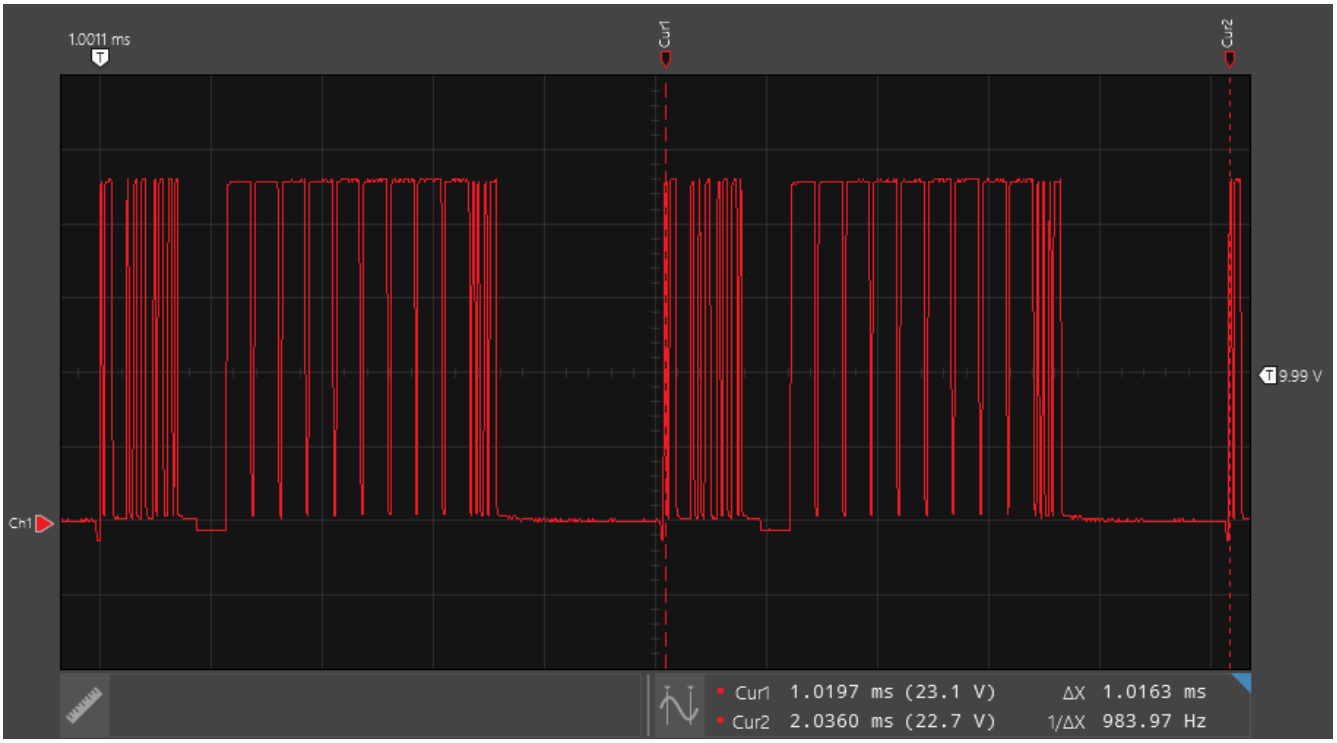


**Figure 3-7. IO-link Communication with TIDA-01437 Stacklight**

Using the IPU cores for timing critical protocols, it is easily possible to integrate a timing critical protocol, for example, IO-Link into a Linux application. A much more accurate timing can be achieved by moving to a separate core, as when implementing the same as Linux application or Kernel module. The IPC framework allows easy software development without the need to modify the Linux Kernel. The Sitara processor offer integrated Arm Cortex-M4 cores that are easy to use and can handle real-time requirements well.

This approach enables the possibility to integrate IO-Link master and device to Linux, for example for vision sensors running HALCON as vision library on top of Linux can be enabled to have IO-Link interface that way.

## 4 Alternate Device Recommendations

**Table 4-1. Alternative Device Recommendations**

| DEVICE | OPTIMIZED PARAMETERS |
|---|---|
| AM5728 | 2x Arm Cortex A15; 2 IPUs with Dual Core Arm Cortex M4 |
| AM5718 | 1x Arm Cortex A15, 2 IPUs with Dual Core Arm Cortex M4 |
| AM6548 | 4x Arm Cortex A53, 2x Arm Cortex R5F |
| AM6528 | 2x Arm Cortex A53, 2x Arm Cortex R5F |

## References


- *Processor SDK Linux Software Developer's Guide Foundational Components 3.7. IPC*
- *Linux Programmer's Manual*

Texas Instruments, *AM572x Sitara Processor Technical Reference Manual*

Texas Instruments, *AM572x Sitara Processors Silicon Revision 2.0 Data Manual*

*8 Port IO-Link Master Reference Design*

*RGB Signal Light With IO-Link Interface Reference Design*

# 5 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

| Changes from Revision * (April 2019) to Revision A (May 2021) | Page |
|---|---|
| • Updated the numbering format for tables, figures and cross-references throughout the document | 1 |

# IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (https:www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265