

The ultimate hands-on eBook:

Getting started with RISC-V by IAR

From curiosity to mastery



Contents

About this eBook	4
Foreword	5
About the Author	5
1. Introduction	7
1.1 Why RISC-V?	7
1.2 Open-Source ISA and RISC-V	8
1.3 What is RISC?	8
1.4 RISC-V instruction set overview	9
1.4.1 RISC-V instruction set basics	9
1.4.2 Instruction extensions and custom instruction	11
1.4.3 General-Purpose registers & Floating-Point registers	11
1.4.4 CSR(Control and Status Register)	12
1.4.5 Modes of operation	12
1.4.6 Simple assembler instructions	13
1.5 Profiles	13
1.5.1 RISC-V profiles	14
1.5.2 RISC-V platform specification	15
1.6 Why should we use RISC-V?	15
1.7 Organization of this document	16
2. Basic operation of the EWRISCV development environment	18
2.1 Precautions when using EWRISCV	18
2.2 Create a project (sample 1)	18
2.2.1 Creating and running a new project	18
2.2.2 Project structure	24
2.2.3 About the manual	24
2.3 Options	26
2.3.1 General Options	26
2.3.2 C/C++ Compiler	32
2.3.3 Output converter	37
2.3.4 Linker	38



2.3.5 Debugger	41
2.4 Understanding the RISC-V project as a whole	42
2.4.1 Creating sample 2	42
2.4.2 Running sample 2	43
2.4.3 About GP relative	45
2.5 C extension instructions	48
2.6 M Extension instructions	50
2.6.1 Creating sample 3	50
2.6.2 Enabling M extension instructions	51
2.6.3 RV32M	53
2.7 A extension instructions	54
2.7.1 Creating sample 4 with A extension instructions	55
2.8 N extension instructions	57
2.9 Custom instructions	58
2.9.1 Opening the IAR information center examples	58
2.9.2 RISC-V operation codes	59
2.9.3 Custom instruction	59
2.9.4 Using custom instructions in code	60
2.9.5 Using custom instructions in the simulator	61
2.10 About function calls/ABIs	62
2.10.1 C language functions	62
2.10.2 Rules for calling functions	63
2.11 About the output of EWRISC-V	64
2.11.1 Executables/libraries	64
2.11.2 Object files	64
2.11.3 List files	64
2.11.4 Browse files	64
2.11.5 MAP files	65
3. Learn RISC-V on real hardware	77
3.1 Using the GigaDevice GD32VF103	77
3.1.1 Debug probe connection	77
3.1.2 Checking the connection with IAR I-jet	79
3.1.3 LED blinking: creating sample 5 using GPIO	81
3.1.4 Set up the debugger and start running	86
3.1.4 Learning about interrupts	88
3.1.5 Let's check the CSR	94
3.2 Using the Renesas FBP-R9A02G021 board	105
3.2.1 Generating and debugging an example project	106
4. Navigating RTOS, automated workflows, and code quality	112
5. Conclusion	115
References	115

About this eBook

Infineon, Qualcomm, Nordic Semiconductor, Bosch, and NXP have already declared their joint intent to explore the RISC-V architecture.

Now, with Renesas announcing the availability of a general-purpose RISC-V MCU, the heat is on, and embedded developers worldwide must quickly skill up.

This is “The ultimate hands-on guide: Getting Started with RISC-V by IAR.”

For numerous years, developers have highly appreciated RISC-V, particularly its open architecture, customization, scalability, and community support. Companies like SiFive, Andes, and GigaDevice already operate a respectable and sizeable business manufacturing and selling RISC-V cores and devices, but still the commercial lift-off has been somewhat missing - until now.

The ecosystem surrounding RISC-V is swiftly maturing, which is also the obvious case when looking at recently announced industry partnerships. As more devices are put to market, performance and efficiency are expected to improve as competition sharpens. Lastly, large vendors like Renesas would not enter the RISC-V arena unless they confidently predict demand. Consequently, and it's been a long way coming, now the cornerstones for RISC-V lift-off appear to be in the right position.

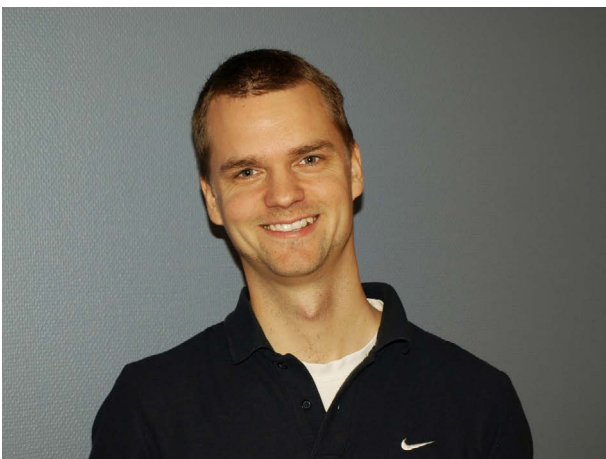
At IAR, we have spent the past 40 years building

solutions and toolchains for a very large number of architectures and devices, serving embedded developers and organizations around the world. This guide is structured around use cases with IAR Embedded Workbench for RISC-V, including references to our Static Analysis tool, IAR C-STAT for RISC-V. As you go through the guide, we strongly recommend that you [download a free evaluation copy](#) of the software.

IAR's solutions for RISC-V are Functional Safety-certified and compliant with standards such as ISO 26262 for automotive applications and IEC 61508 for industrial automation. The solution can be operated by a developer, in a CI pipeline with automated workflows or a combination of both. You decide how you run it: local, virtual, or cloud, we call it Open Choice.

We hope that you enjoy this exhaustive free guide; feel free to share it with peers, and don't hesitate to reach out should you have any additional questions. If you want to dive into more of our RISC-V tips and tricks and best practices, go to our RISC-V thought leadership section located [here](#).

I sincerely hope you are ready to immerse yourself in RISC-V and wish you a fun coding and skill-up experience!



Niklas Källman,
Senior Product Manager,
RISC-V solutions, IAR Systems

Foreword

This book is intended for developers and professionals developing embedded software using RISC-V. We will explain RISC-V features and its capabilities when using IAR Embedded Workbench for RISC-V (from now on referred to as EWRISCV). Thus, to understand, we will also look at the behavior of the CPU's instruction set and stack. The reader is expected to have a basic knowledge of CPUs and understand the basic functionality of assembler instructions. This book employs the C programming language and assembly language. It provides an introduction to C programming that is necessary for the

context of this book. As for assembly language, it focuses on the essentials needed for the RISC-V architecture. For more in-depth knowledge of assembly language, readers are encouraged to consult additional resources.

The evaluation version of IAR Embedded Workbench for RISC-V is available free of charge at <https://www.iar.com/products/architectures/risc-v/iar-embedded-workbench-for-risc-v/>.

With this book, we encourage the reader to learn about the RISC-V architecture while using IAR's RISC-V solution.

About the Author:

Hiroki Akaboshi is a senior field application engineer employed by IAR in Tokyo, Japan. He has achieved numerous academic degrees, including a B.E. (1991), M.E. (1993), and D.E. (1996) in computer science from Kyushu University.

Hiroki's research interests primarily concern CPU architectures and compilers, but his passion lies in embedded software development. For more than 20 years, he has channeled that passion into a wide range of projects focused on safety and functionality in the automotive industry.





1. Introduction



1. Introduction

1.1 Why RISC-V?

RISC-V is an open-source instruction set architecture. Processors and microcontrollers using the RISC-V instruction set are also often referred to as RISC-V. RISC-V was first developed by Krste Asanović in 2010 at the University of California, Berkeley. RISC-V is the fifth RISC instruction set (Instruction Set Architecture: ISA) coming from the University of California, Berkeley. RISC is an acronym for Reduced Instruction Set Computer, which refers to a simple instruction set.

RISC-V has not come out of the blue but has been made with decades of research and practical application in mind. For example, in 1990, David Patterson and John Hennessy published the book “Computer Architecture: A Quantitative Approach”[1] which introduced the DLX architecture, the ancestor of RISC-V. The DLX in this book was used to a certain extent in education. These two professors have popularized RISC in a general sense. John Hennessy had launched MIPS at that time, so there was not much movement from a business perspective with DLX. In this way, the technical content that led to RISC-V already existed 30 years ago. Another significant point is that RISC-V is a project on which David Patterson is involved and continues to distribute RISC-V information (Ref. 7). Since this project is based on the experience and collaboration of such experts, RISC-V solves many problems in legacy instruction sets. For example, the following points have been improved in RISC-V.

- In contrast to CPUs that have been forced to expand the address space, 32-bit, 64-bit, and 128-bit CPUs have been considered from the beginning.
- The instruction set is modular, allowing you to choose to implement only what you need, allowing you to use limited hardware efficiently.
- Elimination of delayed branching, lazy loading, and other mechanisms that assume a pipeline for single instruction execution.

RISC-V International was established in 2015 to promote RISC-V and moved to Switzerland in 2019. RISC-V International publishes the ISA specifications for RISC-V, which are developed by RISC-V International members.

The reason why RISC-V is widely discussed is that the ISA (instruction set) is open source. The specification is distributed under Creative Commons Attribution 4.0 International and is licensed under a license that may be distributed or modified. Please note that because the ISA is open, it does not mean that hardware or IP (Intellectual Property) is free. Since the ISA specification is open, there is no problem no matter who makes a microcontroller with the ISA. Therefore, various companies, institutions, universities, etc., have implemented RISC-V CPUs.

In some cases, design data is released as open source, mainly by universities, but not all designs are open to the public, and not all designs are free. Several companies sell RISC-V IP cores. Examples include Andes Technology, SiFive, Inc., and Codaip. If the instruction set is fixed, you might think that the implementation will be the same, but this is not the case; for example, the number of stages of the pipeline, the number of instructions issued at the same time, the cache, memory access, branch prediction, etc., are each determined by each company.

Furthermore, many people relate GCC (the GNU C Compiler) to open source. Also for RISC-V, GCC is supported and is required for building the Linux kernel and applications. Many people think that GCC is required when developing software with RISC-V, but compilers are also required on a case-by-case basis in developing embedded systems. For example, in systems requiring functional safety, the user must validate and certify the compiler tools, and this is quite tricky. IAR provides compilers certified by a third party, significantly reducing users' time and effort. IAR's compiler is specialized for embedded systems, which also benefits in terms of code size and execution speed.

1.2 Open-Source ISA and RISC-V

One of the features of RISC-V is the open-source licensed instruction set (ISA), but what are the benefits? Many MCUs (Micro Controller Units) and MPUs (Micro Processor Units) sold as chips worldwide are owned by a specific company.

For example, Renesas' RX family is defined by Renesas, which implements its vendor-specific ISAs and sells them as chips. In addition, Arm sells Arm's Cortex-M/Cortex-R/Cortex-A CPU cores as IP (Intellectual Property) to semiconductor manufacturers, who use the IP to make and sell chips. Such MCUs and MPUs have vendor-specific ISAs. Vendor-specific ISAs cannot be added or modified by the user using them.

In the case of Arm, there was a contract to use the Arm IP as it is, and an architecture license that allowed you to develop it yourself. In addition, starting in 2020, the Cortex-X Custom Program has been added, and it is possible to make a contract that can be customized. In contrast to these vendor-specific ISAs, RISC-V has the advantage of being an open-source ISA. For example, you are free to add special instructions to implement your application efficiently. In a vendor-specific ISA, it is generally difficult for users to add their own instructions. In the case of Arm, it may be possible to make a contract, but the user cannot do it alone.

However, there are quite a few projects that not only have an open instruction set but also an open IP. So why is RISC-V getting so much attention? Only creating an open ISA is not that useful. One reason is that RISC-V has successfully built a group of friends called an ecosystem. As of 2023, many companies are participating in the ecosystem, from hardware to software. For example, IAR released Embedded Workbench for RISC-V in 2019, and it has been upgraded 11 times by March 2023. In addition, it has also been released in 2021 and 2023 as a functionally safety-certified toolchain for RISC-V.

If you look at IAR Embedded Workbench for RISC-V, you can also introduce examples such as Azure RTOS ThreadX, FreeRTOS example projects, and SAFERTOS. Considering embedded systems, having a compiler and RTOS allows you to start development with peace of mind. IAR is just one example, but having such an ecosystem will enable us to provide a more comprehensive service. On the other hand, if the RISC-V business is not established in the long term, no companies will be participating in the ecosystem. From that point of view, RISC-V, which is an open ISA, has become a business target.

1.3 What is RISC?

(Reduced Instruction Set Computer)? RISC is an acronym for Reduced Instruction Set Computer. What is RISC? Let me briefly explain the question. In the world of CPUs (computers), an instruction set is defined, and a program is created using that instruction set (side by side). To make the CPU do faster and more sophisticated things, many engineers added more and more instruction sets. Software development is now developed using compilers, and instructions have been added to execute high-level language syntax. This has come to be called the Complex Instruction Set Computer (CISC) because complex instructions have been added.

However, as the research progressed properly (analyzing the execution of the code), it became clear that complex instructions were rarely used and that it took more work for the compiler to use them well. If that is the case, it is easier to design if only simple instructions are implemented, and it has been found that the operation as a CPU can be speedy. What do you do if you don't have complicated instructions? For that, it will be executed by combining simple instructions.

By using CPUs that are limited to simple instructions in this way, the movement to create faster CPUs has been popular since around 1980, and it has become widely used as a Reduced Instruction Set Computer (RISC). Sparc, R2000/R3000, Alpha, i860/i960, PA-RISC, PowerPC, and others were introduced to the market. "Computer Architecture: A Quantitative Approach" was published in 1990 by David A. Patterson and John L. Hennessy as a computer science textbook. One of the authors, Patterson, is also involved in RISC-V.

As the speed of the CPU progresses, the difference between the speed of the memory and the operating speed of the CPU has widened. No matter how fast the CPU is, there is a problem with waiting when retrieving data from memory or writing data to memory. This is called the von Neumann bottleneck. In CISC, there are usually instructions for manipulating data in memory. However, since memory access is slow, the CPU is also affected by the slow access, and the operation speed slows down. RISC-type CPUs are also characterized by adopting a load-store architecture in which general-purpose registers are prepared, and operations are performed on the registers. In the load store architecture adopted by many RISCs, the CPU instruction has a Load instruction that reads data from memory and a Store instruction that writes data to memory. Memory is accessed by Load and Store instructions, and processing such as addition and bit arithmetic is performed on registers.

As the CPU gets faster and faster, a method is taken to close the speed difference between the CPU and memory by using high-speed memory such as cache memory (but only small in size). On modern CPUs, the cache itself is prepared at multiple levels.

1.4 RISC-V instruction set overview

1.4.1 RISC-V instruction set basics

In RISC-V, the instruction set defines basic and extended instructions and which custom instructions can be added. Let's examine them. There are four basic types of instructions: There are three types of addresses: 32-bit, 64-bit, and 128-bit. In addition to 32 general-purpose registers, 16 are also available for small microcontrollers.

- RV32I (32-bit addressing, integer instructions, 32 general-purpose registers)
- RV64I (64-bit addressing, integer instructions, 32 general-purpose registers)
- RV128I (128-bit addressing, integer instructions, 32 general-purpose registers)
- RV32E (32bit addressing, integer instructions, 16 general-purpose registers)

There may currently be little need for the 128-bit RV128I, but it seems that up to 128 bits have been added to the basic instruction in consideration of the future. Since the instruction sets of existing CPUs are based on 32 bits and have been expanded, they often have an awkward structure that has been repeatedly extended and renovated. On the other hand, RISC-V takes advantage of the fact that it is a latecomer and creates a beautiful instruction set by considering 32 bits, 64 bits, and 128 bits together. In addition, there is still room for future expansion.

First, six instruction formats are the basis of RISC-V instructions. Among them, opcode and funct specify instructions, rd, rs1, and rs2 specify registers, and imm represents immediate data. In the figure below, it is interesting to note that rd, rs1, and rs2 are placed in the same position. These points make it easier to design hardware.

31	27	26	25	24	20	19	15	14	12	11	7	6	0		
funct7				rs2			rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type	
imm[11:5]				rs2			rs1		funct3		imm[4:0]		opcode		S-type
imm[12:10:5]				rs2			rs1		funct3		imm[4:1:11]		opcode		B-type
imm[31:12]										rd		opcode		U-type	
imm[20:10:1:11:19:12]										rd		opcode		J-type	

Therefore, allocating 6 to 2 bits of the instruction set is determined as follows. The previous figure

is the part corresponding to the opcode. The lower two bits are $inst[1:0]=11$.

inst[6:5]↓ inst[4:2]→	000	001	010	011	100	101	110	111
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	custom-2/ rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/ rv128	>=80b

Let's look at the actual LOAD instruction, which is as follows on the RV32I. You can see that $inst[6:2]$ is 00000. Even in the Load instruction,

the part that handles byte data (signed and unsigned), half words (signed and unsigned), and word data is distinguished in the $INST[14:12]$ part.

	31	20	19	15	14	12	11	7	6	0
lb (load byte)	offset12 [11:0]				rs1	000	rd	0000011		
lbu (load byte unsigned)	offset12 [11:0]				rs1	100	rd	0000011		
lh (load halfword)	offset12 [11:0]				rs1	001	rd	0000011		
lhu (load halfword unsigned)	offset12 [11:0]				rs1	101	rd	0000011		
lw (load word)	offset12 [11:0]				rs1	010	rd	0000011		

1.4.2 Instruction extensions and custom instruction

RISC-V can have extended instructions and custom instructions. This is what makes RISC-V unique; it is called a modular configuration. Extension instructions are being proposed to RISC-V International, and various extension instructions are being standardized or discussed. For example, in the 2019 ISA Specifications (Volume 1, Unprivileged Specification version 20191213[4]), the extended instructions are described as follows:

Extension	Version	Status
M	2.0	Ratified
A	2.1	Ratified
F	2.2	Ratified
D	2.2	Ratified
Q	2.2	Ratified
C	2.0	Ratified
Counters	2.0	Ratified
L	0.0	Draft
B	0.0	Draft
J	0.0	Draft
T	0.0	Draft
P	0.2	Draft
V	0.7	Draft
Zicsr	2.0	Ratified
Zifencei	2.0	Ratified
Zam	0.1	Draft
Ztso	0.1	Frozen

The meaning of Status at this time is also described.

- Ratified approved; no changes allowed
- Frozen will not be changed without approval and may only be changed if there is a critical issue.
- Drafts are subject to change until approval. The status will change over time, so always check the latest materials.

1.4.3 General-Purpose registers & Floating-Point registers

RISC-V uses a general-purpose register system, but 32 integer registers can be used with the RV32I, RV64I, and RV128I basic instructions. However, one of them will always be a zero register with a zero value. The RV32E is designed

for small microcontrollers and uses only 16 integer registers.

RISC-V has two names for integer registers: the register name and the ABI name. x0~x31 is the register name, and the ABI name is zero,ra, sp, gp,tp, t0~t6, a0~a7, s0~s11. If you are programming in assembler with EWRISCV or using an inline assembler in a C program, use the ABI name.

- zero is a register that reads are always zero, and writes are ignored (so always returns zero) When ra calls a function, set the return address to ra and call it.
- sp is a stack pointer
- gp is a global pointer, and gp is a register used for memory access. To access memory using this GP, you also need to configure the linker settings
- tp is a thread pointer
- t0-t6 are temporary registers
- a0-a7 are used for function arguments and return values
- S0-S11 are the conserving registers

Register name	ABI Name	Content
x0	zero	zero register
x1	ra	return register
x2	sp	stack pointer
x3	gp	global pointer
x4	tp	thread pointer
x5,x6,x7	t0,t1,t2	temporary pointer
x8	s0/fp	save register or frame pointer
x9	s1	save register
x10,x11	a0,a1	function arguments/ return values
x12,x13,x14, x15,x16,x17	a2,a3,a4, a5,a6,a7	function arguments
x18,x19,x20, x21,x22	s2,s3,s4, s5,s6	save register
x23,x24,x25, x26,x27	s7,s8,s9, s10,s11	save register
x28,x29, x30,x31	t3,t4,t5,t6	temporary register

The ABI name may differ somewhat from the group for this register name x0~x31. The sequence of the save registers is divided. It is for the RV32E. Since the RV32E can only use 16 registers, it is devised to operate the program efficiently even when reduced by half.

In RISC-V, it is possible to handle single-precision floating-point numbers with extended instruction F and double-precision floating-point with extended instruction D. In terms of specifications, there are Zfinx and Zdinx that use quadruple precision and integer registers with extended instruction Q, but first, we will proceed with the basic F and D. The floating-point register has 32 registers. Use the lower 32 bits for single

precision and 64 bits for double precision. Floating-point registers can be f0~f31 for single precision and d0~d31 for double precision.

1.4.4 CSR(Control and Status Register)

RISC-V has a group of registers called CSR (Control and Status Register). Since it can be specified as a 12-bit value, various control registers are defined in Reference [5]. However, in reality, the implementation differs depending on the chip. For details, please check the specifications and the microcontroller to be used. Here are some of the most important CSR registers. In Chapter 3, we will check these values while looking at the values on the GigaDevice microcontroller.

0xF11	mvendorid	Vendor ID
0xF12	marchid	Architecture ID
0xF13	mimpid	Implementation ID
0xF14	mhartid	Hardware thread ID
0x300	mstatus	Machine status register
0x301	misa	ISA and extensions
0x304	mie	Machine interrupt-enable register
0x305	mtvec	Machine trap-handler base address
0x341	mepc	Machine exception program counter
0x342	mcause	Machine trap cause
0x343	mtval	Machine bad address or instruction
0x344	mip	Machine interrupt pending

1.4.5 Modes of operation

Reference [5] describes the privileges and describes the modes of operation. If you look at the current embedded microcontrollers, you will see that they have a machine mode and a user mode. level

It is expected that the software and drivers will be running. The machine mode must be implemented in any implementation, but the other modes are optional. Three combinations will be implemented.

Level	Value	Modes of operation
0	00	user/application (U)
1	01	supervisor (S)
3	11	machine (M)

# of support levels	Support mode	Usage
1	M	Simple embedded system (S)
2	M,U	Secure system (S)
3	M,S,U	Unix, android, or Windows

When entered, it will transition to machine mode (there is also a way to transfer, but the explanation will be omitted here). MRET is the instruction used when returning from interrupt processing in machine mode.

1.4.6 Simple assembler instructions

There are not many people who create programs in assembler these days. However, it is still used if you want to implement an RTOS or optimize performance. There are extension and custom instructions in the case of RISC-V, so you should at least remember the assembler when using RISC-V. Remembering operations, memory accesses, and function calls/returns would be best. First, look at two instructions as arithmetic examples: the addition `add` instruction and the additive immediate value `addi` instruction.

- `add a0, a1, a2`
- `addi a0, a1, 1`

The `add` directive specifies three registers, where `a0` is the resulting storage register, `a1` is the first argument, and `a2` is the second argument. If you write it mathematically, it will be like $a0 = a1 + a2$, which is the same as the order in which it is written in assembler. This instruction adds the contents of registers `A1` and `A2` and writes them to `A0`.

In the instruction `addi`, `a0` is the result storage register, `a1` is the first argument, and `1` is the immediate value. Mathematically, $a0 = a1 + 1$, add `1` to register `a1` and write to `a0`. The following are the `lw` and `sw` instructions for memory access.

- `lw a0, -8(gp)`
- `sw a0, -8(gp)`

The instruction `lw` becomes the meaning of the load word, which retrieves the word data from memory and stores it in a register. The address of the memory is the register plus the offset, which is the address of `-8` to the contents of `gp`, and the contents of the address are stored in `a0`. The instruction `sw` becomes the meaning of the store word and writes the value of the register to memory. The address of the memory is the register plus the offset, which is the address of `-8`

to the contents of `gp`, and the contents of register `a0` are written to that address. In the case of a memory access instruction, the first argument is a register, and the second argument is the memory address.

The next is the jump instructions.

- `jal ra, 0x18` (call `0x18` pseudo instruction)
- `jalr zero, ra, 0x0` (ret pseudo instruction)

The instruction `jal` saves the address of the next instruction in the register `ra` and sets (jumps) the PC to the address obtained by adding the immediate `0x18` to the current PC. If you write it as a call pseudo-instruction, it becomes `call 0x18`, but it has exactly the same meaning. The instruction `jalr` stores the next instruction in the zero register and sets the PC to the address obtained by adding the offset `0x0` to the register `ra`. Writing to the zero register is meaningless in RISC-V, so it is used when returning from a simple function. Therefore, the `RET` pseudo-instruction has the same meaning.

At first, assembler may seem difficult to grasp, but once you have the opportunity to use and see it, you will get used to it. To accelerate habituation, it is a good idea to learn using RISC-V standards.

1.5 Profiles

RISC-V has a modular structure that allows extension instruction selection. However, if you create a modular structure too freely, you may end up with many incompatible things. In fact, Dr. David Patterson has also explained these concerns. A blog with Dr. David Patterson listed in the Bibliography [3] is titled “Top Ten Fallacies About RISC-V”.

The sixth part of the misconception is that “6. modularity leads to a more fragmented software ecosystem “ he writes, “ This fallacy has been raised since we first started advocating for RISC-V, so it’s not been neglected”.. It is explained.

Here’s an introduction to the current profile.

1.5.1 RISC-V profiles

RVI20 Profiles, RVA20 Profiles, and RVA22 Profiles are defined in the instruction set (as of 2023). These Profiles are defined in the following six ways:

- RVI20U32
- RVI20U64
- RVA20U64
- RVA20S64
- RVA22U64
- RVA22S64

The first two letters “RV” refer to RISC-V, and the three letters indicate the family name of the profile.

- RVI: INTEGER
- RVM: MICROCONTROLLER
- RVA: Application

A two-digit number indicates the year of approval of this profile. The sixth character specifies the operating mode: M-mode (machine mode), S-mode (supervisor mode), and U-mode (User mode). The two digits at the end specify 32 bits and 64 bits. For example, the RVI20U32 profile consists of three elements: Mandatory Base, Mandatory Extensions, and Optional Extensions.

- In the Mandatory Base, the RV32I is specified as little-endian. In addition, the fence.tso directive is also mandatory.
- None- mandatory extension
- Optional Extensions
 - > M Integer multiplication and division.
 - > A Atomic instructions.
 - > F Single-precision floating-point instructions.
 - > D Double-precision floating-point instructions.
 - > C Compressed Instructions.
 - > Zifencei Instruction-fetch fence instruction.
 - > Misaligned loads and stores may be supported.
 - > Zicntr Basic counters.
 - > Zihpm Hardware performance counters.

Let's also take a look at the RVA22U64 Profile.

- In the Mandatory Base, the RV64I is specified as little-endian. In addition, the fence.tso directive is also mandatory.
- Mandatory Extensions

- > M Integer multiplication and division.
- > A Atomic instructions.
- > F Single-precision floating-point instructions.
- > D Double-precision floating-point instructions.
- > C Compressed Instructions.
- > Zicsr CSR instructions. The presence of F implies these.
- > Zicntr Base counters and timers.
- > Zihpm Hardware performance counters.
- > Ziccif Main memory regions with both the cacheability and coherence PMAs must support instruction fetch, and any instruction fetches of naturally aligned power-of-2 sizes up to min(ILEN,XLEN) (i.e., 32 bits for RVA22) are atomic.
- > Ziccrse Main memory regions with cacheability and coherence PMAs must support Rsrventual.
- > Ziccamoa Main memory regions with cacheability and coherence PMAs must support AMOArithmetic.
- > Zicclsm Misaligned loads and stores to main memory regions with both the cacheability and coherence PMAs must be supported.
- > Za64rs Reservation sets are contiguous, naturally aligned, and have a maximum of 64 bytes.
- > Zihintpause Pause instruction.
- > Zba Address computation.
- > Zbb Basic bit manipulation.
- > Zbs Single-bit instructions.
- > Zic64b Cache blocks must be 64 bytes in size, naturally aligned in the address space.
- > Zicbom Cache-Block Management Operations.
- > Zicbop Cache-Block Prefetch Operations.
- > Zicboz Cache-Block Zero Operations.
- > Zfhmin Half-Precision Floating-point transfer and convert.
- > Zkt Data-independent execution time.
- Optional Extensions
 - > Zfh Half-Precision Floating-Point.
 - > V Vector Extension.
 - > Zkn Scalar Crypto NIST Algorithms.
 - > Zks Scalar Crypto ShangMi Algorithms.

1.5.2 RISC-V platform specification

RISC-V profiles are specified as extension instructions, so it is a simple list. In addition, the RISC-V Platform Specification is defined. This is a meaningful profile. Currently, there are two platform profiles: OS-A platform for relatively rich operating systems such as Linux and Windows and Platform M for RTOS and bare-metal systems that run on MCUs used in embedded systems.

The OS-A Platform defines the OS-A Embedded Platform and the OS-A Server Platform, while the M Platform defines the Base and Physical Memory Protection (PMP) Extension. OS-A requires ISA profiles for RVA22U and RVA22S, and M Platform requires RVM22M. However, at the moment, it is said that the specifications of the RVA are being prioritized, and there is no material about the RVM22M yet.

They seem to be in a hurry to support Linux and Android around here. It is important to understand this situation if you are using RISC-V or creating your own RISC-V core.

1.6 Why should we use RISC-V?

The content introduced so far has explained the technical points of RISC-V. The RISC-V specification can implement not only basic instructions but also extended instructions and custom instructions, making it possible to support various systems. Since it is a new ISA, it has the advantage of being cleaner than the existing instruction set (clean instructions are easy to implement). The advantage of RISC-V is that many other points improve performance. Three factors determine performance.

- Performance = (Time/Program)

It is possible to make a clear assessment that the shorter the time to run the program, the better. If you put a little more effort into the right side, it will be easier to understand. We get the following if we break down the elements on the right side.

- Performance = (Time/Cycle) * (Cycle/Instruction) * (Number of Instructions/Program)

Looking at each, (number of instructions/program) is how many instructions one program can execute, (cycles/number of instructions) is how many cycles one instruction can be executed, and (time/cycle) is how many seconds can one clock cycle be executed. It isn't very easy, but let's think about it.

1. The number of instructions/programs is set in the Instruction Set Architecture (ISA).
2. The microarchitecture determines the number of cycles/instructions. For example, the number of stages in a pipe run, the superscalar, and the cache configuration are factors here.
3. Time/cycle has a significant impact on semiconductor processes. If you try to speed them up, you will use an expensive micro process because it involves the manufacture of semiconductors.

RISC-V is characterized by the fact that 1 and 2 can be changed by the user. In addition, by changing 1 and 2 by the user, it is possible to be implemented in a cheap semiconductor process. Therefore, in the case of the highest degree of freedom, all of 1, 2, and 3 can be changed and selected by the user.

For example, in Arm's Cortex series, the purchased design data (Arm processor core) is often used as is (there is a separate contract for customization). Therefore, changing the semiconductor process is the only way to improve performance.

RISC-V, on the other hand, allows you to add instruction sets and change the pipeline configuration. For example, an app may have a special operation that can be improved by implementing a custom instruction that executes the operation directly. If other companies use general-purpose CPUs, these custom instructions cannot be imitated, so they can significantly contribute to product differentiation. In addition, even if competitors try to make similar products, competing with them at the same clock frequency will be difficult.

A major feature of RISC-V is that it allows you to select a method that is different from conventional CPU cores. When developing semiconductors (or mounting them in FPGAs), it may be possible to improve performance (processing power and low power consumption) by modifying the CPU core itself. In recent years, in order to make competitive products, there has been a movement to make competitive semiconductors by ourselves. Representative movements will be Apple and Google. Apple and Google are developing different businesses on iOS and Android, but both companies have developed their own semiconductors and installed them in their products. It will be interesting to see what happens in the future.

However, RISC-V is an open instruction set. It is also possible to switch to another company's RISC-V. It is also possible to make semiconductors by ourselves. Strictly speaking, we need to think about peripherals and semiconductor processes, but there will be a big change from the place where nothing has been possible so far. The third benefit is the postponement of assignments. Certainly, the merits of RISC-V are understood, but the creation of our own CPU cores is not feasible at the moment. If you are not sure whether you will use custom instructions in the future, you can use RISC-V to prepare for future changes. Even if you don't have a plan now, if you use RISC-V, you will be able to add custom instructions at a certain timing. It is not uncommon for performance to increase several times by inserting new instructions. For that reason, you can prepare for the next one while studying the RISC-V implementation that has been released, and you can also purchase CPU cores as IP (Intellectual Property) without making them yourself.

Let's expand the discussion from business to international politics. In the past, there was a Coordinating Committee for Multilateral Export Controls (COCOM), which was established in 1949, began its activities in 1950, and was disbanded in 1994 due to the end of the Cold War. If you search for COCOM violations, you can confirm that various incidents have occurred. This was a long time ago, but now there is the issue of trade friction between the United States and China. Specifically, there have been cases where CPU technology cannot be exported to China. This friction can also be related to the products you develop. From this point of view, RISC-V with open source ISA, even if a specific microcontroller cannot be exported, there is a possibility that it can be replaced with a locally available one.

1.7 Organization of this document

By now, you should have understood that RISC-V is different from microcontrollers and CPUs provided by conventional semiconductor vendors. In Chapter 2, we will use IAR's development environment called EWRISCV to check the actual instruction set and key points that will be generated.

2. Basic operation of the EWRISCV development environment



2. Basic operation of the EWRISCV development environment

The fastest way to learn RISC-V is to create and run a program. In this case, we will use the IAR development environment. IAR Embedded Workbench for RISC-V (from now on referred to as EWRISCV) is an embedded software development environment for RISC-V. This chapter explains how to use the EWRISCV. There is also a tool called `iarbuild.exe` for running on

the command line, but the basic usage is to run it using the options set in the integrated development environment. The compiler itself is provided as a command line tool called `iccriscv.exe`, the linker is provided by `ilinkriscv.exe`, etc., so you can also use general make commands. However, in this book, we will explain how to use it in an integrated development environment.

2.1 Precautions when using EWRISCV

The following are some points to note when using EWRISCV. The following items are required for a PC to run EWRISCV.

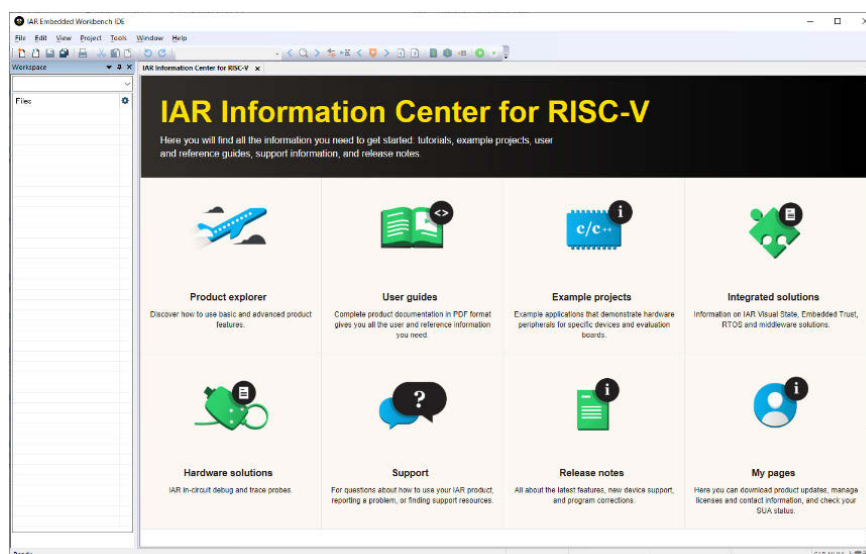
- A Pentium-compatible PC with Windows 7, Windows 10, or Windows 11, 64-bit versions.
- At least 4 GB of RAM and 10 GB of free disk space.
- Adobe Acrobat Reader to access the product documentation

In order to use it comfortably, the PC's memory must be 8 Gbytes or more, and the HDD must be about 20 GB. In addition, for development, we recommend a multi-monitor environment using multiple monitors. Please consider the following points.

