Ma Z, Qiao L, Yang MF *et al.* Verification of real time operating system exception management based on SPARCv8. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 36(6): 1367–1387 Nov. 2021. DOI 10.1007/s11390-021-1644-x

# Verification of Real Time Operating System Exception Management Based on SPARCv8

Zhi Ma[1], Lei Qiao[1,*], *Member, CCF*, Meng-Fei Yang[2], Shao-Feng Li[3], and Jin-Kun Zhang[1]

[1] *Beijing Institute of Control Engineering, Beijing 100190, China*
[2] *China Academy of Space Technology, Beijing 100094, China*
[3] *School of Computer Science and Technology, Xidian University, Xi'an 710071, China*

E-mail: tboymz@outlook.com; fly2moon@aliyun.com; fly2sun@aliyun.com; lsffree@aliyun.com
jinkun1993218@outlook.com

Received May 31, 2021; accepted November 16, 2021.

**Abstract** Exception management, as the lowest level function module of the operating system, is responsible for making abrupt changes in the control flow to react to exception events in the system. The correctness of the exception management is crucial to guaranteeing the safety of the whole system. However, existing formal verification projects have not fully considered the issues of exceptions at the assembly level. Especially for real-time operating systems, in addition to basic exception handling, there are nested exceptions and task switching by exceptions service routine. In our previous work, we used high-level abstraction to describe the basic elements of the exception management and verified correctness only at the requirement layer. Building on earlier work, this paper proposes EMS (Exception Management SPARCv8), a practical Hoare-style program framework to verify the exception management based on SPARCv8 (Scalable Processor Architecture Version 8) at the design layer. The framework describes the low-level details of the machine, such as registers and memory stack. It divides the execution logic of the exception management into six phases for comprehensive formal modeling. Taking the executing scenario of the real-time operating system SpaceOS on the Beidou-3 satellite as an example, we use the EMS framework to verify the exception management. All the formalization and proofs are implemented in the interactive theorem prover Coq.

**Keywords** operating system, exception, Scalable Processor Architecture Version 8 (SPARCv8), Coq, formal verification

## 1 Introduction

OS kernel is the backbone of almost every safety-critical computer system. As the lowest level function module of the OS kernel, the exception management is usually implemented in the assembly language to pursue high efficiency. It is responsible for making abrupt changes in the system's control flow to react to exceptional events in the system. The correctness of the exception management module is the basis for ensuring the security of the OS kernel. Thus, it is highly desirable to formally certify the correctness of these OS kernel programs, especially for real-time operating systems.

Unfortunately, to simplify the formalization job of the target program, the existing OS verification projects either do not model the exception management or use the modeling method based on the abstraction layer to verify the correctness of exception management. For example, seL4[1] and CertiKOS[2] disable most asynchronous exceptions (interrupts) when running the kernel, and implement interrupt points by polling instead of temporarily enabling interrupts. Chen *et al.*[3] and Xu *et al.*[4] used abstract specifications to describe the behavior of the exception management at the C level. Our previous work[5] also used high-level abstractions to verify the exception mana-

Regular Paper

Special Section on Software Systems 2021—Theme: Dependable Software Engineering

This work was supported by the National Natural Science Foundation of China under Grant Nos. 61632005 and 62032004.

*Corresponding Author

gement at the requirement layer. The above work hid the detailed information of the underlying machine, such as registers and stacks. Therefore, the processing logic of exception events has not been wholly modeled, and the exception management module has not actually been verified at the assembly level.

Scalable Processor Architecture Version8 (SPARCv8) [6] is an efficient and reliable microprocessor, and its instruction set architecture has been widely used in workstations and various embedded systems. For example, SpaceOS [7] running on the SPARCv8 processor is a real-time embedded operating system developed by Beijing Institute of Control Engineering (BICE). It has been deployed in the central computer of many space exploration projects such as Chang'e lunar exploration, Beidou navigation, and Mars exploration. SPARCv8 uses some unique mechanisms to improve system performance and reduce power consumption, such as register windows and window rotation mechanisms that are closely related to the exception management. Zha *et al.* [8] and Wang *et al.* [9] performed assembly-level semantic descriptions for these unique mechanisms. However, the former did not involve exception issues, and the latter ignored the impact of the OS level on the system state. For example, during the execution of the Exception Service Routine (ESR) program, there may be OS functions such as OSTaskCreat and OSTaskDelete that can cause task switching. At the same time, the handling of the underlying details of the machine will also change, and the processor needs to save multiple register windows of SPARCv8 instead of performing window rotation actions.

On the whole, the exception management is the most relative OS functional module to the hardware layer. Therefore, we have to consider the challenges posed by both operating systems and hardware platforms at the same time.

First, the real-time operating system has the characteristics of concurrent multi-tasking and frequent interruptions. The preemptible feature of the OS kernel forces the system not only to face basic exception handling, but also to consider nested exceptions and task switching during exception handling.

• *Nested Exceptions.* If a high-priority exception occurs when a low-priority ESR is in execution, the scheduler lets the high-priority exception preempt the CPU resource. The low-priority ESR will be executed after the high-priority ESR is completed. If a low-priority exception occurs when a high-priority ESR is in execution, the scheduler will continue executing the high-priority ESR. This mechanism ensures that more emergent exception is executed prior to the lesser one and allows the preemption operation.

• *Task Switching.* If the ESR program involves task-create or task-delete function, the task switching will occur after completing the ESR program. The relevant data of the new task will be stored in the corresponding register waiting for the processor to schedule, which leads to a change in the data object when the context recovery operation is performed. At this time, the context data recovered is not for the original task but the new task.

Second, unlike the stack push and pop operations performed by ARM and x86 when handling exceptions, SPARCv8 uses a special window register and window wheel rotation mechanism.

• The window roulette contains eight sub-windows. Each sub-window has 32 general registers, which can be divided into four groups: *ins*, *outs*, *local*, and *global*. All sub-windows share the *global* register group. The *outs* register group of the previous sub-window is equivalent to the *ins* register group of the next sub-window. The *local* group is a unique register group for each sub-window. The processor only needs to deal with a signal sub-window used by the previous task for basic exceptions. When task switching occurs, the processor needs to save all sub-windows used by the previous task to ensure the data security of the task context.

In this paper, we tackle these challenges directly and present a novel framework Exception Management SPARCv8 (EMS) to certify the exception management function of the operating system based on the SPARCv8 processor architecture. We strictly follow the SPARCv8 guidelines to model the details of the underlying machine, and the verification framework can be easily used in different operating systems based on SPARCv8, which is essential for scalability of any verification effort and critical for reasoning about the interruptible code. Our paper makes the following new contributions.

• We describe the mathematical model of SPARCv8 ISA (instruction set architecture) and OS memory closely related to the exception management, including general registers, special registers, memory stacks, and global variables that represent flag information.

• We propose the EMS framework, which divides the execution logic of the exception management into six phases for detailed formal modeling. The six phases are as follows: entering the exception management, protecting Context, entering ESR, leaving ESR, restoring

Context, and exiting the exception management.

• Taking the executing scenario of the real-time operating system SpaceOS on the Beidou-3 satellite as an example, we use EMS to prove the correctness of the exception management. Like proving programs with Hoare triples [10], we prove that these programs satisfy the given pre-conditions and post-conditions.

