ISSN: 2688-8645

Design and Implementation of 1553B Bus Simulator - MT

Haixin Jian, Xiaoying Yan

Xi'an Shiyou University, Xi'an Shaanxi, 710000, China

Abstract

This paper focuses on the modeling and simulation of the MT mode of the 1553B bus protocol processor B64843GC. Built on the QEMU platform, a dual-track mechanism is implemented that can be freely switched between word monitor and selective message monitor modes. It not only captures every command, data, and status word on the bus without omission, but also records only critical messages by programming multidimensional filters such as RT address, sub-address, and T/R bit, significantly reducing storage and post-analysis overhead. The system fully supports RT/MT combined mode, allowing the processor to respond to local messages as a remote terminal while still keeping full monitor coverage of communication among other terminals. Complemented by interrupt management, cyclic stack, auto-wrap, and error detection mechanisms, it performs real-time tagging of anomalies like format errors, no-response, word-count mismatch, and message supersession, forming a closed monitor loop that covers the full life cycle of "start-transfer-end". Simulation results verify the stability and real-time performance of the solution under high-load, multi-node, and complex topologies, providing an efficient, flexible, and low-cost virtual verification means for debugging, fault analysis, and data logging of 1553B systems, and can be directly applied to development and test flows in safety-critical fields such as avionics, shipboard, and vehicle platforms.

Keywords

1553B; B64843GC; MT; QEMU; ARM.

1. Introduction

MIL-STD-1553B[1] is a widely adopted digital communication bus standard in the aerospace field, dedicated to data transmission between aircraft and ground-based equipment. It is characterized by high reliability, strong anti-interference capability, and moderate data transfer rate. Traditional 1553B bus testing usually relies on costly hardware devices such as specialized testers or physical equipment[2], whereas virtualized test platforms can significantly reduce development and testing costs. By leveraging virtualization technology, developers are able to simulate the operation of complex 1553B bus devices on ordinary computers, without the need for substantial physical hardware resources. The module designed in this research project is the MT (Bus Monitor Terminal) module in the 1553B[3] protocol. With reference to the MT functionality of the B64843GC[4] chip, it is encapsulated as a functional model. In the virtualized environment, test scenarios can be configured rapidly and adjusted flexibly, and the test results feature high repeatability, facilitating problem localization and optimization.

2. Simulating ARM with QEMU

QEMU is an open-source virtual machine monitor implemented purely in software, which can run independently without hardware-assisted virtualization. It is capable of dynamically translating instructions and simulating multiple processor architectures such as x86, ARM and

ISSN: 2688-8645

PowerPC, as well as peripheral devices including network cards, graphics cards and USB controllers, supporting the booting of complete operating systems. It is commonly used in development and debugging, cloud hosting and firmware testing scenarios.As an open-source virtualization software, QEMU features cross-architecture emulation as its core advantage, which can easily meet the virtualization requirements of various processor architectures like x86 and ARM. Its unique attribute lies in the adoption of a pure software implementation scheme, enabling hardware simulation without hardware assistance: ranging from the translation and execution of CPU instructions, to the virtualized mapping of memory addresses, and then to the functional simulation of I/O devices such as network cards and storage controllers, it fully replicates the hardware environment. This allows ARM virtual machines to run stably on the virtualization layer, satisfying the needs of cross-platform testing, development and other application scenarios.

This research project leverages QEMU to virtualize an ARM platform. The experimental environment configured for this project is as follows:

System: Ubuntu 20.04;

QEMU Version: QEMU 9.0;

Virtual Platform: virt-2.10;

Linux Kernel Version: linux-6.1.111;

Root File System: busybox-1.36.1.

2.1. Installing and Configuring QEMU

2.1.1. Install the Cross-Compilation Toolchain

First, it is necessary to install QEMU and the cross-compilation toolchain on the Ubuntu system. Since the processor of the host computer is an x86 processor, while the processor used in QEMU for this research project is an ARM processor, cross-compilation is required.

Install the cross-compilation toolchain:

wget https://developer.arm.com/-/media/Files/downloads/gnu-a/10.3-2021.07/binrel/gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf.tar.xz

tar -xf gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf.tar.xz

mv gcc-arm-10.3-2021.07-x86_64-arm-none-linux-gnueabihf arm-gnu-toolchain

The decompressed directory comes with a complete toolchain including bin/arm-none-linux-gnueabihf-gcc, g++, ld, gdb and other tools.