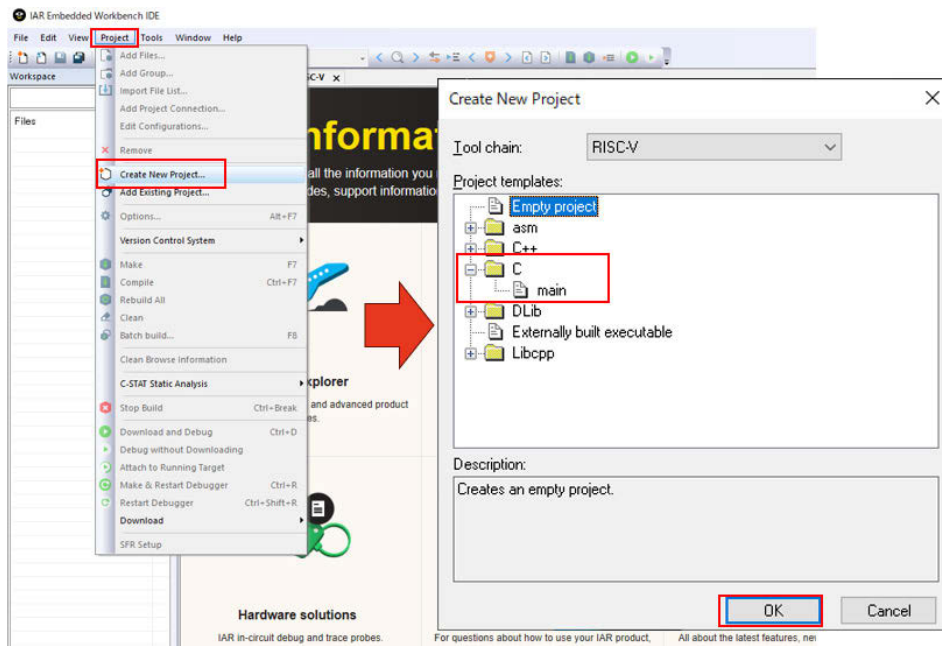
2.2 Create a project (sample 1)

2.2.1 Creating and running a new project

When you start EWRISCV, the following screen will appear:

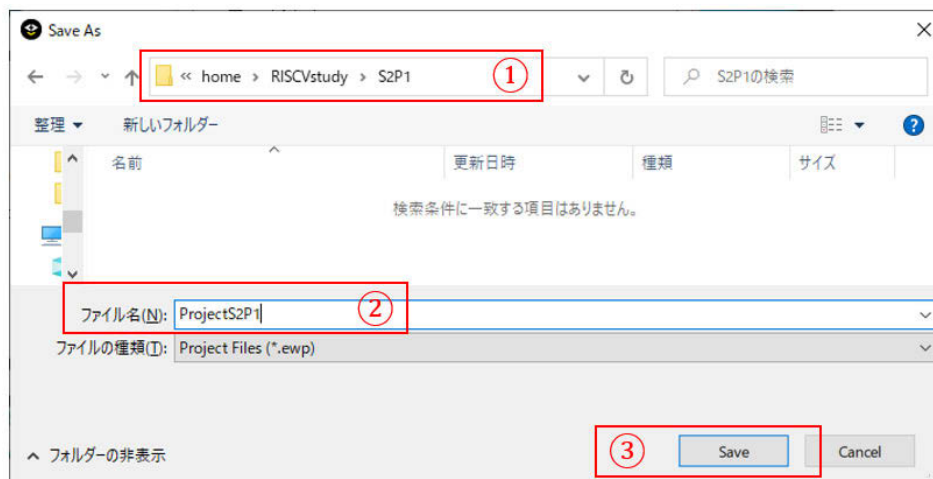


To create a new project, select Project > Create New Project from the toolbar. Select [main] and click [OK].



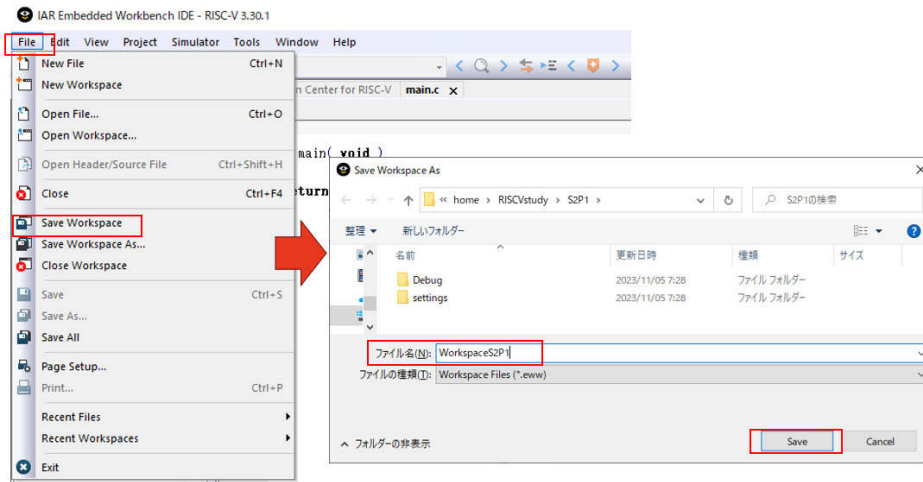
Then, the following screen will appear, so specify the folder where you want to create the project in the input at the top, enter the file name of the

project, and finally click [Save]. In this case, set (2) the project name ProjectS2P1 to the folder (1) / home/RISCVstudy/S2P1, and (3) click [Save].



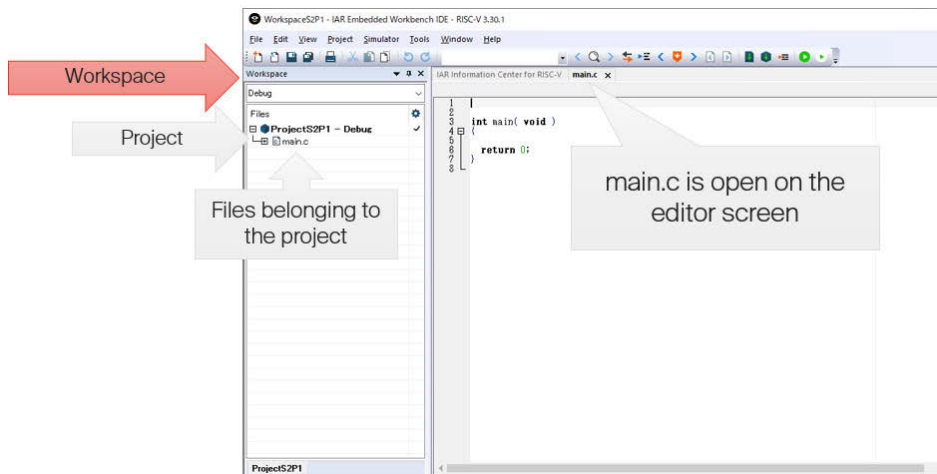
This created a project that contains main.c, but we will also create a workspace now. Select [File]-

[Save Workspace] to open the screen, specify the file name of the workspace, and then click [Save].



Then, the project is created in the following state. In the figure below, there is a workspace screen on the left side, an editor screen on the right side, and main.c is open. EWRISCV is an MDI-type

Windows application with multiple child windows on the main screen. You can change the position of child windows or hide them.



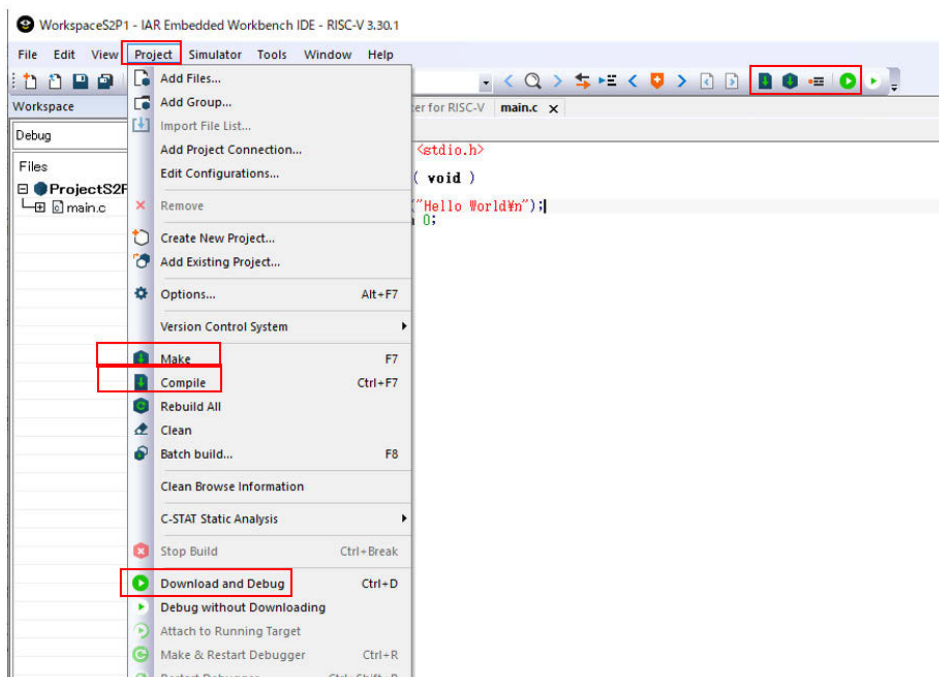
At this point, we have the base of a project to create a C language program with EWRISCV. When you generate a new project, EWRISCV sets the options and other settings to their default state. In this article, we will develop the software

in this default state. Let's create a Hello World program. The Hello World program is the first program in the C programming language. To start with, change the generated main.c to the following.

```
#include <stdio.h>
int main( void )
{
    printf("Hello World\n");
    return 0;
}
```

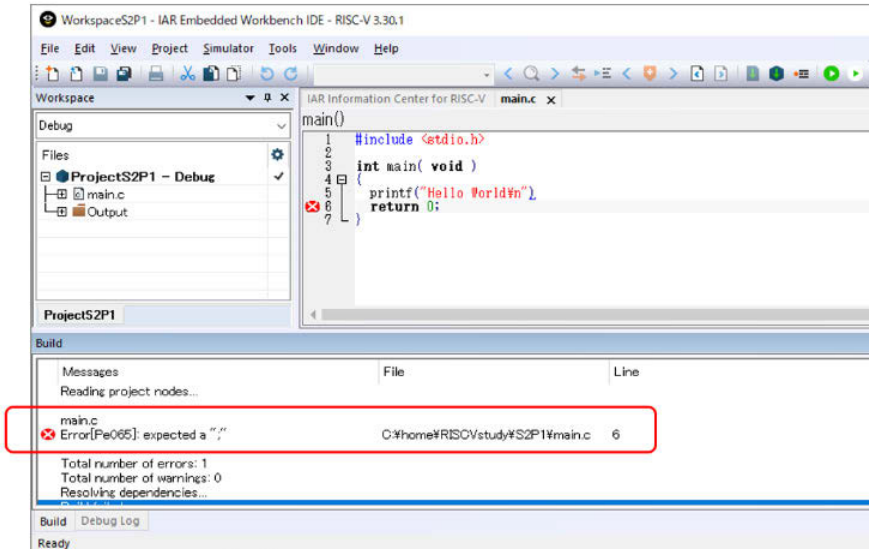
Now that the source code is ready let's build it: Select [Project] - [Make] or click the make button. If you want to compile, click [Project]-[Compile] or click the compile button. The difference between compilation and make will also be explained. The compilation is to compile a file with the extension .c/.cpp to create a .o file. Files with the .o

extension are called object files, but they cannot be executed because the location of variables and functions has not yet been determined. Make compile files with the extension .c/.cpp. Object files (.o) are collected and combined in an executable format output file. In modern microcontrollers, the output format is called ELF.



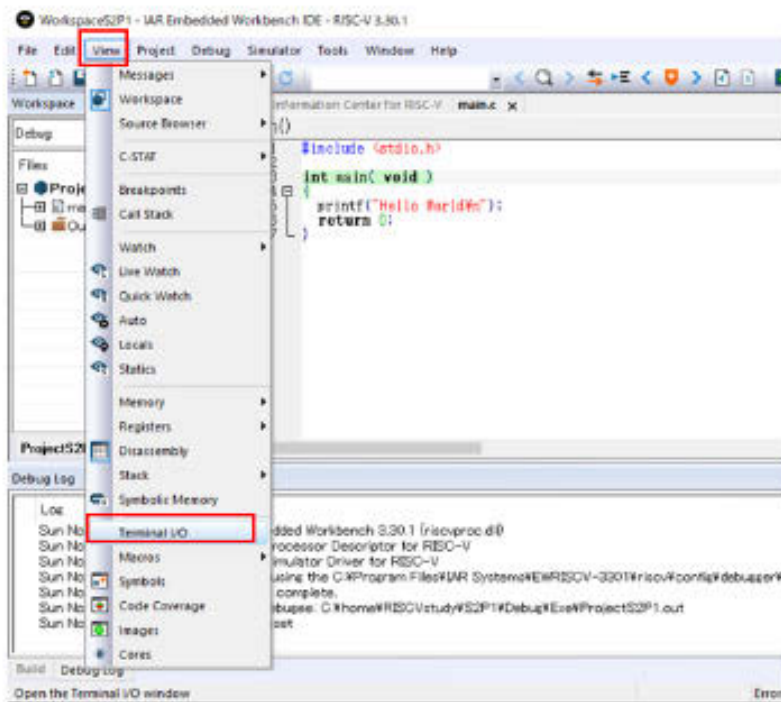
Errors and warnings can also occur at compile time. In EWRISCV, information is output to the Build window. At the bottom of the figure below, there is a Build window, and an error message

appears. The error message shows the file name and the number of lines in which the error occurred. Clicking on this line jumps to the line in the editor where the error occurred.



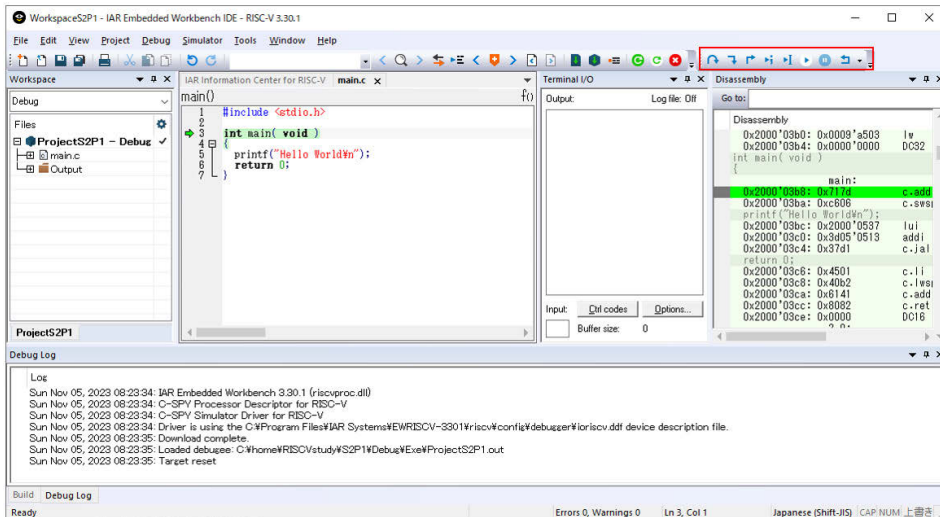
If there is an error, please check for a typo. A common problem occurs when double-byte characters are entered in addition to text strings and comments. For example, spaces, semicolons, double quotation marks, and parentheses are hard to notice.

Once there are no errors and the build is complete, let's start debugging. The simulator is used for debugging by default, so click Download and Debug. This will put the EWRISCV into debug mode. In this case, printf outputs a string, but select [View]—[Terminal I/O] for its display.



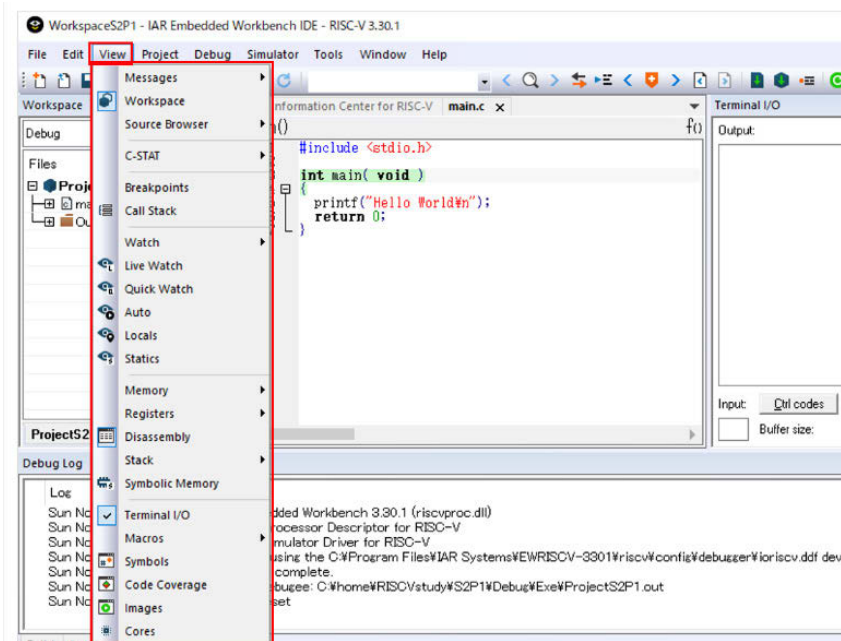
Then, the Terminal I/O screen will open, as shown in the figure, so please execute it. Hello World appears on the Terminal I/O screen. If you execute the program in

detail, you can proceed by combining step over, step into, step out, etc. If you perform a reset, you can perform it again.



Terminal I/O is not the only screen that can be used for debugging. As shown in the figure below, various information can be displayed. Commonly used ones include

Memory, Watch, and Registers. For more detailed information, check the EWRISCV manual.



During debugging, you can edit the file, but you can't actually change the options. When you are finished debugging, click Exit Debugger.

2.2.2 Project structure

So, what is the project structure of EWRISCV? You will see the following.

- Workspace (.eww)
- Project files (.ewp)
- main.c
- [Folder]Debug
 - > [Folder] BrowseInfo
 - > [Folder] EXE
 - > [Folder] List
 - > [Folder] Obj
 - > [Folder] setting

The following files/folders are located in the folder where this project was created. A workspace is a container for managing projects. You can manage multiple projects in one workspace. The project file has two configurations: Debug and Release. Due to the default Debug configuration, there is only a Debug folder at this stage. There are four folders under that folder: BrowseInfo, Exe, List, and Obj. The folder BrowseInfo contains information such as the analysis of the source code. Therefore, there is no need for the user to see it directly. The folder Exe is the location of the Made Executable Format (ELF). When you

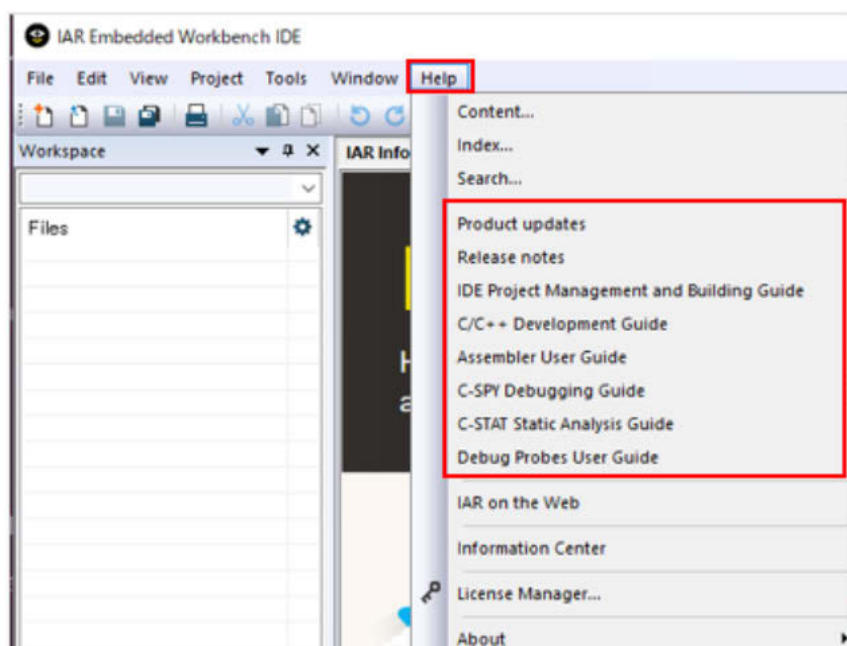
generate a HEX file, etc., it will be generated in this Exe folder by default. In the folder List, MAP files and other files are placed. The folder Obj contains the compiled .o files.

It can be difficult to check in large projects, so let's check what is generated where in these small projects.

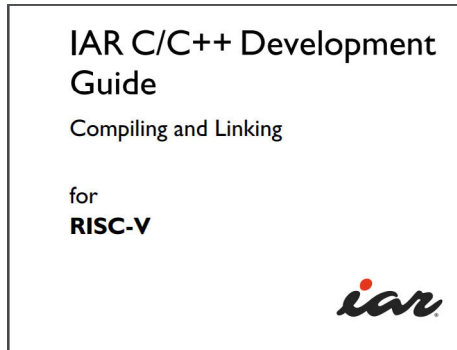
2.2.3 About the manual

When developing software, it is necessary to research how to use compilers, assemblers, etc. EWRISCV provides PDF manuals and online help. If you select [help] on the EWRISCV development screen, you can select manuals, etc. (red box in the figure below).

- For information about the development environment, see the IDE Project Management and Building Guide
- For information about the C/C++ compiler and linker, see the C/C++ Development Guide
- For details about assembler, see the Assembler User Guide
- For debugging information, see C-SPY Debugging Guide
- For details about debug probes, see the Debug Probe User Guide



For example, when you open the C/C++ Development Guide, you will see a PDF file with the following display: This C/C++ Development



Guide contains information about the following: Part 1 explains the basic concepts, and Part 2 describes the options and other details.

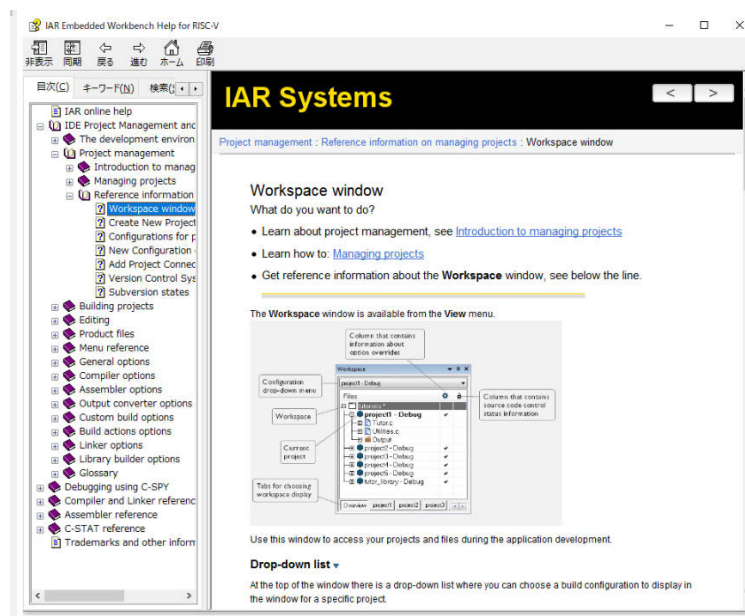
Part2. Reference Information

Part1. How to use the development tools

- Development environment
- Development of embedded applications
- About storing data About Functions
- Links using ILINK
- About the Runtime Environment
- Interface with assembler, etc.

- Compiler options
- Linker options
- About data types
- Extended keywords
- About Pragmas
- Built-in functions
- About linker settings
- About sections used by the compiler
- About the Stack Analysis Settings File
- C++/C Language Implementation Dependencies

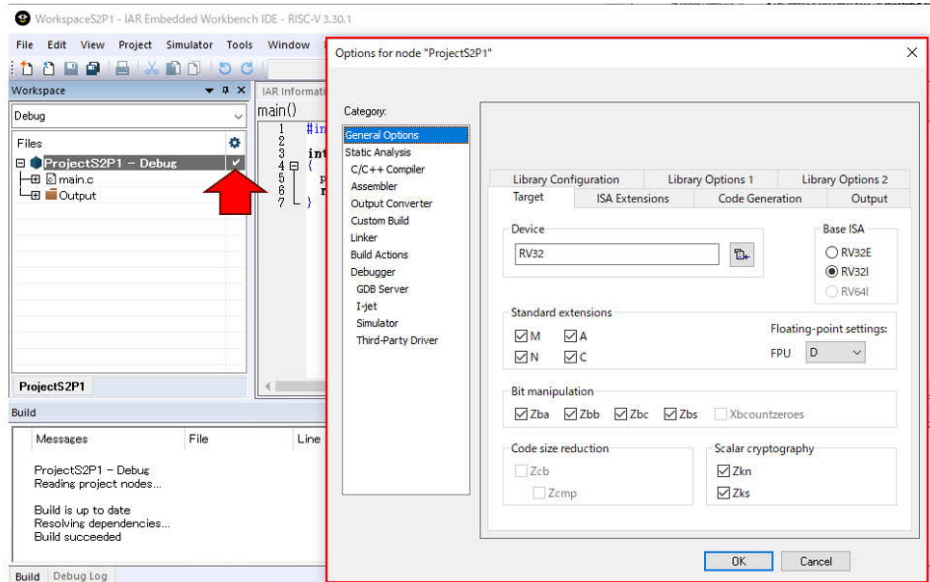
When you press the F1 key in the EWRISCV window, the online help is displayed.



2.3 Options

First, let's take a look at how to set project options and what they do. To open the project options in EWRISCV, double-click ✓ (above the red arrow

below) to the right of the project name on the workspace window, or select the project name in the workspace window and select [Project]-Options.



The part surrounded by the red frame in the above figure is the Options screen. This Option screen has categories on the left and a place to set options on the right.

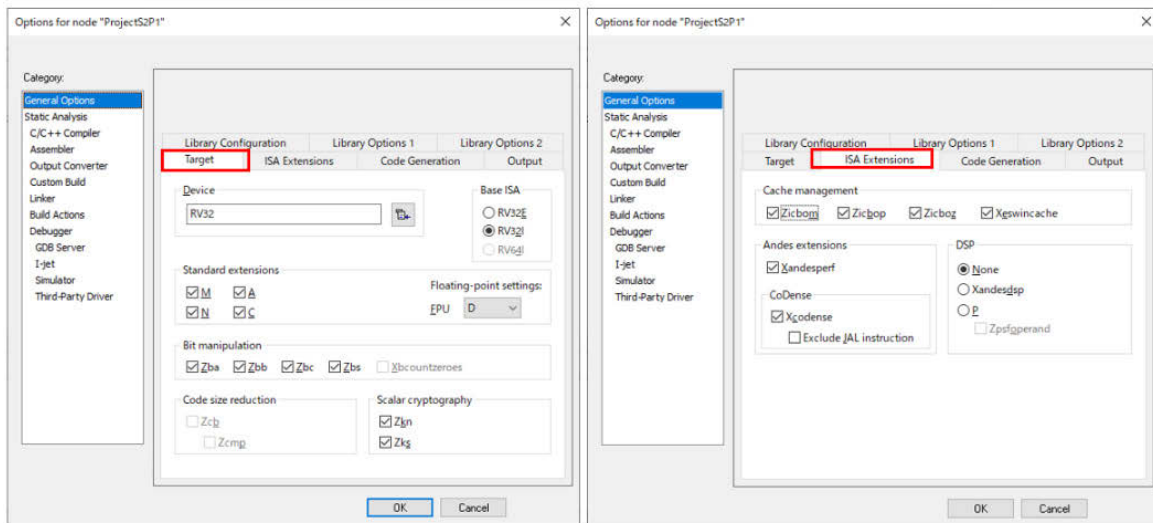
2.3.1 General Options

In General Options, you can 1) specify options related to the RISC-V instruction set, 2) stack,

heap size, and code model, 3) library settings, and 4) output format and folder settings.

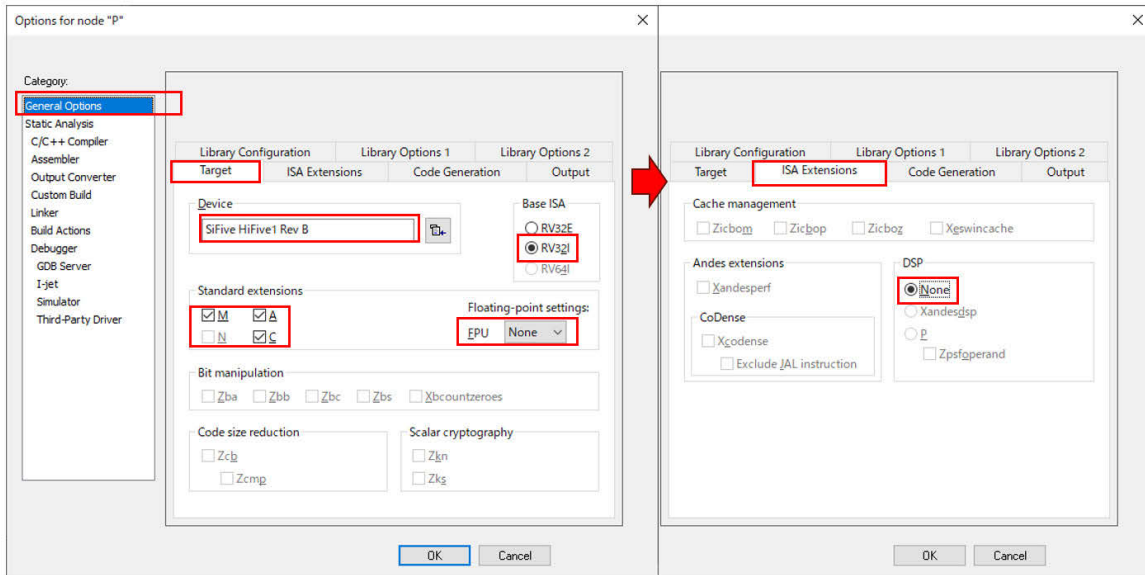
On the Project Options

On the Project Options screen, select General Options under Category. Tabs for Target and ISA Extensions allow you to set options related to the instruction set.



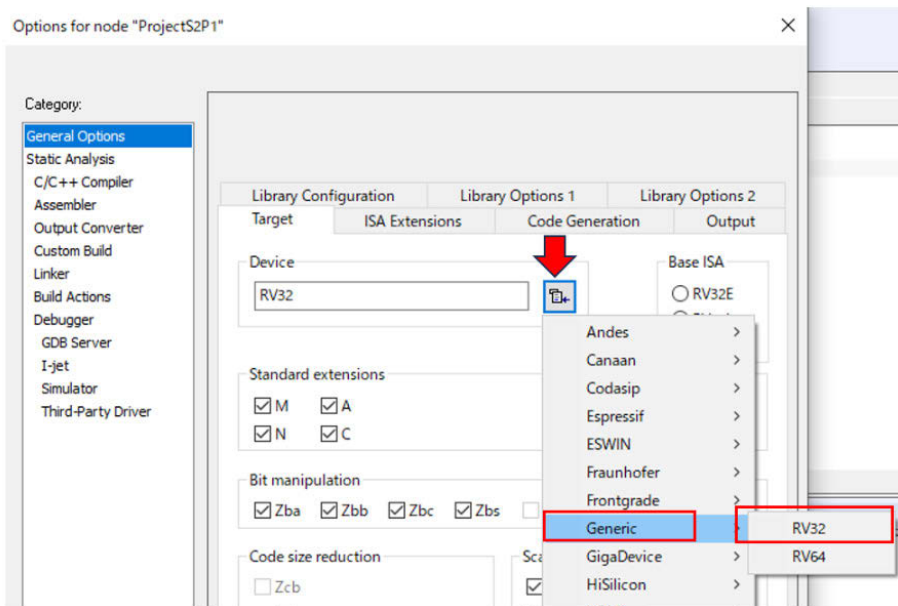
The most important part is [Device], and if the microcontroller to be used is decided, it can be specified here to set the extension instructions automatically. For example, if you select SiFive's HiFive1 Rev B, the instruction set options are

set as follows: If you check the implementation instructions for HiFive1 Rev B, it says RV32IMAC, so you can see that the RV32I + M + A + C option is set (marked with a green box).



In EWRISCV, basic and extended instructions can be specified individually. To do this, click the Device selection button (arrow in the figure below), select [Generic], and select [RV32] or [RV64]. For other extension instructions, you can

select any extension instruction by choosing to check or not check one by one (of course, there are also dependent instructions, so it is not completely free to choose).



You can choose from the following three basic instruction sets.

- RV32E: 32-bit, 16 general-purpose registers
- RV32I: 32-bit, 32 general-purpose registers
- RV64I: 64-bit, 32 general-purpose registers

In the Target option, the following can be selected as standard extension instructions:

- M: Integer multiplication/division
- A: Atomic Instruction
- C: Compressed (16-bit) instruction
 - N: User-level interrupt
- B: Bit manipulation instruction
 - Zba: Instructions for address calculation
 - Zbb: Underlying bit instruction
 - Zbc: Carryless multiplication
 - Zbs: Single-bit manipulation instruction

The following floating-point extension instructions are available:

- F: Single-precision floating-point arithmetic in floating-point registers
- Zfinx: Single-precision floating-point arithmetic on integer registers
- D: Double-precision floating-point arithmetic in floating-point registers
- Zdinx: Double-precision floating-point arithmetic on integer registers

You can choose from the following options for code size reduction:

- Zcb: 16-bit extension instruction to add C extension
- Zcmp: 16-bit extension instruction that makes the code smaller with stack manipulation instructions

For cryptographic extensions, you can choose from:

- Zkn: Extensions for NIST Algorithms
- Zks: Extended instructions for ShangMi Algorithms

Cache management enhancements include the following options:

- Zicbom: Cache Block Management
- Zicbop—Cache prefetch operation
- Zicboz: Cache Block Zero Processing
- Xeswincache: Non-standard cache management extensions

Andes extensions can be selected from the following:

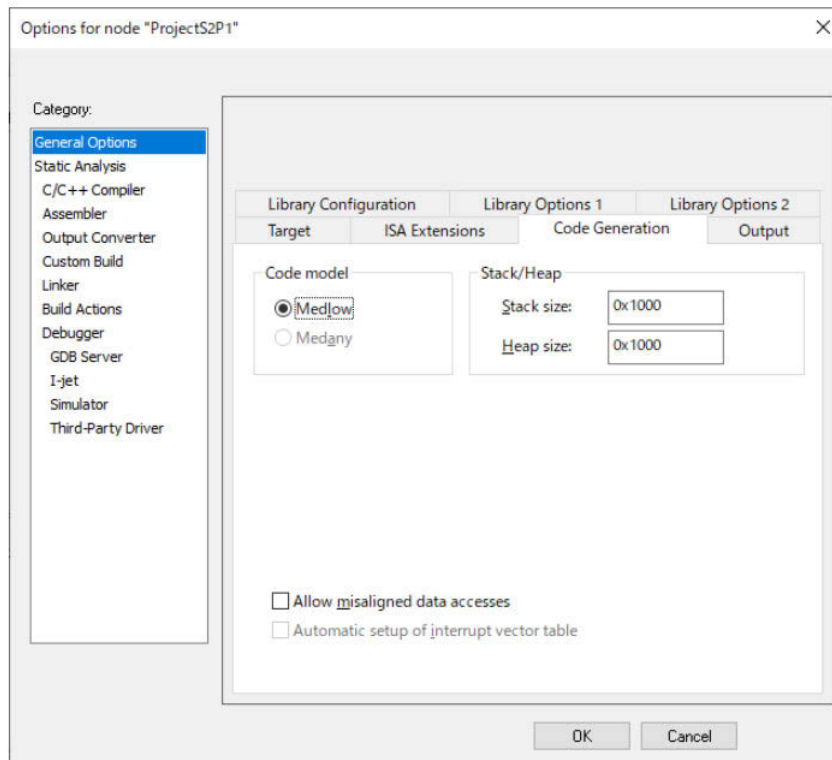
- Xandesperf: Andes' performance enhancement AndeStar V5 Performance
- Xcodense: Andes' code size compression extension AndeStar V5 CoDense

The following DSP extensions are available:

- None: Do not use DSP extensions
- Xandesdsp: DSP extension from Andes
- P: Uses a subset of the P extensions, Zpn and Zbpbo
- Zpsfoperand: Use all P extensions

Stack, heap size, and code model

The CodeGeneration options screen allows you to configure the code model and set the stack/heap size.