• All of the formalization and proofs have been implemented in Coq, which contains around 15 000 lines of the Coq code in total.

The remainder of this paper is organized as follows. Section 2 introduces the exception control flow and abstract model of three different exceptions in real-time OS. Section 3 models the registers and memory stack involved in the exception management. Section 4 introduces the design and implementation of the EMS verification framework. Section 5 uses EMS to verify the exception management function of SpaceOS. Section 6 introduces related work. Section 7 summarizes the paper and proposes prospects for future research.

## 2 Overview of Exceptions

Computer systems must react to changes in system state that are not captured by internal program variables and are not necessarily related to the program's execution [11]. For example, a hardware timer goes off at regular intervals and must be dealt with; packets arrive at the network adapter and must be stored in memory. Modern systems respond to the above situations by causing sudden changes in the control flow. People refer to these abrupt changes in general as "exception control flow", which occurs at all levels of the computer system. For example, at the operating system level, the kernel transfers control from one user task to another through context switching. At the application level, a task can send a signal to another task that abruptly transfers control to a signal handler in the recipient.

The exception is a form of exception control flow, and the handling of the exception is implemented partly by the hardware and partly by the operating system. The part implemented by the operating system is the exception management, and the design for it varies with the hardware platform. It is worth noting that the terminology for each category of exceptions varies from system to system. Processor ISA [12, 13] specifications often distinguish between asynchronous "interrupts" and synchronous "exceptions" yet which provide no umbrella term to refer to these very similar concepts. To avoid confusion, we use the word "exception" as the general term in this paper and distinguish between asynchronous exceptions (interrupts) and synchronous exceptions (traps) only when it is appropriate.

Every type of exception that may occur in the system is assigned a unique non-negative integer type of exception label. Some of these labels are allocated by the designer of the processor, such as division by zero, memory address misalignment and window overflow, while the rest of the labels are allocated by developers of the operating system, such as system calls and task switching. When the computer is started (restart or power on), the operating system will allocate and initialize a jump table called exception vector [14], which contains the entry address of the ESR corresponding to each exception. Before executing each instruction, the system checks for pending exceptions. If any exceptions are present, the system selects the one with the highest priority, and then the control flow jumps to the exception management.

The operating system abstracts the operation of the CPU as a task and presents it to the user. As shown in Fig.1, this is a basic exception handling process. If the system receives an emergency request signal for the exception event, task $I$ in the running state will be interrupted (the processor will wait for the execution of the current instruction to complete before responding to the asynchronous exception event). The processor first records the flag information of the exception event, then searches for the ID number matching the current exception event in the exception vector table, and finally guides the control flow to jump into the exception management. There are three main missions to complete for the exception management:

• protect the context data of task $I$;

• guide the control flow to jump into the entry address of the corresponding ESR, and set the correct return address;

• restore the context data of task $I$ and guide the control flow to exit the exception management.

In order to improve the system's ability to respond to critical events, the real-time OS must consider nested exceptions. As shown in Fig.2, the ESR program of exception event Exce 1 is interrupted by the more urgent exception event Exce 2, which causes the nested exceptions. The processor first suspends ESR 1 and protects its context data, then runs ESR 2 until the program is completed, and finally restores the context data of ESR 1. It is noteworthy that in the exception management process, the system is always in the critical zone, shielding all exception events. The purpose is to prevent data

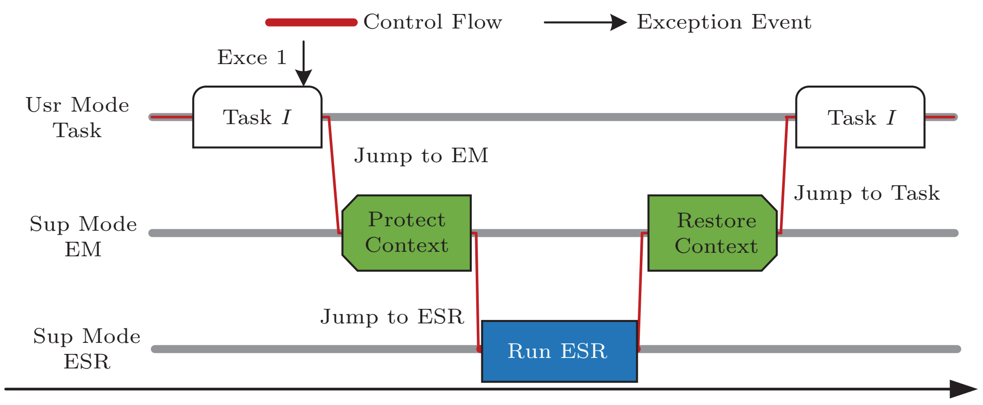

Fig.1. Abstract model of basic exception. EM represents the control flow of exception management. Exce represents an exception event. Usr: user; Sup: supervision.

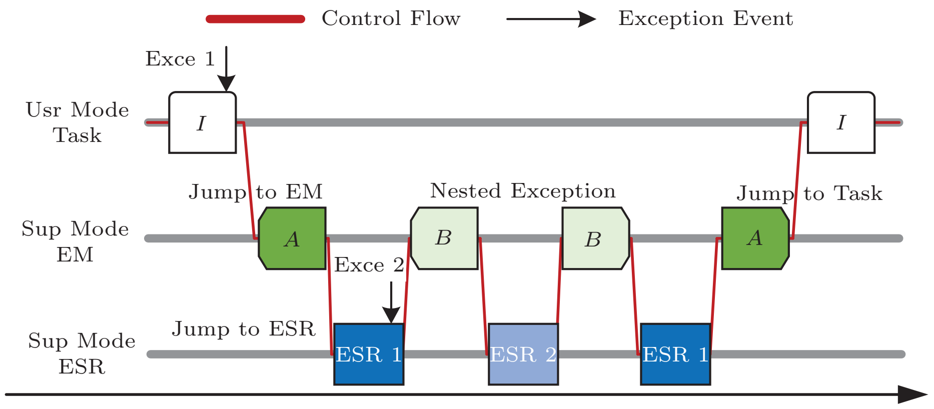

Fig.2. Abstract model of nested exceptions. EM represents the control flow of exception management. Exce represents an exception event. Usr: user; Sup: supervision.

loss caused by interruption when the system is saving and restoring context information. Therefore, exception nesting can only occur during the execution of the ESR program.

The real-time operating system has the characteristics of being preemptible. Sometimes there are OS functions such as OSTaskCreate and OSTaskDelete during the execution of the ESR program, which may cause task switching. For example, if the new task created by ESR has a higher priority than the current task, it will be set to the ready state by the processor during the context restore phase. Moreover, when the control flow exits the exception management, the CPU control will be transferred to the new task. As shown in Fig.3, task $I$ is interrupted by an exception event, and the control flow jumps to the exception management to protect the context data of task $I$. Then the ESR program is executed to create a new task $H$ with a high priority. In the context restore phase, the processor stores the context data of task $H$ in the corresponding register and stack. Finally, the CPU control is assigned to task $H$ by the system when the exception management ends. It should be noted that the object has been changed in the context restore phase. In order to prevent the loss of context data of task $I$, it is necessary to back up the memory address of its data to the task control block.