For convenience, the folder is renamed to arm-gnu-toolchain. After the installation is completed, you can verify the installation result.

2.1.2. Install QEMU Version 9.0.0

Run the command wget https://download.qemu.org/qemu-9.0.0.tar.xz to download the package. After the download is completed, proceed with configuration and compilation.

2.2. Installing and Configuring the Kernel

The Linux kernel in this context is intended to run on QEMU. Since QEMU is configured for the ARM architecture, the kernel must be an ARM-targeted program and thus needs to be compiled using the cross-compiler downloaded earlier. The default configuration is adopted for this setup.

Download and extract the kernel source code:

wget https://mirror.bjtu.edu.cn/kernel/linux/kernel/v6.x/linux-6.1.111.tar.xz

tar -xvf linux-5.10.tar.xz

In the decompressed directory, generate the configuration file for the virt-2.10 development board and compile the kernel:

ISSN: 2688-8645

cd linux-6.1.111
make CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm vexpress_defconfig
make CROSS_COMPILE=arm-linux-gnueabihf- ARCH=arm
The generated kernel image is located at arch/arm/boot/zImage, and the device tree is located at arch/arm/boot/dts/virt.dts.

2.3. Building the Root File System

2.3.1. Download and Compile BusyBox:

wget http://www.busybox.net/downloads/busybox-1.36.1.tar.bz2
tar -xvf busybox-1.36.1.tar.bz2
cd busybox-1.36.1
make defconfigmake CROSS_COMPILE=arm-linux-gnueabi-
make install CROSS_COMPILE=arm-linux-gnueabi-

2.3.2. Create the Root Directory Structure and Copy BusyBox Files:

mkdir -p rootfs/{bin,sbin,etc,proc,sys,usr/{bin,sbin}}
cp -av /home/busybox-1.36.1/_install/* rootfs/
Then enter the rootfs directory and create a symbolic link:
cd rootfsln -sf bin/busybox init

2.3.3. Create the Device Folder and Device Files (mem, tty2, tty3, tty4):

mkdir dev
cd dev
sudo mknod -m 660 mem c 1 1
sudo mknod -m 660 tty2 c 4 2
sudo mknod -m 660 tty3 c 4 3
sudo mknod -m 660 tty4 c 4 4
In Linux, everything is a file. Accessing hardware devices essentially involves reading from and writing to device files. The /dev folder stores these device files, and the mknod command is used to create device files (also called device nodes).

2.3.4. Package the Root File System

Return to the parent directory of rootfs and package all files into rootfs.cpio.gz:
cd ../find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../rootfs.cpio.gz

2.4. Starting the QEMU Virtual Machine

sudo ./qemu-system-arm \
-M virt-2.10 \
-m 256M \
-kernel /home/linux-6.1.111/arch/arm/boot/zImage \
-initrd /home/rootfs.cpio.gz \
-dtb /home/virt.dtb \
-nographic \
-append "console=ttyAMA0 rdinit=/bin/sh"
QEMU starts successfully when the following prompts appear:
Starting kernel ...
Booting Linux on physical CPU 0x0...
Please press Enter to activate this console.

ISSN: 2688-8645

/ # ← Entered the BusyBox shell

3. MT Module

3.1. Word Monitor

In Word Monitor terminal mode, the terminal monitors both 1553B buses [5]. After completing initialization and the monitor-start sequence, it stores every command, status, and data word received from either bus. For each word received from either bus, a pair of words is written into its shared RAM.

The first word is the 16-bit data of the received word.

The second word is the Monitor Identifier (ID) or "tag" word. The ID word contains information about the bus channel, word validity, and the inter-word gap time.

The data word and ID word are stored in a circular buffer within the shared RAM address space.