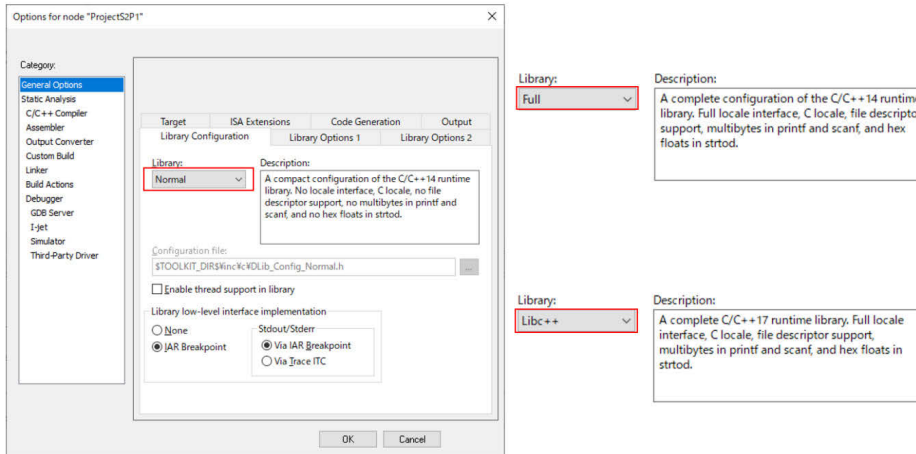


Code Model is an option to select for 64-bit RV64I. In terms of functionality, Medlow is an option to address with 32-bit absolute values. Because it is signed and processed, the area of memory that can be accessed is 0x0000000000000000 to 0x000000007fffffff or 0xffffffff80000000 to 0xffffffffffff.

Medany, on the other hand, addresses in relation to the PC. At this time, you can specify a relative address from -2GB to +2GB. Medlow had a space of +-2 GB centered on the 0x0, but because it is PC-relative, the space that can be referenced is larger than Medlow's. However, you can only refer to -2 GB to +2 GB addresses from the PC, so be careful when using RV64I.

Library

Library Configuration, Library Option1, and Library Option2 configure the library's settings. First, in Library Configuration, you can choose from none, Normal, FULL, libc++, and custom libraries.

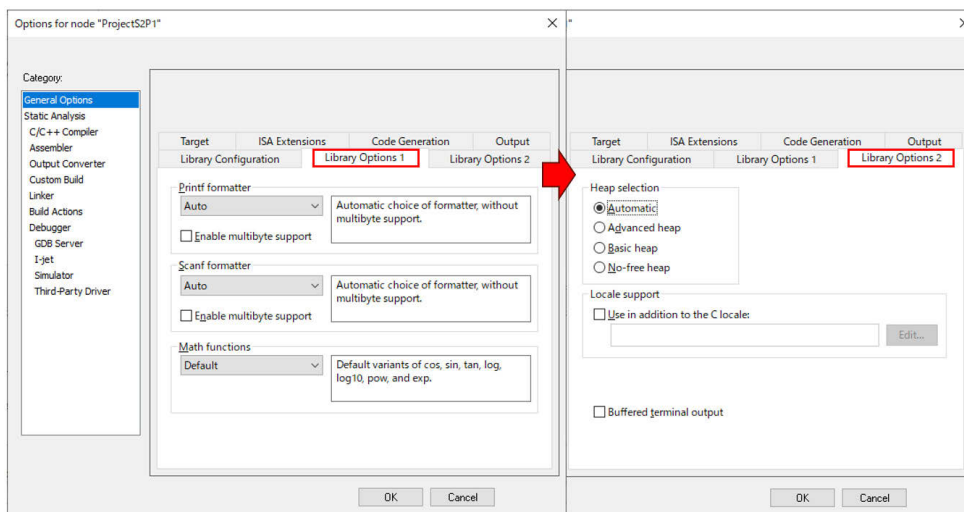


- Normal: C/C++14 libraries (C locale only, no multibyte support, no FILE support)
- FULL: C/C++14 libraries (full set)
- libc++: C/C++17 libraries (full set)
- custom: This is a specification when using your own C/C++ library

Library Configuration allows you to specify libraries for low-level interfaces. This is a setting for whether or not to use a file on the PC when the debugger is connected or to use a function for the debugger that executes

standard input/output, such as printf, to the debugger screen. Specify None if you don't want to use it or IAR Breakpoint if you're going to use it. For stdout/sterr, specify how to output standard output to the debugger. Normally, you would choose Via IAR Breakpoint, which outputs data at break, but for devices with an Instrumentation Trace Component (ITC), you can set Via Trace ITC.

Library settings can be set for printf/scanf, math functions, and the heap.



Libraries have a significant impact on the size of the code implemented, depending on the features they use. If there is a feature that you do not use, please select the correct option. The

following is the format specification in printf. The NoMB at the end of the name means “do not use multibytes”.

printf features	Tiny	Small/ SmallNoMB	Large/ LargeNoMB	Full/ FullNoMB
Basic specifiers c, d, i, o, p, s, u, X, x, and %	YES	YES	YES	YES
Multibyte	NO	YES/NO	YES/NO	YES/NO
Floating-point specifiers a and A	NO	NO	NO	YES
Floating-point specifiers e, E, f, F, g, and G	NO	NO	YES	YES
Conversion specifier n	NO	NO	YES	YES
Format flag +, -, #, 0, and space	NO	YES	YES	YES
Length modifiers h, l, L, s, t, and Z	NO	YES	YES	YES
Field width and precision, including *	NO	YES	YES	YES
long long support	NO	NO	YES	YES
wchar_t support	NO	NO	NO	YES

The scanf format specification is as follows. The scanf format specification is as follows.

scanf Description	Small/ SmallNoMB	Large/ LargeNoMB	Full/ FullNoMB
Basic specifiers c, d, i, o, p, s, u, X, x, and %	YES	YES	YES
Multibyte support	YES/NO	YES/NO	YES/NO
Floating-point specifiers a, and A	NO	NO	YES
Floating-point specifiers e, E, f, F, g, and G	NO	NO	YES
Conversion specifier n	NO	NO	YES
Scan set [and]	NO	YES	YES
Assignment suppressing *	NO	YES	YES
long long support	NO	NO	YES
wchar_t support	NO	NO	YES

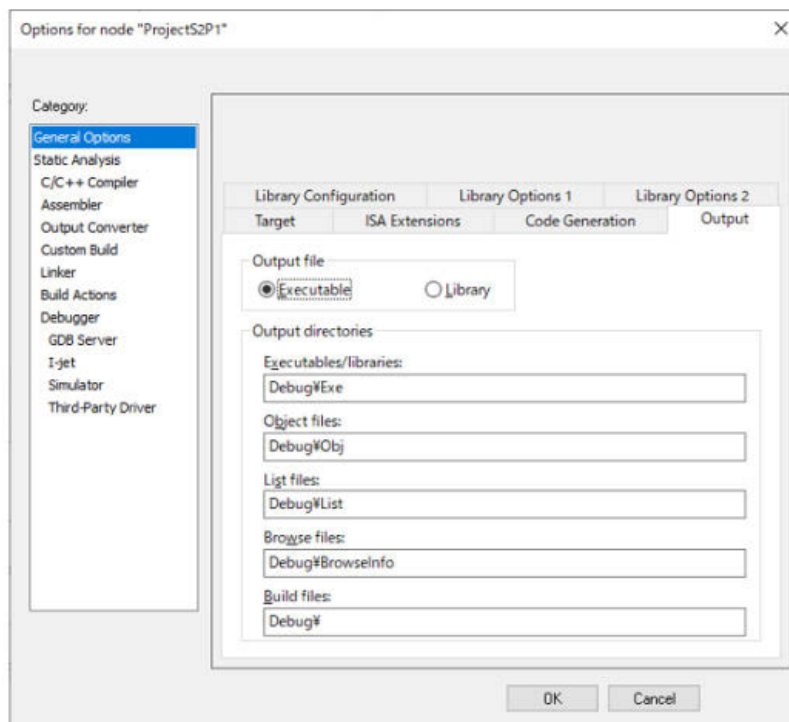
The heap algorithm can also be optionally selected. The heap is the memory used for malloc/free in C and new/delete in C++.

- Automatic: IAR determines the situation and selects one of the following three options:
- Advanced Heap: The best option for heavy HEAP users

- Basic Heap: Recommended option if you don't use HEAP too much
- No-free Heap: Recommended option if you don't want to free up memory

Output Format & Folder Settings

The Output option specifies the output as a library or executable format and specifies the output folder for each product. If there are no particular problems, it is better to operate it as it is. The final product (executables and libraries) is stored in the Executables/Libraries folder, and the map files are stored in the List Files folder.



2.3.2 C/C++ Compiler

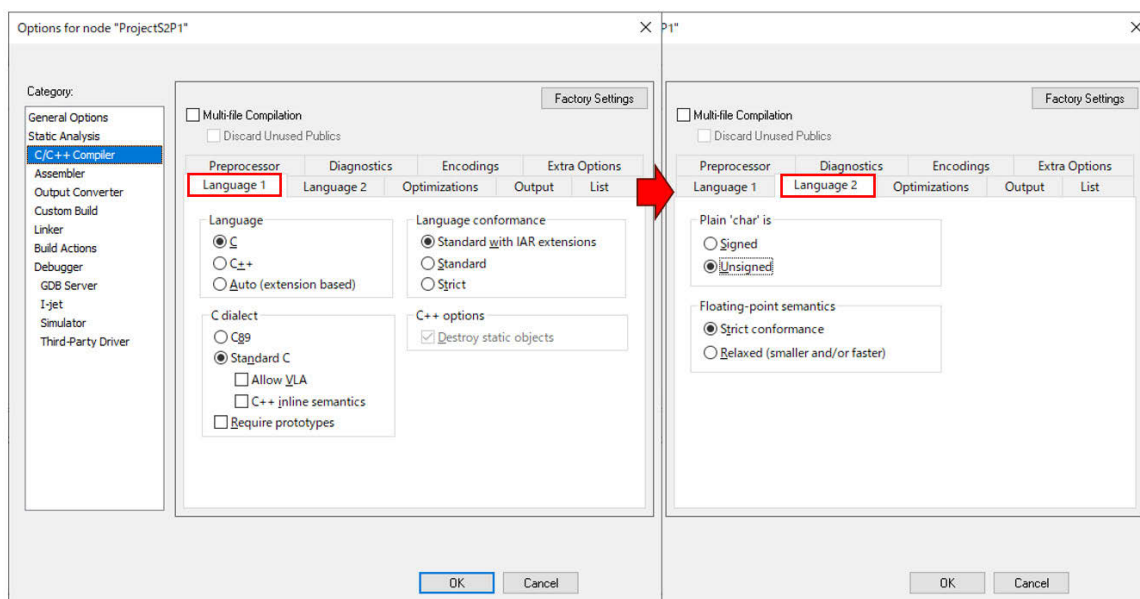
C/C++ compiler options are specified here. This section describes 1) language settings, 2) optimization settings, 3) output settings, 4) preprocessor settings, and 5) diagnostics settings.

Language settings

EWRISCV can compile C or C++ languages. Therefore, it is necessary to choose the language C/C++. You can set the Language on the options screen of Language1. If the extension is .c, it is C language, and if the extension is .cpp, it is C++ language, Auto (Extension based) can also be used. In the C dialect, you can choose between C89 and the current C18 standard. In Language Conformance, you can choose to select from the following three levels of compliance with the standard. In an embedded system, you may want to use CPU-specific instructions, but it may be troublesome to write a program in assembler, so IAR has prepared language extensions for embedded systems. For example, to declare an interrupt handler in C, we have the keyword `__interrupt`.

- Standard with IAR extensions—If you want to enable IAR extensions
- Standard— Disables IAR extensions, but does not adhere strictly to the C or C++ dialect you have selected. Some very useful relaxations to C or C++ are still available.
- Strict: When strictly adhering to the language specification

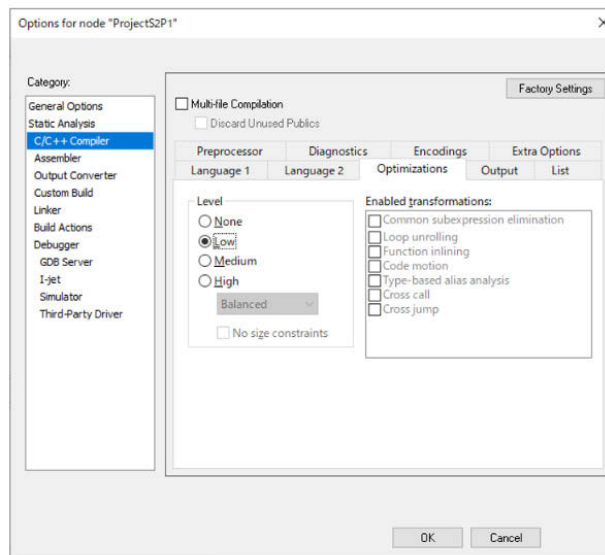
For language 2, some points need to be confirmed. If you look at C language books, you will see some that say, “char is signed, and if only positive integers are handled, it is defined as unsigned char,” but that is not correct. In the C language standard, the sign of char is left to the implementation. There are many cases where char is used as an unsigned char. EWRISCV allows the user to specify it. Define it with the type you want to use.



Optimization settings

In the Optimizations tab, you can choose to optimize as None, Low, Medium, or High, depending on the level. Keep in mind that optimizations can make debugging in source

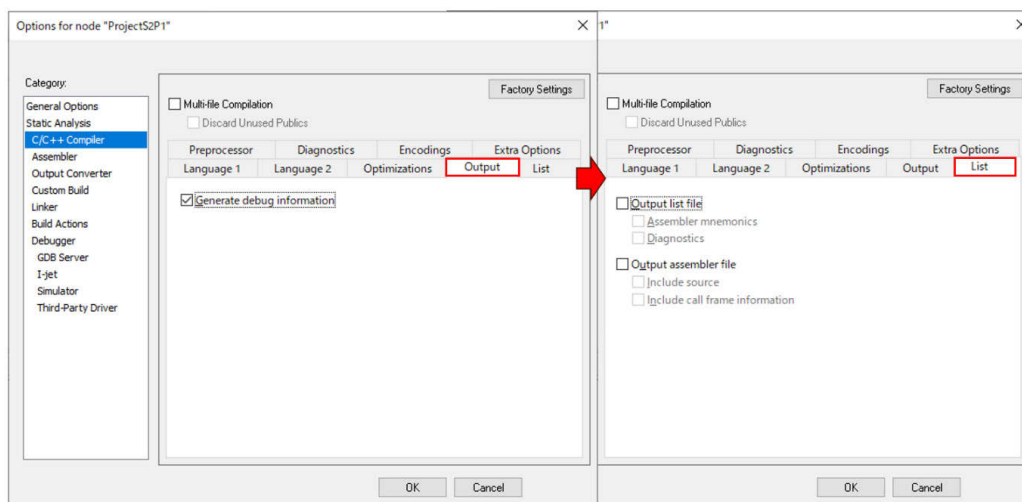
code more difficult. Optimizations can change the order of execution, expand loops, or perform calculations in advance, making it difficult to step through them in the order in which they are described. If you want to debug at the source code level, select None or Low.



Output settings

In the Output tab, there is an Output setting called Generate Debug Information. To debug the source code, be sure to check Generate Debug Information.

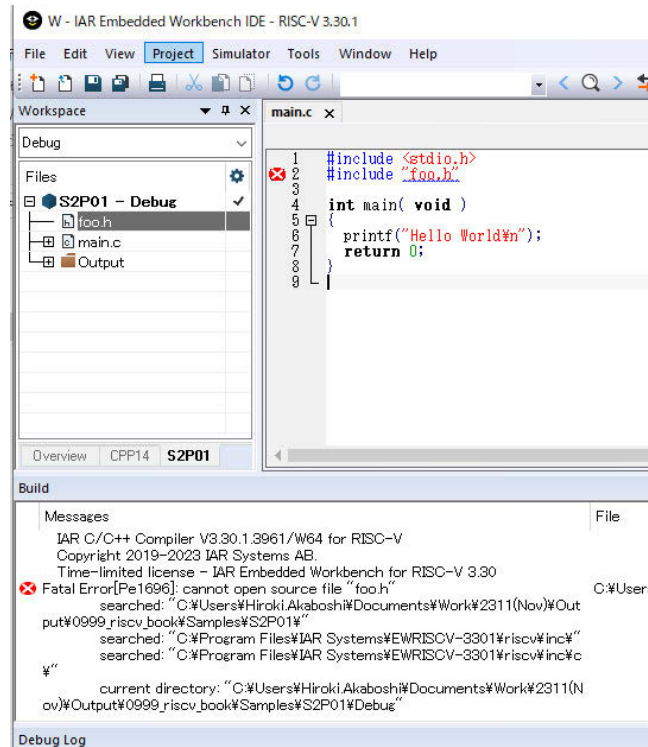
In the List tab, you can specify the output of the compiled list file and create assembler output. It is a good idea to use a list file to create assembler output to check how a C program is compiled. Please note that assembler output is not possible in the evaluation version.



Configuring the preprocessor

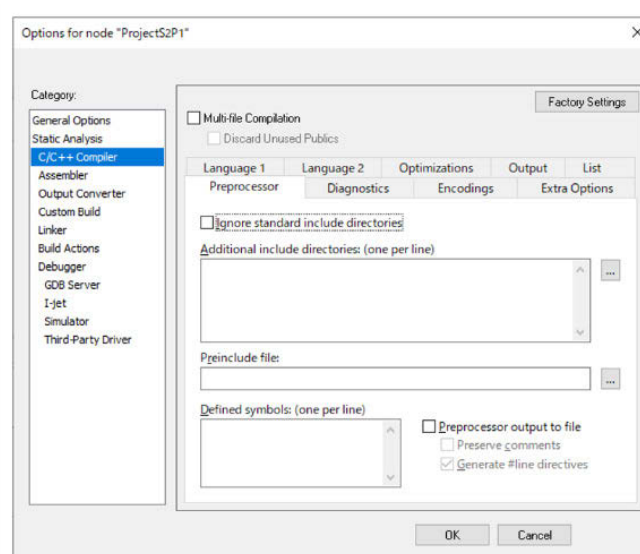
In EWRISCV, you can add a header file to the Workspace window as shown below, but if you do

not configure the include paths correctly, an error will occur if the header is not present, and you will not be able to compile.



In EWRISCV, you need to specify the folders where header files are located. If nothing is specified selected, the folder with the project file, the inc folder, and the inc/c folder of the

EWRISCV toolchain will be searched. If you want to search other folders, specify the folders in the Additional include directories field of the Preprocessor tab.



Avoid specifying a folder with an absolute path, as this will lead to many problems when sharing the project. EWRISCV can refer to the location of the project folder (the folder where the .ewp file is located) in a relative fashion, with \$PROJ_DIR\$, so if you specify:

```
$PROJ_DIR$\inc
```

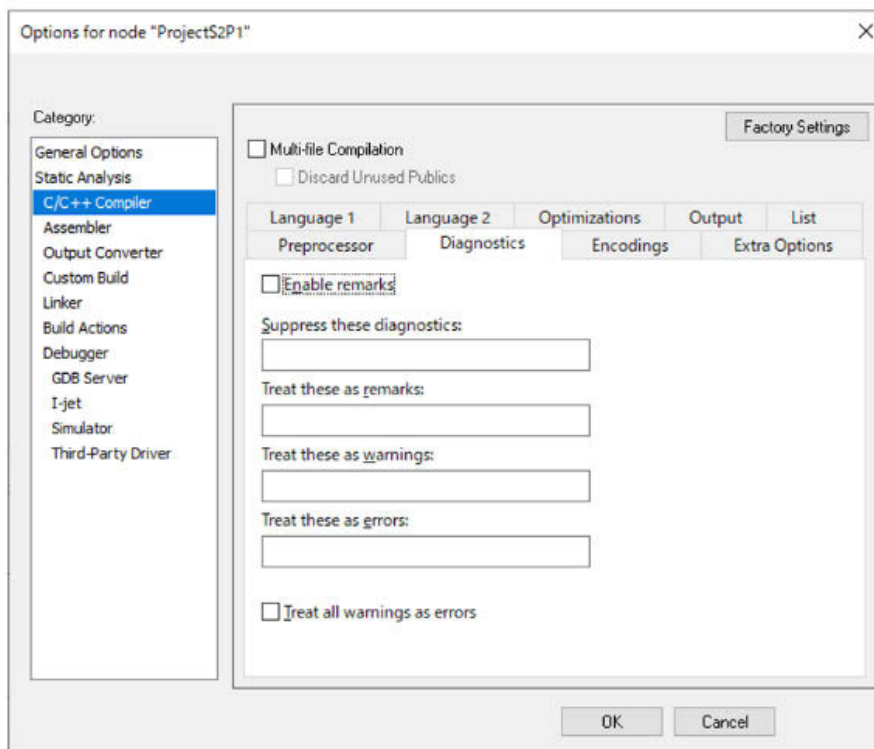
It will also search the inc folder directly under the project folder. If there is more than one, add it on a different line.

Also, in order to set the equivalent of #define SYM (1) at compile time, specify the define in the Preprocessor tab , Define Symbols window.

If you simply define SYM, the value is set to 1. If you want to set a value other than 1, use SYM=100.

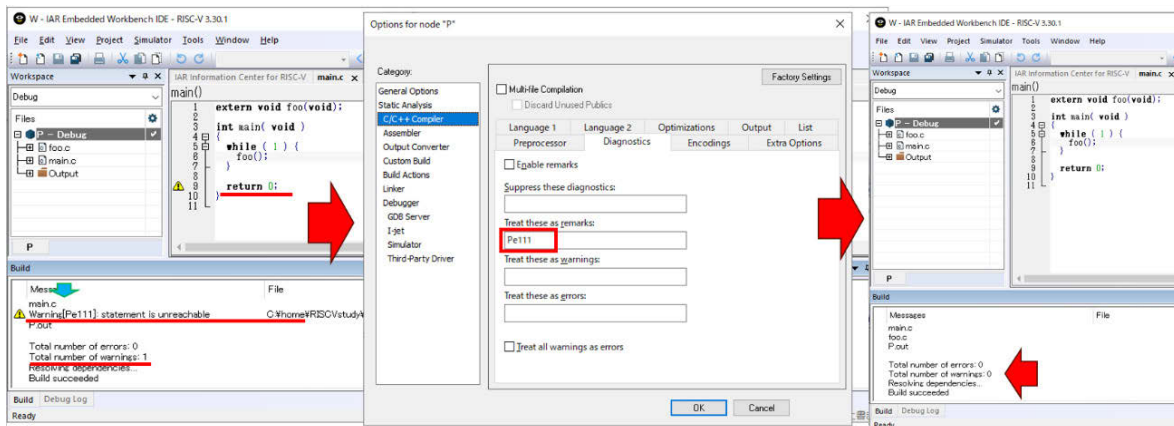
Diagnostics

In EWRISCV, the settings for errors and warnings can be changed. First, in the Diagnostics option, you can enable remarks. A remark is not an error but a message to the effect that you should be careful. If you want to make your program more reliable, you can look at this output and change the code in some cases. You can also change the level of errors and warnings or stop the output.



For example, in the built-in, the main function continues processing in a while(1) loop, so no further processing is performed. In the program on the left below, there is a description of “return 0;” after the while loop, but this line is never executed. In EWRISCV, the warning is output as [Pe111]. Since there is no problem as a program,

it is possible to ignore this warning even if it appears, but you can change the diagnostic settings to change this warning to another remark. By specifying Pe111 in the middle of the figure below, you can confirm that the warning disappeared at the time of build by remarking.

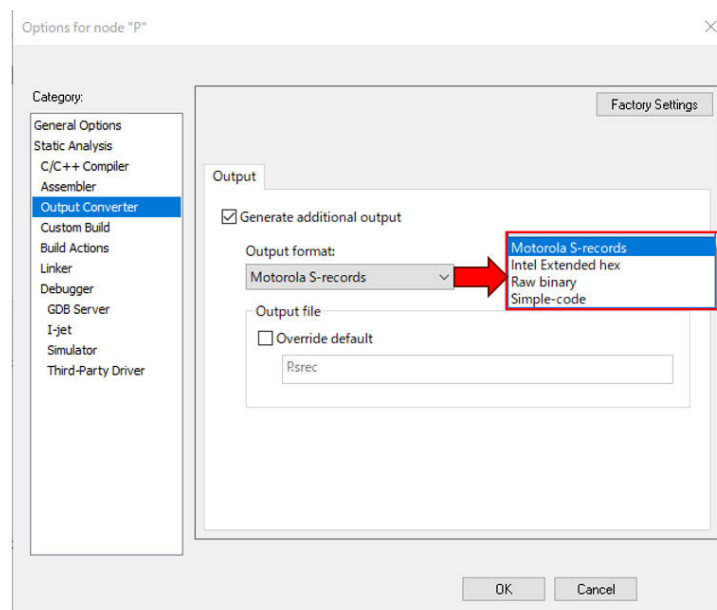


2.3.3 Output converter

In EWRISCV, the file created when building is an executable file in ELF format. However, if you want to use it with another device, such as a ROM writer, you may need to output in HEX or SREC format. At that time, the Output Converter option allows you to specify additional outputs.

You can choose from the following four file formats. The most commonly used formats are Intel and Motorola.

- Motorola S format
- Intel Format
- Binary
- Simple-code



2.3.4 Linker

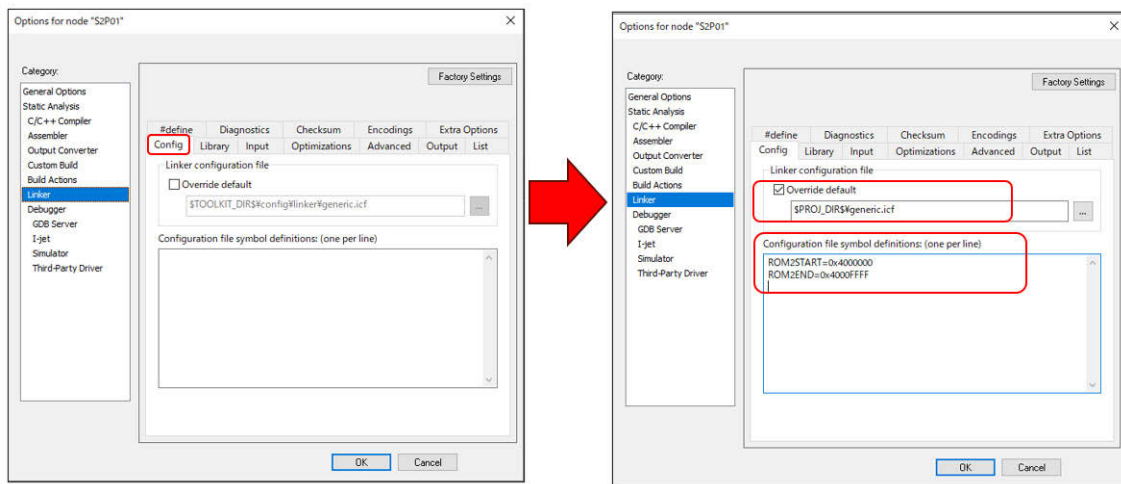
The linker integrates the .o file created by the C/C++ compiler and the necessary data and outputs the program in an executable format. The file that describes the linker configuration is called the linker configuration file, but in IAR, it has a .icf extension.

IAR provides a linker configuration file for the microcontroller so that you can set it with the Config option. On the left side of the figure below, we will use the default Linker Configuration file. This default linker configuration file is located in:

```
\riscv\config\linker
```

under the EWRISCV installation folder.

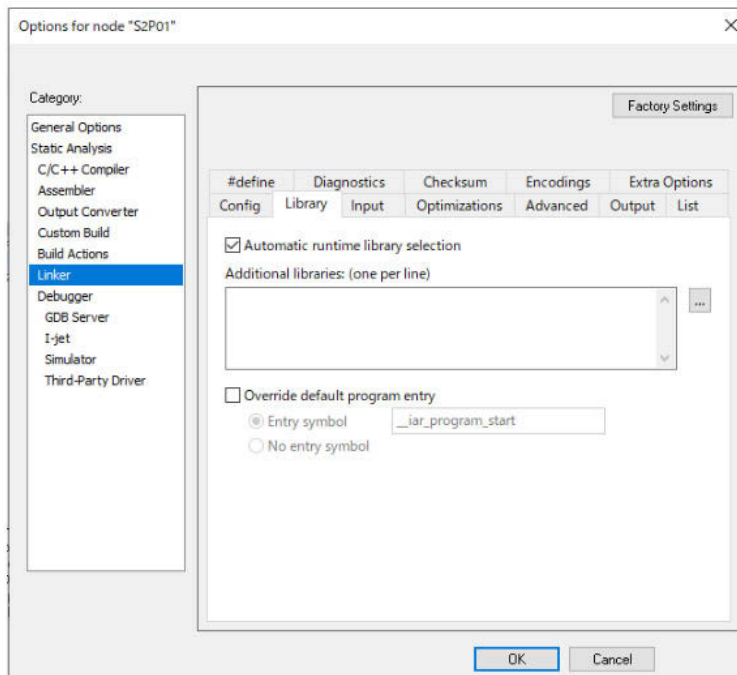
However, in reality, linker settings are often specified individually for each project. Copying is frequently used to copy the default file to the project folder. Since the folder where the project file is located is the project, it can be indicated by the folder \$PROJ_DIR\$. So, on the right side of the figure below, \$PROJ_DIR\$generic.icf is specified as the linker configuration file. You can also define the symbols to be used in the linker configuration file under it.



Library option setting, but by default, Automatic runtime library selection is enabled. Some people don't like the fact that the library is automatically built, but since the compiler is devising code generation, please basically enable and use it here (or if you can fully understand the code generated by the compiler and map the mapping yourself, it is possible manually).

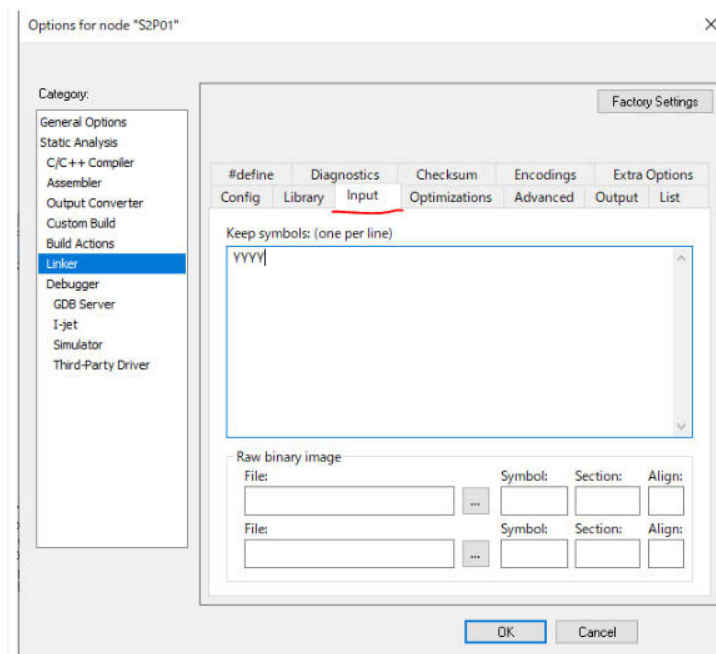
Although it is under the Library option, it is possible to specify the Program Entry. Again, by default, it is __iar_program_start, but the user can

change it. This is the information that the linker needs to create an executable file. By specifying the function (label) to be executed immediately after the reset, the functions and variables necessary for executing the program are placed from here. Therefore, functions that are not explicitly called or variables that are not explicitly accessed are not linked because the linker determines that they are not needed. In such cases, specify them individually with -keep (which can be specified with the following Input option).



In the Input option, you can specify that the symbol should be kept so that functions without explicit calls and variables without explicit access are not deleted (left in the executable form). In the figure below, "YYYY" is specified.

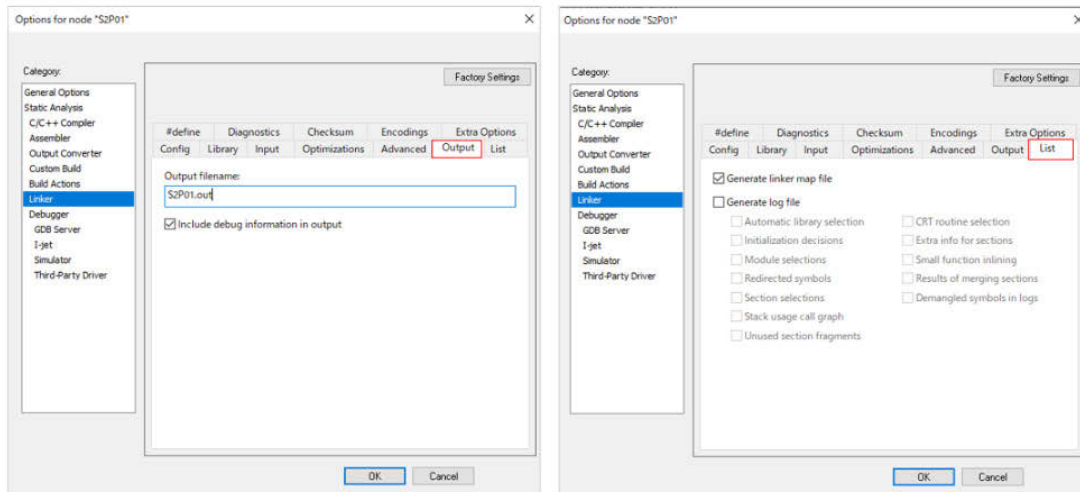
In EWRISCV, when importing binary data such as image files and audio files into an executable format, specify it in Raw binary Image under the Input option. That data specifies the symbol name, section, and alignment.