## 3 Machine Modeling

In the computer system, the CPU provides the operating power, which is most directly reflected in the change of the instruction register. When the computer is powered on, the reset logic circuit writes a fixed address into the instruction register, and the CPU reads and executes instructions from this address. During the execution of the instruction, the CPU hardware logic completes the state change according to the instruction fetched. This change consists of two parts: 1) the change of the CPU's own state, including instruction registers, general registers, status registers and control registers; 2) changes in memory status, including global variables and task stacks. The update of the instruction

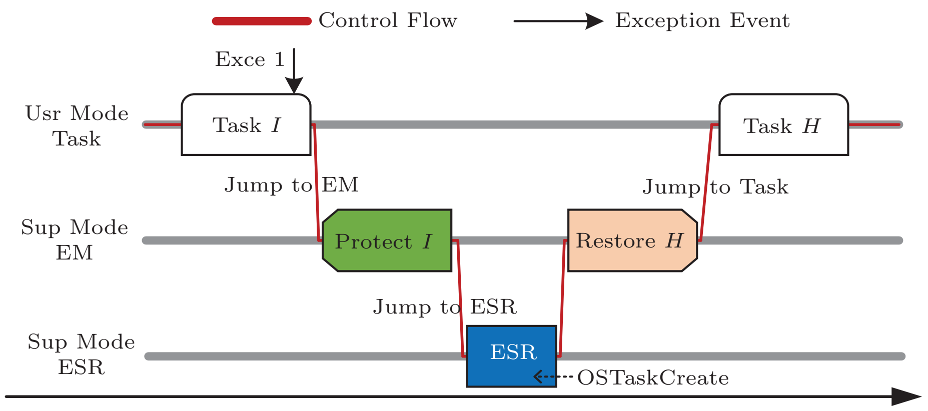

Fig.3. Abstract model of task switching in exception. EM represents the control flow of exception management. Exce represents an exception event. Usr: user; Sup: supervision.

register is commonly referred to as "instruction address plus one" or "transfer address generation". The driving force of system operation is formed by this automatic update. Driven by this force, the CPU moves on the sequence of instructions in order, which we call "control flow". During the control flow, the CPU constantly changes its own and external state, which is what we usually call the execution of the program. Therefore, the operating state of the computer can be represented by a two-tuple (register state, memory state).

### 3.1 Register Modeling

The types and usage of registers are closely related to the hardware platform. This article strictly follows the SPARCv8 standard to describe the register mathematically. The register status $R$ (*RegStat*) of the system consists of general and special registers. Fig.4 shows the characteristics of general registers. There are a total of eight register sub-windows, which are represented by $w_7$ to $w_0$. When the system executes instructions, it will switch windows in sequence from $w_7$ to $w_0$. These windows overlap each other and are connected in a circular loop. Each window has 32 general registers, of which eight registers are global registers, all windows share this *global* register group, and the other 24 registers can be divided into three groups: *outs*, *ins*, and *local*. Adjacent windows share the *ins* and *outs* groups.

When the task is running, the data saved in the window register is the context information of the task. At any time, the system uses only one register sub-window. Users can use the RESTORE and SAVE instructions to control the switching of the window. The RETT instruction and exception events in the supervisor mode can also cause the switching of the window. The register field *cwp* represents the window number currently in use by the system. The WIM register in Fig.4 indicates which register sub-windows are used by the current task. If a SAVE instruction occurs when all sub-windows have been used, the system will trigger an overflow trap in the window register. The right rotation of the window caused by the exception event will also cause the window to overflow. However, it will not trigger the overflow trap because the system has entered the critical area at this time. Hence, the exception management needs to handle the overflow of the window register actively. When a RESTORE or RETT instruction occurs, if the corresponding bit of the WIM register is invalid, the underflow trap of the window register will be triggered. The formal definition of general registers is shown in Fig.5. $W$ ($WinReg$) represents a register sub-window, including four register groups: *ins*, *outs*, *local*, and *global*. It is worth noting that some general registers of SPARCv8 have fixed usage, for instance the value of $g0$ is always 0, and $i6$ and $o6$ are used as *fp* and *sp* registers.

In addition to general registers used for data calculation, SPARCv8 also has some special registers to reflect certain system states. We use $S$ (*SpeReg*) to represent these special registers. *psr* (*Psrreg*) represents the processor status register, and it contains various fields that control the processor and hold status information. There are four fields related to the exception management: the field *et* is used to indicate whether the processor can respond to exception events. When *et* is 0, the system shields all exception events; when it is 1, the system can respond to exception events. The field *pil* indicates the lowest priority of exception events that the system can accept. The field *cwp* indicates which register sub-window is currently being used by the system. *wim* is a window invalid mask register, usually

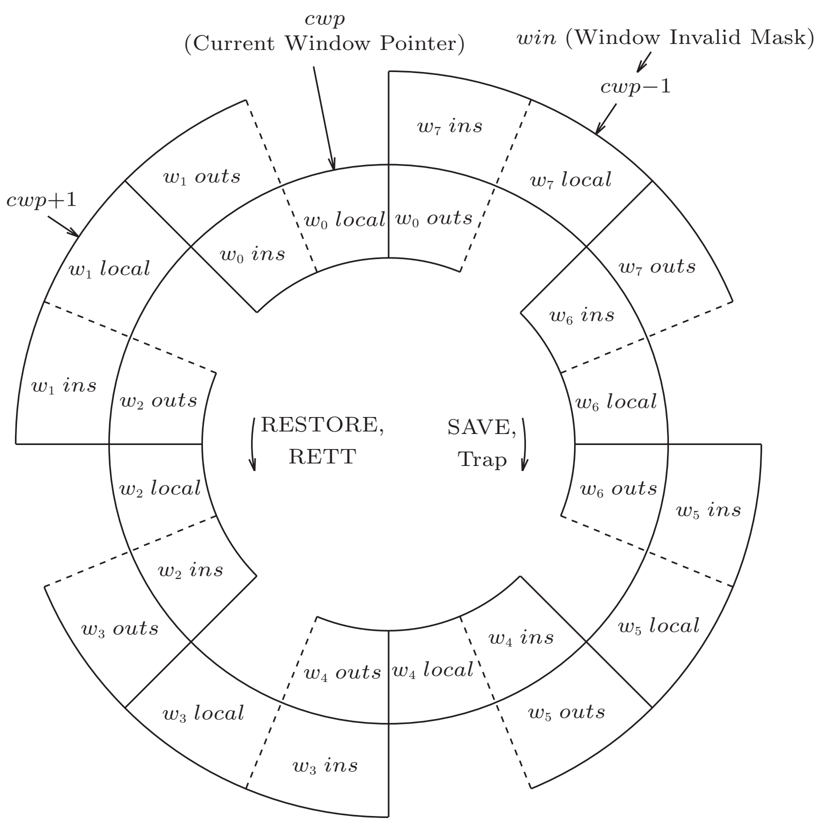

Fig.4. Feature of SPARCv8 window register.

