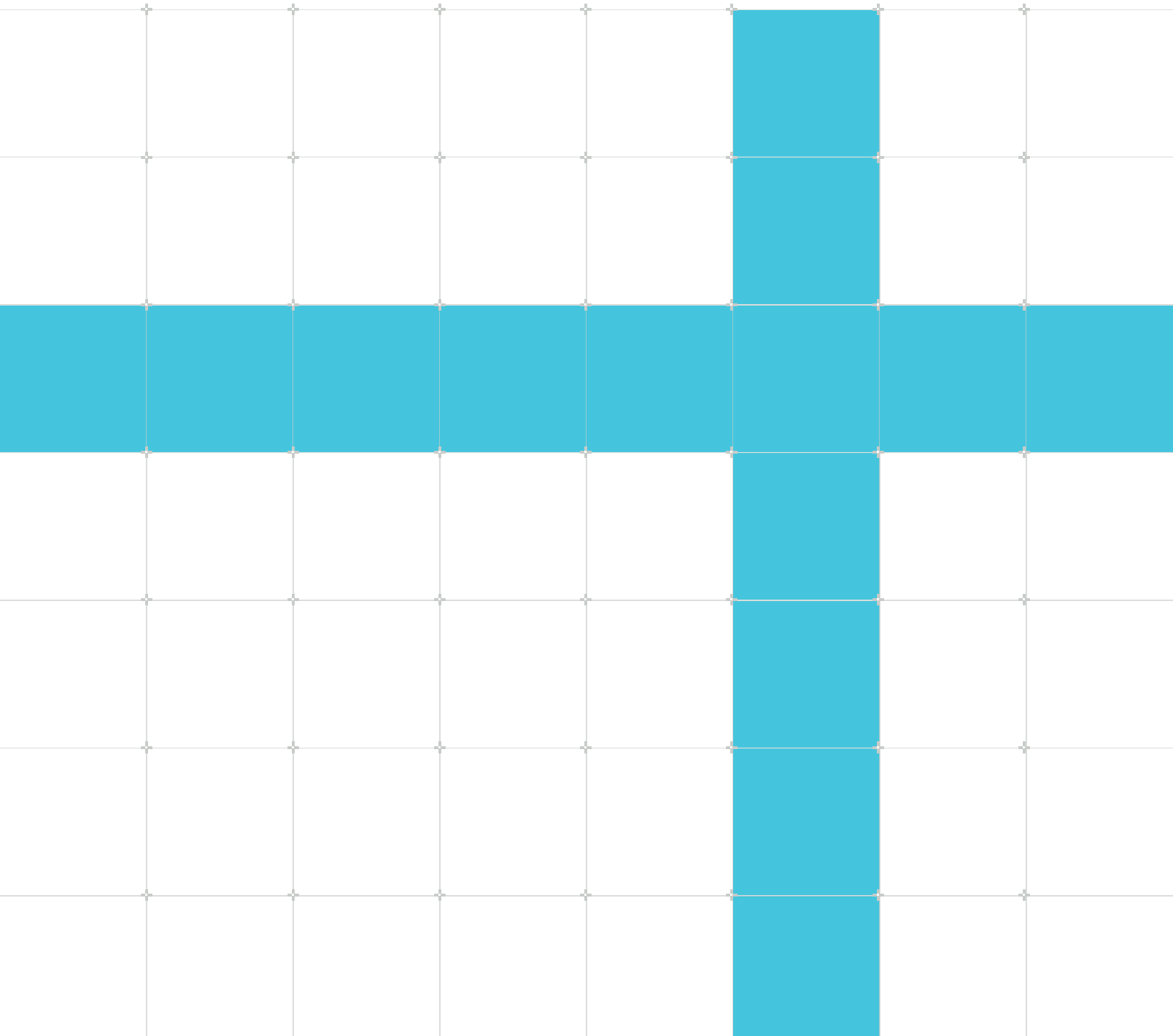
Creating an event-driven embedded image

Version 1.0

Issue 03
102439_0100_03_en



Creating an event-driven embedded image

Release information

Document history

Issue	Date	Confidentiality	Change
0100-03	1 January 2021	Non-Confidential	Initial release

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349)

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm® welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Overview

This guide shows how to write event-driven embedded system code.