Table 1. MT Memory Map

Hex Address	Function
0x0000	First received 1553 word
0x0001	First ID word
0x0002	Second received 1553 word
0x0003	Second ID word
0x0004	Third received 1553 word
0x0005 …	Third ID word …
0x0100 … 0xFFFF	Stack pointer (fixed location)

3.1.1. Word Monitor Trigger

Table 2. Word Monitor Truth Table with Trigger (Configuration Register #1, bits 10, 9, 7)

Start On Trigger (Bit 10)	Stop On Trigger (Bit 9)	External Trigger (Bit 7)	Word MT Start and Store Operation
0	0	0	Software Start, MT stores all data on the bus.
0	0	1	Software or External Trigger Start, MT stores all data on the BUS
0	1	0	Software Start, MT stores all data until the Command Word matches the word stored in the MT Trigger Register (MTR). This Command Word is the last word stored in the MT Stack.
0	1	1	Software or External Trigger Start stores all data until the Command Word matches the word stored in the MT TriggerRegister (MTR). ThisCommand Word is the last word stored in the MT Stack.
1	0	0	Software Start, stores all data after a valid Command Word matches the word stored in the MT Trigger Register (MTR). This Command Word is stored in the MT Stack.
1	0	1	Software or External Trigger Start, stores all data after a valid Command Word matches the word stored in the MT Trigger Register (MTR). This Command Word is stored in the MT Stack.
1	1	0	Software Start, stores ONLY the Command Word that matches the word stored in the MT Trigger Register.
1	1	1	Software or External Trigger Start,stores ONLY theCommand Word that matches the wordstored in the MT Trigger Register.

ISSN: 2688-8645

In Word Monitor mode [6], pattern recognition triggers and pattern recognition interrupts are available. The 16-bit comparison word for both the trigger and the interrupt is held in the MT Trigger Word register. Set the MT_PATT_TRIG bit (bit 9) in the Interrupt Mask register to enable the pattern-recognition interrupt. Set the TRIGGER_ENABLE bit (bit 11) in Configuration Register #1 and select either START-ON-TRIGGER (bit 10) or STOP-ON-TRIGGER (bit 9) to enable the pattern-recognition trigger.

3.2. Selective Message Monitor

The enhanced terminal provides a flexible interface that allows selective monitoring of 1553 messages based on RT address [7], T/R bit and sub-address with almost no intervention from the host processor. In Message-Monitor mode the enhanced terminal reproduces every command/response exchange on both 1553 bus channels A and B and, under control of user-programmable filters(RT address, T/R bit and sub-address), stores the traffic into shared RAM. The monitor may be used stand-alone or in a combined RT/Monitor mode.

The message monitor contains two stacks—a Command Stack and a Data Stack—that are independent of the BC/RT command stack. Pointers to these stacks reside at fixed locations in RAM.

In monitor-only mode the selective message monitor is started by issuing an MT-start command (i.e., setting bit 1 of the Start/Reset register).

3.2.1. Selective-Monitor Operation

Table 3. Message Monitor Selection Table Address

Bit	Description
15(MSB)	LOGIC “0”
14	LOGIC “0”
13	LOGIC “0”
12	LOGIC “0”
11	LOGIC “0”
10	LOGIC “0”
9	LOGIC “1”
8	LOGIC “0”
7	LOGIC “1”
6	RTAD_4
5	RTAD_3
4	RTAD_2
3	RTAD_1
2	RTAD_0
1	T/R*
0(LSB)	Subaddress_4

When operating in selective message monitor mode [8] the enhanced terminal, after receiving a valid command word, consults the Selective-Monitor Lookup Table (a fixed RAM area) to determine whether that command is enabled. The address of the lookup entry is computed by extracting the RT address, the T/R* bit and sub-address bit 4 from the current command word and adding them to the base address 0x0280 (see figure). The bit position within the 16-bit word at that address is given by command-word sub-address bits 3-0.

If the selected bit is logic “0” the command is disabled; the enhanced terminal aborts processing of the current message and begins searching for the next command word. (Note: the enhanced terminal may mis-interpret an RT status word as a new command word.)
If the selected bit is logic “1” the command is enabled and the enhanced terminal:

ISSN: 2688-8645

Creates an entry in the monitor command stack at the location pointed to by the Monitor Command Stack Pointer; Stores the data associated with the command in consecutive locations of the monitor data stack.