| | | |
|---|---|---|
| $(RegStat)$ | $R$ | $::= (S, W)$ |
| $(WinReg)$ | $W$ | $::= ins \mid outs \mid local \mid global$ |
| $(Insreg)$ | $ins$ | $::= i0 \mid ... \mid fp \mid i7$ where $fp = i6$ |
| $(Outsreg)$ | $outs$ | $::= o0 \mid ... \mid sp \mid o7$ where $sp = o6$ |
| $(Locreg)$ | $local$ | $::= l0 \mid ... \mid ... \mid l7$ |
| $(Gloreg)$ | $global$ | $::= g0 \mid ... \mid ... \mid g7$ |
| $(SpeReg)$ | $S$ | $::= psr \mid wim \mid tbr \mid pc \mid npc \mid \tau$ |
| $(Psrreg)$ | $psr$ | $::= cwp \mid et \mid ps \mid s \mid pil$ |
| $(Tbrreg)$ | $tbr$ | $::= tba \mid tt$ |

$$\begin{aligned}
exce\_enable(R) &\twoheadrightarrow R(et) \neq 0\\
exce\_disable(R) &\twoheadrightarrow R(et) = 0\\
sup\_mode(R) &\twoheadrightarrow R(su) \neq 0\\
usr\_mode(R) &\twoheadrightarrow R(su) = 0\\
has\_exce(R) &\twoheadrightarrow R(\tau) = 1\\
enable\_exce(R) &\overset{\text{def}}{=\!=} R\{1 \rightsquigarrow et\}\\
disable\_exce(R) &\overset{\text{def}}{=\!=} R\{0 \rightsquigarrow et\}\\
to\_usr(R) &\overset{\text{def}}{=\!=} R\{0 \rightsquigarrow su\}\\
to\_sup(R) &\overset{\text{def}}{=\!=} R\{1 \rightsquigarrow su\}\\
clear\_exce(R) &\overset{\text{def}}{=\!=} R\{0 \rightsquigarrow \tau\}
\end{aligned}$$

Fig.5. SPARCv8 register model.

used to judge window overflow or underflow. *tbr* is an exception base address flag register used to indicate the essential attributes of the exception. *pc* and *npc* represent program counters, *pc* holds the address of the instruction being executed by the system, and *npc* holds the address of the next instruction to be executed. The register $\tau$ indicates whether there is an exception event in the system. If $\tau$ is zero, it means that it does not exist. Otherwise, it indicates that there are exception events waiting to be responded. Finally, we use $\beta$ and $[\beta]$ to represent the addressing mode of the SPARCv8 register.

$$\beta = \begin{cases} reg, \\ reg + reg, \\ reg + simm, \\ reg - simm, \\ simm, \\ simm + reg. \end{cases} \quad [\beta] = \begin{cases} M[reg1 + g0], \\ M[reg1 + reg2], \\ M[reg + simm], \\ M[reg - simm], \\ M[simm + g0], \\ M[reg + simm]. \end{cases}$$

Due to specific implementation limitations, SPARCv8 real machines put these fields information representing different functions into the same register.

However, we split these fields to express the state of the system in mathematical modeling for expressing the state of the system more clearly. As shown in Fig.5, we use the formal notation "$\rightsquigarrow$" for the assembly instruction MOV, which represents the modification of the value of a register field. Different values in the register field also reflect different states of the system, which we use the symbol "$\twoheadrightarrow$" to represent. The function *enable_exce* means that the field *et* is set to 1, and the system can respond to exception events (*exce_enable*). The function *disable_exce* means that the field *et* is set to 0, and the system shields all exception events (*exce_disable*). The function *to_sup* means that the field *su* is set to 1, and the system enters the supervision mode (*sup_mode*). The function *to_usr* indicates that the field *su* is set to 0, and the system enters the user mode (*usr_mode*). The function *has_exce* means there is a pending exception signal in the system.

The *ins* and *outs* register groups are shared between two adjacent register windows of SPARCv8. In order to describe the characteristics of window switching, we introduce the register window buffer $B$ (*BuffWinReg*) shown in Fig.6, which contains two register groups: *bins* and *bouts*. When the window performs the switching action, *ins* and *outs* of the current window are first saved to *bins* and *bouts* respectively, which are defined as the function *save_ios*. Then the processor determines the direction of rotation of the window, if the window turns to the right, *cwp* is reduced by 1, and the contents of *bins* groups are stored in the *outs* groups, described by the function $win\_right$. If the window turns to the left, *cwp* is incremented by 1, and the contents of the *bouts* register groups are stored in the *ins* register groups, defined as the function $win\_left$.

## 3.2 Memory Modeling

The memory status $M$ (*MemStat*) involved in the exception management consists of two parts. One is the address of a global variable defined in the operating system, and the other is the memory stack address used when protecting and restoring the context data. We use $V$ (*MemoVar*) to represent global variables in memory, and *nest* means the number of nesting layers of exceptions. If *nest* is greater than 1, it means that a nesting exception has occurred in the system, and $ne\_flag$ (see Fig.7) returns the status true; otherwise, it returns false. *curtcb* represents the current task control block, and *pretcb* represents the previous task control block, which will be used when the task is switched. *ossm* represents the ESR address table, which stores the entry addresses of all exception service routines. *swflag* is used to determine whether a task switching has occurred during the execution of ESR. If $swflag$ is 0, it means that no switch has occurred. *osregl*1 and *osregtt* represent the memory variable addresses used to back up register *l*1 and field *tt* respectively. $H$ (*Stack*) is the stack address used in the exception management, which is composed of the register *sp*, representing the top of the stack and the address offset $O$.

## 4 EMS Framework

This section designs and implements the EMS framework. In order to model it in detail, we divide the framework construction process into six phases: entering the exception management, protecting context, entering ESR, leaving ESR, restoring context, and exiting the exception management. They are executed sequentially. We use the functions defined in this section to describe the actions of the processor at each stage. At the end of each subsection, we use structured operational semantics to describe the state transition process corresponding to this phase.

### 4.1 Entering the Exception Management

Before the occurrence of the exception signal, the system is always in user mode to execute program instructions, and the processor will check whether there are pending exception requests before executing each

$$\begin{array}{lll}
(BuffWinReg) & B & ::= bins \mid bouts \\
(Binsreg) & bins & ::= bi0 \mid ... \mid ... \mid bi7 \\
(Boutsreg) & bouts & ::= bo0 \mid ... \mid ... \mid bo7
\end{array}$$

$$\begin{aligned}
save\_ios(R,B) \overset{\text{def}}{=\!=\!=} & R\{[i0,...,i7],[o0,...,o7]\} \rightsquigarrow \\
& B\{[bi1,...,bi7],[bo1,...,bo7]\}
\end{aligned}$$

$$\begin{aligned}
win\_left(R,B) \overset{\text{def}}{=\!=\!=} & \text{let}(R,B') \ ::= save\_ios(R,B) \text{ in} \\
& \text{let}(R',B') ::= R\{cwp+1 \rightsquigarrow cwp\} \text{ in} \\
& (B'\{bi1,...,bi7\} \rightsquigarrow R'\{o0,...,o7\})
\end{aligned}$$

$$\begin{aligned}
win\_right(R,B) \overset{\text{def}}{=\!=\!=} & \text{let}(R,B') \ ::= save\_ios(R,B) \text{ in} \\
& \text{let}(R',B') ::= R\{cwp-1 \rightsquigarrow cwp\} \text{ in} \\
& (B'\{bo1,...,bo7\} \rightsquigarrow R'\{i0,...,i7\})
\end{aligned}$$