This guide is the third in a collection of related guides:

- Building your first embedded image
- Retargeting output to UART
- Creating an event-driven embedded image (this guide)
- Changing Exception level and Security state in an embedded image

Embedded systems typically monitor inputs waiting for an event, which then triggers a response by the system. You need to write code that listens for these events and acts on them. For example, an embedded system in a thermostat might monitor room temperature until it drops below a specified threshold. When the threshold is reached, the embedded system turns on the heating system.

To add meaningful functionality to an embedded system, you must enable asynchronous exceptions: IRQs, FIQs, and SErrors. This guide does not explore all the relevant architectural features, but a guide and online course are available for readers who are not familiar with them. Asynchronous exceptions are taken when the CPU needs to handle something that is external to the current flow of execution. For example, if a user flips a power switch, the processor must stop what it is doing, and branch to a handler that ensures that the shutdown is done correctly.

By the end of this guide, you will have written an event-driven program that:

1. Configures the physical timer within the CPU to generate an exception in a few seconds.
2. Once running, the code waits until the timer interrupt occurs.
3. When the exception occurs, it is dealt with by the exception handler, and a suitable message is sent to the UART interface.

Before you begin

To complete this guide, you will need to have Arm Development Studio Gold Edition installed. If you do not have Arm Development Studio, you can download a 30-day free trial.

Arm Development Studio Gold Edition is a professional quality tool chain developed by Arm to accelerate your first steps in Arm software development. It includes both the Arm Compiler 6 toolchain and the FVP_Base_Cortex-A73x2-A53x4 model used in this guide. We will use the command-line tools for most of the guide, which means that you will need to configure your environment in order to run Arm Compiler 6 from the command-line.

The individual sections of this guide contain some code examples. These code examples are available to download as a ZIP file:

- CommonTasks-EventDrivenEmbeddedImage.zip

Reviewing the summary of the instructions and registers in the architecture might be useful to help you understand this guide.

2. Configure exception routing

To keep things simple, this guide will specify that all exceptions are taken at the highest exception level, EL3.

A summary of the instructions and registers in the architecture will help you to understand this section of the guide. In addition, you must keep in mind some rules that exceptions obey:

- An exception routed to a higher exception level cannot be masked, apart from EL0 to EL1 which can be masked with PSTATE.
- An exception routed to a lower exception level is always masked.
- An exception routed to the current exception level can be masked by PSTATE.

To configure exception routing, you need to perform the following tasks:

- Configure the Secure Configuration Register, SCR_EL3, to enable exception routing to EL3.
- Set the Vector Based Address Register, VBAR_EL3, to point to a vector table.
- Disable masking (or ignoring) of exceptions at EL3 by PSTATE.

To do these things, add this code to `startup.s`:

```
// Configure SCR_EL3
// -----------------
MOV  w1, #0              // Initial value of register is unknown
ORR  w1, w1, #(1 << 3)   // Set EA bit (SError routed to EL3)
ORR  w1, w1, #(1 << 2)   // Set FIQ bit (FIQs routed to EL3)
ORR  w1, w1, #(1 << 1)   // Set IRQ bit (IRQs routed to EL3)
MSR  SCR_EL3, x1


// Install vector table
// --------------------
.global vectors
LDR  x0, =vectors
MSR  VBAR_EL3, x0
ISB

// Clear interrupt masks
// ---------------------
MSR  DAIFClr, #0xF
```

3. Configure the vector table

When any exception is taken, the processor must handle the exception correctly. This means that the processor must detect the type of exception, then branch to an appropriate handler.