In the Selective-Monitor Lookup Table: For even addresses (Subaddress_4 = A0 = 0), bit 15 enables monitoring for sub-address 15, bit 14 for sub-address 14, ..., bit 0 for sub-address 0.For odd addresses (Subaddress_4 = A0 = 1), bit 15 enables monitoring for sub-address 31, bit 14 for sub-address 30, ..., bit 0 for sub-address 16.

Programming a bit to logic "0" disables monitoring of the corresponding RT-address / T/R* / sub-address combination; the message is ignored. Programming a bit to logic "1" enables monitoring; the message will be stored.

3.2.2. Message Monitor Formats

3.2.2.1 Monitor Operation and Memory Map

In selective message-monitor mode the enhanced terminal, after receiving a valid command word, consults the Selective-Monitor Lookup Table (a fixed RAM block) to decide whether the command is enabled. If the command is disabled, the terminal ignores the message (no storage occurs).

3.2.2.2 Reading Monitored Messages

To retrieve a monitored message from the enhanced terminal, perform the following steps:

(1) Block Status Word – determines whether the message is an RT-to-RT transfer.

(2) Command Word – identifies message format (receive, transmit, mode code, broadcast, etc.) and word count.

(3) Data Pointer – points to the data-block entry.

3.2.2.3 Data-Block Format and Storage

Different transfer types store data and status words in distinct arrangements, for example:

(1) BC-to-RT Receive transfer – contains the Status Word followed by the data words.

(2) RT-to-BC Transmit transfer – contains the data words followed by the transmitting RT's Status Word.

(3) RT-to-RT transfer – includes both the receive and transmit command words, status words, and data words.

Table 4. Selective Message Monitor Memory Map (shown for 4K RAM for "Monitor Only" Mode.)

Hex Address	Description
0x0101	Unused
0x0102	Monitor Command Stack Pointer A (fixed location)
0x0103	Monitor Data Stack Pointer A (fixed location)
0x0104–0105	Unused
0x0106	Monitor Command Stack Pointer B (fixed location)
0x0107	Monitor Data Stack Pointer B (fixed location)
0x0108–027F	Unused
0x0280–02FF	Selective Monitor Lookup Table (fixed area)
0x0300–03FF	Unused
0x0400–07FF	Monitor Command Stack A
0x0400–07FF	Monitor Data Stack A

3.3. RT/Selective Message Monitor

RT / Selective Message Monitor Mode provides full RT functionality for the enhanced terminal's own RT address while simultaneously offering selective monitoring for all other RT addresses.

ISSN: 2688-8645

When the enhanced-terminal run-time library is used, post-processing is performed to merge the separate RT and MT stacks into a single RT/MT combined stack that contains all selective-monitor activity.

As in RT mode, the selective monitor's command stack stores every command word, time-tag word, block-status word, and a data-block pointer for each monitored message. Both the command and data stacks can be programmed through bits 12–8 of Configuration Register 3.

3.3.1. RT / Selective Message Monitor Behavior

For an RT-to-RT transfer command while the Enhanced Mini-ACE is in RT/MT mode and is the receiving RT: Because the device is occupied servicing the receive command, the monitor logs no data. The RT stack sets the "RT-RT Transfer" bit in the receiving RT's Block Status Word, signalling that the received command belongs to an RT-RT sequence started by the BC.

For an RT-to-RT transfer command while the Enhanced Mini-ACE is in RT/MT mode and is the transmitting RT: The monitor stack holds one command whose Block Status Word equals 0x4000, indicating Start-of-Message (SOM).In this entry the "RT-RT Transfer" bit is cleared; the monitor treats it as the transmit command within the command-word portion of the stack entry.The monitor command stack then generates a second entry for the same RT-RT command. In this second entry the Block Status Word sets the End-of-Message (EOM), Error Flag, Format Error, and Command Word Content Error bits; the "RT-RT Transfer" bit remains cleared.