Fig.6. Windows buffer model.

$$
\begin{array}{llll}
(MemStat) & M & ::= (V, H) \\
(MemVar) & V & ::= nest \mid curtcb \mid pretcb \mid swflag \mid osregl1 \mid osregtt \\
(StackOffset) & O & ::= \{0\ ,\ 4\ ,\ 8\ ,\ \dots\ ,\ 128\} \\
(Stack) & H & \in O \to sp,\ \text{where } sp = R(o6)
\end{array}
$$

$$
ne\_flag(V) \overset{\text{def}}{=\!=} \begin{cases} \text{true}, & \text{if } 1 < V[nest], \\ \text{false}, & \text{if } 0 \leqslant V[nest] \leqslant 1 \end{cases} \qquad sw\_flag(V) \overset{\text{def}}{=\!=} \begin{cases} \text{true}, & \text{if } V[swflag] \neq 0, \\ \text{false}, & \text{if } V[swflag] = 0 \end{cases}
$$

Fig.7. OS memory model.

instruction. The processor first checks whether the system allows exception events when it receives a request for it. If not allowed, the processor directly ignores the exception signal, resets the value of register $\tau$, and continues to execute the next instruction. If the system allows exceptions to occur, the state of itself is changed through the following actions:

• store the exception information in $l4$ and $tt$ (*res_exce*, see Fig.8); $w$ and $irl$ represent exception flag information and priority respectively;

• set the field $et$ to 0, and shield other exception events;

• set the field $su$ to 0 to make the system enter the supervision mode to execute instructions;

• turn the register wheel to the right;

• back up the contents of *psr, pc, npc* to $l0$, $l1$, $l2$ of the current sub-window respectively to prepare for the exit phase of the exception management.

We define all the above actions as the function *enter_exce* shown in Fig.8. After the state of the system is adjusted, the processor directs the control flow to jump to the address saved in the field *tba*. The OS kernel has stored the entry address of the exception management in the field *tba* during the system initialization process, and thus the control flow will jump into the exception management to continue executing instructions. We define the action of the control flow jump as the function *jmp_exce* shown in Fig.8.

We use (1) and (2) to describe the state transition process of entering the exception management phase. The formal notation "⇒" indicates logical implications. The corresponding transition diagram is shown in Fig.9. The system is initially in user mode to perform user tasks (*usr_mode*). When the processor detects an exception signal (*has_exce*), it first judges the value of the field *et*. Suppose *et* is 1 (*exce_enable*). In that case, the exception flag information and priority are stored in the corresponding register (*res_exce*). The processor adjusts the system state (*set_exce*) and finally guides the control flow into the exception management. If *et* is 0 (*exce_disable*), the system shields all exception signals, and resets the value of register $\tau$ (*clear_exce*). Finally, the processor continues to perform user tasks.

$$
\frac{\begin{array}{c} usr\_mod(R)\ \ exce\_enable(R)\ \ has\_exce(R) \\ res\_exce(w, irl, R) = R'\ \ set\_exce(R', B') = R'', B' \\ jmp\_exce(R'') = R''' \end{array}}{instr_{usr}(R, B) \Rightarrow exce(R''', B')}, \tag{1}
$$

$$
\frac{\begin{array}{c} usr\_mod(R)\ \ exce\_disable(R) \\ has\_exce(R)\ \ clear\_exce(R) = R' \end{array}}{instr_{usr}(R) \Rightarrow usr(R')}. \tag{2}
$$

## 4.2 Protecting Context

In the previous phase, the processor rotates the register window to the right, but does not check whether the window overflow occurs. Hence, the processor needs to check whether the adjacent register sub-window has been used before saving the context. The action of checking whether the sub_window overflows is defined as the function *winof_flag* shown in Fig.8, where *cwp* represents the register sub_window number currently in use and *wim* stores the sub_window number that has been used. If $cwp - 1$ is not equal to *wim*, it means that the next register sub-window has not been used yet, and we set *winof_flag* to false and save the task context. Otherwise, it means that the next register sub_window has been used, and we set *winof_flag* to true, and execute the window overflow processing action.

The window overflow handling action is defined as the function *instr_of* shown in Fig.8. The processor first turns the window to the right (*win_right*), and then saves the context data of the overflow sub-window. Because of the feature that adjacent windows share the data, the processor only needs to save the contents of the *local* and *ins* register groups. The stack top address of the overflow window is stored in the *sp* register, and the processor saves $l0$–$l7$, $i0$–$i7$ to the corresponding stack according to *sp* (*save_ofwin*, as shown in Fig.8, we use the formal notation "↦" to denote the data exchange action between registers and memory addresses). Next, the processor adjusts *wim* because the

$$res\_exce(w, irl, B) \overset{\text{def}}{=\!=} (w, irl) \rightsquigarrow R(l4, tt) \qquad jmp\_exce(w, irl, B) \overset{\text{def}}{=\!=} R\{(tba \rightsquigarrow pc), (tba+4 \rightsquigarrow npc)\}$$

$$winof\_flag(R) \overset{\text{def}}{=\!=} \begin{cases} \text{true} \\ \text{false} \end{cases} \text{let } R\{cwp \rightsquigarrow l5, wim \rightsquigarrow l4\} \text{ in} \begin{cases} \text{if } l5-1 = l4 \\ \text{if } l5-1 \neq l4 \end{cases}$$

$$save\_ofwin(R, H) \overset{\text{def}}{=\!=} R\{l0, ..., l7\} \mapsto H\{sp, [0, ..., 28]\}, R\{i0, ..., i7\} \mapsto H\{sp, [32, ..., 60]\}$$

$$save\_con(R, H) \overset{\text{def}}{=\!=} \text{let}(R', H') ::= R(fp - 128 \rightsquigarrow sp) \text{ in } save\_win$$

$$\begin{aligned} Nep(R, V) \overset{\text{def}}{=\!=} &\text{let}(R', V) ::= V[nest] \mapsto R(l5) \text{ in} \\ &\text{let}(R'', V) ::= R'(l5+1 \rightsquigarrow l5) \text{ in} \\ &R''(l5) \mapsto V[nest] \end{aligned} \qquad \begin{aligned} save\_win(R, H) \overset{\text{def}}{=\!=} &R\{l0, ..., l2\} \mapsto H\{sp, [0, ..., 28]\}, \\ &R\{i0, ..., i7\} \mapsto H\{sp, [32, ..., 60]\}, \\ &R\{g1, ..., g7\} \mapsto H\{sp, [64, ..., 88]\} \end{aligned}$$

$$instr\_of(R, B, H) \overset{\text{def}}{=\!=} \begin{cases} \text{let}(R', B') ::= win\_right(R, B) \text{ in} \\ \text{let}(R'', H') ::= save\_ofwin(R', H) \text{ in} \\ (R''\{wim \rightsquigarrow l4, l4-1 \rightsquigarrow l5, l5 \rightsquigarrow wim\}, \\ win\_left(R'', B'), \quad \text{if } winof\_flag = \text{true}, \\ \\ \perp, \qquad\qquad \text{if } winof\_flag = \text{false} \end{cases} \qquad \begin{aligned} &enter\_exce(R, B) \\ \overset{\text{def}}{=\!=} \; &\text{let}(S') ::= disable\_exce(S) \text{ in} \\ &\text{let}(S'') ::= to\_sub(S') \text{ in} \\ &\text{let}(S''', W', B') ::= wind\_right(S'', W, B) \text{ in} \\ &S'''(psr, pc, npc) \rightsquigarrow W'(l0, l1, l2) \\ &\text{where } R = (S, W) \end{aligned}$$

Fig.8. Functions in phase 1 and phase 2.