Let's look at the code in `vectors.s`. Here we can see how a vector table is configured to branch `fiqFirstLevelHandler` when an FIQ event occurs:

```
   .section  VECTORS,"ax"
   .align 12

   .global vectors
 vectors:

   .global fiqHandler

 // -----------------------------------------------------------
 // Current EL with SP0
 // -----------------------------------------------------------

   .balign 128
 sync_current_el_sp0:
   B        .                    //        Synchronous

   .balign 128
 irq_current_el_sp0:
   B        .                    //        IRQ

   .balign 128
 fiq_current_el_sp0:
   B        fiqFirstLevelHandler //        FIQ

   .balign 128
 serror_current_el_sp0:
   B        .                    //        SError

   //
   // NB, CODE OMITTED!
   //

fiqFirstLevelHandler:
  STP      x29, x30, [sp, #-16]!
  STP      x18, x19, [sp, #-16]!
  STP      x16, x17, [sp, #-16]!
  STP      x14, x15, [sp, #-16]!
  STP      x12, x13, [sp, #-16]!
  STP      x10, x11, [sp, #-16]!
  STP      x8, x9, [sp, #-16]!
  STP      x6, x7, [sp, #-16]!
  STP      x4, x5, [sp, #-16]!
  STP      x2, x3, [sp, #-16]!
  STP      x0, x1, [sp, #-16]!

  BL       fiqHandler

  LDP      x0, x1, [sp], #16
  LDP      x2, x3, [sp], #16
  LDP      x4, x5, [sp], #16
  LDP      x6, x7, [sp], #16
  LDP      x8, x9, [sp], #16
  LDP      x10, x11, [sp], #16
  LDP      x12, x13, [sp], #16
  LDP      x14, x15, [sp], #16
  LDP      x16, x17, [sp], #16
  LDP      x18, x19, [sp], #16
```

```
LDP      x29, x30, [sp], #16
ERET
```

In this case, only an FIQ entry and handler have been defined. In fact, `fiqFirstLevelHandler` merely branches to `fiqHandler`, a procedure that we will define later in C. Here, we have used the `BL` instruction, or branch with link, which branches to the given label. The `BL` instruction saves the current value of the program counter plus four bytes, the next instruction address, to register x30.

Our C procedure ends with a return statement which compiles to a `RET` instruction. By default (that is, if no other register is specified) `RET` branches to the address stored in register x30.

The handler also saves and restores all the general-purpose registers. This is because these registers may be modified by the procedure call.

The remaining entries branch to self, because the handlers have not been written. When you create your own image, you will need to define and configure your vector table to handle all required event types.

4. Configure the interrupt controller

Vector tables have a relatively small and fixed number of entries, because the number and type of exceptions are architecturally defined.

But we might require a great number of different interrupts that are triggered by different sources. An additional piece of hardware is needed to manage these interrupts. The Arm Generic Interrupt Controller (GIC), does exactly this. We will not discuss the GIC and its features in this guide, but you can learn more in our programmers guide.