For output, take a look at the Output and List options. The left side of the figure below is the Output option, but it is possible to specify the file name for the output. By default, it will be project_name.out. Below that, there is a check to see if you want to include debugging information, but basically, you should include it. If you don't include debug information, you won't be able to back up the source code. Some people mistakenly think that the final ROM size will be larger because

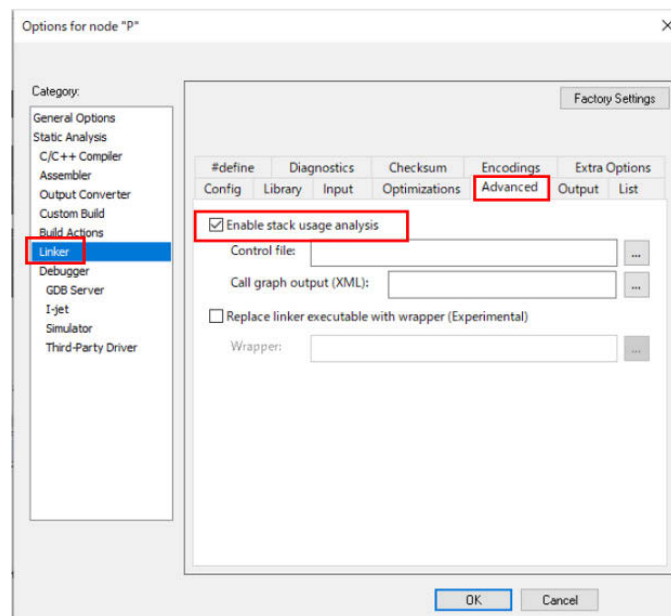
the file size will be larger. After all, there is debug information, but the ROM size is the same with or without debug information. It is also possible to remove the debug information (using the command ielftool provided by EWRISCV).

If you also look at the List option, you can output a MAP file by enabling the "Generate linker map file." If you have completed the build but do not have a MAP file, please check here.



EWRISCV can also perform stack analysis, which is specified in the linker options. Check [Linker]—[Advanced]—[Enable stack usage analysis] to

perform the analysis. The analysis results are output to MAP, but they can also be output in XML.

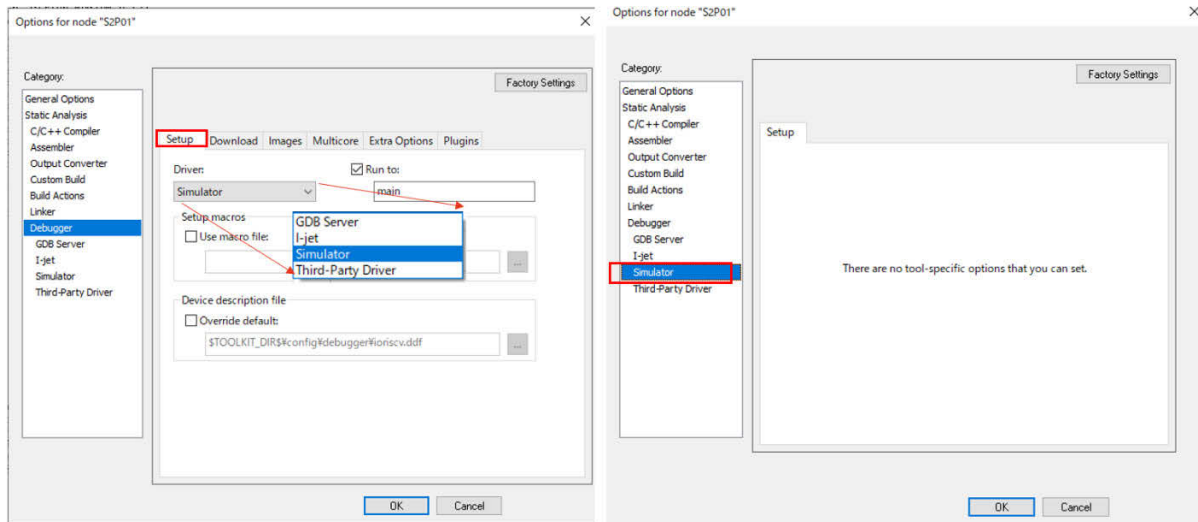


2.3.5 Debugger

EWRISCV has several debug interfaces, which can be switched between options. In this section, we will check the settings for using the simulator. Select the category Debugger and check the Setup option. In the Device section,

you can select GDB Server, I-jet, Simulator, etc. In Chapter 2, we will use the simulator to check the operation.

On the right side of the figure below, there is also a Simulator in the category, but there are no extra options even if you select it.



2.4 Understanding the RISC-V project as a whole

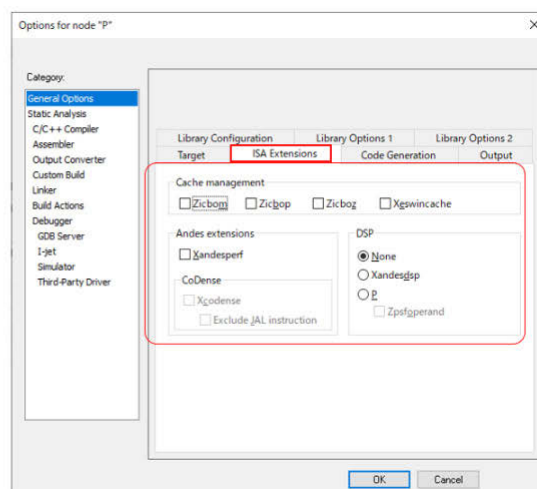
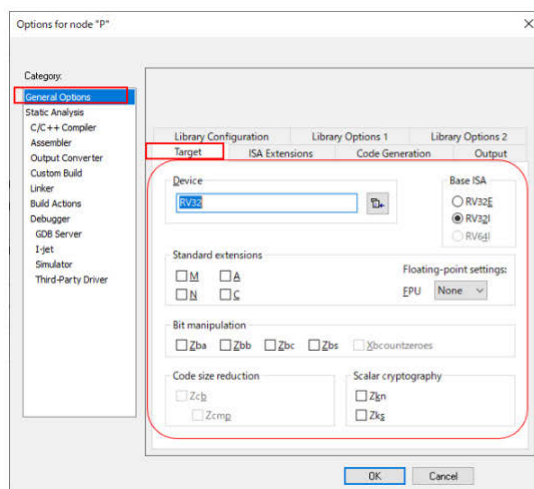
In embedded software, it is not enough to create a C language source program. You need to configure the CPU cores in the startup code, configure the peripheral hardware, and initialize the variables. To do this, you also need to create a linker configuration file. Here, we will explain the basics.

2.4.1 Creating sample 2

First, take a look at the following simple C program. This is a simple program that adds the values of two arrays, da, and db, to the array ha, subtracts the values of the arrays da and db, stores them in the array hb, and updates the arrays da and db with the result.

```
#define N (10)
int da[N]={11,12,13,14,15,16,17,18,19,20};
int db[N]={ 1, 2, 3, 4, 5, 6, 7, 8, 9,10};
int ha[N];
int hb[N];
int main( void )
{
    int i;
    for ( i=0 ; i < N ; i++ ) {
        ha[i]=da[i]+db[i];
        hb[i]=da[i]-db[i];
    }
    for ( i=0 ; i < N ; i++ ) {
        da[i]= ha[i];
        db[i]= hb[i];
    }
    return 0;
}
```

At this time, the following settings are made so that only RV32I instructions are used in the program.

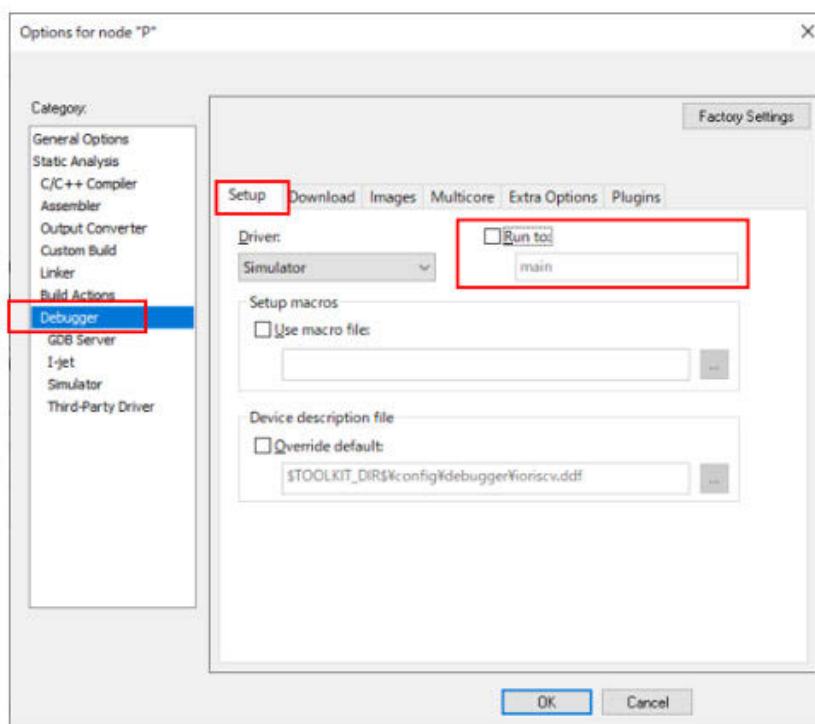


2.4.2 Running sample 2

After doing Make, click Download and Debug. In the debugger, select the variable da in the editor, as shown below, right-click, and select “add to watch.” Do this for the variables db, ha, and hb. Then, when you arrive at the main function, you can see that all the variables have been initialized. Who did this? Other things are essential to run the program but are not done in the program. The following content is implemented in the startup code.

- Arrays da and db need to be initialized with initial values.
- Arrays ha and hb are zero and need to be initialized.
- Configuring the Stack Pointer
- Configuring the Global Pointer
- Interrupt vector setting (you don't have to do it because you won't use an interrupt this time)

However, with the default settings, the debugger is running as far as the main function at the start, so you need to change the optional setting to see the startup code run. As shown in the figure below, uncheck [Run to] in [Debugger]-[Setup].



If you start debugging again with [Download and Debug], you can check the operation of the program immediately after the reset. The illustration below is illustrated, but an important point will be added. RISC-V provides a register called GP (Global Pointer). When accessing the memory, it can be executed faster than without such a mechanism by having it calculated at the offset from the GP value. At this time, you can see that it is set with two instructions, LUI instruction and ADDI instruction.

The lui instruction is an instruction that allows you to set a 20-bit immediate value on the upper

side of the register. In addition, addi can specify a 12-bit value immediately. By using these two instructions, the 32-bit address is set to GP. With IAR, you can choose between two options: leave the compiler to initialize variables or let the user do it in their code. In this case, the linker configuration file is set to leave the initialization to the compiler, so the variable initialization is performed in the function `__iar_data_init2`. For details, please refer to the ILINK documentation [2].

```

_iar_program_start:
0x2000'0004: 0x0000'0013  nop
_iar_cstart_init_gp:
0x2000'0008: 0x8000'01b7  lui    gp, 0x80000
0x2000'000c: 0x0501'8193  addi  gp, gp, 0x50
0x2000'0010: 0xf140'2573  csrr  a0, mhartid
0x2000'0014: 0x8000'1537  lui    a0, 0x80001
0x2000'0018: 0x0a05'0513  addi  a0, a0, 0xA0
0x2000'001c: 0xff05'7113  andi  sp, a0, -0x10
0x2000'0020: 0x2000'0537  lui    a0, 0x20000
0x2000'0024: 0x3005'0513  addi  a0, a0, 0x300
0x2000'0028: 0x3055'1073  csrw  mtvec, a0
0x2000'002c: 0x2100'00ef  call20 _low_level_init
0x2000'0030: 0x0005'0483  beqz  a0, 0x2000'0038
0x2000'0034: 0x2100'00ef  call20 _iar_data_init2
0x2000'0038: 0x0000'0013  nop
0x2000'003c: 0x0000'0513  mv    a0, zero
0x2000'0040: 0x00c0'00ef  call20 main
0x2000'0044: 0x23c0'00ef  call20 exit
0x2000'0048: 0x0000'006f  j     0x2000'0048

```

GP settings (0x80000050)

SP settings (0x800010a0)

Vector table settings
mtvec(0x2000'0300)

Call of the initialization routine

Call of the main function

In the startup code, the global pointer value, the stack pointer value, and the vector table value were set. At this time, the method for checking the address utilized by the startup code will be explained. The addresses of variables and functions placed by the linker are listed in the MAP file. Once the MAP file is made, it will be displayed under Output on the Workspace screen. In this case, it is a file called P.map, so you can open it in the editor screen by double-clicking it.

In the MAP file, where there is an ENTRY LIST, you can check the placement address of variables and functions (see the figure below). \$\$Base indicates the beginning of the region, and \$\$Limit indicates the next address of the region. For example, in CSTACK, the 0x800000a0 to 0x800010a0 area is defined as a stack. It is actually used in the area up to 0x800000a0~0x8000109.

The screenshot shows the IAR Embedded Workbench workspace with the linker map file (P.map) open. The workspace tree on the left shows the project structure. The main window displays the linker map file content with the following annotations:

- Area with variable initialization information .iar.init_table
- Stack area CSTACK
- Interrupt handler
- GP pointer when accessing memory

Now `_iar_static_base`, let's take a look at `$$GPREL` as there are things to confirm. In order to access memory using a global pointer, EWRISCV needs it to be specified in the linker configuration file. The specific specification is shown in the figure below. To briefly describe the linker configuration file, define the memory area to be used in the "define region." Here, we

have defined the RAM and ROM area. What is important is the definition of the block `RW_DATA` on line 38. In this block, the data of the read-write attribute is placed, but here, it is specified that it is accessed relative to GP because it is with static base `GPREL`. This `RW_DATA` is placed in the memory area `RAM_region32`. For more information on `ILINK`, see Reference [6].

```

IAR Information Center for RISC-V | main.c | P.map | P.out | generic.icf x | _dbg_break.c [RO]
1 ////////////////////////////////////////////////////////////////////
2 // RISC-V ilink configuration file.
3 //
4
5 build for rom;
6
7 define exported symbol _link_file_version_2 = 1;
8 keep symbol __iar_cstart_init_gp; // defined in cstartup.s
9
10 define memory mem with size = 4G;
11
12 define region RAM_region32 = mem:[from 0x80000000 to 0x8003FFFFFF];
13 define region ROM_region32 = mem:[from 0x20000000 to 0x3FFFFFFF];
14

```

Define RAM area from 0x80000000 to 0x8003FFFF

```

36 }
37
38 define block RW_DATA with static base GPREL { rw data };
39
40 "CSTARTUP32" : place at start of ROM_region32 { ro section .cstartup };
41
42 "ROM32" : place in ROM_region32 { ro,
43 block MINTERRUPTS };
44
45 "RAM32" : place in RAM_region32 { block RW_DATA,
46 block HEAP,
47 block CSTACK };
48

```

Place data with readwrite attribute in block RW_DATA. Declare that it will be placed using GPREL (GP relative).

Place block RW_DATA, heap and stack in RAM area. The heap was eventually removed since it was not used.

2.4.3 About GP relative

If you create software without being aware of assembler instructions on a daily basis, you may not be able to understand the GP relative. Here, how are C variables accessed GP-relative? Consider the case where you don't use GP relative.

In the previous program, we extracted the part implemented in GP relative. The value set for GP is `__iar_static_base` 0x80000058 denoted by `$$GPREL`. On the other hand, the variable `ha` is 0x80000054, and the variable `hb` is x8000007c. In order to access the variable `ha` or the variable `hb`, this address must be set in a register.

To set a 32-bit address in a register, for example, you can execute a combination of LUI and ADDI instructions. However, if you set it every time you access a variable, the code size will increase, and the execution time will be slow.

GP relative is a method of calculating the address of the memory access starting from the address in GP. In the figure below, in order to write `ha[i]`, the value of the GP register is first copied to register `a2`, the value of index `ix4` (because it is an int type) is added to `a2`, and the value is written with the `sw` instruction. Where the index is multiplied by 4, it is implemented by shifting to the left.

Placement and size of variables

<code>__iar_static_base\$\$GPREL</code>	0x8000'0058
<code>da</code>	0x8000'0000
<code>db</code>	0x8000'0028
<code>ha</code>	0x8000'0054
<code>hb</code>	0x8000'007c

// The part where `ha[i]` is assigned in `ha[i]=da[i]+db[i]`;

```

addi a2, gp, -4
slli a3, a0, 2
add a2, a2, a3
sw a1, 0(a2)

```

Subtracting -4 from GP becomes the start address of ha

Since a0 has the index of the array, use x4 to calculate the size of int type

Add the value of index x4 to the first address of a2, calculate the address of ha[i], and store it in a2

Write the operation result to address a2.

Let's see what happens if we don't use GP relative. In order not to use GP relative, it is necessary to change the linker settings. By default, the linker settings of IAR were set to use GP relative, as shown in the upper part of the figure below. Until now, the data to be accessed

relative to GP (in this case, the read-write data attribute data) was stored in a block `RW_DATA`, and the block was placed in the `RAM_region32`. In order to avoid using GP relative, we stopped defining block `RW_DATA` and placed `rw data` in the `RAM_region32` as it is.

```
define block RW_DATA with static base GPREL { rw data };

"RAM32":place in RAM_region32 { block RW_DATA,
                                block HEAP,
                                block CSTACK};
```



```
// define block RW_DATA with static base GPREL { rw data }; //コメントアウト

"RAM32":place in RAM_region32 { rw data,
                                block HEAP,
                                block CSTACK};
```

If you leave it as it is, you will get an error in the startup code. In addition, the part where the GP is

configured is commented out in the startup code (cstartup. s).

```
136 PUBLIC __iar_cstart_init_gp
137 __iar_cstart_init_gp:
138 cfi ?RET Undefined
139 EXTERN __iar_static_base$$GPREL
140 .option push
141 .option norelax
142 ;; lui gp, %hi(__iar_static_base$$GPREL)
143 ;; addi gp, gp, %lo(__iar_static_base$$GPREL)
144 // la gp, __iar_static_base$$GPREL
145 .option pop
146 REQUIRE ?cstart_init_gp
147
148 CfiEnd !|
149
```

As a result, the generated code has changed as follows: It is supposed to use the lui instruction and addi to set the address. The point is that the number of instructions when executing the same

process is different when GP relative and this GP relative are not used. In many cases, using GP relative produces shorter code. Shorter code is generated, which also results in faster execution.

da	0x8000'0000
db	0x8000'0028
ha	0x8000'0050
hb	0x8000'0078

// Write ha[i]=da[i]+db[i]; to ha (retains the result added to a1)

```
lui a3, 0x80000
addi a3, a3, 0x50
slli a2, a0, 2
add a2, a3, a2
sw a1, 0(a2)
```

Array ha is 0x8000 0050, so set a2 with lui and addi instructions

Since the array is of int type, 4 times the index is required, and the access address is calculated as a2=a2+s3.

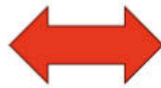
Write the operation result a1 to the address held by a2

Let's examine the output instruction sequence with and without GP relative. It is clear that the part that sets the variable's address is GP relative,

which can be done with one instruction. However, if GP relative is not used, two instructions, lui instruction, and addi instruction, are required.

With GP relative
reference

```
addi  a2, gp, -4  
slli  a3, a0, 2  
add   a2, a2, a3  
sw    a1, 0(a2)
```



Without GP relative
reference

```
lui   a3, 0x80000  
addi  a3, a3, 0x50  
slli  a2, a0, 2  
add   a2, a3, a2  
sw    a1, 0(a2)
```

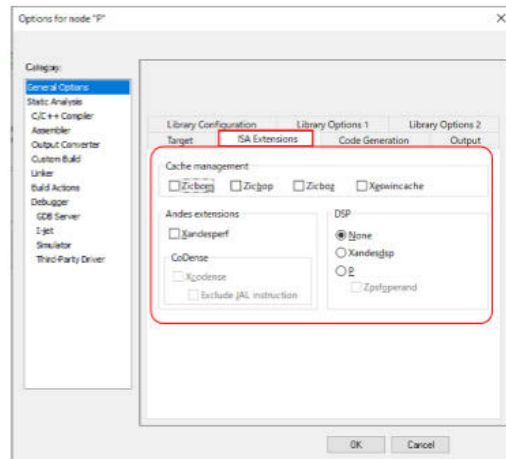
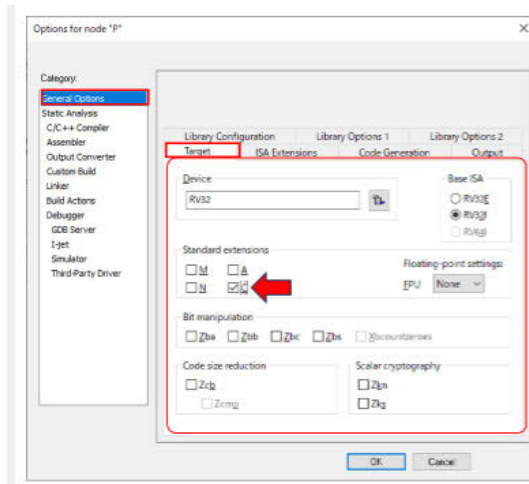
In summary, GP relative is an important feature of using RISC-V, and it may result in a smaller code size and faster execution speed. In order to generate GP-relative code in EWRISCV, you need

to define a block with a static base GPREL in the linker configuration and place the block with a static base.

2.5 C extension instructions

The basic instruction of RISC-V is a 32-bit instruction. However, this C extension provides 16-bit instructions. As a result, the program's code size can be reduced, so the limited FLASH (ROM) area can be used effectively. Let's see how much of a difference it actually makes.

The project created previously in 2.4 used the RV32I instruction set. This time, an attempt will be made to enable the use of the C extension instruction as well. In the project options, check [C] in [General Options]-[Target]-[Standard Extensions].



Let's compare the results. It affects the code memory and the read-only data in the arrangement. In terms of code, 704 bytes have been drastically reduced to 472 bytes.

Since the amount of memory that can be used in embedded systems is smaller than that of personal computers, C expansion is very effective.

	RV32I	RV32I+C
readonly code memory	708	476
readonly data memory	108	112
readwrite data memory	4,300	4,344

Let's use a different example. The following code is a function that adds eight arguments and changes the result.

```
int hoo(int a, int b, int c, int d, int e, int f, int g, int h ) {
    return a+b+c+d+e+f+g+h;
}
```

This is the left of the figure below when it is compiled with RV32I, and the right figure below when it is compiled with RV32I + C.

Extended instructions are prefixed with c., so addition becomes c.add instructions. On the RV32I, the add instruction specified the operation with three operands, but the c.add instruction is now 16 bits, so it can only take two operands. The first operand is both the register that defines the

result and is also the input. Suppose you write the instruction c.add a0 a1 in mathematical terms, $a0 = a0 + a1$. If you check the machine language in the figure below, you can see that the RV32I is 32 bits (8 hexadecimal digits), but the C extension is 16 bits (4 hexadecimal digits).

RV32I

```

hoo:
0x00b50533 add a0, a0, a1
0x00c50533 add a0, a0, a2
0x00d50533 add a0, a0, a3
0x00e50533 add a0, a0, a4
0x00f50533 add a0, a0, a5
0x01050533 add a0, a0, a6
0x01150533 add a0, a0, a7
0x00008067 ret
Machine code

```

RV32I+C

```

hoo:
0x952e c.add a0, a1
0x9532 c.add a0, a2
0x9536 c.add a0, a3
0x953a c.add a0, a4
0x953e c.add a0, a5
0x9542 c.add a0, a6
0x9546 c.add a0, a7
0x8082 c.ret
Machine code

```

There is a possibility of reducing the code size by using C extension instructions (Compress). Not all instructions have extended instructions. If you are coding this Compress instruction manually, the important point is to “use or not to use,” but for those who use C or C++, you

can switch with the compiler option. In order to make effective use of FLASH (ROM), which is limited in embedded systems, we would like you to consider using it on the premise that it will be used.

2.6 M Extension instructions

The M extension instruction allows you to perform integer multiplication and division. If there is no M extension, multiplication and division will be performed by software. Looking only at the C/C++ level, wouldn't it be good if it could be executed? You might think. So, the focus now is to look at the actual code here and see what the

code generated with and without the M extension instruction will look like.

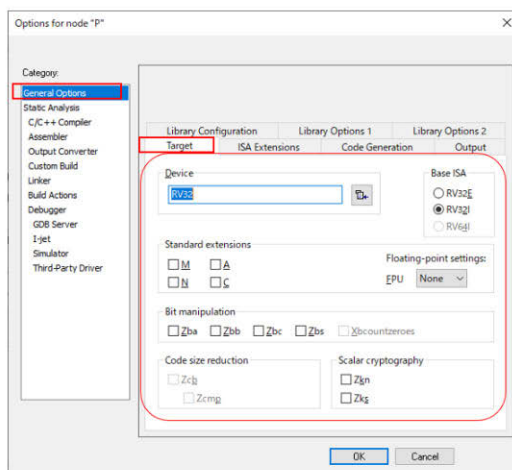
2.6.1 Creating sample 3

Now, let's look at the generated code for the following program. In this project, we will first use only the RV32I.

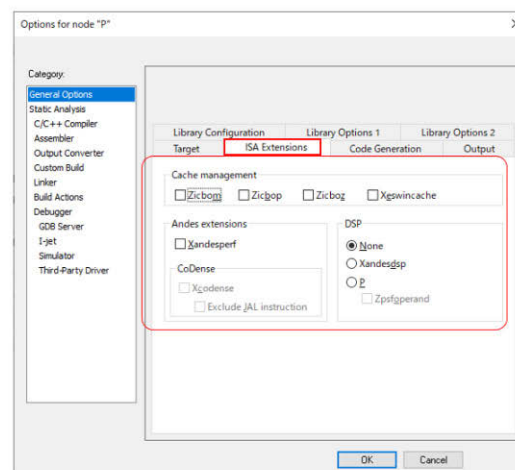
```
int a=100;
int b=4;
volatile int c;
volatile int d;
int main( ) {
    c = a*b;
    d = a/b;
    return 0;
}
```

In this case, we are using multiplication (*) and division (/) in the program. In this project, settings

are configured for RV32I. There are no hardware multiplication or division instructions.



In the figure below, the variables and functions related to this time from the map file on the left have been extracted. `__iar_imul` to calculate multiplication and `__iar_idivmod` to calculate



division are implemented. On the right side of the figure below, the generated code for the part to be multiplied and divided is included.

Placement and size of variables and functions

__iar_imul	0x2000'01fc	
__iar_idivmod	0x2000'0048	
__iar_static_base\$\$GPREL		
	0x8000'0008	
a	0x8000'0000	0x4
b	0x8000'0004	0x4
c	0x8000'0008	0x4
d	0x8000'000c	0x4

Generated code

```
// c = a*b;
lw  a0, -8(gp)
lw  a1, -4(gp)
jal  t0, __iar_imul
sw  a0, 0(gp)

// d = a/b;
lw  a0, -8(gp)
lw  a1, -4(gp)
jal  t0, __iar_idivmod
sw  a0, 4(gp)
```

GP-8 becomes the address of variable a and loads from there

GP-4 becomes the address of variable b and loads from there

Multiplication is performed using a function called __iar_imul

Since the calculation result is in a0, GP+0 becomes variable c and the value is written there

GP-8 becomes the address of variable a and loads from there

GP-4 becomes the address of variable b and loads from there

Perform division using a function called __iar_idivmod

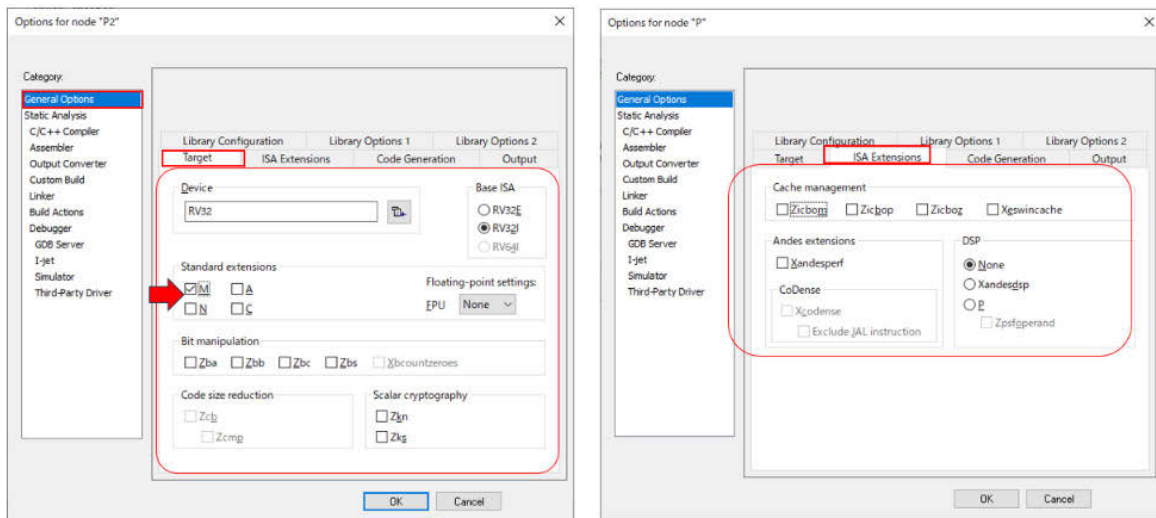
Since the calculation result is in a0, GP+4 becomes variable d, and the value is written there

Make a note of the code size at this time.

```
820 bytes of readonly code memory
32 bytes of readonly data memory
4'236 bytes of readwrite data memory
```

2.6.2 Enabling M extension instructions

Now, let's try to enable the M extension instruction. Set the options as follows:



The generated code should look like this: The map file on the right side of the figure below does not have a software multiplication/division

function. In the generated code on the right of the figure below, you can see that the mul and div instructions of the M extension are used.

Placement and size of variables and functions

_iar_static_base\$\$GPREL		
	0x8000'0008	
a	0x8000'0000	0x4
b	0x8000'0004	0x4
c	0x8000'0008	0x4
d	0x8000'000c	0x4

Generated code

```
// c = a*b;
lw  a0, -8(gp)
lw  a1, -4(gp)
mul  a0, a0, a1
sw  a0, 0(gp)

// d = a/b;
lw  a0, -8(gp)
lw  a1, -4(gp)
div  a0, a0, a1
sw  a0, 4(gp)
```

GP-8 becomes the address of variable a and loads from there

GP-4 becomes the address of variable b and loads from there

Execute multiplication instruction mul

Since the calculation result is in a0, GP+0 becomes variable c and the value is written there

GP-8 becomes the address of variable a and loads from there

GP-4 becomes the address of variable b and loads from there

Execute division instruction div

Since the calculation result is in a0, GP+4 becomes variable d and the value is written there

Looking at the code size at this time, it was as follows.

```
500 bytes of readonly code memory
32 bytes of readonly data memory
4'168 bytes of readwrite data memory
```

Only the code memory is different with the M extension. 820 bytes equals 500 bytes, which makes the code size about 60%. However, it is not always 60%. It means that the number of libraries executed by the software has increased by about 300 bytes, so the impact will be smaller for large-scale software.

When the execution cycle was examined at this time, the following was discovered. The execution cycle fluctuates depending on the data, so it's good to look at this example as a reference, but the multiplication and division performed by hardware can be performed at high speed.

	RV32I	RV32I+M
c = a*b;	24	4
d = a/b;	59	4

2.6.3 RV32M

The extended instructions on the RV32M are as follows. I will also explain how to name the instructions at this time. MUL, DIV, and REM are the multiplication, division, and extras of basic 32-bit instructions. In multiplication, 32-bit and 32-bit results take 64-bit values. The instruction MULL stores the result of the upper 32 bits.

At this time, it is necessary to use MULH, MULHSU, and MULHU, depending on the value of the source register signed or unsigned. In the case of MULH, the two sources are signed, in the case of MULHU, the two sources are unsigned, and in MULHSU, rs1 is signed and rs2 is unsigned.

- MUL rd,rs1,rs2
- MULH rd,rs1,rs2
- MULHSU rd,rs1,rs2
- MULHU rd,rs1,rs2
- DIV rd,rs1,rs2
- DIVU rd,rs1,rs2
- REM rd,rs1,rs2
- REMU rd,rs1,rs2

2.7 A extension instructions

The RV32A's atomic instructions are the following. These are required when multiple processes are running simultaneously. Typically, in a multitasking environment with an operating system (including RTOS), various tasks are used to access shared data correctly. When the OS is not used, it is generally used to manipulate shared data in interrupt processing.

- AMOSWAP.W rd, rs2,(rs1)
- AMOADD.W rd, rs2,(rs1)
- AMOXOR.W rd, rs2,(rs1)
- AMOAND.W rd, rs2,(rs1)
- AMOOR.W rd, rs2,(rs1)
- AMOMIN.W rd, rs2,(rs1)
- AMOMAX.W rd, rs2,(rs1)
- AMOMINU.W rd, rs2,(rs1)
- AMOMAXU.W rd, rs2,(rs1)
- LR.W rd, (rs1)
- SC.W rd,rs1,(rs2)

Instructions preceded by AMO perform Read-Modify-Write. AMO is an abbreviation for atomic memory operation. For example, AMOOR.W rd, rs2, (rs1) performs the following actions inseparably:

1. Read the value from the memory address indicated by 1.rs1
 2. OR with the value read from memory and the register value of rs2
 3. At the same time as writing the result of the OR to memory, the value of rs2 is stored in register rd. LR.W and SC.W are Load-Reserved and Store-Conditional instructions.
- LR.w rd, (rs1) Reads the memory value that is rs1 and stores it in register rd. Record the reservation for that memory.
 - SC.w rd,rs2,(rs1) If there is a reservation for the address indicated by rs1, write the contents of rs2, and if the store is successful, the value of rd is set to zero. Otherwise, it writes a non-zero error code.

The following examples of how to use the Load-Reserved and Store-Conditional instructions are shown in the specification.

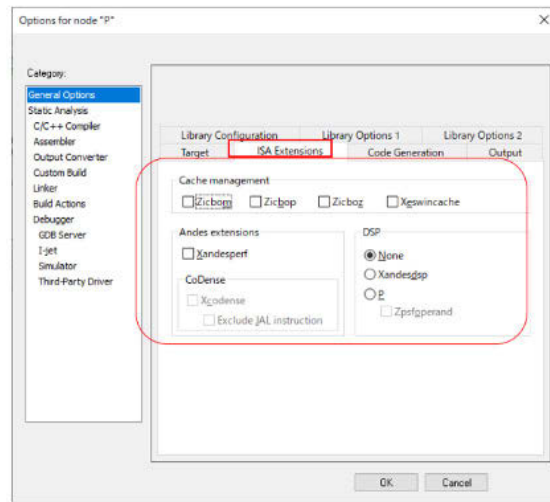
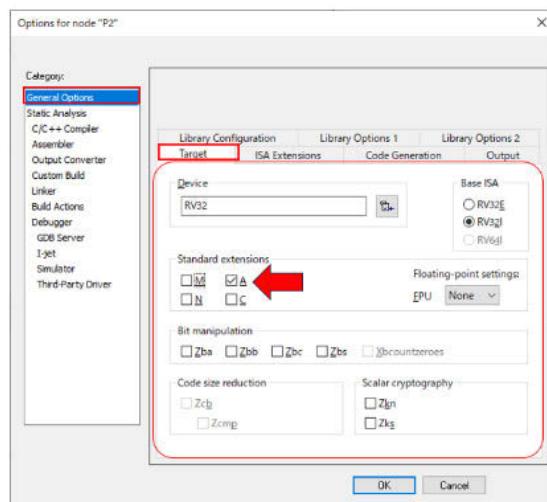
```
# a0 holds address of memory location
# a1 holds expected value
# a2 holds desired value
# a0 holds return value, 0 if successful, !0 otherwise
cas:
    lr.w t0, (a0) # Load original value.
    bne t0, a1, fail # Doesn't match, so fail.
    sc.w t0, a2, (a0) # Try to update.
    bnez t0, cas # Retry if store-conditional failed.
    li a0, 0 # Set return to success.
    jr ra # Return.
fail:
    li a0, 1 # Set return to failure.
    jr ra # Return.
```


2.7.1 Creating sample 4 with A extension instructions

A extension instructions are not generated from code written in C/C++ language. It must be written using assembler instructions or an inline assembler. Let's actually look at a sample that

calls assembler from C and runs the AMOADD instruction in the assembler.