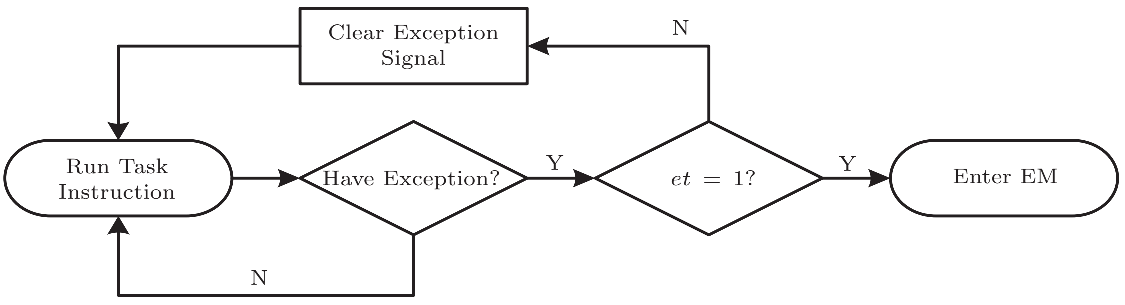

Fig.9. State transition of phase 1. EM stands for the control flow of exception management.

context content of the window has been successfully saved. After that, the processor deletes the window number in *wim*, which means this window can be used. Finally, the processor turns the register window to the left ($win\_left$) and returns to the initial state of the previous stage.

The context saving action is defined as the function *sav_con*. The processor creates a 128-byte space for the register sub-window and uses the current task stack to save the context data. The registers that need to be saved are $l0$–$l2$, $i0$–$i7$, $g1$–$g7$, and the other remaining unused registers do not need to be saved. Nesting layers of exception events are represented by *nest*, and it is always 0 when the system is under user mode. The number of nesting layers will increase with the times that the control flow enters the exception management. We define the above action as a function $Nep$, and we will describe the modeling of nested exceptions in detail in the next stage.

We use (3) and (4) to describe the state transition process in the context protection phase. The corresponding transition diagram is shown in Fig.10.

$$\frac{\begin{array}{c} winof\_flag(R) = \text{true} \; instr\_of(R, B, H) = (R', B', H') \\ save\_con(R', H') = (R'', H'') \; Nep(R'', V) = (R''', V') \end{array}}{instr_{exce}(R, B, V, H) \Rightarrow (R''', B', H'', V')}, \tag{3}$$

$$\frac{\begin{array}{c} winof\_flag(R) = \text{false} \;\; instr\_of(R, B, H) = \perp \\ save\_con(R, H) = (R', H') \;\; Nep(R', V) = (R'', V') \end{array}}{instr_{exce}(R, B, V, H) \Rightarrow (R'', B', H'', V').} \tag{4}$$

The processor first judges whether the adjacent sub-window has been used. If $winof\_flag$ returns true, the processor executes the function $instr\_of$, turns the register window to the right, and saves the sub-windows

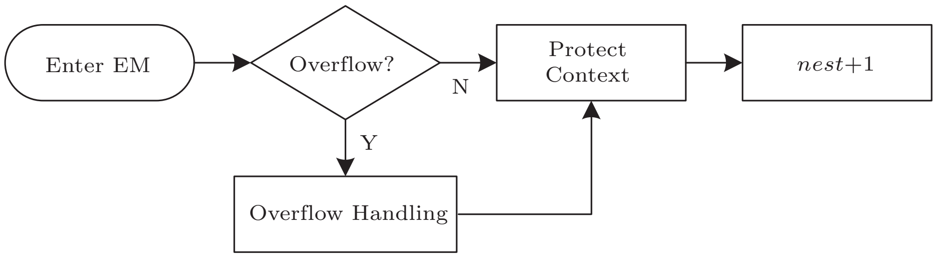

Fig.10. State transition of phase 2. EM stands for the control flow of exception management.

general register content to the corresponding stack. After that, it adjusts the value of *wim*, returns the sub-window position, saves the context data, and finally increases the number of exception nesting layers by 1. If $winof\_flag$ returns false, the processor skips the window overflow handling function and executes subsequent actions. It is worth noting that the field *et* has not changed in this phase, and the system is still in an exception shielding state.

## 4.3 Entering ESR

After the task context is saved, the control flow enters the ESR jump stage, which is mainly to make some preparations for task switching and exception nesting in the exception management. The real-time OS allows task start and task delete instructions in ESR, which may cause the control flow to enter a new task when exiting the exception management, as shown in Fig.3. The OS kernel switches tasks by changing the current task control block, and *curtcb* represents task $I$ interrupted by the exception event. If ESR causes a task switching, the system will assign the new task $H$ to *curtcb*. Therefore the processor needs to back up task $I$ at this phase, store *curtcb* in *pretcb*, and store *sp* in the position $i6$ of the task control block at the same time. When task $I$ is resumed, the processor can find the context data saved in the task stack from $i6$. The above action is defined as the function *sw_tcb* shown in Fig.11. It is worth noting that the prerequisite for performing task switching is that the system does not have nested exceptions.

On the other hand, during the execution of the ESR program, if a higher priority exception event occurs, the system has nested exceptions, and the control flow will enter the exception management again, as shown in Fig.2. When the processor responds to exception events for the first time, it only needs to determine whether *et* is 0. If the system wants to determine whether exception nesting occurs, it also needs to compare the priority of the exception event with the system's priority threshold. The function *set_pil* means to set the priority threshold of the system according to the field *tt*. If *tt* is between 1 and 15, it means that the current exception is synchronous. Therefore, the processor changes the value of *pil* to $l6$, and shields all asynchronous excep-

$$sw\_tcb(R,V) \overset{\text{def}}{=\!=\!=} \begin{cases} \text{let} \quad (R',V') ::= V[curtcb] \mapsto R(16), R(l6) \mapsto V[pretcb] \ \text{ in} \\ \qquad\qquad\qquad R'(sp) \mapsto V'[pretcb+56], \\ \qquad\qquad\qquad \text{if } \ ne\_flag = \text{true}, \\ \perp, \qquad\qquad\quad \text{if } \ ne\_flag = \text{false} \end{cases}$$

$$jmp\_esr(R) \overset{\text{def}}{=\!=\!=} R\{npc \rightsquigarrow o7, l6 \rightsquigarrow pc\} \qquad jmp\_back(R) \overset{\text{def}}{=\!=\!=} R\{o7 \rightsquigarrow pc\}$$

$$set\_pil(R) \overset{\text{def}}{=\!=\!=} \begin{cases} R\{l6 \rightsquigarrow l4, l4 \rightsquigarrow pil\}, & \text{if } \ 1 \leqslant R(tt) \leqslant 15, \\ R\{tt \rightsquigarrow l5, l5 \rightsquigarrow pil\}, & \text{if } \ 16 \leqslant R(tt) \leqslant 31 \end{cases}$$

$$bac\_exce(R,V) \overset{\text{def}}{=\!=\!=} R(l1) \mapsto V[osregl1], R(tt) \mapsto V[osregtt]$$

$$find\_esr(R,V) \overset{\text{def}}{=\!=\!=} let(R',V) ::= V(ossm) \rightsquigarrow R(l4) \ \text{ in } \ R(l4+l3 \rightsquigarrow l6)$$

$$esr\_exce(w,irl,R) \overset{\text{def}}{=\!=\!=} \begin{cases} (w,isr) \rightsquigarrow R(l3,tt), & \text{if } \ exce\_enable(R) \wedge \\ & sup\_mode(R) \wedge (irl < pil), \\ \perp, & \text{otherwise} \end{cases}$$

Fig.11. Functions in phase 3 and phase 4.

tions. Otherwise, it means that the current exception is asynchronous, and the process passes *tt* into the field *pil*. In addition, some information about the exception needs to be recorded in the corresponding global variable (*bac_exce*, see Fig.11), because the exception handling process is invisible to the user and can only be analyzed through the information recorded by the global variable.