3.3.2. Combined Mode Memory Map

Table 5. RT/Selective Message Monitor Memory Map (shown for 64K RAM for combined RT/MT mode)

Address (Hex)	Description
0000-00FF	RT Command Stack A (256 words)
100	RT Command Stack Pointer A (fixed location)
101	RESERVED
102	Monitor Command Stack Pointer A (fixed location)
103	Monitor Data Stack Pointer A (fixed location)
104	Stack Pointer B (fixed location)
105	RESERVED
106	Monitor Command Stack Pointer B (fixed location)
107	Monitor Data Stack Pointer B (fixed location)
0108-010F	Mode Code Selective Interrupt Table (fixed area)
0110-013F	Mode Code data (fixed area)
0140-01BF	Lookup Table A (fixed area)
01C0-023F	Lookup Table B (fixed area)
240.0247	Busy Bit Lookup Table (fixed area)
0248-025F	(not used)
0260-027F	RT Data Block 0
0280-02FF	Selective Monitor Lookup Table (fixed area)
0300-03FF	Command Illegalizing Table (fixed area)
0400-07FF	Monitor Command Stack A (1024 words)
0800-081F	RT Data Block 0
0820-083F	RT Data Block 1
.	.
7FE0-7FFF	RT Data Block 959
8000-FFFF	Monitor Data Stack (32,768 words)

The memory map for Combined Mode is shown in Table 5. The size of the Monitor Command Stack can be programmed to 256, 1K, 4K, or 16K words through bits 11 and 12 of Configuration Register #3. The size of the Monitor Data Stack can be programmed to 512, 1K, 2K, 4K, 8K, 16K, 32K, or 64K words through bits 8 to 10 of Configuration Register #3.

ISSN: 2688-8645

3.4. Interrupt Status Queue

Selective-Monitor and Combined RT/Selective-Monitor modes implement an Interrupt Status Queue that furnishes a time-sequenced history of interrupt-generating conditions and events.

3.4.1. Queue and Filtering

The queue offers programmable filtering: only valid messages, only invalid messages, or both can create entries. Bits 8 (disable invalid) and 7 (disable valid) of Configuration Register #6 control this.

The queue is 64 words deep and can hold up to 32 monitored-message entries.

3.4.2. Queue Pointer

The pointer is held in the Interrupt Vector Queue Pointer register. The host must initialize it; thereafter the RT message handler increments it. Addresses wrap modulo-64, so the pointer automatically rolls over after the 64th word.

3.4.3. Event-triggered Two-word Entries

Each interrupt-producing event writes a two-word entry:

Word 1 – Interrupt Vector: encodes the event/condition.

Word 2 – Parameter: For message events: pointer to the first word (Block Status Word) of the RT or MT command-stack descriptor. For non-message events (RAM parity error, etc.): the offending RAM address; for RT-address parity, self-test complete, time-tag rollover, etc., the value is 0x0000.

3.4.4. Event Types

Message interrupts: End-of-Message, selected Mode Codes, Format Error, Sub-address Control Word, RT circular-buffer rollover, handshake failure, RT command-stack rollover, transmitter timeout, MT data-stack rollover, MT command-stack rollover, plus 50 % rollover variants of the above.

Non-message interrupts: Time-tag rollover, RT-address parity error, RAM parity error, self-test complete.

3.4.5. Interrupt Status Word Format

Bit 0 of the Interrupt Status Word identifies the class:

1 = message interrupt event

0 = non-message interrupt event

An entry can never represent both classes simultaneously.

4. Device Mounting

4.1. Device Creation

Register QOM type:static const TypeInfo b64843gc_info = {

```
    .name          = TYPE_B64843GC,      // Device type name
    .parent        = TYPE_SYS_BUS_DEVICE,   // System bus equipment
    .instance_size = sizeof(B64843gcState),     // The size of the equipment status structure
    .class_init    = b64843gc_class_init,       // Class initialization function
};
type_register_static(&b64843gc_info);
```

The QEMU build system scans all source files for type_register_static calls, adds the B64843GC type to QEMU's internal type table, and makes the new -device B64843GC option available after compilation.