Set the following to enable the use of the A extension in the project options.



The C language and assembler programs are shown below. Let's start with the C code on the left. The fact that the @ 0x80000000 is appended to the declaration of the variable d2 is an extension of IAR. This function places the address of the number after @ in the variable. This places the variable d2 in the 0x80000000. Inside the main function, we are calling function f001, but the function f001 is defined in the assembler.

Let's take a look at the assembler code on the right. Many people have not seen many assembler programs, so that will be explained in detail here. In assembler, the items described from the first column are described as labels. Items other than labels should be listed in the second column or later. In this assembler file,

only the fifth line, f001, is listed in the first column. In assembler, both variables and functions are labeled with names, so keep this in mind. Then, in the second line, public f001, the label f001 can be referenced from an external file. If this public definition is not present, it will be accessed only by file.

In the fourth line, the section is set to place the object to be described from here. The section name is ".text" and the memory type is CODE. There are three types of memory: CODE, CONST, and DATA. They mean codes, constants, and variables, respectively. The NOROOT after it is an instruction that if this part is not needed, it can be deleted. If you want to keep functions that you don't use, use the keyword ROOT. The last (2) is aligned with 2 bytes.

```

main.c x
main()
1 #include <stdio.h>
2
3 extern void fool();
4
5 int d2 @ 0x80000000;
6
7 int main( void ) {
8     int t;
9     t=f001();
10    printf("d2=%d, ret=%d\n", d2,t);
11    t=f001();
12    printf("d2=%d, ret=%d\n", d2,t);
13    return 0;
14 }
15

test.S x
1
2     public f001
3
4     SECTION `.text`:CODE:NOROOT(2)
5 f001:
6     lui t0, 0x800000;
7     addi t0,t0, 0x000;
8     addi t1, zero, 0x001;
9
10    amoadd.w a0,t1,(t0)
11    ret
12
13
14    END
15

```

From line 6 onwards, it is the part that is programmed using assembler instructions. This explanation is attached to the figure below. Note

also that the assembler file will result in an error if there is no END at the end.

```

f001:
    lui t0, 0x80000;
    addi t0,t0, 0x000;
    addi t1, zero, 0x001;

    amoadd.w a0,t1,(t0)
    ret

END

```

Set register t0 to 0x8000 0000
 Set 0x001 to register t1 (zero+0x001)
 Store the value of address t0 in a0, and write the result of that value + t1 back to address t0
 Return from function f001, return value is in a0

The result of the terminal I/O that will be executed is also attached. You can see that amoadd returns the value before the change.

```

main.c x
main()
1 #include <stdio.h>
2
3 extern void fool();
4
5 int d2 @ 0x80000000;
6
7 int main( void ) {
8     int t;
9     t=f001();
10    printf("d2=%d, ret=%d\n", d2,t);
11    t=f001();
12    printf("d2=%d, ret=%d\n", d2,t);
13    return 0;
14 }
15

Terminal I/O
Output:
d2=1, ret=0
d2=2, ret=1

Input:
Ctrl codes Options...
Buffer size: 0

```

2.8 N extension instructions

As explained previously in the 1.4.5 operating mode, interrupts cannot be received in user mode. This N extension enforces interrupts

and exceptions at the user level. CSR has been extended, and instructions have been added for interrupting at the user level.

The following are the enhancements of CSR.

CSR address	Name	memo
0x000	ustatus	User Status Register
0x004	uie	User level interrupt enable register
0x005	utvec	User interrupt handler base address
0x040	uscratch	Scratch register for user interrupts
0x041	uepc	user interrupt PC
0x042	ucase	User exception cause
0x043	utval	User trap value register
0x044	uip	User interrupt pending register

The following instructions are added as additional instructions: This is an instruction to return from an interrupt at the user level.

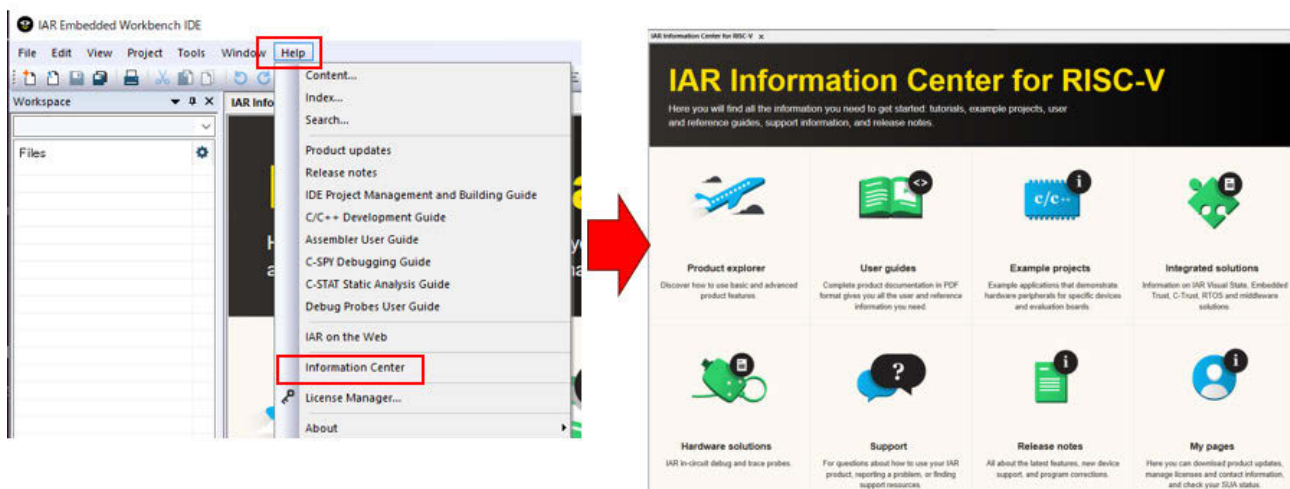
- uret

2.9 Custom instructions

RISC-V is characterized by modular configuration and custom instructions. EWRISCV incorporates a mechanism for managing custom instructions, which will be introduced here.

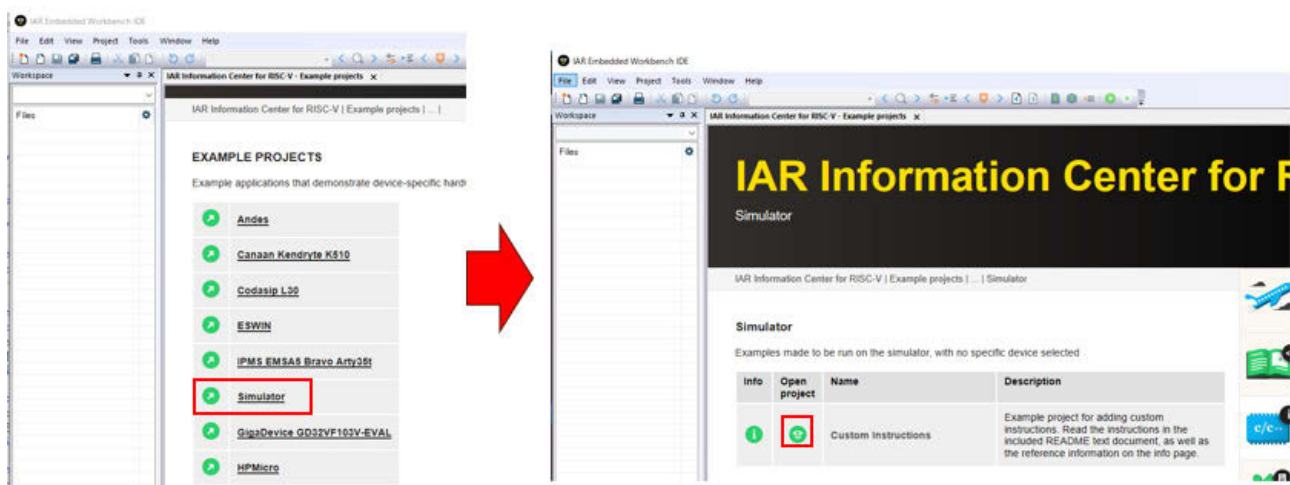
2.9.1 Opening the IAR information center examples

From the EWRISCV screen, open [Help]—[Information Center]. The Information Center provides a wide range of information, including a sample project. Now, select Example Projects.



Then, the screen will change, select [Simulator], and then select [Open Project] under [Custom Instructions]. You will then be asked which folder

to store, so please specify it. This will open the project.



2.9.2 RISC-V operation codes

RISC-V allows custom instructions. However, the following operation codes are determined. In this example, we are using Reserved space.

This was created with reference to [6], and while it has not been modified, if it were to be actually made, it might be advisable to utilize the sections described as custom-0 / custom-1 / custom-2 / custom-3.

inst[4:2]	000	001	010	011	100	101	110	111
inst[6:5]								(> 32b)
00	LOAD	LOAD-FP	custom-0	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	custom-1	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	reserved	custom-2/rv128	48b
11	BRANCH	JALR	reserved	JAL	SYSTEM	reserved	custom-3/rv128	≥ 80b

Table 24.1: RISC-V base opcode map, inst[1:0]=11

Before you can create custom instructions, you need to understand the six instruction types in RISC-V. Depending on the instruction, an immediate value may be specified, or a register may be specified, and the number of registers is different for each.

- R type: An instruction type that specifies two register inputs and one register output, such as R=R+R
- I type: An instruction type that specifies one register input, an immediate value, and one register output, such as R=R+I.
- S-type: Instruction type that specifies how to write (store) to memory, etc.
- Type B: An instruction type that specifies two registers, such as a conditional branch
- U type: Instruction type that handles 20-bit immediate values
- J type: Instruction type used for unconditional jumping, etc.

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7			rs2		rs1	funct3		rd		opcode				R-type
imm[11:0]					rs1	funct3		rd		opcode				I-type
imm[11:5]			rs2		rs1	funct3		imm[4:0]		opcode				S-type
imm[12:10:5]			rs2		rs1	funct3		imm[4:1 11]		opcode				B-type
imm[31:12]								rd		opcode				U-type
imm[20:10:1 11 19:12]								rd		opcode				J-type

2.9.3 Custom instruction

Now, in the sample based on reference [6], we implement the integer mod operation (remainder). Let's decide on the custom instruction at that time. The mod arithmetic instruction is of the form R = R%R, so it is an R type. For R-type instructions, it is necessary to

determine inst (the part described as an opcode), funct3, and funct7. For rd, rs1, and rs2, specify the register number in the same way as other R instructions.

- The value of inst[6:0] is 1101011 (0x6B) in binary.
- The value of funct3 is 000 (0x0) in binary.
- The value of funct7 is 0010000 (0x10) in binary.

2.9.4 Using custom instructions in code

In EWRISCV, it is possible to write inline assembler code that corresponds to custom instructions. Please refer to the manual for details, but here, let's check the implementation of the mod operation instruction decided in 2.9.3. I'm using an inline assembler to use custom instructions in C. `.insn r` is an R-type instruction,

followed by the values of `inst`, `funct3`, and `funct7` as `0x6B`, `0x0`, `0x10`, and then the registers. `%0` is the destination and `%1` and `%2` are the sources. Behind that, we are mapping variables with `%0,%1,%2`. For `%0`, `%1`, and `%2`, the compiler allocates them to the appropriate registers, which is a convenient notation when using an inline assembler in C.

```
int modulo(int a, int b) {
    int r;
    __asm(".insn r 0x6B, 0x0,0x10,  %0,%1,%2" : "=r"(r) : "r"(a), "r"(b) );
    return r;
};
```

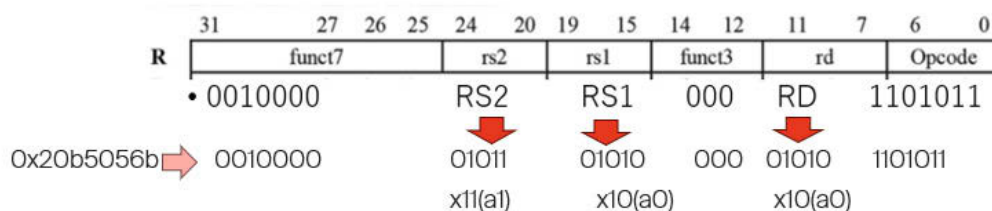
If you look at the actual build result with this, it will be as follows. The instructions in question are implemented in 32-bit instructions and

are `0x20b5056b`. Unfortunately, for custom instructions, the EWRISCV disassembler display is `Unknown32`.

```
__asm(".insn r 0x6B, 0x0,0x10,  %0,%1,%2" : "=r"(r) : "r"(a), "r"(b) );
modulo:
0x2000'0320: 0x20b5'056b  Unknown32
return r;
0x2000'0324: 0x8082      c.ret
```

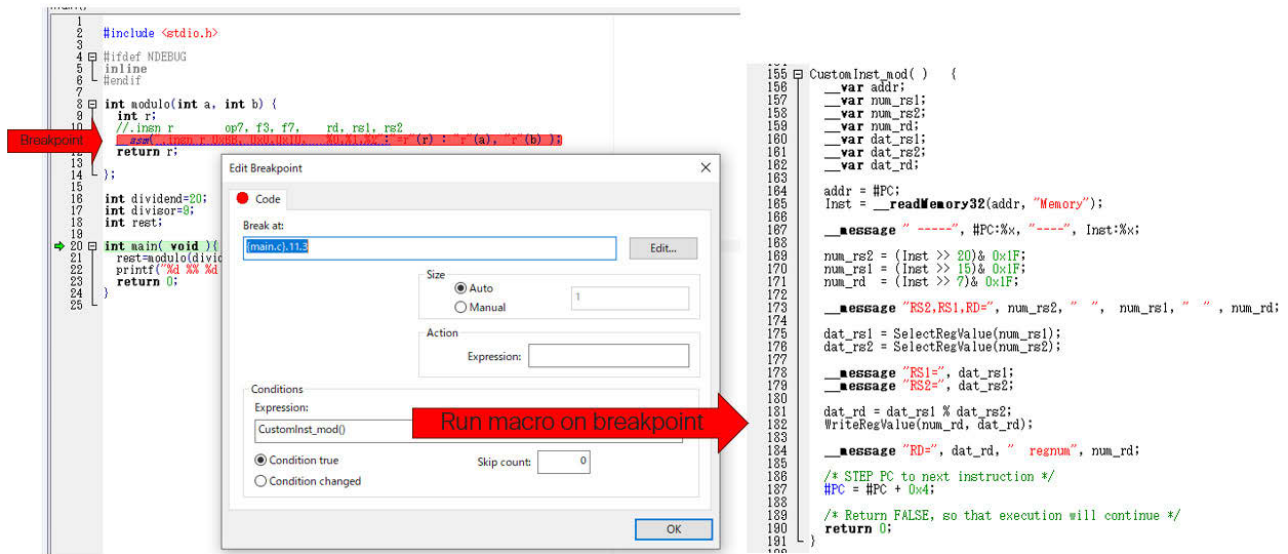
The analysis of the `0x20b5056b` machine language is as follows. In machine language, it is specified by the register name, so `rs1` is `x10`

and `rs2` is `x11`. If you write the ABI name, `rs1` is `a0`, and `rs2` is `a1`. `A0` and `A1` are the first and second arguments of the function.



2.9.5 Using custom instructions in the simulator

The EWRISCV simulator is not capable of executing custom instructions. But this project can be executed properly. In fact, we have a breakpoint at the address where we put the custom instruction, and we have a macro function that executes when the breakpoint is reached.



Here, we will introduce only a part, so if necessary, please check it while actually running the sample with EWRISCV. A breakpoint is set, and the macro function `CustomInst_mod` is executed when it is set. In this macro function,

registers `rd`, `rs1`, and `rs2` are analyzed to simulate MOD operations. Also, when this macro function is executed, `PC=PC+4`. This makes it possible for the simulator to execute custom instructions without knowing them.

2.10 About function calls/ABIs

2.10.1 C language functions

Let's take a look at what kind of RISC-V instructions are used when calling and returning from C functions.

```
int cnt=0;
void add1( )
{
    cnt++;
}
void foo() {
    add1();
}
```

The following is the output assembler instruction. Here, the optimization is compiled on low. To call the function add1 from the function foo, we are calling call add1. In fact, there is no call in the RISC-V instruction explained earlier. The call is

a pseudo-instruction for the JAL ra imm. ret is a pseudo-instruction of jalr x0, 0(x1) and is the return instruction from the function. x0 is the zero register, and x1 is the ra (return address).

```
add1:
    lw      a0, 0x18(gp)
    addi   a0, a0, 1
    sw     a0, 0x18(gp)
    ret
foo:
    addi   sp, sp, -0x10
    sw     ra, 0xC(sp)
    call20 add1
    lw     ra, 0xC(sp)
    addi   sp, sp, 0x10
    ret
```

2.10.2 Rules for calling functions

Here, we will use the RV32I as an example to explain the ABI. First, we will classify the registers into three categories.

1. Scratch registers t0~t6, ft0~ft11, a0~a7, fa0~fa7
2. Storage registers s0~s11 and fs0~fs11,
3. Application-specific registers SP/x2, GP/x3, RA/x1

The scratch register is a function call, and its value can be changed within the function to become an OK register. On the other hand, if you want to change the value of a save register in a function, you must save it programmatically and return it when the function ends. The special-purpose registers are the stack pointer sp, the

global pointer gp (used for variable access), and the return register ra, which holds the function's return value.

Now, let's take a look at what happens when the function is called. Integer values and pointer values are passed using registers a0~a7. If it is missing, use the stack. The return value on the RV32I is a0. For 64-bit data, use a0 and a1 to return.

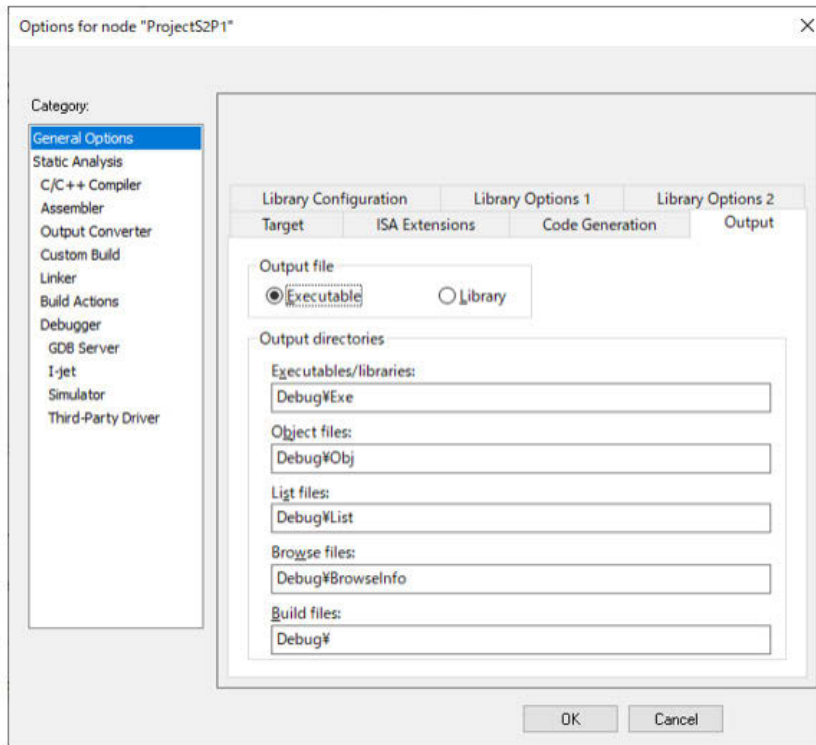
Now, let's see how the function call is realized.

The following function has 12 arguments of type int. Up to 8 arguments can be passed, so four are passed in a stack. The following is the generated assembler code, but at the beginning of the function, we use the lw instruction to load the values of four variables into the temporary register via the stack. The other eight variables are passed in the a0~a7 registers.

```
f:
    lw    t0, 0(sp)
    lw    t1, 4(sp)
    lw    t2, 8(sp)
    lw    t3, 0xC(sp)
    add   a0, a0, a1
    add   a0, a0, a2
    add   a0, a0, a3
    add   a0, a0, a4
    add   a0, a0, a5
    add   a0, a0, a6
    add   a0, a0, a7
    add   a0, a0, t0
    add   a0, a0, t1
    add   a0, a0, t2
    add   a0, a0, t3
    ret
```

2.11 About the output of EWRISC-V

At the conclusion of this chapter, a summary of EWRISCV's output will be provided. The output destination folder is specified in [General Options]-[Output] of the project options.



2.11.1 Executables/libraries

In the folder where Executables/Libraries is specified, an ELF format executable format or LIBRARY is generated. It will also be output here when the HEX/SREC format is output with Output Converter.

2.11.2 Object files

The .o file that results from compiling the C/C++ file is output to this Object Files folder.

2.11.3 List files

MAP files and LIST files that can be generated at compile time are output to this List Files folder. If you generate assembler files during C/C++ compilation, they will also be generated in this folder.

2.11.4 Browse files

EWRISCV has a source code input support function. For example, it suggests the members of a structure or completes functions and variables while typing. You will see the following choices: The information is currently stored in this folder.

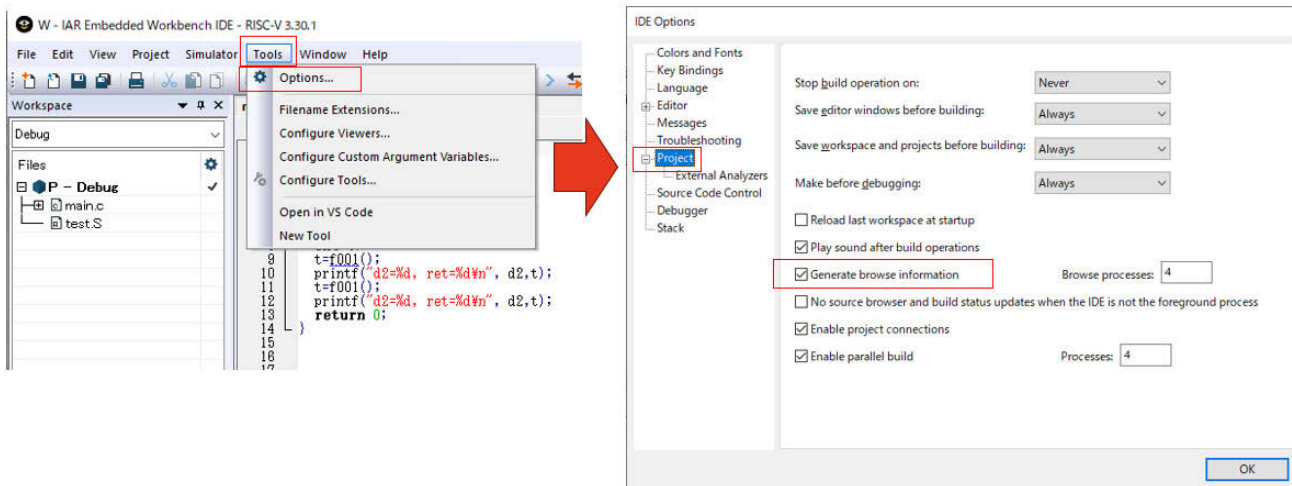
```

3 int main( void )
4 {
5     printf("Hello World\n");
6     printf
7 }
8
9
fo printf(const char *restrict, ...) int
fo snprintf(char *restrict, size_t, const char *restrict, ...) int
fo sprintf(char *restrict, const char *restrict, ...) int
fo vprintf(const char *restrict, __Va_list) int
fo vsnprintf(char *restrict, size_t, const char *restrict, __Va_list) int
fo vsprintf(char *restrict, const char *restrict, __Va_list) int
int
_Winnt
fo main() int

```

This source code analysis function can be turned on and off. If it takes a lot of time to analyze the source code, EWRISC-V may be stopped. At that time, it is possible to find the cause by

temporarily stopping the analysis function. You can open the IDE options with [Tools]-[Options] and control them with [Project]-[Generate Browse Information].



2.11.5 MAP files

Here is a brief description of the MAP file generated by EWRISC-V. Some of the output may increase or decrease depending on options and code, but the basic part will be explained. The basic parts are the following eight parts.

- LINKER Configuration OPTION
- RUNTIME MODEL ATTRIBUTES
- HEAP SELECTION
- PLACEMENT SUMMARY
- INIT TABLE
- STACK USAGE
- MODULE SUMMARY
- ENTRY LIST

The MAP generated in sample 2 will be explained.

MAP: Linker Configuration Options

At the top of the MAP file, you will find the linker (ILINK) options. Since EWRISCV is often used in an integrated development environment (IDE), it is necessary to open the screen to check the options, but it is also possible to do it with this MAP.

```
# IAR ELF Linker V3.20.1.3352/W64 for RISC-
V
      30/Aug/2023  09:10:27
# Copyright 2019-2023 IAR Systems AB.
#
#   Output file =
#       C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\Exe\P.o
ut
#   Map file    =
#       C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\List\P.
map
#   Command line =
#       -f
#       "C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\Exe\P.
out.rsp"
#       ("C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\Obj\m
ain.o"
#       --config_def CSTACK_SIZE=0x1000 --config_def HEAP_SIZE=0x1000
#       --no_out_extension -o
#       "C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\Exe\P.
out"
#       --define_symbol __iar_static_base$$GPREL=0 --map
#       "C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\List\P
.map"
#       --config
#       "C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\generic.icf"
#       --entry __iar_program_start --vfe --enable_stack_usage --
text_out
#       locale --debug_lib --core=RV32I)
#
#####
#####
```


MAP:RUNTIME MODEL ATTRIBUTES

In EWRISCV, we link the runtime libraries, but we record the options and versions at that time. In the following, DLib is used for __SystemLibrary, so the standard library of EWRISCV is specified, but users can also include their libraries.

```
*****
***  RUNTIME MODEL ATTRIBUTES
***
__SystemLibrary           = DLib
__dlib_file_descriptor    = 0
__dlib_version            = 6
__iar_optimize_for_size   = 1
__iar_riscv_atomic        = *
__iar_riscv_base_isa     = rv32i
__iar_riscv_compact       = *
__iar_riscv_div           = *
__iar_riscv_enum_size    = *
__iar_riscv_extension_atomic = *
__iar_riscv_extension_div = 0
__iar_riscv_extension_mul = 0
__iar_riscv_fpu           = none
__iar_riscv_i_version     = 2.0
__iar_riscv_mul           = *
__iar_riscv_xbcountzeroes = *
__iar_riscv_xlen         = 32
__iar_riscv_zba           = *
__iar_riscv_zbb          = *
__iar_riscv_zbc          = *
__iar_riscv_zbe          = *
__iar_riscv_zbf          = *
__iar_riscv_zbm          = *
__iar_riscv_zbp          = *
__iar_riscv_zbr          = *
__iar_riscv_zbs          = *
__iar_riscv_zbt          = *
__rt_version             = 1
```

MAP:HEAP SELECTION

EWRISCV allows you to choose the algorithm to be used on the heap. This is also output to MAP so that you can understand which library was used when linking.

```
*****
***  HEAP SELECTION
***
The basic heap was selected because --advanced_heap
was not specified, and the application did not appear to
be primarily optimized for speed.
```

MAP:PLACEMENT SUMMARY

THE PLACEMENT SUMMARY CONTAINS INFORMATION ABOUT THE PLACEMENT OF SECTIONS, BLOCKS, AND SO ON The PLACEMENT SUMMARY contains information about the placement of sections, blocks, and so on. The first half contains information about the linker configuration file, and the second half contains information about the placement of sections.

First of all, the information about the linker configuration file in the first half, such as block information, placement information, initialization information, etc. Block information is defined as a group that specifies the size or groups of sections. For example, a block called MVECTOR defines a block (with 128 alignments) that contains read-only in the section .mintvec. Blocks such as HEAP and CSTACK are defined with a size of 4 KB and an alignment of 16. Placement

is performed in a statement that includes place, and HEAP and CSTACK of read-write data and blocks are specified to be placed in the memory area 0x8000'0000~0x8003'ffff. These descriptions reflect the contents of the linker configuration file.

The important point is that “No sections matched the following patterns.” This is output when there is a description in the linker configuration file to place it, but it is not in the program. Here, it is stated that there is no read-only data in the section .mintvec placed in the block MVECTOR. Whether this is correct or incorrect can only be known by the person who created the program. This time, we didn't use the section .mintvec in the program. However, EWRISCV deletes variables and functions that are not directly referenced, so it is important to check whether variables and functions that are indirectly referenced have been deleted.

```
*****
*** PLACEMENT SUMMARY
***
build for rom;
"CSTARTUP32":
    place at start of [from 0x2000'0000 to 0x3fff'ffff] {
        ro section .cstartup };
define block MVECTOR with alignment = 128 { ro section .mintvec };
define block MINTERRUPTS
    with maximum size = 64K { ro section .mtext, block MVECTOR };
"ROM32":
    place in [from 0x2000'0000 to 0x3fff'ffff] { ro, block MINTERRUPTS
};
define block HEAP with size = 4K, alignment = 16 { };
define block CSTACK with size = 4K, alignment = 16 { };
"RAM32":
    place in [from 0x8000'0000 to 0x8003'ffff] {
        rw data, block HEAP, block CSTACK };
initialize by copy { rw };
keep symbol __iar_cstart_init_gp;
No sections matched the following patterns:
ro section .mintvec in block MVECTOR
```

The second half of the placement summary describes where sections and blocks were actually placed. There are five columns here: Section, Kind, Address, Size, and Object. The Section column contains the section or block's

name, the Kind column contains information about the region, including its Address and Size, and the Object column is the location (.o) where the file is defined.

Section	Kind	Address	Size	Object
-----	----	-----	----	-----
"CSTARTUP32":			0x48	
.cstartup	ro code	0x2000'0000	0x48	cstartup.o [5]
	-	0x2000'0048	0x48	
"ROM32":			0x334	
.text	ro code	0x2000'0048	0xe8	main.o [1]
.text	ro code	0x2000'0130	0xa0	__dbg_abort.o [2]
.text	ro code	0x2000'01d0	0x4	__dbg_break.o [2]
.text	ro code	0x2000'01d4	0x24	__dbg_xxexit.o [2]
Initializer bytes	const	0x2000'01f8	0x50	<for RAM32-1>
.text	ro code	0x2000'0248	0x8	low_level_init.o [4]
.text	ro code	0x2000'0250	0x3c	data_init.o [4]
.text	ro code	0x2000'028c	0x4	exit.o [4]
.text	ro code	0x2000'0290	0x20	cexit.o [4]
.text	ro code	0x2000'02b0	0x30	copy_init.o [4]
MINTERRUPTS		0x2000'0300	0x20	<Block>
.mtext	ro			
code	0x2000'0300	0x20	default_interrupt_handler.o [3]	
.iar.init_table	const	0x2000'0320	0x1c	- Linker created -
.text	ro code	0x2000'033c	0x20	main.o [1]
.text	ro code	0x2000'035c	0x20	zero_init.o [4]
	-	0x2000'037c	0x334	
"RAM32", part 1 of 3:			0x50	
RAM32-1		0x8000'0000	0x50	<Init block>
.data	inited	0x8000'0000	0x28	main.o [1]
.data	inited	0x8000'0028	0x28	main.o [1]
	-	0x8000'0050	0x50	
"RAM32", part 2 of 3:			0x50	
.bss	zero	0x8000'0050	0x28	main.o [1]
.bss	zero	0x8000'0078	0x28	main.o [1]
	-	0x8000'00a0	0x50	
"RAM32", part 3 of 3:			0x1000	
CSTACK		0x8000'00a0	0x1000	<Block>
CSTACK	uninit	0x8000'00a0	0x1000	<Block tail>
	-	0x8000'10a0	0x1000	
Unused ranges:				
From	To	Size		
----	--	----		
0x2000'037c	0x3fff'ffff	0x1fff'fc84		
0x8000'10a0	0x8003'ffff	0x3'ef60		

The following is an explanation of the most commonly used kinds.

- ro code: The code of the Readonly attribute
- const: Constants and other data
- initd: Non-zero initialization variables or regions

- zero: Initialization variable or region at zero
- uninit: Uninitialized variables or regions

MAP:INIT TABLE

INIT TABLE prints information about the initialization of variables. This area is output when the compiler initializes variables. Also, if there are no variables to initialize, it will not be output. In the C language, there are two types of variable initialization: zero initialization and non-zero initialization. In general, zero initialization is a .bss area, and non-zero initialization is a variable

initialization of .data. In the following description, the initialization function `__iar_zero_init2` to be used for the Zero part and the beginning address and size of the area to be initialized by zero are described, and the initialization function `__iar_copy_init2` to be used in the Copy part and the beginning address and size of the area to be initialized to non-zero are described. If you change the startup code or write your own, this initialization function must be called correctly.

```
*****
***  INIT TABLE
***
Address      Size
-----      ----
Zero (__iar_zero_init2)
  1 destination range, total size 0x50:
    0x8000'0050  0x50
Copy (__iar_copy_init2)
  1 source range, total size 0x50:
    0x2000'01f8  0x50
  1 destination range, total size 0x50:
    0x8000'0000  0x50
```

MAP:STACK USAGE

STACK USAGE is output by enabling the stack analysis function in the linker options. Since it is not specified in detail, EWRISCV analyzes the stack amount from the program entry. First, three types of interrupt, Program entry, and Uncalled

function are displayed. For interrupts, analysis is performed using the handler specified in the __interrupt as an interrupt. Program entry analyzes the stack usage from the program entry that starts execution from the time of reset. If there is an Uncalled function that is not called by any function, it will also be displayed.

```
*****
***  STACK USAGE
***

Call Graph Root Category  Max Use  Total Use
-----
interrupt                 112      112
Program entry              48       48
Uncalled function          0         0

Program entry
  "__iar_program_start": 0x2000'0000
Maximum call chain                48 bytes
  "__iar_program_start"              0
    "exit"                            0
    "_exit"                           16
    "__exit"                           32
    "__DebugBreak"                    0

interrupt
  "__iar_default_minterrupt_handler": 0x2000'0300
Maximum call chain                112 bytes
  "__iar_default_minterrupt_handler"  16
    "abort"                            64
    "__exit"                           32
    "__DebugBreak"                    0

Uncalled function
  "hoo": 0x2000'033c
Maximum call chain                0 bytes
  "hoo"                                0
```

STACK USAGE is output by enabling the stack analysis function in the linker options. Since it is not specified in detail, EWRISCV analyzes the stack amount from the program entry. First, three types of interrupt, Program entry, and Uncalled function are displayed. For interrupts, analysis is performed using the handler specified in the __interrupt as an interrupt. Program entry analyzes the stack usage from the program entry that starts execution from the time of reset. If there is an Uncalled function that is not called by any function, it will also be displayed. The following cases include cases where EWRISCV analysis cannot be performed correctly.

- If you use a function pointer or create a program with a fixed jump address
- Contains functions written in assembler
- When the same function is executed multiple times, e.g., in a recursive function

Also, if you use a real-time OS (RTOS), you will need to use stack usage for each task.