We define the state transition of the preparation work before entering ESR as (5), (6), and (7). The corresponding transition diagram is shown in Fig.12.

First, the processor judges the number of exception nesting layers if the system has nested exceptions, and backs up the control block address of the original task (*sw_tcb*). Otherwise, the processor sets the status of the register *psr* according to the exception attribute information (*set_pil*) and then stores some important attribute information in corresponding global variables (*bac_exce*). Finally, the processor sets *et* to 1, making the system able to respond to new exception events.

$$\frac{\begin{array}{c} ne\_flag = \text{true} \;\; sw\_tcb(R,V) = (R',V') \\ set\_pil(R') = (R'') \;\; bac\_exce(R'',V') = (R''',V'') \\ enable\_exce(et) = (1 \rightsquigarrow et) \end{array}}{instr_{exce}(R,V,et=0) \Rightarrow (R''',V'',et=1)}, \tag{5}$$

$$\frac{\begin{array}{c} ne\_flag = \text{false} \;\; sw\_tcb(R,V) = \perp \\ set\_pil(R) = (R') \;\; bac\_exce(R',V') = (R'',V') \\ enable\_exce(et) = (1 \rightsquigarrow et) \end{array}}{instr_{exce}(R,V,et=0) \Rightarrow (R'',V',et=1)}. \tag{6}$$

The OS kernel stores the ESR entry address in the *ossm* table during initialization. We define the operation of finding the entry address as *find_esr* shown in Fig.11. The processor has stored the unique identification information of the exception in the register *l*3 in the first phase. Therefore, the entry address of the ESR corresponding to the exception can be found through *l*3, and then the processor stores the address in the register *l*6. The action to enter the ESR program is defined as function *jmp_esr* shown in Fig.11. Different from *jmp_exce*, the processor sets the current *npc* as the return address. The control flow can jump back to the exception management through this address when the ESR program has finished running.

$$\frac{find\_esr(R,V) = (R',M) \;\; jmp\_esr(R') = (R'')}{instr_{exce}(R,V,et=0) \Rightarrow esr(R',V,et=1)}. \tag{7}$$

## 4.4 Leaving ESR

In order to facilitate the modeling of the exception management, we assume that the initial state of the system is in user mode. The design paradigm of exception management requires that exceptions of the same priority cannot be nested. For exception requests from different sources, nesting can improve the system's ability to respond to critical events. In this phase, the field *et* has been set to 1, and the system can respond to exceptions typically. However, because the system is in supervision mode, it is also necessary to determine whether to respond to exception events according to the field *pil*. The function *esr_exce* shown in Fig.11 indicates that if the priority of the new exception event is higher than *pil*, the system will store the exception's flag information and priority in *l*3 and *tt* respectively. Otherwise, *esr_exce* returns a null state.

The exception management is responsible for guiding the control flow into the correct ESR execution entry, and does not involve the execution process of the ESR program. Because most systems allow users to edit the ESR program, we assume the ESR program is correct in this paper. The control flow will return to the exception management program according to the address stored in *o*7, when the ESR program ends. It is worth noting that not all ESRs will return to the exception management, such as instruction errors ESR, and memory address misalignment ESR will cause the system to reset. At this time, there is no need to perform the subsequent stages, but this does not affect our modeling and verification of the exception management.

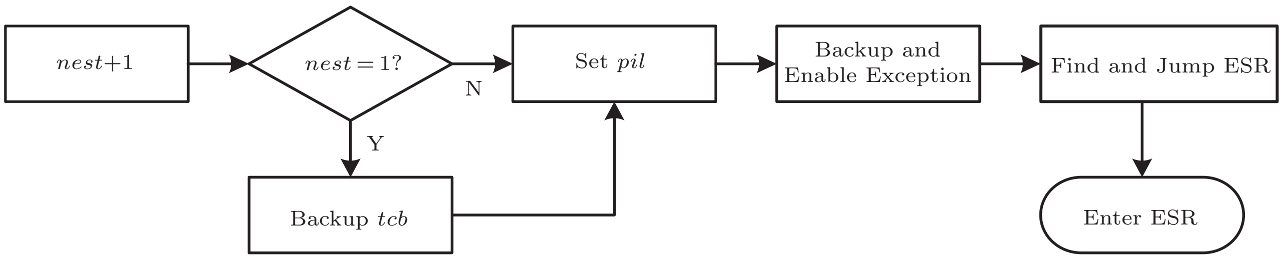

Fig.12. State transition of phase 3.

We use (8) and (9) to illustrate the state transition at this stage. The corresponding transition diagram is shown in Fig.13.

$$\frac{\begin{array}{c} \mathit{sup\_mode}(R)\ \mathit{exce\_enable}(R)\ \mathit{has\_exce}(R) \\ \mathit{esr\_exce}(w, \mathit{irl}, R) = (R')\ \mathit{set\_exce}(R', B) = (R'', B') \\ \mathit{jmp\_exce}(R'') = (R''') \end{array}}{\mathit{instr}_{esr}(R, B) \Rightarrow \mathit{exce}_{new}(R''', B')}, \tag{8}$$

$$\frac{\begin{array}{c} \mathit{sup\_mode}(R)\ \ \mathit{exce\_enable}(R)\ \ \mathit{has\_exce}(R) \\ \mathit{esr\_exce}(w, \mathit{irl}, R) = \perp\ \ \mathit{clear\_exce}(R) = (R') \\ \mathit{jmp\_back}(R') = (R'') \end{array}}{\mathit{instr}_{esr}(R) \Rightarrow \mathit{exce}(R'')}. \tag{9}$$

If a new exception event occurs during the operation of the ESR program, the processor will compare its priority *irl* with *pil*. If the event meets the requirements, the processor will store its related information in the corresponding register and interrupt the execution of the current ESR program. Then the processor sets the state of the system (*enter_exce*), and finally the control flow enters the first phase of the exception management again to execute the relevant instructions (*jmp_exce*). The context saved in the exception management is the data of the current register window being used by the previous exception. In fact, exception nesting is an iterative process. If *irl* is less than *pil*, *esr_exce* will return an empty state and then clear the exception existence flag. Finally, the ESR program will continue to execute and return to the exception management (*jmp_back*).