ISSN: 2688-8645

Table 6. RT/Monitor Interrupt Status Word (for interrupt status queue)

Bit	Definition For Message Interrupt Event	Definition For Non-Message Interrupt Event
15	TRANSMITTER TIMEOUT	NOT USED
14	ILLEGAL COMMAND	NOT USED
13	MONITOR DATA STACK 50% ROLLOVER	NOT USED
12	MONITOR DATA STACK ROLLOVER	NOT USED
11	RT CIRCULAR BUFFER 50% ROLLOVER	NOT USED
10	RT CIRCULAR BUFFER ROLLOVER	NOT USED
9	MONITOR COMMAND (DESCRIPTOR) STACK 50% ROLLOVER	NOT USED
8	MONITOR COMMAND (DESCRIPTOR) STACK ROLLOVER	NOT USED
7	RT COMMAND (DESCRIPTOR) STACK 50% ROLLOVER	NOT USED
6	RT COMMAND (DESCRIPTOR) STACK ROLLOVER	NOT USED
5	HANDSHAKE FAIL	NOT USED
4	FORMAT ERROR	TIME TAG ROLLOVER
3	MODE CODE INTERRUPT	RT ADDRESS PARITY ERROR
2	SUBADDRESS CONTROL WORD EOM	PROTOCOL SELF-TEST COMPLETE
1	END-OF-MESSAGE (EOM)	RAM PARITY ERROR
0	"1" FOR MESSAGE INTERRUPT EVENT; "0" FOR NON-MESSAGE INTERRUPT EVENT	

4.2. Attaching to the System Bus

hw/arm/virt.c – add create_64843():
Prior declaration:[VIRT_B64843GC] = 45

```
static void create_64843(const VirtMachineState *vms)
{
   DeviceState *dev = qdev_new(TYPE_B64843GC);
   sysbus_realize_and_unref(SYS_BUS_DEVICE(dev), &error_fatal);
       /* Register region: 0x0B00_0000, length 0x100 */
       sysbus_mmio_map(SYS_BUS_DEVICE(dev), 0, 0x0b000000);
       /* Wire IRQ-45 to the GIC */
       int irq = vms->irqmap[VIRT_B64843GC];
     sysbus_connect_irq(SYS_BUS_DEVICE(dev), 0,
                qdev_get_gpio_in(vms->gic, irq));
}
```

The guest physical address range starting at 0x0b00_0000 (256 B) is mapped as the device register area, and interrupt line 45 is connected to the GIC.

ISSN: 2688-8645

4.3. Updating the Device Tree

Modify the previously generated device-tree snippet to declare the device at base address 0x0B000000, 256 bytes in size, and interrupt number 45:

```
zdev@1000 {
  compatible = "zdev, interrupt-device";
  reg = <0x0 0x0B000000 0x0 0x100>;
  interrupts = <0 45 4>;
};
```

4.4. Driver Development

4.4.1. Device Registration

When the kernel boots it scans the device tree for any node whose compatible string is "zdev,interrupt-device".

It then looks up its driver database and finds that this hardware is handled by example_driver. The kernel automatically invokes the driver's probe() routine, which performs all necessary hardware initialization; at this point the driver officially "takes over" the device. If the driver is later unloaded or the hardware is removed, the kernel automatically calls the driver's remove() routine to clean up, achieving fully automatic unload.

```
static const struct of_device_id example_of_match[] = {
  { .compatible = "zdev,interrupt-device" },
  {}
};
MODULE_DEVICE_TABLE(of, example_of_match);
static struct platform_driver example_driver = {
  .probe = example_probe,
  .remove = example_remove,
  .driver = { .of_match_table = example_of_match },
};
```

4.4.2. Register Read/Write

4.4.2.1 Read Operation

The read logic is implemented in device_read().User-space issues read() on /dev/b64843dev0.

The driver first reads the value from the pre-programmed register address that was set during a previous write operation (commands 10-12).

Address convention: the read does not carry an explicit address; it always uses the read_addr established by the last write—this is a driver-level contract.

Function used: ioread16(), the kernel helper for 16-bit register reads; it avoids the hazards of direct pointer dereference.

Data returned: the 16-bit value is split into two bytes (low byte first, high byte second) and copied to user-space, which re-assembles the full 16-bit word.

4.4.2.2 Write Operation

The write logic is implemented in device_write().

User-space issues write() on /dev/b64843dev0 with a 9-byte packet:

– Byte 0: command type

– 0 = write REG

– 1 = write MEM

– 2 = write GPIO

ISSN: 2688-8645

– Bytes 1-4: register offset (base-address + offset → physical address)
– Bytes 5-6: 16-bit data to write (bytes 7-8 reserved/unused)

The driver uses iowrite16() to commit the 16-bit value to the target register—safer than direct pointer assignment and immune to cache/endian issues.