In `gic.s`, add the following code to enable the GIC, and define a source of interrupts from the physical timer:

```
  .global gicInit
  .type gicInit, "function"
gicInit:
  // Configure Distributor
  MOV      x0, #GICDbase  // Address of GIC

  // Set ARE bits and group enables in the Distributor
  ADD      x1, x0, #GICD_CTLRoffset
  MOV      x2,     #GICD_CTLR.ARE_NS
  ORR      x2, x2, #GICD_CTLR.ARE_S
  STR      w2, [x1]

  ORR      x2, x2, #GICD_CTLR.EnableG0
  ORR      x2, x2, #GICD_CTLR.EnableG1S
  ORR      x2, x2, #GICD_CTLR.EnableG1NS
  STR      w2, [x1]
  DSB      SY

  // Configure Redistributor
  // Clearing ProcessorSleep signals core is awake
  MOV      x0, #RDbase
  MOV      x1, #GICR_WAKERoffset
  ADD      x1, x1, x0
  STR      wzr, [x1]
  DSB      SY
1:   // We now have to wait for ChildrenAsleep to read 0
  LDR      w0, [x1]
  AND      w0, w0, #0x6
  CBNZ     w0, 1b

  // Configure CPU interface
  // We need to set the SRE bits for each EL to enable
  // access to the interrupt controller registers
  MOV      x0, #ICC_SRE_ELn.Enable
  ORR      x0, x0, ICC_SRE_ELn.SRE
  MSR      ICC_SRE_EL3, x0
  ISB
  MSR      ICC_SRE_EL1, x0
  MRS      x1, SCR_EL3
  ORR      x1, x1, #1  // Set NS bit, to access Non-secure registers
  MSR      SCR_EL3, x1
  ISB
  MSR      ICC_SRE_EL2, x0
  ISB
  MSR      ICC_SRE_EL1, x0

  MOV      w0, #0xFF
  MSR      ICC_PMR_EL1, x0 // Set PMR to lowest priority
```

```
  MOV      w0, #3
  MSR      ICC_IGRPEN1_EL3, x0
  MSR      ICC_IGRPEN0_EL1, x0

//--------------------------------------------------------------
//Secure Physical Timer source defined
  MOV      x0, #SGIbase       // Address of Redistributor registers

  ADD      x1, x0, #GICR_IGROUPRoffset
  STR      wzr, [x1]          // Mark INTIDs 0..31 as Secure

  ADD      x1, x0, #GICR_IGRPMODRoffset
  STR      wzr, [x1]          // Mark INTIDs 0..31 as Secure Group 0

  ADD      x1, x0, #GICR_ISENABLERoffset
  MOV      w2, #(1 << 29)     // Enable INTID 29
  STR      w2, [x1]           // Enable interrupt source

  RET

// -------------------------------------------------------------

  .global readIAR0
  .type readIAR0, "function"
readIAR0:
  MRS       x0, ICC_IAR0_EL1  // Read ICC_IAR0_EL1 into x0
  RET

// -------------------------------------------------------------

  .global writeEOIR0
  .type writeEOIR0, "function"
writeEOIR0:
  MSR        ICC_EOIR0_EL1, x0 // Write x0 to ICC_EOIR0_EL1
  RET
```

We have defined the functions `readIAR0()` and `writeEOIR0()`, using the `.global` and `.type` assembler directives. The `.global` directive makes the label visible to all files given to the linker, while the `.type` directive allows us to declare that the label is a function.

As you will see in Rebuild and test, when you modify `hello_world.c` to call these functions, using these directives lets us call these assembly functions from C code, following the Procedure Call Standard (PCS). The PCS defines a number of things, including how values are passed and returned. In particular:

- Arguments are passed in `x0` to `x7` in the same order as the function prototype.
- Values are returned to the registers `x0` and `x1`.

Using `readIAR0()`, we read the value of the Interrupt Controller Interrupt Acknowledge Register 0, ICC_IAR0_EL1. The lower 24 bits of this register give the interrupt identifier, INTID. By calling `readIAR0()` in C, we can get the INTID from the GIC and then handle different interrupts case by case. Later in the C code `fiqHandler()` is defined, and you will see a call to `writeEOIR0()`. The INTID is passed to `x0`. INTID is then written to the Interrupt Controller End of Interrupt Register 0, ICC_EOIR0_EL1, which tells the processor that that interrupt is complete.

5. Configure the timer to generate interrupts

So far, we have enabled the GIC, and defined a source of interrupts from a secure physical timer. We have a system timer, which we read using a comparator in the processor. We can also tell the hardware to generate an interrupt request after a set number of system ticks. Now we need a way to disable the comparator, so that it does not continue to interrupt the processor after the ticks have elapsed.