If you want to analyze correctly or want the necessary information, there are two ways to do it with EWRISCV. There are two methods: #pragma, which is to specify it in the source code or a separate file. Please refer to the manual for details, but for a simple example, a sample below is provided. If stack analysis is required separately, such as when task1 and task2 below are tasks on the RTOS, specify the #pragma call_graph_root before the function. If EWRISCV cannot understand the function call due to a function pointer, etc., specify the function to be called with #pragma calls.

```
#pragma call_graph_root
void task1(void ) {
    . . .
}
#pragma call_graph_root
void task2(void ) {
    . . .
}
extern int (*fp)(void);
int main( void )
{
    #pragma calls=foo,goo
    fp();
}
```

It is also possible to provide information for stack analysis in a separate file. A separate file has more content that can be specified. Of the following three, the top two are #pragma and have the same specifier content. The third is for recursive calls. If there is a recursive call, the compiler does not know how many times it will be

executed, but by specifying the upper limit, it is possible to perform stack analysis correctly.

```
call graph root: task1, task2;
possible calls fp: foo, goo;
max recursion depth recursive: 32;
```


MAP: MODULE SUMMARY

In addition to each file, the MODULE SUMMARY outputs the size of each library by dividing it into three categories: code size for each library, read-only data (data that does not change, such as constants), and read-write data (variables, etc.). Since the size of the module may fluctuate slightly with the final executable file due to optimization by the linker,

First, the size of the user code is displayed, and then it becomes a library. Libraries are meaningful in their names. In the case of `dbg-rv32i.a`, the instruction set is RV32I in the I/O library for debugging. `di-rv32i` provides a default interrupt handler, `dl-rv32i.a` includes `printf/scanf` and `dmath-rv32i.a` is a C math library. This may change the library used by the extension instructions.

```
*****
***  MODULE SUMMARY
***

Module                ro code  ro data  rw data
-----
command line/config:
-----
Total:
C:\Users\EWRISCV\Documents\WB\Output\Samples\S2P02\Debug\Obj: [1]
  main.o                264      80      160
-----
Total:                  264      80      160
dbg-rv32i.a: [2]
  __dbg_abort.o         160
  __dbg_break.o          4
  __dbg_xxexit.o        36
-----
Total:                  200
di-rv32i.a: [3]
  default_interrupt_handler.o  32
-----
Total:                  32
dl-rv32i.a: [4]
  dexit.o                32
  copy_init.o            48
  data_init.o            60
  exit.o                  4
  low_level_init.o       8
  zero_init.o            32
-----
Total:                  184
dmath-rv32i.a: [5]
  cstartup.o             72
-----
Total:                  72
Linker created                28  4'096
-----
Grand Total:                752  108  4'256
```

MAP:ENTRY LIST

The part that embedded engineers use the most is this Entry List. In the Entry List, you can check the placement status of functions and variables. An Entry is a function, variable, or label name, and to the right of it is the address, size, type, Gb/Wk, and object. In terms of type, Code is a function, and Data is a constant or variable. Gb is a globally defined variable/function, Wk is a function/variable with a weak attribute, and Lc is a locally defined variable/function.

As for objects, some created by the linker have a specified file name. If it looks like [2], you can see which file was linked from below. Even if you use a library, it is originally a C or assembler file, so it has a *.o name, and the entity is indicated in []. Finally, there is information about the size of the program, as well as the number of warnings and errors during the build.

```
*** ENTRY LIST
***
Entry                Address  Size  Type  Object
-----
.iar.init_table$$Base 0x2000'0320  --  Gb - Linker created -
.iar.init_table$$Limit 0x2000'033c  --  Gb - Linker created -
CSTACK$$Base         0x8000'00a0  --  Gb - Linker created -
CSTACK$$Limit        0x8000'10a0  --  Gb - Linker created -
MINTERRUPTS$$Base    0x2000'0300  --  Gb - Linker created -
MINTERRUPTS$$Limit   0x2000'031c  --  Gb - Linker created -
Region$$Table$$Base  0x2000'0320  --  Gb - Linker created -
Region$$Table$$Limit 0x2000'033c  --  Gb - Linker created -
__DebugBreak         0x2000'01d0  0x4  Code Gb __dbg_break.o [2]
__exit                0x2000'01d4  0x28 Code Gb __dbg_xxexit.o [2]
__iar_copy_init2     0x2000'02b0  0x30 Code Gb copy_init.o [4]
__iar_cstart_init_gp 0x2000'0004  Code Gb cstartup.o [5]
__iar_data_init2     0x2000'0250  0x3c Code Gb data_init.o [4]
__iar_default_minterrupt_handler
                    0x2000'0300  0x24 Code Gb default_interrupt_ha
```

```

ndler.o [3]
__iar_program_start      0x2000'0000      Code  Wk  cstartup.o [5]
__iar_static_base$$GPREL {Abs}
                        0x0      Data  Gb  <internal module>
__iar_zero_init2        0x2000'035c  0x20  Code  Gb  zero_init.o [4]
__low_level_init        0x2000'0248  0x8   Code  Gb  low_level_init.o [4]
__exit                   0x2000'0290      Code  Gb  cexit.o [4]
abort                    0x2000'0130  0xa8  Code  Gb  __dbg_abort.o [2]
da                        0x8000'0000  0x28  Data  Gb  main.o [1]
db                        0x8000'0028  0x28  Data  Gb  main.o [1]
exit                     0x2000'028c  0x8   Code  Gb  exit.o [4]
ha                        0x8000'0050  0x28  Data  Gb  main.o [1]
hb                        0x8000'0078  0x28  Data  Gb  main.o [1]
hoo                       0x2000'033c  0x20  Code  Gb  main.o [1]
main                      0x2000'0048  0xe8  Code  Gb  main.o [1]

[1] = C:\Users\Documents\Work\Output\riscv_book\Samples\S2P02\Debug\Obj
[2] = dbg-rv32i.a
[3] = di-rv32i.a
[4] = dl-rv32i.a
[5] = dlmath-rv32i.a
[6] = dln-rv32i.a

752 bytes of readonly code memory
 108 bytes of readonly data memory
4'256 bytes of readwrite data memory

Errors: none
Warnings: 1

```


A man with a beard and glasses is shown in profile, looking down at a computer system. He is holding a pen over a keyboard. The background is a server rack with various components. A white grid pattern is overlaid on the entire image. The text '3. Learn RISC-V on real hardware' is written in white at the bottom left.

3. Learn RISC-V on real hardware



3. Learn RISC-V on real hardware

3.1 Using the GigaDevice GD32VF103

We will use the Wio Lite RISC-V board equipped with Gigadevice's RISC-V, which is inexpensively available as a RISC-V evaluation board. As of March 2024, this product is sold at Digikey for 11 USD. It will be introduced as it serves as a suitable example, including how to connect the debugger.

This board is equipped with a WiFi module and ESP8266 from Espressif, and a Gigadevice GD32VF103, but only the GD32VF103 is used here.

GD32VF103CBT6 specifics: :

- RISC-V compliant little-endian RV32IMAC (32GPRs);
- Machine (M) and User (U) Privilege levels support;

- Single-cycle hardware multiplier and Multi-cycles hardware divider support;
- Misaligned load/store hardware support;
- Atomic instructions hardware support;
- Non-maskable interrupt (NMI) support;
- WFI (Wait for Interrupt) support;
- WFE (Wait for Event) support;

3.1.1 Debug probe connection

A hardware (debug) probe is required to download software to the GD32VF103 mounted on the board and debug it. Here, we will use the I-jet from IAR. It is connected and used as shown in the figure below. On the PC, the software is built using EWRISCV and then downloaded into the MCU board using I-jet.

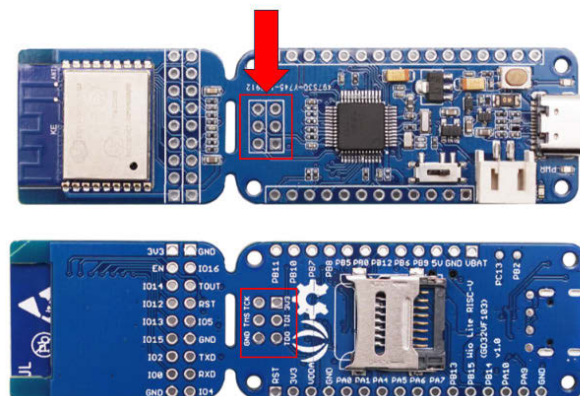


I-jet can be connected with a JTAG connection, but let's check the pins coming out of the board at this time. As shown in the figure below, there are pins related to JTAG on the board. Six ports (signals) are available in the area enclosed in the red box below.

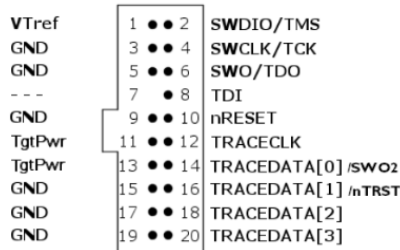
- TCK: JTAG clock

- TMS: Used to transition JTAG TAP controllers
- TDO—JTAG data output
- TDI: JTAG data entry
- 3V3: 3V output used as power supply
- GND: Ground

On the evaluation board, it is placed in the area surrounded by the red line below.

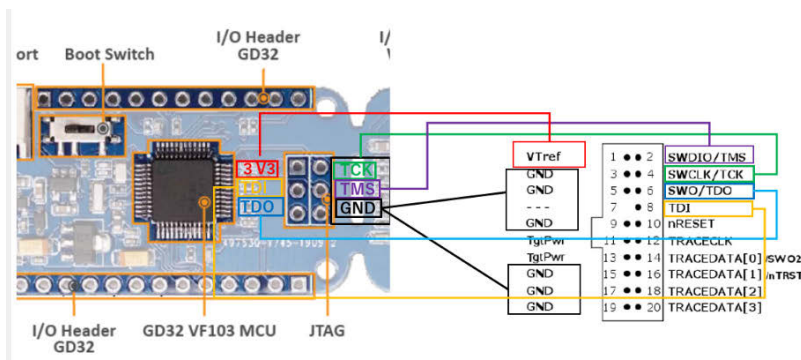


On the other hand, let's examine the I-jet side. The figure below shows the MIPI-20 connector, which has a 20-pin signal.



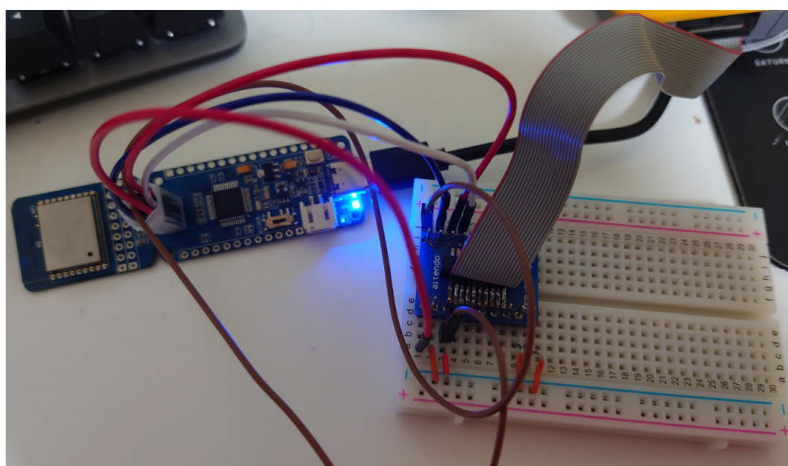
TCK, TMS, TDO, TDI, and GND can be connected to the corresponding pins on the board, and the

I-jet's VTref pin is connected to 3V3. This VTref is mandatory to set the signal level.



Since the small MIPI-20 connector is difficult to handle, we prepared a pitch conversion board and connected it to the GD32 JTAG pins using a

breadboard and jumper wires. Note that this type of connection is not recommended for high-frequency signals.



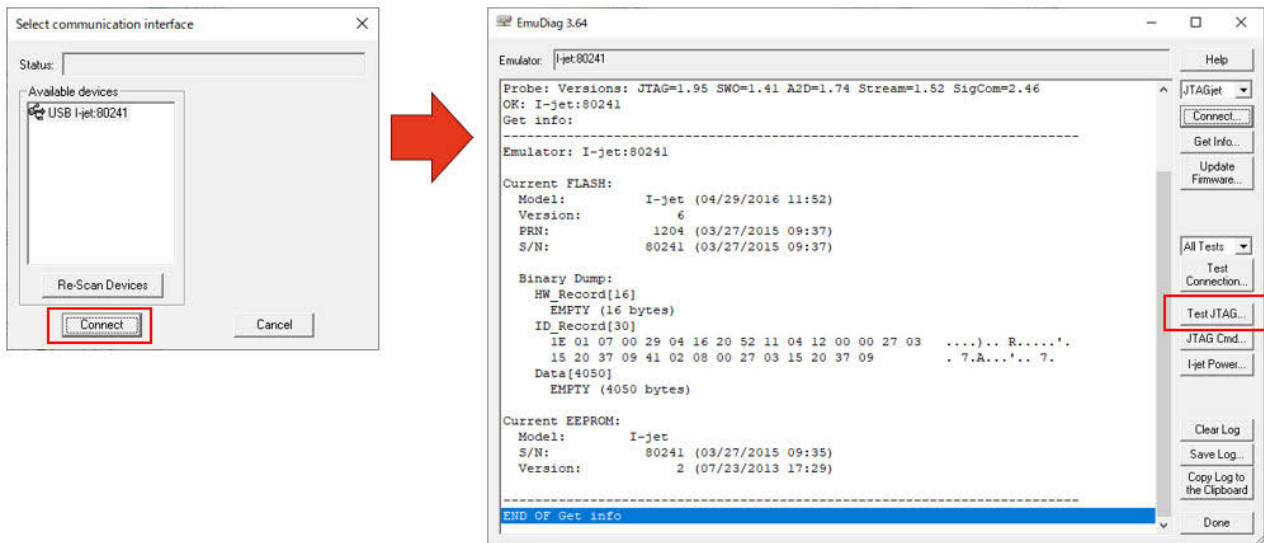
3.1.2 Checking the connection with IAR I-jet

Once the board, I-jet, and PC are connected, we should check whether the connection is correct. An evaluation board with a pre-mounted MIPI-20 connector always makes a reliable JTAG connection. Still, in this example, the connection may be bad because we used a breadboard and connected the pins manually.

When you install EWRISCV, you will find a program called Emudiag.exe in

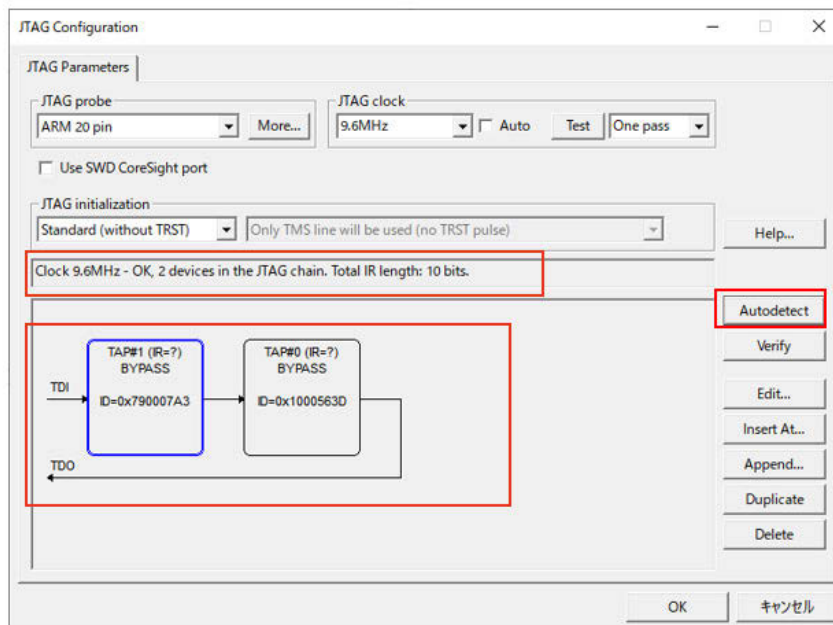
```
\riscv\bin\jet
```

under the folder where the tool was installed. When it starts, click [Connect] and click Test JTAG... Click.



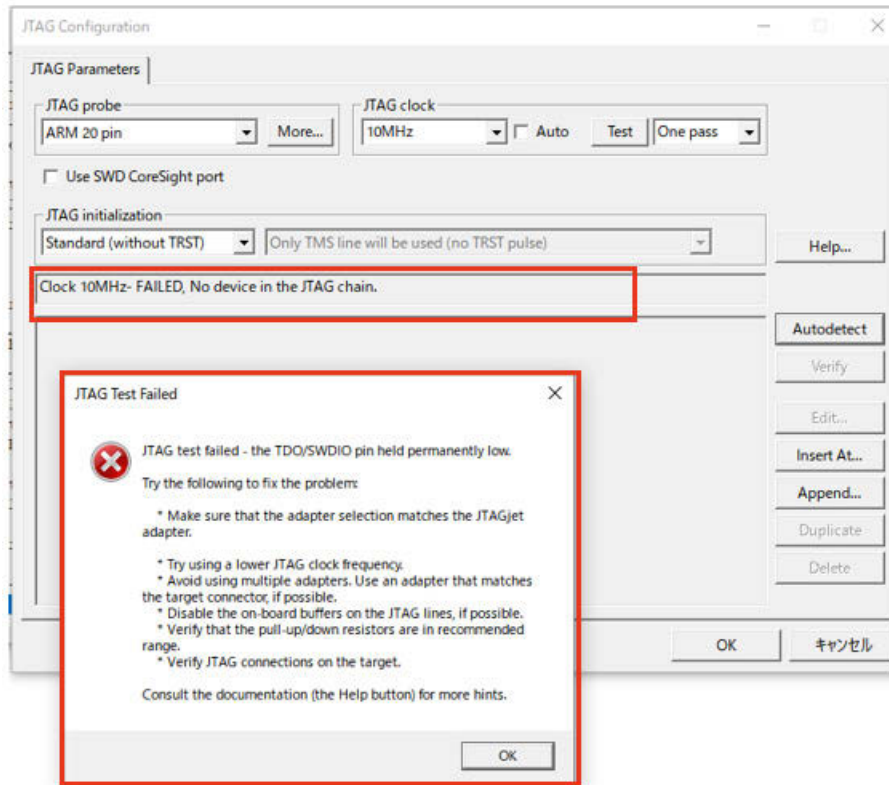
When the JTAG Configuration screen appears, click [Autodetect]. After clicking, you will see

the area circled in green. If the connection is successful, you will see these results.



An example of a connection not being established successfully will also be shown. This is an example of disconnecting the TMS from a correct connection. If JTAG is not able to connect

properly, you will get an error like this. In such a case, please stay calm and make sure that the connection is properly checked.



When the analysis is complete, exit the Emudiag application with [OK]-[Done]. If this is running, you may not be able to connect from EWRISCV.

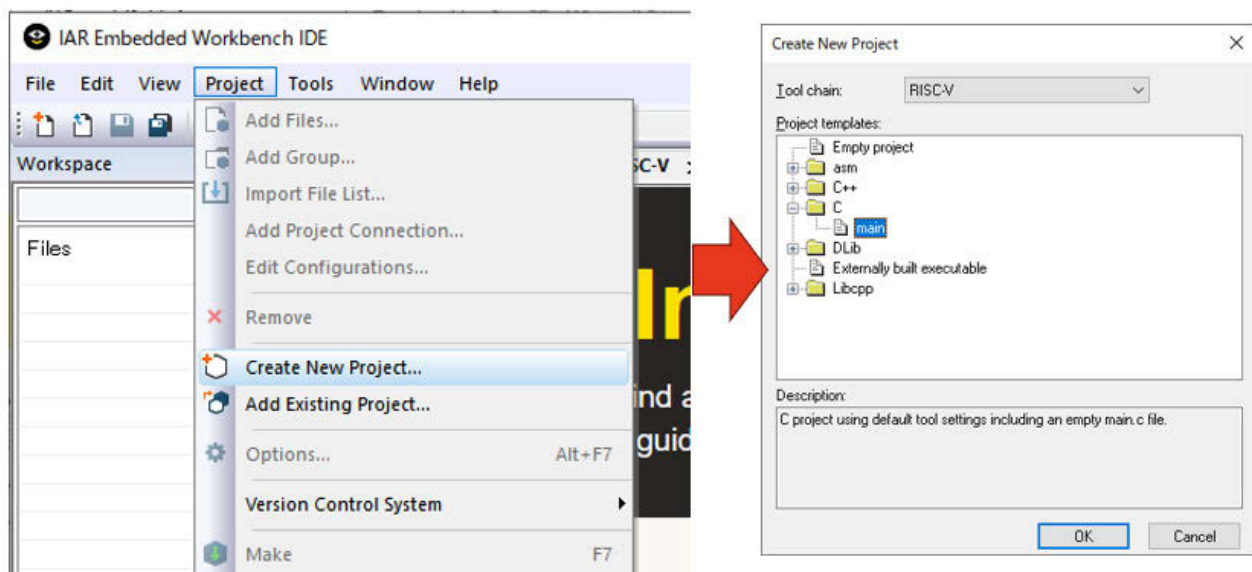
3.1.3 LED blinking: creating sample 5 using GPIO

Look at the schematic of the Wio Lite RISC-V board to check the LED connections. Since the LED is connected to the PA8, we will flash this LED.



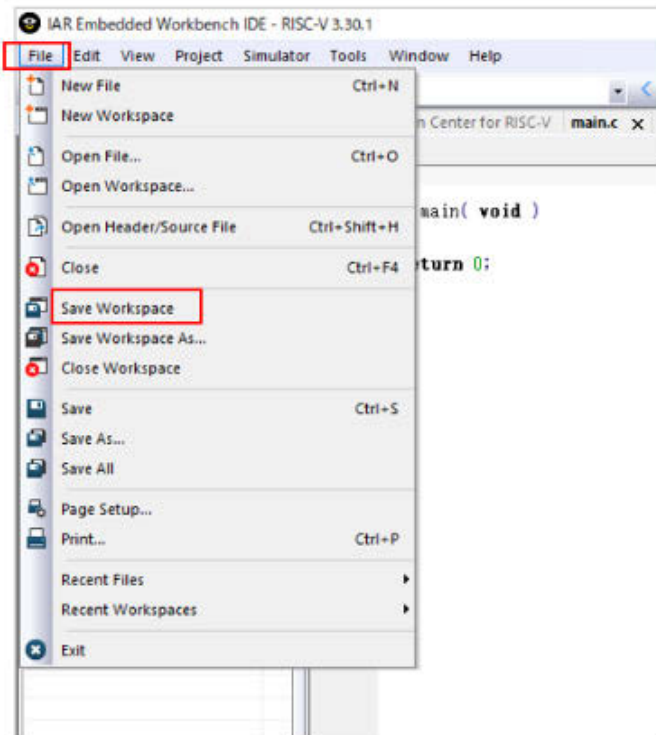
Now, let's create a project. From EWRISCV, select [C]-[main] in [Project]-[Create New Project].

Specify the project file name as appropriate. This time, we have named it S3P01.

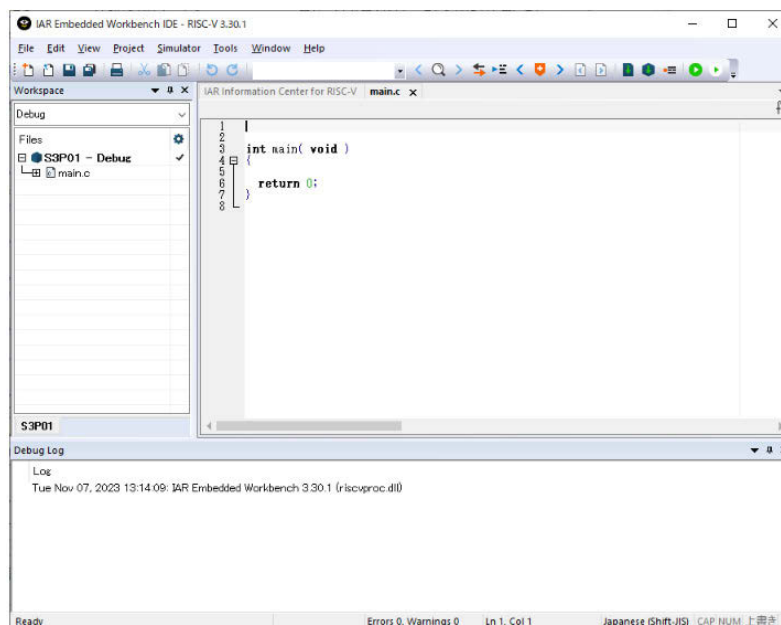


Select the folder you want to create a project and give it a project name. Next, execute [File]-[Save

Workspace] to name the workspace. Please specify the workspace file name here as well.

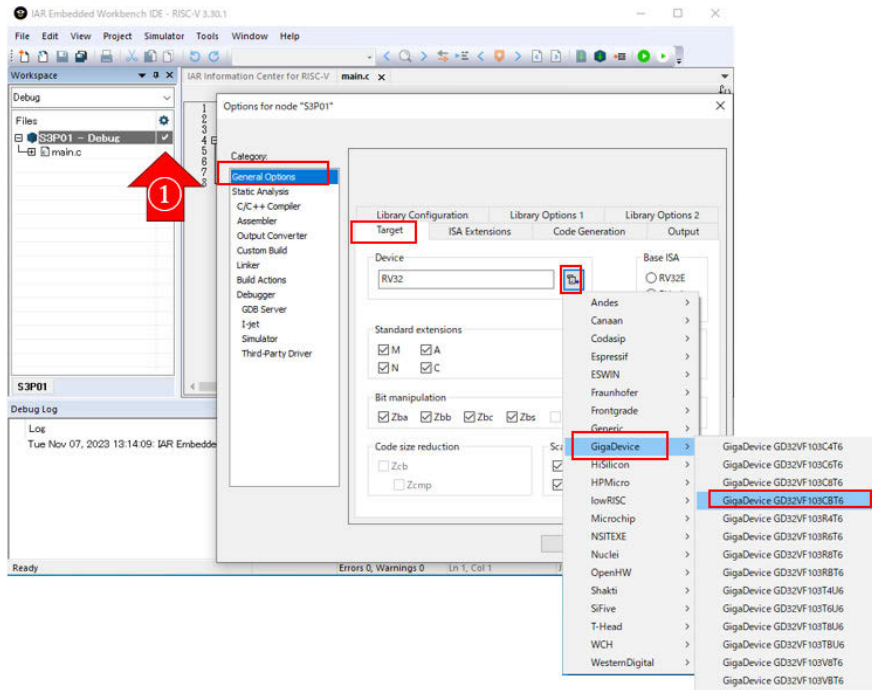


You now have a project and a workspace with one main.c.



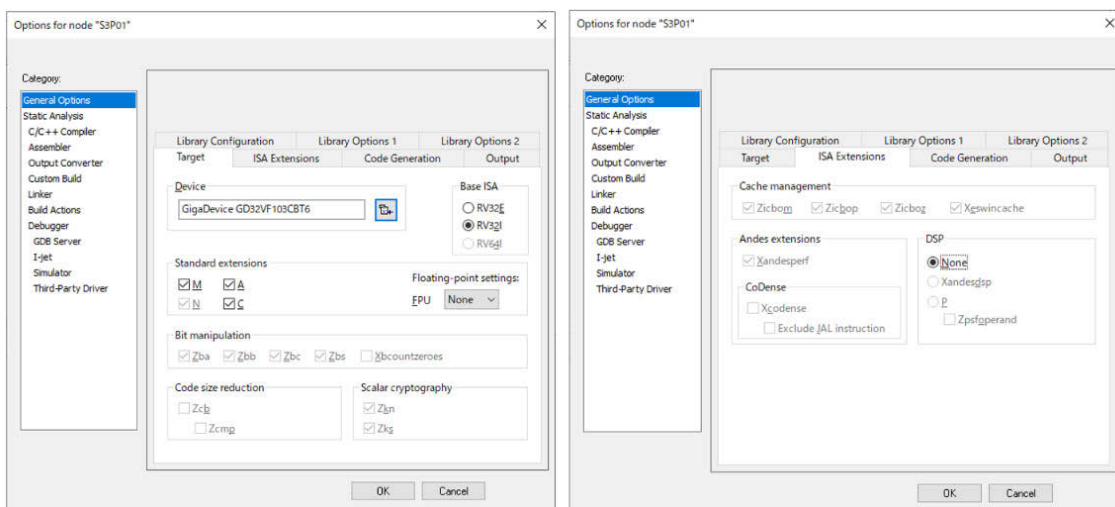
For EWRISCV, let's specify the microcontroller to be used this time. Double-click the check next to the project name in the workspace to open the option settings screen. In the figure

below, the project name is "S3P01." Double-click the red arrow 1. On the options screen, select the GigaDevice GD32VF103CBT6 in [General Options]—[Target]—[Device].



In chapter 2, detailed settings were required whether to use extended instructions, etc., but those supported by EWRISCV can be set by simply specifying the device. As for the settings related to instructions, it is okay to leave the default. However, if it is necessary to align the

options in order to use different microcontrollers, the user should change them. The following are the settings for the options related to the instructions when the current GD32VF103CBT6 is selected.



The advantage of the EWRISCV device support is that header files for peripheral access can also be used. #include <gigadevice/ioGD32VF103.h> allows peripheral access.

In order to blink the LED this time, it is necessary to enable the GPIOA clock and set the PA8 to output and PushPull settings.

```
#include <gigadevice/ioGD32VF103.h>
void delay(void ) {
    volatile int i;
    for (i=0; i< 500000;i++);
}
int main( void ){
    RCU_APB2EN_bit.PAEN=1; /* GPIO port A clock enable */
    GPIOA_CTL1_bit.MD8 = 0x1; /*PA8: Output 10MHz */
    GPIOA_CTL1_bit.CTL8 = 0x0; /*PA8: push-pull */

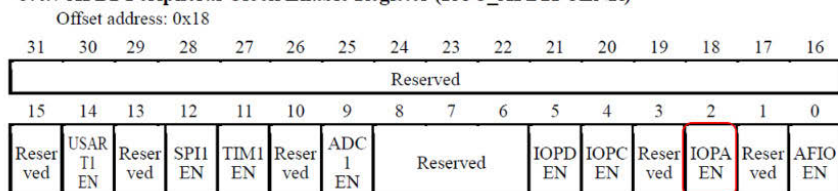
    while( 1 ) {
        GPIOA_OCTL_bit.OCTL8 = 0; /* Drive Low(0) */
        delay();
        GPIOA_OCTL_bit.OCTL8 = 1; /* Drive High(1) */
        delay();
    }
    return 0;
}
```

Let's take a look at the definition in the header file in the APB2 Peripheral Clock Enable Register (APB2EN). The left side of the figure below shows the part where APB2EN is defined. It is a union, and if you want to access it with 32 bits, use RCU_APB2EN. If you want to access it with a bitfield,

use RCU_APB2EN_bit. Use XXX, where XXX is the name of a separate field. In the previous example, only the PAEN bit is set, so RCU_APB2EN_bit. PAEN=1; it is described in. The same is true for the GPIOA's CTL1 and OCTL.

```
no_init volatile union
{
    unsigned int RCU_APB2EN;
    struct
    {
        unsigned int AFEN : 1;
        unsigned int PAEN : 1;
        unsigned int PBEN : 1;
        unsigned int PCEN : 1;
        unsigned int PDEN : 1;
        unsigned int PEEN : 1;
        unsigned int : 2;
        unsigned int ADC0EN : 1;
        unsigned int ADC1EN : 1;
        unsigned int TIMEROEN : 1;
        unsigned int SPI0EN : 1;
        unsigned int : 1;
        unsigned int USARTOEN : 1;
        unsigned int : 17;
    } RCU_APB2EN_bit;
} e 0x40021018;
```

3.4.7 APB2 Peripheral Clock Enable Register (RCC_APB2PCENR)



For devices that support EWRISCV, a linker configuration file is usually provided in addition to the header file. This time, the linker configuration file is also selected by selecting the device as

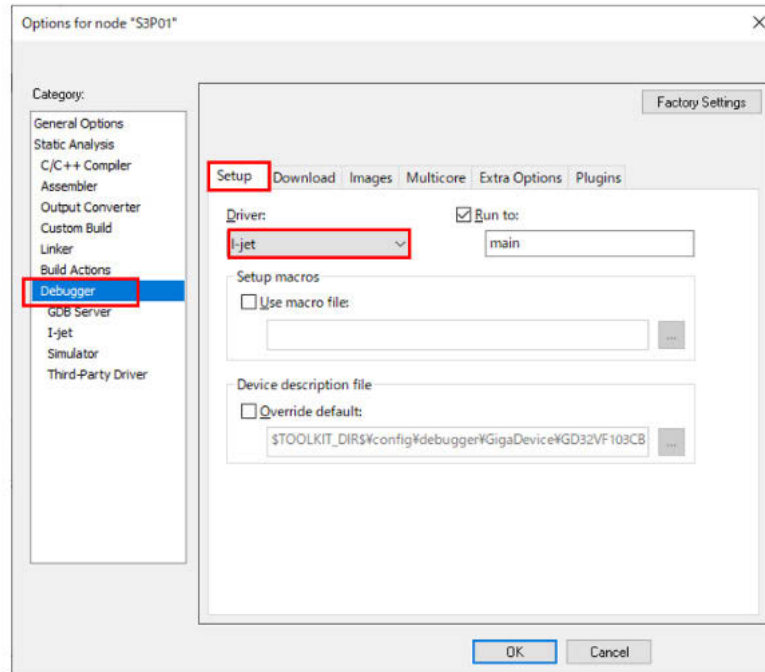
GD32VF103CBT6. You don't have to understand the contents of this file at first to create a program.

```
////////////////////////////////////
// RISC-V ilink configuration file
// for the GD32VF103CBT6
//
define exported symbol _link_file_version_2 = 1;
define exported symbol _auto_vector_setup = 1;
define exported symbol _max_vector = 96;
if (isdefinedsymbol(_disable_clic))
{
    define exported symbol _CLINT = 1;
}
else
{
    define exported symbol _uses_clic=1;
    define exported symbol _CLIC_GIGADEVICE = 1;
}
keep symbol __iar_cstart_init_gp; // defined in cstartup.s
keep { ro section .alias.hwreset };
define memory mem with size = 4G;
define region ROM_region32 = mem:[from 0x08000000 to 0x0801FFFF];
define region RAM_region32 = mem:[from 0x20000000 to 0x20007FFF];
initialize by copy { rw };
do not initialize { section *.noinit };
define block CSTACK with alignment = 16, size = CSTACK_SIZE { };
define block HEAP with alignment = 16, size = HEAP_SIZE { };
define block MVECTOR with alignment = 128, size = _max_vector*4 { ro
section .mintvec };
if (isdefinedsymbol(_uses_clic))
{
    define block MINTERRUPT with alignment = 128 { ro section .mtext };
    define block MINTERRUPTS { block MVECTOR,
                                block MINTERRUPT };
}
else
{
    define block MINTERRUPTS with maximum size = 64k { ro section .mtext,
                                                midway block
MVECTOR };
}
define block RW_DATA with static base GPREL { rw data };
"CSTARTUP32" : place at start of ROM_region32 { ro section
.alias.hwreset,
                                                ro section .cstartup };
"ROM32":place in ROM_region32 { ro,
                                block MINTERRUPTS };
"RAM32":place in RAM_region32 { block RW_DATA,
                                block HEAP,
                                block CSTACK };
```

3.1.4 Set up the debugger and start running

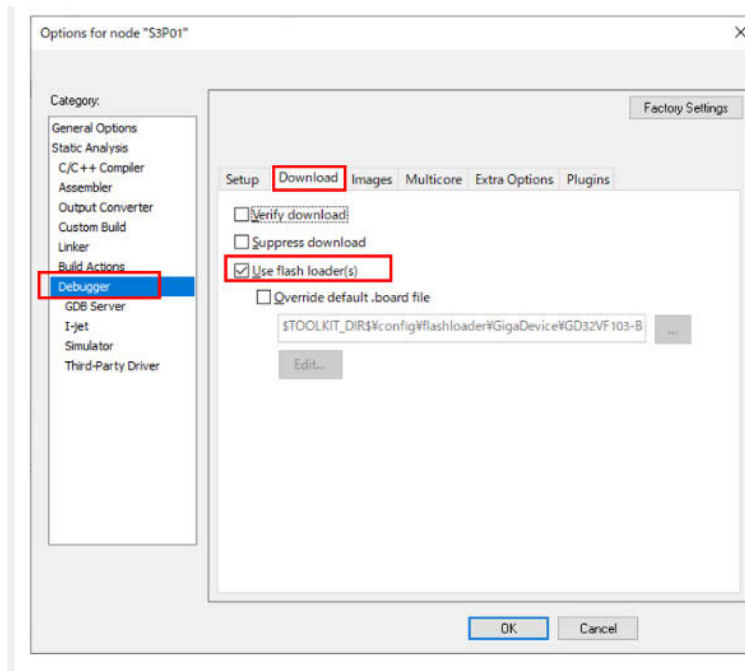
Now that the program is ready, it is intended to be downloaded and executed. In this case, I-jet

will be used for the hardware probe, so select [Debugger]-[Setup]-[Device] as [I-jet].