After every write the driver calls iounmap() to release the temporarily mapped virtual address; register accesses are short-lived and do not keep mappings permanently.

4.4.3. Interrupt Handling

During initialization, example_probe extracts the IRQ number from the device tree and registers example_interrupt_handler with request_irq(), binding the interrupt line to the driver.

User space enables asynchronous notification via fcntl(), which invokes button_fasync() to initialize button_async and establish the signal channel.

When the device asserts the interrupt, example_interrupt_handler runs and calls kill_fasync() to deliver SIGIO to the registered user-space process(es).

A debug sysfs entry /sys/kernel/example_interrupt/trigger is provided; writing to it invokes trigger_interrupt() for manual interrupt generation.

On driver removal, example_remove calls free_irq() to release the interrupt line and prevent resource leakage.

4.5. Socket Communication

Each 1553B device is assigned an IP address and a port number; devices exchange data through UDP/TCP sockets.

5. Testing

The 1553B bus simulator was used to exercise the MT block. Three terminal windows were opened, each assigned an IP address and port number, and then launched as separate QEMU VMs configured as BC, RT, and MT respectively. Custom test code exercised all three MT operating modes; every MT function defined in the B64843GC data sheet was verified, and message traffic fully complied with MIL-STD-1553B.

On the live bus, the complete MT sub-feature set—selective monitoring, stack rollover, interrupt queue, error detection, external trigger, etc.—was validated with 100 % test coverage; observed behavior matches the official data-sheet exactly.

The figure shows the MT response when a message's RT address is not listed in the Selective-Monitor lookup table.

6. Summary

This model completely reproduces the internal logic and timing of the B64843GC Message Monitor (MT) module in pure software. Running as a process/thread on a virtualized platform, it delivers full 1553B bus monitoring without any physical hardware. Comparative tests show that the software model equals or surpasses traditional cards in key metrics such as message-capture ratio, timestamp accuracy, and interrupt latency, while offering easy deployment, effortless scalability, and simultaneous multi-card simulation. It provides a highly reliable, low-cost solution for large-scale simulations and holds significant engineering value and broad commercial promise.

ISSN: 2688-8645

References

[1] LIANG Shuangyu, LI Tiantian, SHAO Yunsong. Design of Simulation Board Based on 1553B Protocol IP Core[J]. Integrated Circuit Applications, 2025, 42(01):100-101. DOI:10.19339/j.issn.1674-2583.2025.01.034.

[2] WANG Yuetao, ZHAO Dongqing, WU Huijun. Design and Verification of 1553B Bus RT/MT Functions Based on FPGA and B61580S3[J]. Electronic Design Engineering, 2023, 31(02): 179-183. DOI: 10.14022/j.issn1674-6236.2023.02.038.

[3] ZHANG Di, JIANG Zhidong, GAO Weiwei, et al. Design and Implementation of an Integrated 1553B Bus Comunication Simulation Platform[J]. Electronic Design Engineering, 2022, 30(10):139-143+149. DOI:10.14022/j.issn1674-6236.2022.10.029.

[4] ZHU Xisong, WANG Jianjun, LIU Shiquan. Application of 1553B Bus Based on FPGA and 64843[J]. Electronic Quality, 2016, (03):60-63.

[5] YANG Weiwei, YANG Ping. Design of MIL-STD-1553B Bus Remote Terminal Based on BU-64843[J]. Computer Measurement & Control, 2015, 23(11):3837-3838+3852. DOI: 10.16526/j.cnki.11-4762/ tp.2015.11.080.

[6] LIU Dun. CAN and 1553B Bus Equipment and Network Virtualization Technology[D]. Zhejiang University, 2018.

[7] WEN Feng, DU Zhimei, XUE Zhichao, et al. Function Implementation of Bus Monitor Based on BU-61580 Transparent Mode[J]. Journal of Test and Measurement Technology, 2021, 35(01):74-78.

[8] SUN Quanyu, SHAO Guowei. Design of Dual-Channel 1553B Remote Terminal Conversion Module Based on FPGA and BU-61580[J]. Industrial Control Computer, 2020, 33(05):29-31.