To enable the timer and define its behavior, add this code to `timer.s`:

```
  .section  AArch64_GenericTimer,"ax"
  .align 3

// -------------------------------------------------------------

  .global setTimerPeriod
  // void setTimerPeriod(uint32_t value)
  // Sets the value of the Secure EL1 Physical Timer Value Register (CNTPS_TVAL_EL1)
  // w0 - value - The value to be written into CNTPS_TVAL_EL1
  .type setTimerPeriod, "function"
setTimerPeriod:
  MSR     CNTPS_TVAL_EL1, x0
  ISB
  RET

// -------------------------------------------------------------

  .global enableTimer
  .type enableTimer, "function"
enableTimer:
  MOV    x0, #0x1            // Set Enable bit, and clear Mask bit
  MSR    CNTPS_CTL_EL1, x0
  ISB
  RET

// -------------------------------------------------------------

  .global disableTimer
  .type disableTimer, "function"
disableTimer:
  MSR    CNTPS_CTL_EL1, xzr // Clear the enable bit
  ISB
  RET
```

6. Rebuild and test

Next we need to modify `hello_world.c` to call the assembly functions we created in the earlier steps:

```
#include <stdio.h>
#include <stdint.h>
#include "pl011_uart.h"

extern void gicInit(void);
extern uint32_t readIAR0(void);
extern void writeEOIR0(uint32_t);

extern void setTimerPeriod(uint32_t);
extern void enableTimer(void);
extern void disableTimer(void);

volatile uint32_t flag;

int main () {
  uartInit((void*)(0x1C090000));
  gicInit();

        printf("hello world\n");

  flag = 0;
  setTimerPeriod(0x1000);  // Generate an interrupt in 1000 ticks
  enableTimer();

  // Wait for the interrupt to arrive
  while(flag==0){}

  printf("Got interrupt!\n");

        return 0;
}

void fiqHandler(void) {
  uint32_t intid;
  intid = readIAR0(); // Read the interrupt id

  if (intid == 29) {
    flag = 1;
    disableTimer();
  } else {
    printf("Should never reach here!\n");
  }

  writeEOIR0(intid);
        return;
}
```

Here we have defined `fiqHandler()` to produce the desired behavior when the interrupt is triggered.

Build and run the project using these instructions:

```
$ armclang -c -g --target=aarch64-arm-none-eabi startup.s
$ armclang -c -g --target=aarch64-arm-none-eabi vectors.s
$ armclang -c -g --target=aarch64-arm-none-eabi gic.s
$ armclang -c -g --target=aarch64-arm-none-eabi timer.s
$ armclang -c -g --target=aarch64-arm-none-eabi hello_world.c
```

```
$ armclang -c -g --target=aarch64-arm-none-eabi pl011_uart.c
$ armlink --scatter=scatter.txt --entry=start64 startup.o vectors.o gic.o timer.o
 hello_world.o pl011_uart.o
```

If you include the flag `-C bp.refcounter.non_arch_start_at_default=1`, the system counter on the model is enabled. If you run the image now, you will see:

```
$ FVP_Base_Cortex-A73x2-A53x4 -C bp.refcounter.non_arch_start_at_default=1 -a
 __image.axf
```

Figure 6-1: Terminal showing output of running test

7. Related information

Here are some resources related to material in this guide:

- Architecture specifications
- Arm Community
- ARM Cortex-A Series Programmer's Guide for ARMv8-A (chapter on exception handling)
- Arm Generic Interrupt Controller Architecture Specification
- The Armv8-A Architecture training course
- GICv3 and GICv4 Software Overview
- Architecture exploration tools
- Procedure Call Standard for the ARM 64-bit Architecture (AArch64)
- Armv8-A Learn the Architecture series of guides

8. Next steps

This guide is the third in a series of four guides on the topic of building an embedded image.

This guide introduced the configuration of exception routing, the vector table, and the interrupt controller, and how to configure the timer to generate interrupts.

You can continue learning about building an embedded image in the next guide in the series:

- Changing Exception Level and Security State in an Embedded Image

In case you missed them, the previous guides in the series are:

- Building your First Embedded Image
- Retargeting Output to UART