When the device is supported by EWRISCV, the flash loader can be used as an option in the [Debugger]-[Download] part. This may be the case with only the internal flash or with the external flash memory used on a

particular evaluation board. In the case of GD32VF103CBT6, an on-chip flash is supported. Please note that if you do not select Use Flash Loader, you will not be able to write to the flash memory area.



If you want to support external flash memory even with a compatible device, or if you're going to write a program to the flash memory of a device that does not yet support it, you need to do it yourself. In this case, there is a manual called [FlashLoaderGuide.ENU .pdf] in the folder where EWRISCV is installed:

```
\riscv\doc
```

So, you can implement it by referring to this. There are also examples of flash loader source code in the folder:

```
\riscv\src\flashloader
```

For example:

```
\riscv\src\flashloader\GigaDevice\GD32VF103
```

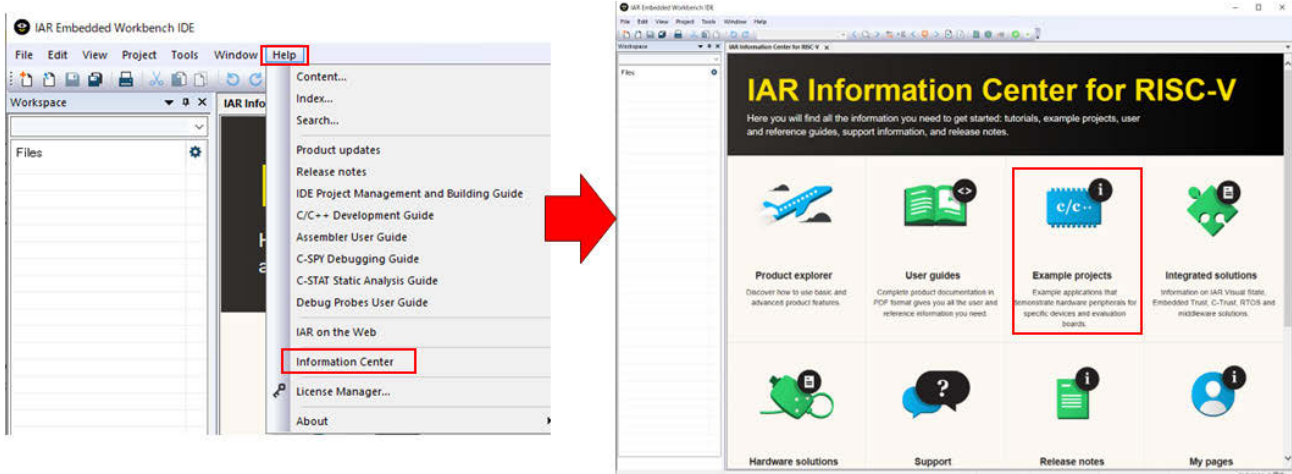
Finally, the LED blinks when you run the program.



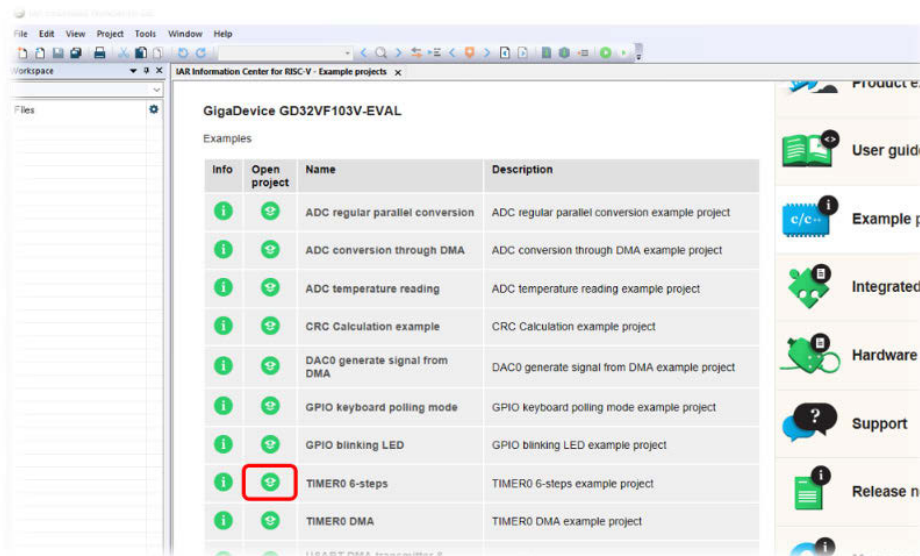
3.1.4 Learning about interrupts

It is quite difficult to create an interrupt program with a microcontroller for the first time. EWRISC-V has a sample of the GigaDevice GD32VF103V-EVAL board, so even though the board is different, an attempt will be made to use it in

this example. It is difficult to create an interrupt program for the microcontroller you are using from scratch, so the fastest way to learn is to use a sample. The sample is in the information center, so open the information center of EWRISC-V. Then, click Example Projects.

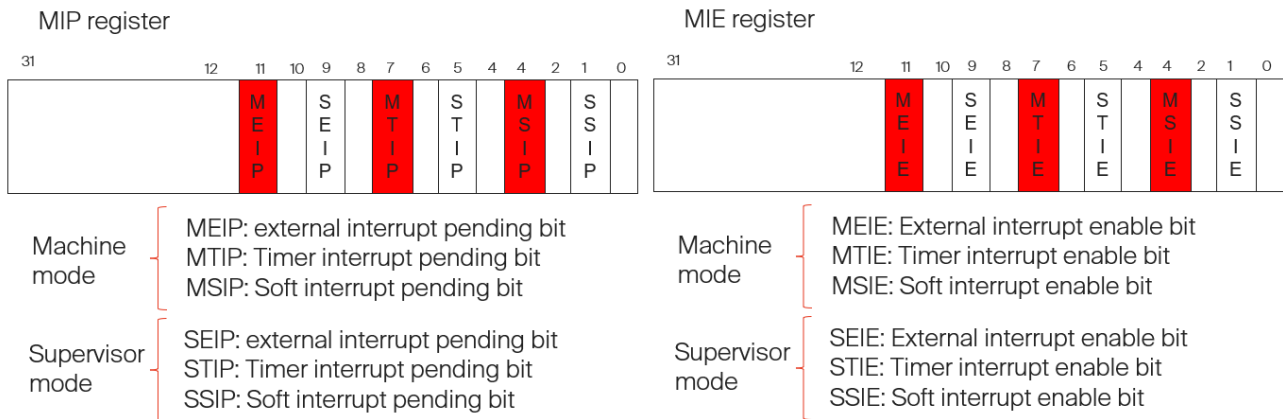


When you come to "Example Projects", open [GigaDevice GD32VF103V-EVAL]-[TIMER0 6-steps] (see figure below).



This will open this project in EWRISCV by specifying the save folder. There is not only one interrupt in the GD32VF103, but it can be switched and used in the settings. RISC-V defines CLIC (Core Local Interrupt Controller) and CLINT (Core Local INTerrupt), ACLINT, etc. CLIC and CLINT will be better

understood by looking at references [7]. CLINT is a simple interrupt controller that handles internal interrupts and handles external interrupts together with MEIP/MEIE. The PLIC (Platform-Level Interrupt Controller) is configured to handle external interrupts at that time. MIP and MIE of CSR manage interrupts in the figure below.



On the other hand, the CLIC is managed using management registers without using MIP/MIE. You can set the priority of the interrupt, the trigger type, and the vector response for each interrupt. In GD32VF103, ECLIC (Enhanced Core Local Interrupt Controller) is implemented based on CLIC. Not everything can be explained here, but the important parts will be covered. So, if you want to investigate in detail, please check the operation while looking at the sample.

- `__disable_interrupt()`—Disables CPU interrupts.
- `__set_bits_csr(reg, val)`: Sets the bit with the value of `val` for the CSR specified by `reg`.
- `__clear_bits_csr(reg, val)`—Clears the bits with the value of `val` for the CSR specified by `reg`.
- `__write_csr(reg, val)`: Sets the value `val` for the CSR specified by `reg`.
- `__read_csr (reg)`: Reads the CSR specified by `reg` and returns the contents.

First, let's examine the functions used in the sample code. The following functions are called intrinsics functions provided by EWRISCV and define functions that cannot be written in C but are often used.

In the sample, the basic settings for interrupts are implemented in the function `__low_level_init`. `MTVT`, `MTVT2`, and `MTVEC` are configured. As explained in the figure below, it is set to operate in CLIC mode and call `irq_entry` when an interrupt occurs and `trap_entry` when `TRAP/NMI` occurs.

```

int __low_level_init()
{
    disable_interrupt();
    /* Set the NMI base to share with mtvec by setting CSR_MMISC_CTL */
    /* li t0, 0x200 */
    /* csrcs CSR_MMISC_CTL, t0 */
    __set_bits_csr(/*CSR_MMISC_CTL*/ 0x7D0, 0x200);

    /* Initialize the mtvt */
    /* la t0, vector_base */
    /* csrcw CSR_MTVT, t0 */
    __write_csr_CSR_MTVT, ((unsigned int)&gd_vector_base));
    /* Initialize the mtvt2 and enable it */
    /* la t0, irq_entry */
    /* csrcw CSR_MTVT2, t0 */
    /* csrcs CSR_MTVT2, 0x1 */
    __write_csr(/*CSR_MTVT2*/ 0x7EC, 0x1 | ((unsigned int)&irq_entry));

    /* Initialize the CSR MTVEC for the Trap and NMI base addr*/
    /* la t0, trap_entry */
    /* csrcw CSR_MTVEC, t0 */
    /*
    __write_csr_CSR_MTVEC, 0x03 | ((unsigned int)&trap_entry));

    /* Enable mvcycle_minstret */
    __clear_bits_csr(/*CSR_MCOUNTINHIBIT*/ 0x320, 0x5);
    return 1;
}

```

Disable CPU interrupts using the intrinsic function provided by EWRISCV

Setting to generate an interrupt when NMI occurs

Setting the address used in vector mode (vector mode is not used in this sample)

Register the vector that handles external interrupts in MTVT2. Call irq_entry when an interrupt occurs. By setting 0x1, the mtvec2 address will be used during interrupts. Setting the lower 1 bit to 0x0 uses mtvec.

Register vectors to MTVEC to be executed when TRAP or NMI occurs. Call trap_entry when TRAP or NMI occurs. It is set to CLIC mode with 0x03. If the lower 2 bits are other than 0x3, it enters CLINT mode.

On the other hand, interrupt vectors are created in arrays, and zeros are set to parts where there are no handlers. Functions are specified in the

array, and the interrupt handler calls individual functions depending on the interrupt factor.

```

typedef void(*_fp)();
const _fp gd_vector_base[96] =
{
    0,
    0,
    0,
    eclic_msip_handler,
    0,
    0,
    0,
    eclic_mtsp_handler,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    eclic_bwei_handler,
    eclic_pmovi_handler,
    WWDGT_IRQHandler,
    LVD_IRQHandler,
    TAMPER_IRQHandler,
    RTC_IRQHandler,
    FMC_IRQHandler,
    RCU_IRQHandler,
    EXTIO_IRQHandler,
    EXTI1_IRQHandler,
    EXTI2_IRQHandler,
    EXTI3_IRQHandler,
    EXTI4_IRQHandler,
    DMA0_Channel0_IRQHandler,
    DMA0_Channel1_IRQHandler,
    DMA0_Channel2_IRQHandler,
    DMA0_Channel3_IRQHandler,
    DMA0_Channel4_IRQHandler,
    DMA0_Channel5_IRQHandler,
    DMA0_Channel6_IRQHandler,
    ADC0_1_IRQHandler,
    CAN0_TX_IRQHandler,
    CAN0_RX0_IRQHandler,
    CAN0_RX1_IRQHandler,
    CAN0_EWMC_IRQHandler,

```

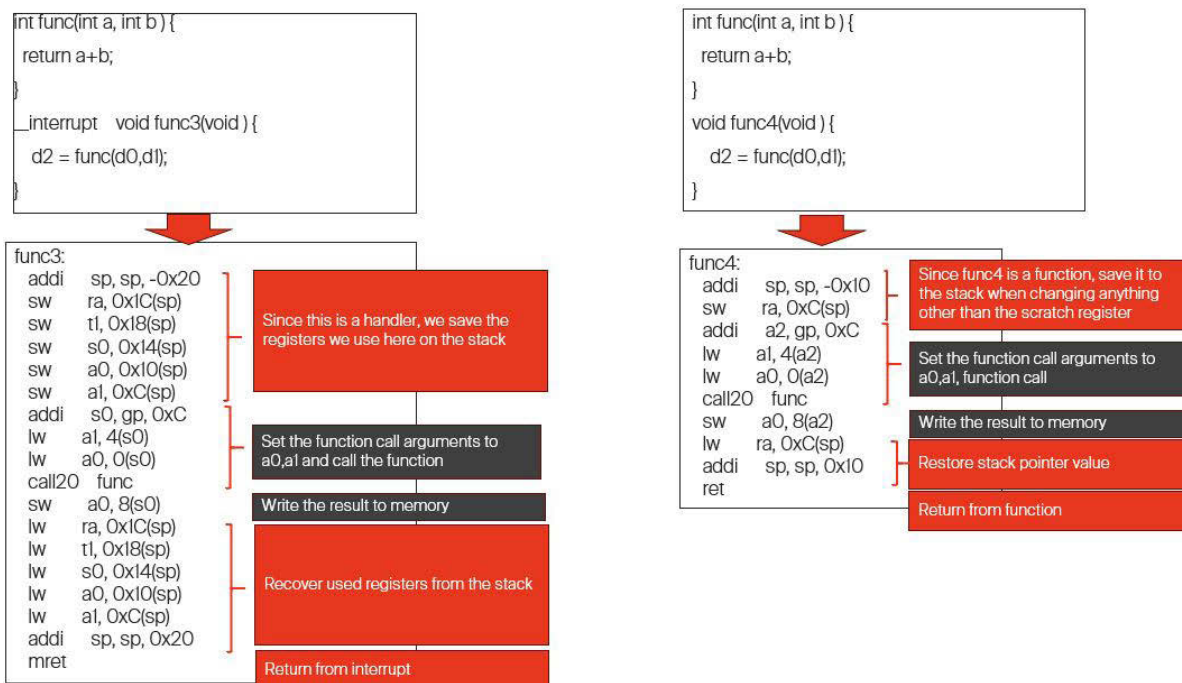

Now, let's move on to the interrupt handler. We will explain how to define a handler in EWRISCV. EWRISCV provides `__interrupt` as an extended keyword. You can define it as a handler by adding the following `__interrupt` before defining the function. The interrupt handler has no arguments and no return value, so the arguments and return value are always void.

```
__interrupt void my_interrupt_handler(void);
```

So why should you use this `__interrupt`? Describes the following: If this `__interrupt` does not exist, it will be a normal function. As explained in [2.10.2 Rules for Calling Functions], the use of registers when calling functions is as follows. The scratch register is a register that the function can freely rewrite. When called as a function, the caller of the function saves the value of the scratch register on the stack and calls the function. This allows the called function to change the value of the scratch register freely.

1. Scratch registers t0~t6, ft0~ft11, a0~a7, fa0~fa7
2. Storage registers S0~S11 and FS0~FS11,
3. Application-specific registers sp/x2, gp/x3, ra/x1

The registers used by the handler must be saved to the stack when an interrupt occurs, and the value of the register must be returned when the handler exits. Otherwise, when the interrupt handler returns, the value of the register will change, and the behavior will be strange. Of particular concern is the scratch register part. The figure below shows how the generated code changes depending on the presence or absence of `__interrupt`. On the left is the interrupt handler with the `__interrupt`, and on the right is the normal function. On the interrupt handler side, all registers used in the handler are saved, but only the stack pointer is saved in the function. You can see that calling the function `func` itself is the same as calling it from the function when calling it from the handler (green part). After the processing of the function `func` is completed and the result is stored in memory, it is a return process of the saved register.



The actual interrupt handler definition is shown in the figure below. In this program, external interrupts are `irq_entry`, and TRAP and NMI are `trap_entry` interrupt handlers. Therefore, you can see that `irq_entry` and `trap_entry` are `__interrupt`. In `irq_entry`, the CSR MCAUSE, MEPC, MSUBM, etc., are saved, and the handler is called. After

that, the saved register is restored, and the interrupt returns. This sample appears to be set up for handling multiple interrupts; however, it does not fully implement this capability, preventing the system from effectively managing concurrent interrupts.

```

uintptr_t handle_trap(uintptr_t mcause, uintptr_t sp)
{
    _fp fp;
    mcause &= 0xFFF;
    fp = gd_vector_base[mcause];
    if (fp)
        fp();
    return 0;
}

__interrupt void trap_entry()
{
    uintptr_t mcause = __read_csr(_CSR_MCAUSE);
    handle_trap(mcause, 0);
}

__interrupt void irq_entry()
{
    uintptr_t mcause = __read_csr(CSR_MCAUSE);
    uintptr_t mepc = __read_csr(CSR_MEPC);
    uintptr_t msubm = __read_csr(0x7C4);
    handle_trap(mcause, 0);
    //asm("csrrw ra,0x7ED, ra");
    __disable_interrupt();
    __write_csr(CSR_MCAUSE, mcause);
    __write_csr(CSR_MEPC, mepc);
    __write_csr(0x7C4, msubm);
}

```

Select and call the handler to process using the number stored in mcause

In EWRISCV, interrupt handlers are marked with `__interrupt`

Since the factor is included in CSR's MCAUSE, save it in a local variable

The interrupt return destination is saved in the MEPC of CSR, and its value is saved in a local variable

Save the current interrupt state and the previous interrupt state to MSUBM of CSR. Save this to a local variable.

Call `handle_trap` with the interrupt factor as an argument

After that, it is necessary to configure the CLIC, and in the GD32V103, the ECLIC setting. It's hard to explain everything here, so I'll explain the point. For details, please read the manual. ECLIC provides the following registers: Does clicintattr receive interrupts as vectors for each interrupt?

The interrupt type can be set to level, edge, etc. For each individual interrupt, a clicintie and a clicintip are brought in as permission and hold flags. For this reason, we do not use CSR MIE, MIP, etc.

Offset	Permission	Register	Bit	Explanation
0x0000	RW	cliccfg	8	Level/priority bit width specification
0x0004	R	clicinfo	32	CLIC implementation information
0x000b	RW	mth	8	Defining interrupt thresholds
0x1000+4*i	RW	clicintip[i]	8	i-th interrupt pending flag
0x1001+4*i	RW	clicintie[i]	8	i-th interrupt enable flag
0x1002+4*i	RW	clicintattr[i]	8	i-th interrupt type specification/vector setting
0x1003+4*i	RW	clicintctl[i]	8	Setting the i-th level/priority

The configuration of this CLIC in the sample is as follows: By continuing to read with reference to the sample, your understanding will deepen.

Hence, the expectation is for EWRISCV to be utilized.

```
int main(void)
{
    gpio_config();
    eclic_global_interrupt_enable();
    eclic_set_nbits(ECLIC_GROUP_LEVEL3_PRI01);
    eclic_irq_enable(TIMERO_TRG_CMT_IRQn, 6, 1);
    timer_config();

    while(1)
    {
        delay_1ms(10);
        timer_event_software_generate(TIMERO, TIMER_EVENT_SRC_CMTG);
    }
}
//END
```

Allow CPU interrupts

Set CLIC interrupt level and priority

Enable TIMERO_TRG_CMT_IRQ for CLIC, set level=6, priority=1

Generate a TIMER event for TIMERO from software

3.1.5 Let's check the CSR

In RISC-V, Control and Status Registers (CSRs) are important for checking the operating status and controlling interrupts. Here, we will check how to access CSRs and check some CSR registers in GD32VF103CBT6.

RISC-V does not have any arithmetic flags

Before we get into CSR, let's talk about arithmetic flags. Cortex-M, which is often used in embedded systems, has the following five operation flags, but RISC-V does not have these operation flags. Therefore, please understand that there is no arithmetic flag.

- N flag: A negative flag that is equal to 1 if the result of the operation is negative.
- Z-flag: A zero flag, which is 1 if the result of the operation is zero.
- C-flag: Carry/Borrow flag, which is 1 if the result of the operation is a carry or a borrow.
- V flag: An overflow flag that is equal to 1 in the event of an overflow.
- Q flag: The saturation flag, which is 1 if saturation occurs in the saturation operation.

Machine-Level CSR

In reference [5], the Machine-Level CSR is described in the Machine-Level ISA (Machine Level Instruction Set) chapter.

Number	Name	Memo
Machine Information Registers		
0xF11	mvendorid	Vendor ID
0xF12	marchid	Architecture ID
0xF13	mimpid	Implementation ID
0xF14	mhartid	Hardware thread ID
0xF15	mconfigptr	Pointer to configuration data structure
Machine Trap Setup		
0x300	mstatus	Machine status register
0x301	misa	ISA and extensions
0x302	medeleg	Machine exception delegation register
0x303	mideleg	Machine interrupt delegation register
0x304	mie	Machine interrupt-enable register
0x305	mtvec	Machine trap-handler base address
0x306	mcounteren	Machine counter enable
0x310	mstathsh	Additional machine status register(RV32 only)
Machine Trap Handling		
0x340	mscratch	Scratch register for machine trap handlers
0x341	mepc	Machine exception program counter
0x342	mcause	Machine trap cause
0x343	mtval	Machine bad address or instruction
0x344	mip	Machine interrupt pending
0x34A	mtinst	Machine trap instruction (transformed)
0x34B	mtval2	Machine bad guest physical address

The rest of the list is not explained in this book, but it is defined as a standard.

Number	Name	Memo
Machine Configuration		
0x30A	menvcfg	Machine environment configuration register
0x31A	menvcfgh	Additional machine env. conf. register(RV32 only)
0x747	mseccfg	Machine security configuration register
0x757	mseccfgh	Additional machine security conf. register(RV32 only)
Machine Memory Protection		
0x3A0	pmpcfg0	Physical memory protection configuration
0x3A1	pmpcfg1	Physical memory protection configuration(RV32 only)
0x3A2	pmpcfg2	Physical memory protection configuration
0x3A3	pmpcfg3	Physical memory protection configuration(RV32 only)
...
0x3AE	pmpcfg14	Physical memory protection configuration
0x3AF	pmpcfg15	Physical memory protection configuration(RV32 only)
0x3B0	pmpaddr0	Physical memory protection address register
0x3B1	pmpaddr1	Physical memory protection address register
...
0x3EF	pmpaddr63	Physical memory protection address register
Machine Counter/Timers		
0xB00	mcycle	Machine cycle counter
0xB02	minstret	Machine instructions-retired counter
0xB03	mhpmcounter3	Machine performance-monitoring counter
0xB04	mhpmcounter4	Machine performance-monitoring counter
...
0xB1F	mhpmcounter31	Machine performance-monitoring counter
0xB80	mcycleh	Upper 32 bits of mcycle(RV32 only)
0xB82	minstreth	Upper 32 bits of minstret(RV32 only)
0xB83	mhpmcounter3h	Upper 32 bits of mhpmcounter3(RV32 only)
0xB84	mhpmcounter4h	Upper 32 bits of mhpmcounter4(RV32 only)
...
0xB9F	mhpmcounter31h	Upper 32 bits of mhpmcounter31(RV32 only)
Machine Counter Setup		
0x320	mcountinhibit	Machine counter-inhibit register
0x323	mhpmevent3	Machine performance-monitoring event selector
0x324	mhpmevent4	Machine performance-monitoring event selector
...
0x33F	mhpmevent31	Machine performance-monitoring event selector

Debug/Trace Registers (shared with Debug Mode)		
0x7A0	tselect	Debug/Trace trigger register select
0x7A1	tdata1	First, Debug/Trace trigger data register
0x7A2	tdata2	Second Debug/Trace trigger data register
0x7A3	tdata3	Third, Debug/Trace trigger data register
0x7A8	mcontext	Machine-mode context register
Debug Mode Registers		
0x7B0	dcsr	Debug control and status register
0x7B1	dpc	Debug PC
0x7B2	dscratch0	Debug scratch register 0
0x7B3	dscratch1	Debug scratch register 1

Programmatic access to CSRs

Specific instructions also exist for accessing the CSRs. They are defined as Zicsr in reference [4]. The basic instruction is the CSRRW. The contents of the specified CSR are output to register rd, and the contents of the CSR specified in register rs1 are rewritten.

- CSRRW rd, csr, rs1; Read/Write
- CSRRS rd, csr, rs1; Read and Set bit
- CSRRC rd, csr, rs1; Read and Clear bit
- CSRRWI rd, csr, imm; Read/Write Immediate Value
- CSRRSI rd, csr, imm; Read and Set bit Immediate Value
- CSRRCI rd, csr, imm; Read and Clear bit Immediate Value

However, many people are not comfortable programming in assembler, so EWRISCV provides the following intrinsic functions to access CSR. Each function specifies a CSR and performs the operation. All functions return the value before the CSR operation in the return value.

- `__clear_bits_csr (csr, value)`: Clears the specified CSR with value.
- `__set_bits_csr (csr, value)`: Set the specified CSR with value.
- `__read_csr (CSR)`: Reads the specified CSR
- `__write_csr (csr, value)`: Reads the specified CSR and rewrites the value.

Now, let's create and execute a program that accesses CSR. I'm using the intrinsics function to read the CSR and then using printf to write it to standard output. For the part where the CSR is specified, the #define from csr.h was used. CSRs that are not defined there can always be specified directly by their CSR number. This is the part that got a runtime error when executing the application. The reason might be that the specification has been revised, or it may be up to the device vendor whether the CSR is implemented or not, so this may occur. In the code example below, the problematic parts have been commented out.

```
#include <intrinsics.h>
void print_csr_status(void ) {
    /* Machine Information Registers */
    printf("MVENDORID    0x%x\n", __read_csr( _CSR_MVENDORID ) );
    printf("MARCHID      0x%x\n", __read_csr( _CSR_MARCHID ) );
    printf("MIMPID       0x%x\n", __read_csr( _CSR_MIMPID ) );
    printf("MHARTID     0x%x\n", __read_csr( _CSR_MHARTID ) );
    // printf("mconfigptr  0x%x\n", __read_csr( 0xF15 ) );
}
```

```

/* Machine Trap Setup */
printf("MSTATUS      0x%x\n", __read_csr( _CSR_MSTATUS ) );
printf("MISA         0x%x\n", __read_csr( _CSR_MISA ) );
printf("MEDELEG       0x%x\n", __read_csr( _CSR_MEDELEG ) );
printf("MIDELEG       0x%x\n", __read_csr( _CSR_MIDELEG ) );
printf("MIE           0x%x\n", __read_csr( _CSR_MIE ) );
printf("MTVEC         0x%x\n", __read_csr( _CSR_MTVEC ) );
printf("MCOUNTEREN    0x%x\n", __read_csr( _CSR_MCOUNTEREN ) );
// printf("MSTATUSH    0x%x\n", __read_csr( 0x310 ) );

/* Machine Trap Handling */
printf("MSCRATCH      0x%x\n", __read_csr( _CSR_MSCRATCH ) );
printf("MEPC          0x%x\n", __read_csr( _CSR_MEPC ) );
printf("MCAUSE        0x%x\n", __read_csr( _CSR_MCAUSE ) );
printf("MTVAL         0x%x\n", __read_csr( _CSR_MTVAL ) );
// printf("mtinst     0x%x\n", __read_csr( 0x34A ) );
// printf("mtval2     0x%x\n", __read_csr( 0x34B ) );
}

```

The result of running the above code is shown below.

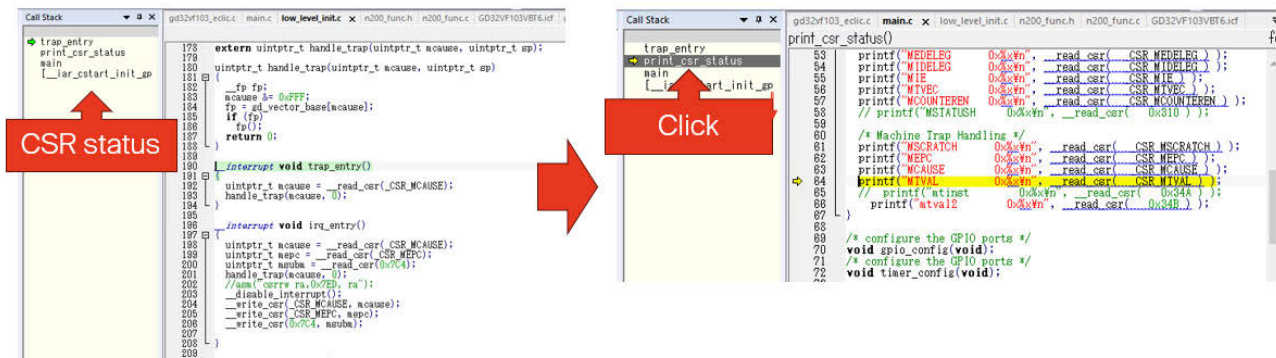
```

VENDORID      0x31e
ARCHID        0x80000022
IMPID         0x100
HARTID        0x0
MSTATUS       0x0
MISA          0x40901105
MEDELEG       0x0
MIDELEG       0x0
MIE           0x0
MTVEC         0x8000b83
MCOUNTEREN    0x0
MSCRATCH      0x0
MEPC          0x80011d4
MCAUSE        0x0
MTVAL         0x34b02573

```

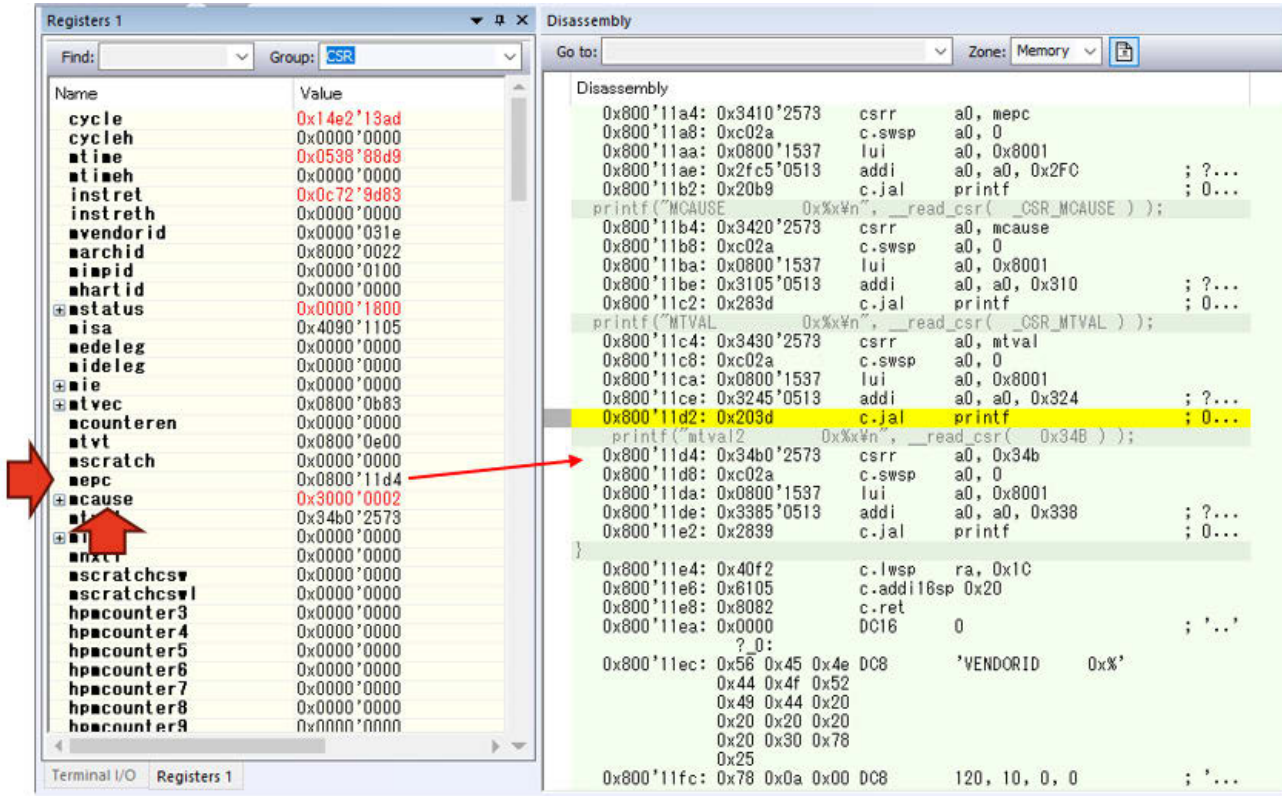
Since the opportunity presents itself, the situation when the error appears will be checked. Check for errors when accessing mtval2 of the CSR. An error occurred, and an exception occurred, and the transition to trap_entry was made. Still, unfortunately, since an interrupt vector was only proposed for the normal system, it resulted in an infinite loop at the trap_entry. Where did it occur at that time in EWRISCV? It can be displayed by the Call Stack when debugging. The left side

of the figure below is the state when the break/halt is applied. The call stack shows how we got there. While the main function is executing, the print_csr_status is called, and the trap_entry is occurring there. In the call stack, you can click on a displayed function to indicate where it occurred (approximate location). When you run the print_csr_status this time, it will look like the one on the right.



MEPC and MCAUSE

From there, you can find out more about the CSR value. In EWRISCV, you can check the CSR value on the register screen.



MEPC stands for Machine exception program counter, so it records the PC at the time of interruption. It can be confirmed that it occurred when CSR a0, 0x34B was issued. Check

MCAUSE (Machine trap cause) to see what the cause is. The GD32V103 has been extended from the standard specification to include the following bit settings:

Field	bit	Memo
INTERRUPT	31	0: Exception or NMI, 1: Interrupt
MINHV	30	Indicate processor is reading interrupt vector table
MPP	29~28	privilege mode before interrupt
MPIE	27	interrupt enable before interrupt
MPIL	23~16	Previous interrupt level
EXCCODE	11~0	Exception/Interrupt Encoding

Since the value of mcause is 0x30000002, INTERRUPT=0 causes EXCEPTION or TRAP, MPP=3 (machine mode before interrupt), and EXCCODE causes the exception to 2. Upon investigating the nature of exception 2, which

was not listed in the microcontroller specification, the RISC-V specification (reference [5]) was consulted, revealing the following: Illegal instructions are being issued for accessing CSR that should not be accessed. mstatus

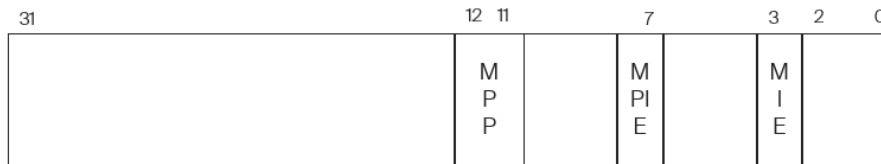
INTEERUPT	Code	memo
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Reserved
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved
0	15	Store/AMO page fault
0	16~23	Reserved
0	24~31	Designated for custom use
0	32~47	Reserved
0	48~63	Designated for custom use
0	≥64	Reserved

- **mstatus**

indicates the state of execution in machine mode. The GD32V is as follows. Set the permission of the CPU itself to interrupt in MIE (1: allowed, 0: forbidden), and MPIE indicates the

state of MIE before the interrupt is entered. The MPP indicates the privileged status before the interrupt is given in 2 bits. Displays 00 for user mode and 11 for machine mode.

mstatus: Machine Status Registers



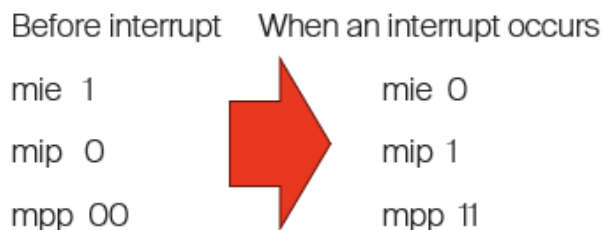
MIE: Interrupt enable bit

MPIE: Save previous value of MIE bit

MPP: Previous Privilege State

In fact, before and after interruption, it looks like this: When running in machine mode, it is possible to accept interrupts when MIE is 1. When an interrupt is entered, the MPP is entered in machine mode (11), and the MPIE is entered with the value of the MIE before the interrupt (1). In RISC-V, when an interrupt is entered, the MIE is

0 because the next interrupt cannot be received. Note: In the case of Cortex-M, it is possible to receive the next interrupt immediately after the interrupt. If you have been using Cortex-M, please pay attention to this point. When the interrupt returns, it returns to the left side.



Here, let's take a look at CSRs that are independent of execution.

- **mvendorid**

mvendorid is the JEDEC manufacturer ID. Upon examining the read value, it was found to be 0x0000031E. After some research was conducted, reference [8] was found to indicate that this value is the code for Andes

Technology Corporation. This point was not initially understood. Upon further research into GD32V103, a page containing references [9] was discovered. On the Nuclei website, GigaDevice's GD32V is presented as an example of a customer. Furthermore, the N22 RISC-V processor from Andes is introduced. Thus, it appears that the processors from GigaDevice, Nuclei, and Andes in China are related.

- **marchid**

marchid indicates the ID of the microarchitecture to be executed. If the most significant bit (MSB) of marchid is zero, the vendor sets the ID for an open-source project, and if it is 1, it is a commercial project. The ID of the open-source project is given in reference [10].

In the results of the GD32V103 run, Marchid was 0x80000022. The MSB is 1, and the Architecture ID is 0x22. In the case of the Andes A45 core mentioned earlier, it is 0x8a45, so it is conceivable that this pertains to the Andes N22 core.

Field	bit	memo
MSB	31	0 : OSS, 1 : Business Project
Architecture ID	30~0	Architecture ID

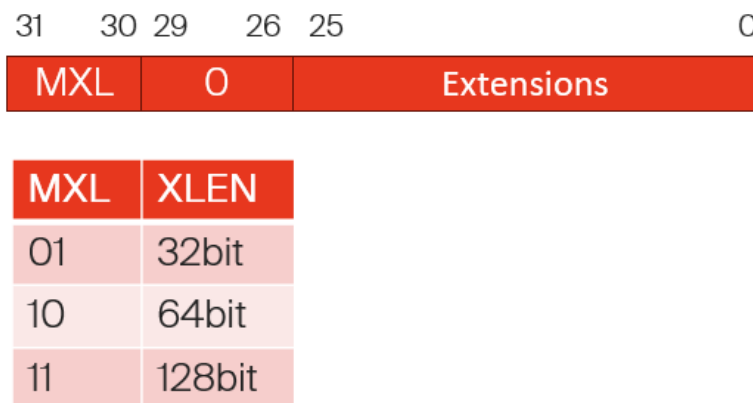
- **mimpid and mhartid**

mimpid seems to return the ID at the time of implementation. The results of the GD32V103 execution were 0x00000100. Andes' A45 naming method is Major=1, Minor=0, Extension=0. mhartid indicates the ID of the hardware thread on which the code is executed. Some recent microcontrollers are capable of multi-threaded execution that executes multiple codes at the same time in hardware. Still, in such a case, it is possible to check which hardware thread

is currently running. It is 0x0 in the result of executing the GD32V103. This is by design, and it is stated that a hardware thread must return zero. If there is only one thread of execution, that thread will return a 0x0 as in this case.

- **misa**

misa provides information about the instruction set. Basically, it is specified by bit length and instruction set (Extensions), as shown in the figure below.

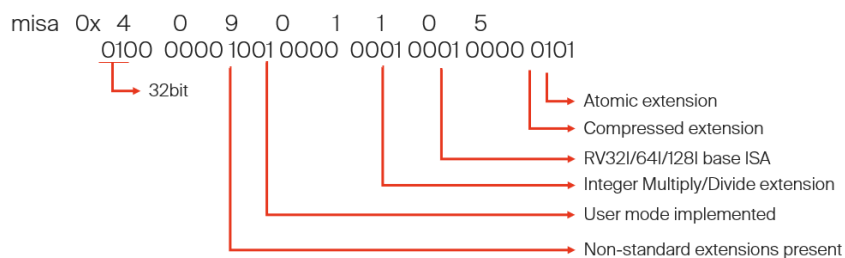


The extensions part is defined as follows.

Bit	Name	Description
0	A	Atomic extension
1	B	Tentatively reserved for Bit-Manipulation extension
2	C	Compressed extension
3	D	Double-precision floating-point extension
4	E	RV32E base ISA
5	F	Single-precision floating-point extension
6	G	Reserved
7	H	Hypervisor extension
8	I	RV32I/64I/128I base ISA
9	J	Tentatively reserved for Dynamically Translated Languages extension
10	K	Reserved
11	L	Reserved
12	M	Integer Multiply/Divide extension
13	N	Tentatively reserved for User-Level Interrupts extension
14	O	Reserved
15	P	Tentatively reserved for Packed-SIMD extension
16	Q	Quad-precision floating-point extension
17	R	Reserved
18	S	Supervisor mode implemented
19	T	Reserved
20	U	User mode implemented
21	V	Tentatively reserved for Vector extension
22	W	Reserved
23	X	Non-standard extensions present
24	Y	Reserved
25	Z	Reserved

In the case of GD32V103, it was 0x40901105, so it will be an extended instruction described below. As for how to write RISC-V, it is in the form of corresponding to RV32IMAC. It is written in

the manual which extended instructions are supported, but it may be useful to know that these contents are included in the CSR.



3.2 Using the Renesas FBP-R9A02G021 board

Here, we will use Renesas' FBP-R9A02G021 board, which has Renesas' first general-purpose 32-bit RISC-V MCU mounted on it.



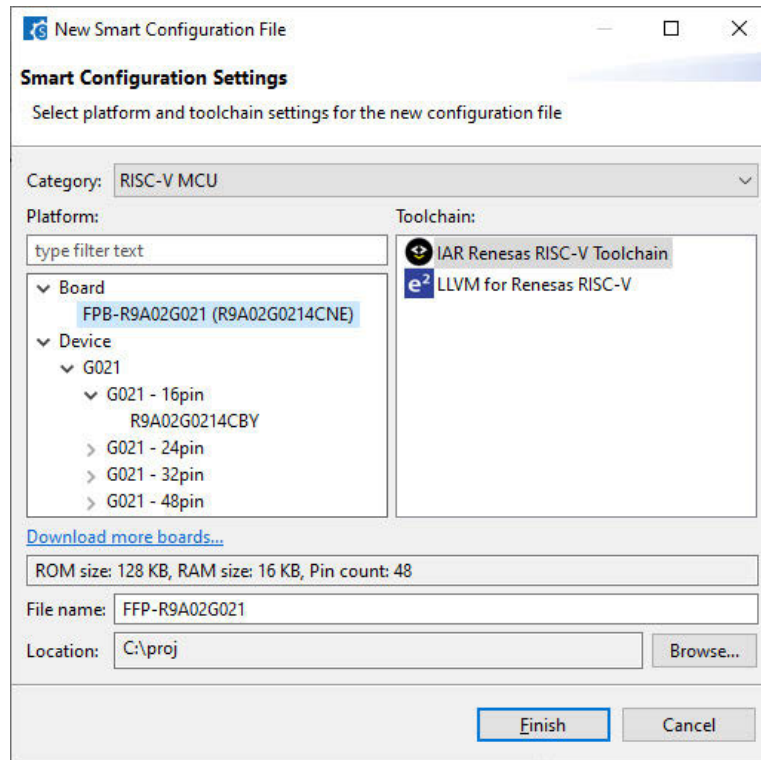
The FBP-R9A02G021 has an on-board debugger and a connector for connecting an external debugger probe. When using EWRISCV, the external debugger connection with I-jet is more

functional, so let's use the external debugger first. At the end of this section, we will also explain the on-board connection.

3.2.1 Generating and debugging an example project

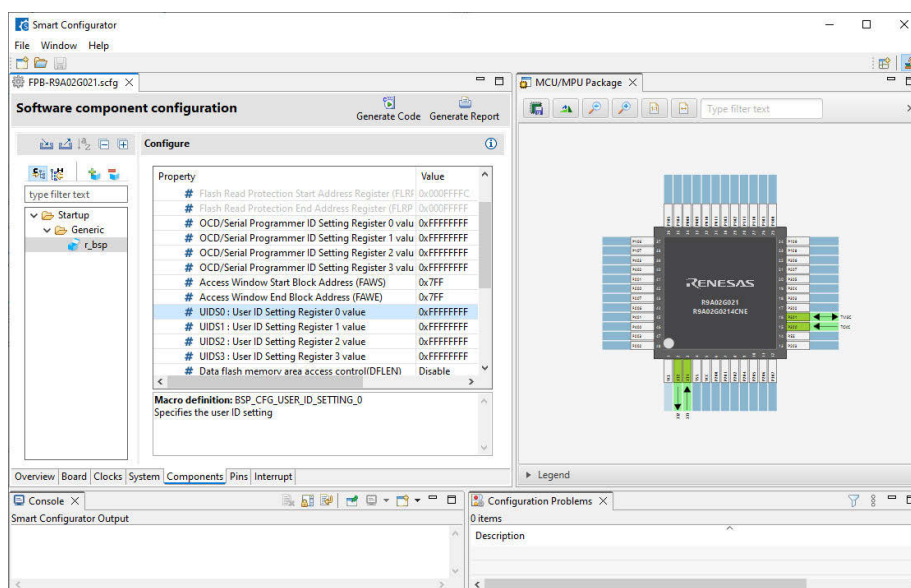
To generate an example and BSP code for the Renesas FBP-R9A02G021 board, it is recommended to use the Renesas Smart

Configurator tool. To create a new project for EWRISCV and FBP-R9A02G021, start the Smart Configurator tool and do File > New. The following dialog will appear:



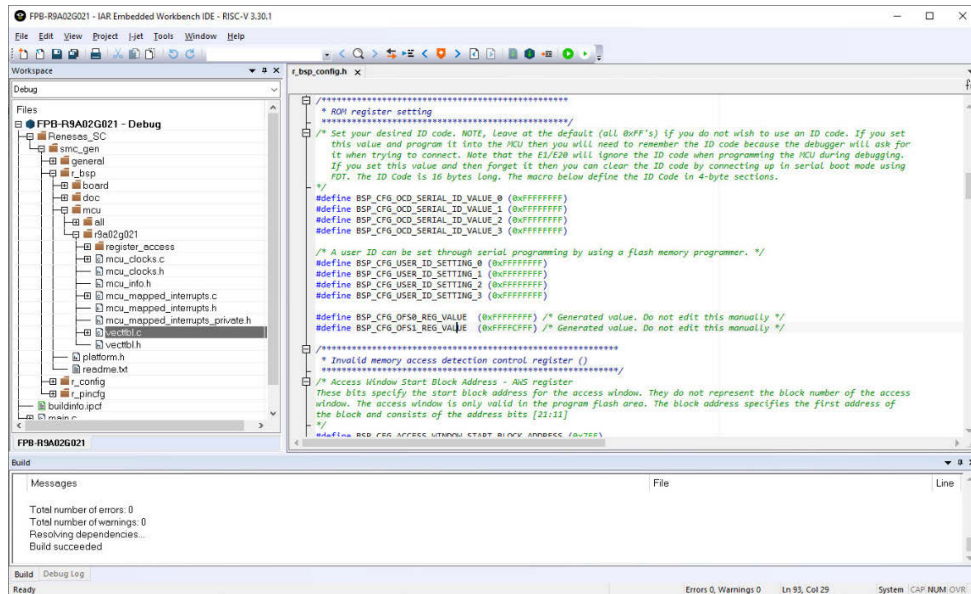
Select the FPB- R9A02G021 board and the IAR Renesas RISC-V Toolchain. After this, it is

possible to configure system details, such as clocks, components, and pin settings:



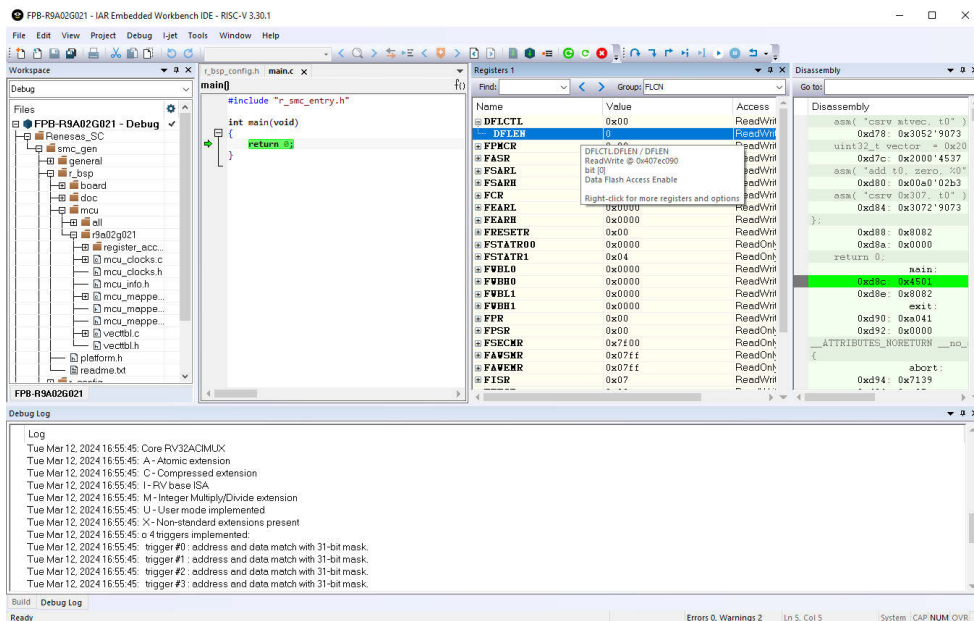
The standard settings are often OK to start with, so it is fine just to click the Generate Code button. The tool will generate code and produce a clickable link to the resulting files in the Console tab in the Smart Configurator Output window.

In the example above, the generated EWRISCV workspace file will be called FPB-R9A02G021. eww. Open the workspace file in EWRISCV, and the following will be shown:



As we can see, the UID50: User ID Setting Register value that is set to 0xFFFFFFFF in the Smart Configurator can be found in the generated code, in r_bsp_config.h, as macro definition BSP_CFG_USER_ID_SETTING_0.

After this, it is possible to do Project > Make. The project is normally pre-configured to use the I-jet debug probe. If not, go to Project > Options > Debugger and set the Driver to I-jet. Now, to download the generated application to the board, do Project > Download and Debug, and a debug session will start. This is how it looks:



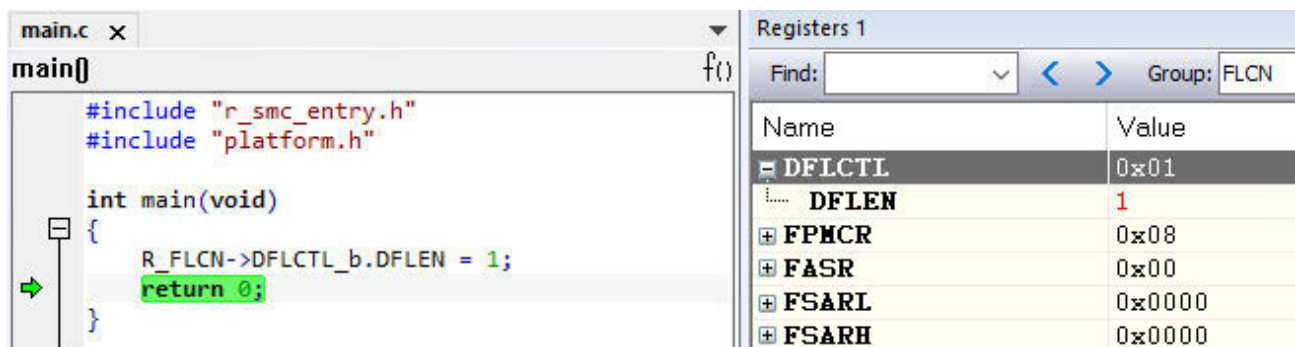
In the debug session, we can study various CSRs in the Registers window, such as the Data flash memory area access control register DFLEN in the FLCN register group. Here, we can see that access to the data flash memory area is disabled since the value is 0.

It is also possible to see that the connected core is an RV32ACIMU core in the Debug Log window. That means that the core implements the Atomic, Compressed, Integer, Multiplication, and User Mode extensions.

To study a simple code change in the debugger, we can add #include "platform.h" in our main.c file, and set the CSR bit DFLEN to 1 by adding R_FLCN->DFLCTL_b.DFLEN = 1 to the main function. The register definition can be found in the generated R9A02G021-specific BSP file iodefine.h in the:

```
r_bsp\mcu\r9a02g021\register_access
```

folder.



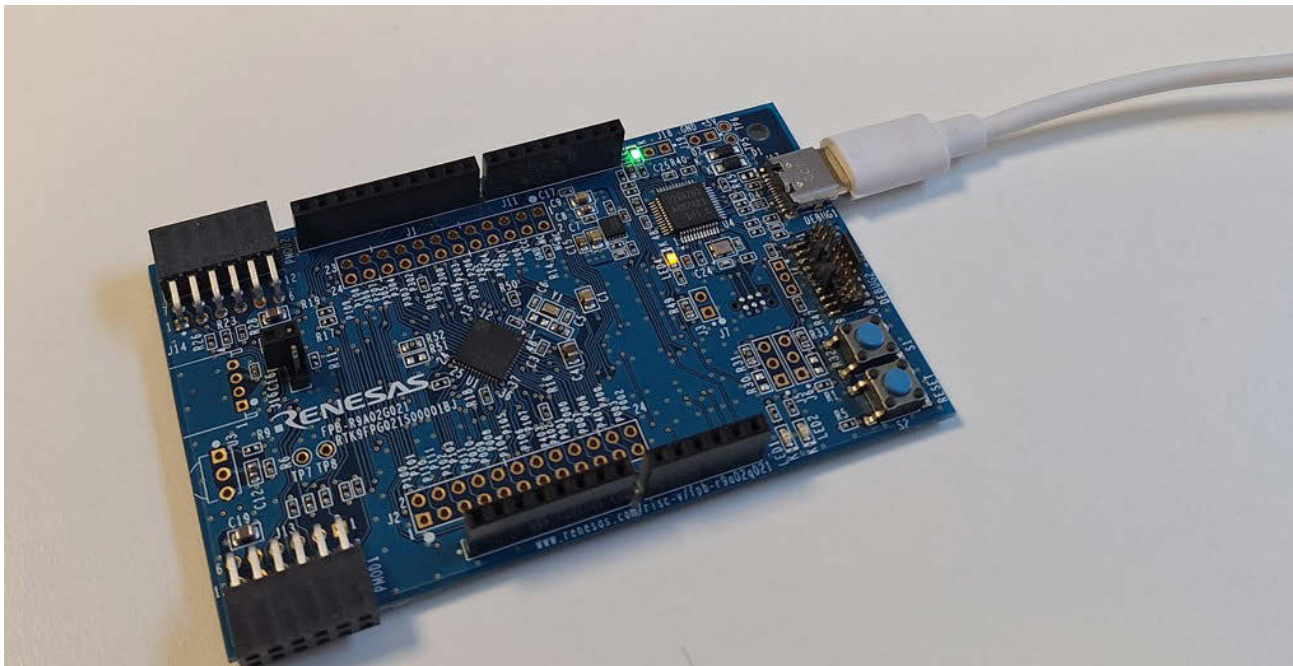
The screenshot shows an IDE with a C code file named 'main.c' and a 'Registers 1' window. The code in 'main.c' includes 'r_smc_entry.h' and 'platform.h', and sets 'R_FLCN->DFLCTL_b.DFLEN = 1;'. The 'Registers 1' window shows the 'DFLEN' register in the 'DFLCTL' group with a value of '1'.

```
main.c x
main()
#include "r_smc_entry.h"
#include "platform.h"

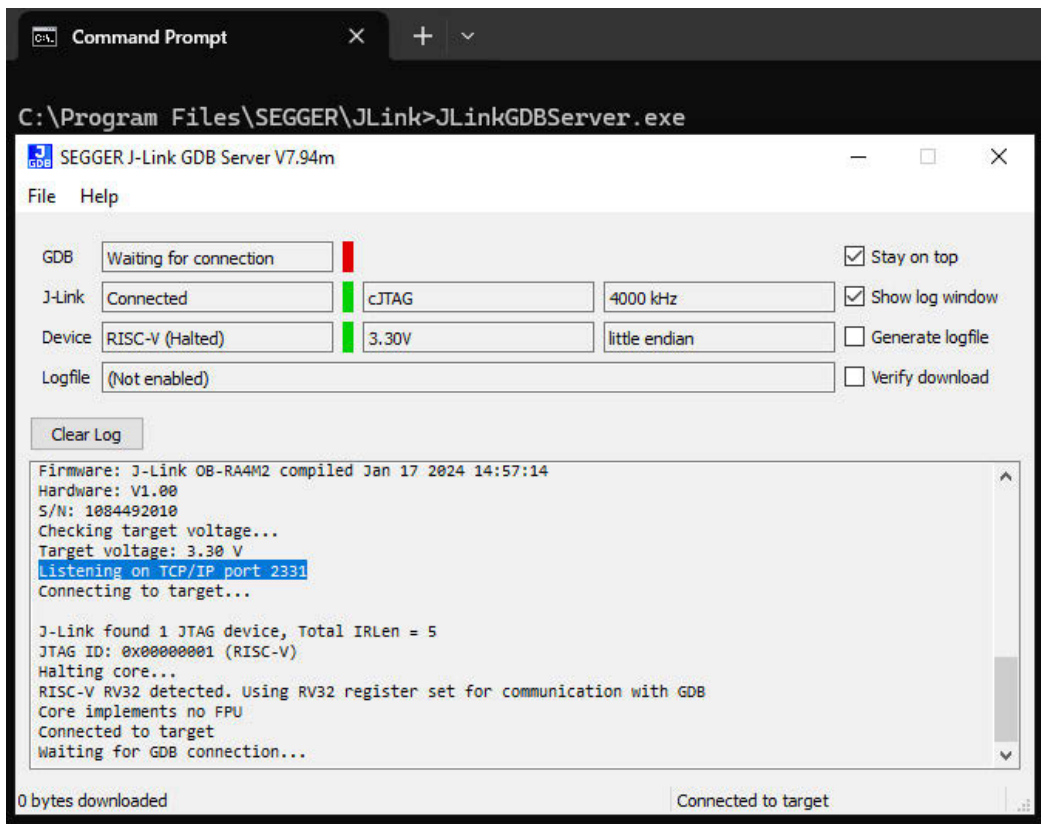
int main(void)
{
    R_FLCN->DFLCTL_b.DFLEN = 1;
    return 0;
}

Registers 1
Find: [ ] Group: FLCN
Name Value
DFLCTL 0x01
DFLEN 1
FPMCR 0x08
FASR 0x00
FSARL 0x0000
FSARH 0x0000
```

There is also an on-board J-Link debug probe on the FPB- R9A02G021 board. To use it, connect it to the USB-C port like this:

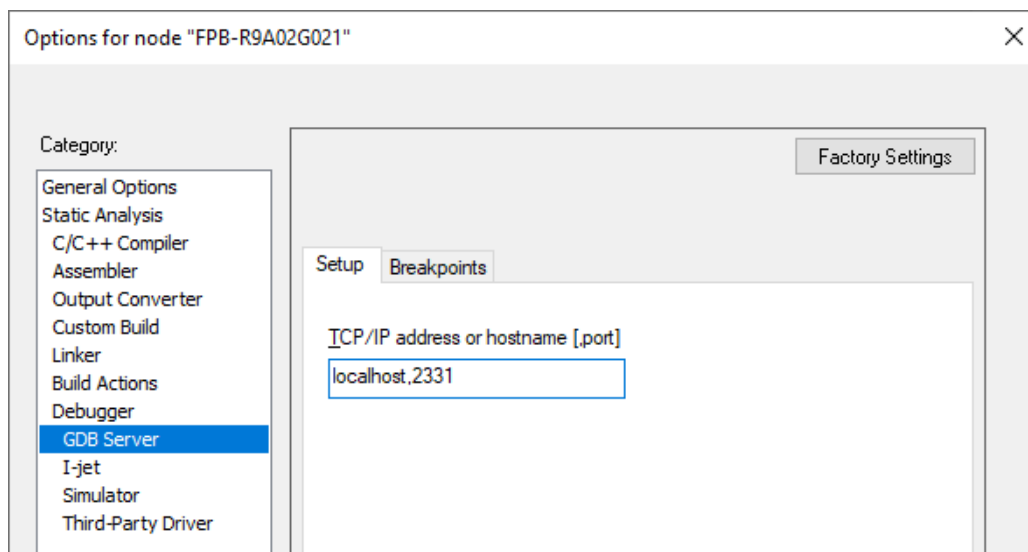


Then, start SEGGER's JLinkGDBServer.exe application and let it connect to the RISC-V core:

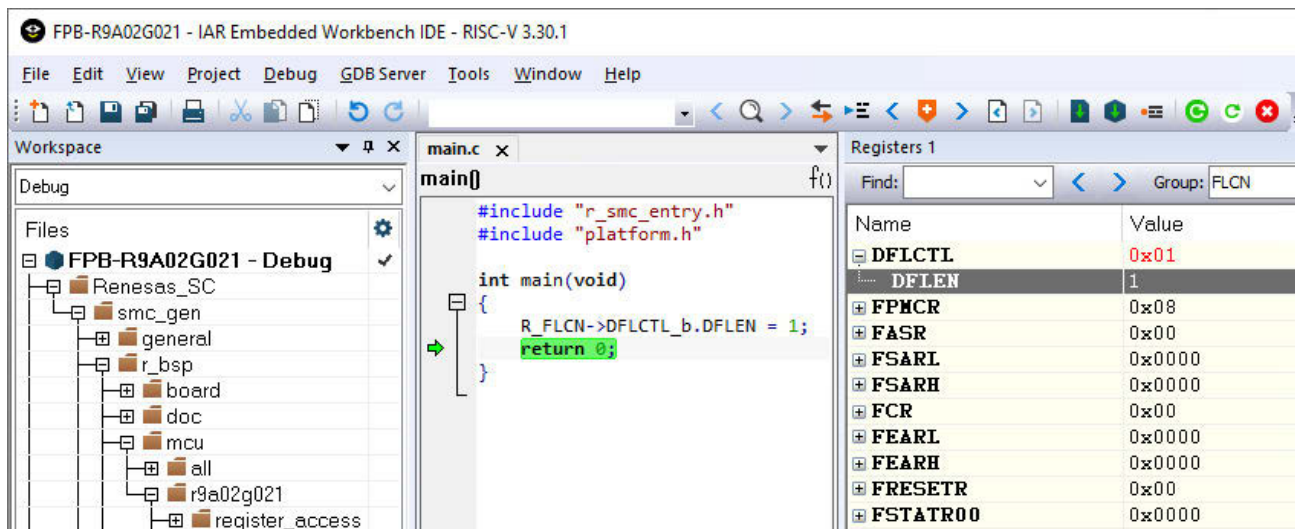


Now, the JLinkGDBServer.exe application waits for a TCP/IP connection on port 2331. In EWRISCV, go to Project > Options > Debugger

and select the GDB Server driver. Configure the GDB Server driver as follows (localhost,2331):



After this, we can start the debug session, and the same register values can be studied:



Note that currently, the GDB Server performance and functionality are somewhat limited in EWRISC-V. For example, it is not possible to study certain CSRs while using the EWRISC-V GDB Server implementation. This will be improved in later releases. With an I-jet debug probe, no such limitations exist.

A hand holding a smartphone is the central focus, set against a dark blue background. The background features a white grid pattern and various glowing digital elements, including lines, dots, and faint text, creating a high-tech, futuristic atmosphere. The lighting is soft, highlighting the hand and the phone.

4. Navigating RTOS, automated workflows, and code quality



4. Navigating RTOS, automated workflows, and code quality

When diving into the RISC-V architecture and its ecosystem, understanding and utilizing the right tools is crucial for effective learning and development. Among these, Real-Time Operating Systems (RTOS) play a vital role, especially as software projects grow in complexity. RTOSs like Azure RTOS (ThreadX) and FreeRTOS, available as sample implementations with the IAR Embedded Workbench for RISC-V (EWRISCV), offer structured resource, task, and timing management essential for real-time applications. Additionally, SAFERTOS provides a pre-certified, deterministic RTOS solution for applications requiring the utmost safety and reliability.

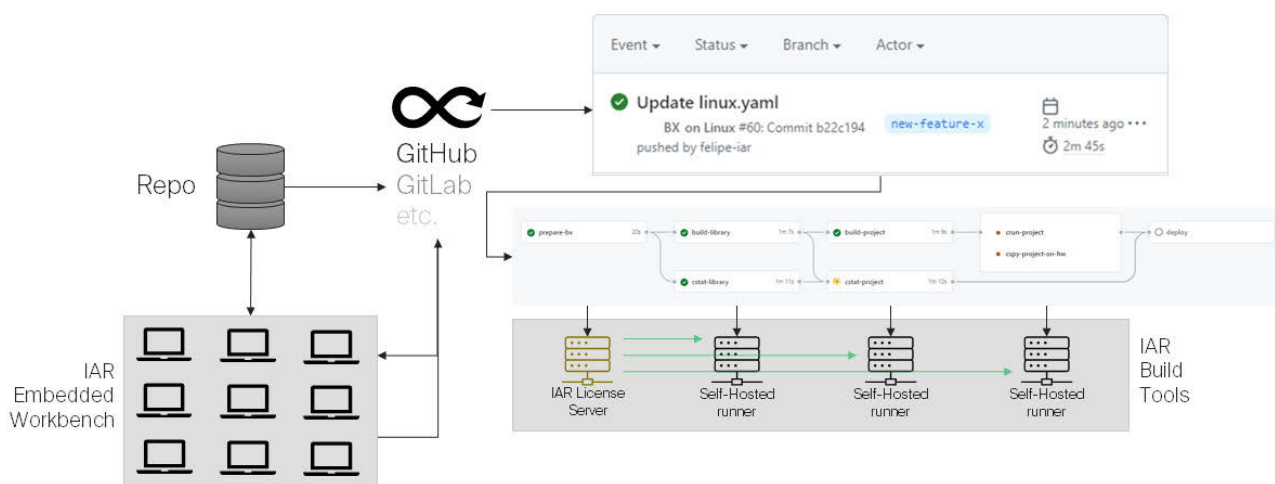
This book will not delve into the specifics or step-by-step details of the RTOSes. We encourage developers to experiment with the provided out-of-the-box examples and seek additional resources on the RTOS vendors' websites or directly on GitHub.

Additionally, to address common development challenges associated with modern workflows, automation, and Continuous Integration/Continuous Deployment (CI/CD) pipelines, IAR enhances the ecosystem by offering the IAR Build Tools for RISC-V, featuring:

Efficient Software Building and Testing: The comprehensive tool suite, including the IAR C/C++ Compiler, Assembler, Linker, and IARBuild, facilitates efficient building and testing of critical software on a large scale, ensuring reliability and performance in deployment.

Adaptability and Performance Across Environments: Designed to adapt to various organizational needs, these tools can be deployed on small build servers with a few licenses or scaled to support hundreds of parallel builds, ensuring high performance regardless of the scale.

Integration with Modern Development Workflows: Built with modern software development practices in mind, the IAR Build Tools seamlessly integrate into CI/CD pipelines, supporting Virtual Machines, Containers (Docker), and Self-hosted Runners. This compatibility ensures that developers can maintain efficient, continuous integration and deployment processes, which is crucial for modern software development.



By leveraging the RTOS options and the IAR Build Tools for RISC-V, developers can navigate the complexities of RISC-V development with a comprehensive set of resources designed to address key challenges and enhance the quality and efficiency of software projects.

Finally, in addition to RTOS and automated workflows, ensuring code quality and facilitating code reuse are pivotal for the sustainability and efficiency of software projects. IAR addresses

these aspects with its powerful static analysis tool, C-STAT, fully integrated into the IAR Embedded Workbench. C-STAT performs an exhaustive analysis on the source code level, identifying potential issues early in the development process. This proactive approach to code quality helps developers adhere to industry-standard coding practices, including MISRA, CWE, and CERT C/C++ Secure Coding Standards.



5. Conclusion



5. Conclusion

In conclusion, this book serves as a comprehensive guide for developers and professionals navigating the intricacies of embedded software development within the RISC-V ecosystem. By elucidating the features and capabilities of RISC-V and the extensive toolset provided by the IAR Embedded Workbench for RISC-V (EWRISCV), it lays a solid foundation for understanding CPU instruction sets, stack behaviors, and the pivotal role of Real-Time Operating Systems (RTOS) in managing complex software projects.

The book emphasizes the importance of selecting the right tools, such as the IAR Build Tools for RISC-V and the C-STAT static analysis tool, to enhance development workflows, ensure code quality, and facilitate code reuse.

These solutions offer a range of functionalities from efficient software building and testing to adherence to industry-standard coding practices, thereby mitigating security risks and coding errors.

With practical insights into the use of RISC-V on hardware along with a focus on modern development practices including CI/CD pipelines, this guide encourages experimentation and further exploration beyond its pages. It underscores the value of certified compilers like those offered by IAR, which streamline the development process, especially in systems requiring functional safety.

By leveraging the resources and examples provided, along with the IAR Embedded Workbench's powerful capabilities, developers are well-equipped to tackle the challenges of RISC-V development, ensuring their projects are not only efficient and reliable but also maintainable and secure for future endeavors.

References

1. Computer Architecture: A Quantitative Approach to Design, Implementation, and Evaluation, David A. Patterson, John L. Hennessy
2. IAR C/C++ Development Guide, Linking using ILINK https://wwwfiles.iar.com/riscv/EWRISCV_DevelopmentGuide.ENU.pdf
3. Top Ten Fallacies About RISC-V, David Patterson <https://riscv.org/blog/2023/03/top-ten-fallacies-about-risc-v/>
4. The RISC-V Instruction Set Manual Volume I: Unprivileged ISA
5. The RISC-V Instruction Set Manual Volume II: Privileged Architecture Document Version 20211203
6. <https://hsandid.github.io/posts/risc-v-custom-instruction/>
7. <https://github.com/riscv/riscv-fast-interrupt/blob/master/clic.adoc>
8. AndesCore AX45MP-1C Processor Reference Manual, <https://www.andestech.com/wp-content/uploads/AX45MP-1C-Rev.-5.0.0-Datasheet.pdf>
9. Nucleisys Customer Cases, <https://www.nucleisys.com/product/rvipes/lcxp/>
10. Open-Source RISC-V Architecture IDs, <https://github.com/riscv/riscv-isa-manual/blob/latex/marchid.md>