### 4.5 Restoring Context

After the ESR program ends, the control flow returns to the exception management. In this phase, the processor also needs to deal with the task switching to recover the context data. The context recovery function *resto_win* is the left inverse function of the context saving function *save_win*, which means restoring the data saved in the stack to the corresponding register.

$$\begin{aligned} & \mathit{resto\_win}(R, H) \overset{\text{def}}{=\!=} \\ & H\{sp, [0, ..., 28]\} \mapsto R\{l0, ..., l2\}, \\ & H\{sp, [32, ..., 60]\} \mapsto R\{i0, ..., i7\}, \\ & H\{sp, [64, ..., 88]\} \mapsto R\{g1, ..., g7\}. \end{aligned}$$

Before data recovery, the processor first needs to determine if the system exists nested exceptions. If the system survives nested exceptions, the processor will restore the last exception context data. Otherwise, the processor needs to determine whether it needs to switch tasks according to *swflag*. If ESR does not involve instructions OSTaskCreate and OSTaskDelete, *swflag* will be set to 0, which means no task switching, and the processor will restore the context data of the original task. If a task switching occurs, the processor needs to save all the register sub-windows used by the original task. Supposing that *cwp* points to sub-window $w_3$ and *wim* points to sub-window $w_7$, it means that the sub-windows used by the original task are $w_4$, $w_5$, and $w_6$. In phase 2, function *sav_win* has saved the context data of the sub-window $w_3$ to the corresponding stack. Therefore, the processor only needs to store the contents of the remaining three register windows in the corresponding locations.

Function *save_sw* (see Fig.14) means to save the data of all register sub-windows for the original task. If the state of *wim_cwp* (see Fig.14) is true, the processor rotates the window to the left, and then saves the *local*, *outs* and *ins* register groups of current sub-window. The data of the global register group is already saved to the stack, and the system can use the $g0$–$g7$ registers for data calculation; if the status of *wim_cwp* is false, it means that all the register sub-windows used by the original task have been saved, and a null value is returned. During the execution of ESR, the entry address of the new task is stored in *cutrcb*, which is why we need to back up the original task control block

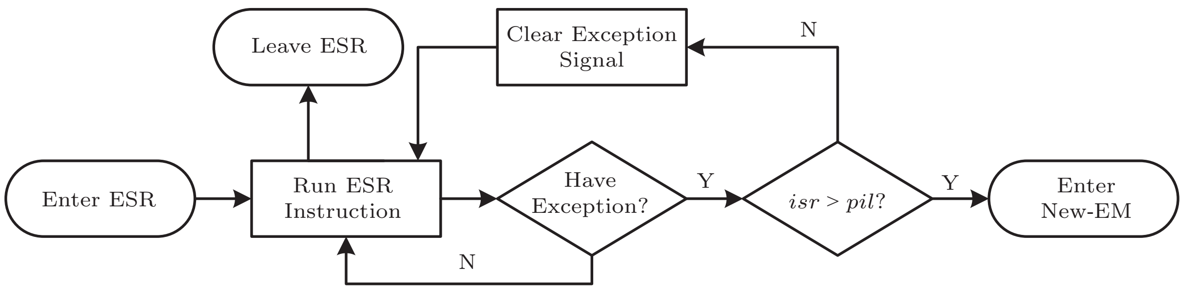

Fig.13. State transition of phase 4. EM stands for the control flow of exception management.

$$wim\_cwp(R) \overset{\text{def}}{=\!=} \begin{cases} \text{true} \\ \text{false} \end{cases} \text{let } R\{cwp \rightsquigarrow g4, wim \rightsquigarrow g7\} \text{ in} \begin{cases} \text{if } g4+1=g7 \\ \text{if } l5+1 \neq g7 \end{cases}$$

$$winuf\_flag(R) \overset{\text{def}}{=\!=} \begin{cases} \text{true} \\ \text{false} \end{cases} \text{let } R\{cwp \rightsquigarrow o5, wim \rightsquigarrow o4\} \text{ in} \begin{cases} \text{if } o5+1=o4 \\ \text{if } o5+1 \neq o4 \end{cases}$$

$$instr\_uf(R,B,H) \overset{\text{def}}{=\!=} \begin{cases} \text{let}(R',B') ::= win\_left(R,B) \text{ in} \\ \text{let}(R'',H') ::= save\_ufwin(R',H) \text{ in} \\ R''\{wim \rightsquigarrow o4, o4-1 \rightsquigarrow o5, o5 \rightsquigarrow wim\}, \\ win\_right(R'',B') \ \text{ if } \ winuf\_flag = \text{true}, \\ \\ \bot, \qquad\qquad \text{if } \ winuf\_flag = \text{false} \end{cases}$$

$$save\_sw(R,B,H) \overset{\text{def}}{=\!=} \begin{cases} \text{let}(R',B') ::= win\_left(R,B) \text{ in } save\_ofwin(R',H), \\ \qquad\qquad \text{if } \ wim\_cwp = \text{true}, \\ \bot, \qquad\quad \text{if } \ wim\_cwp = \text{false} \end{cases}$$

$$\begin{array}{ll} & quit\_exce(R,V) \\ \overset{\text{def}}{=\!=} & \text{let}(S',W) ::= W\{l0,l1,l2\} \rightsquigarrow S(psr,pc,npc) \ \text{ in} \\ & \text{let}(S'') ::= enable\_exce(S) \ \text{ in } \ wind\_left(S'',W,B) \\ & \text{where } \ R=(S,W) \end{array} \qquad \begin{array}{ll} & Nem(R,V) \\ \overset{\text{def}}{=\!=} & \text{let}(R',V) ::= V[nest] \mapsto R(l4) \ \text{ in} \\ & \text{let}(R'',V) ::= R'(l4-1 \rightsquigarrow l4) \ \text{ in} \\ & R''(l4) \mapsto V[nest] \end{array}$$

$$resto\_ufwin(R) \overset{\text{def}}{=\!=} H\{sp,[0,...,28]\} \mapsto R\{l0,...,l7\}, H\{sp,[32,...,60]\} \mapsto R\{i0,...,i7\}$$

$$resto\_sp(R) \overset{\text{def}}{=\!=} V[curtcb+56] \mapsto R(sp)$$

Fig.14. Functions in phase 5 and phase 6.

during the ESR entering phase (*sw_tcb*). The design paradigm of the SpaceOS task start function requires the system to find the task context from the position $i6$ of the task control block. If the processor needs to restore the context data of the new task, it must pass the content of the control block $i6$ to $sp$ (*resto_sp*, see Fig.14), and then execute the function *resto_con* to restore the context data of the new task.

We use (10), (11), and (12) to express the state transition process at this phase. The corresponding transition diagram is shown in Fig.15. First, the processor shields all exception signals to protect data security, and then perform exception nesting judgment. If *nest* is not 1, the processor restores the previous exception context data. Otherwise, it continues to judge whether task switching has occurred in the ESR. If the task switching has not occurred in the ESR, the processor restores the context data of the original task. Otherwise, it saves all register windows used by the original task and then saves the address of the new task into the register *sp*. Finally, it restores the context data of the new task to the corresponding registers.

$$\frac{\begin{array}{c} disable\_exce(et) = (0 \rightsquigarrow et) \\ ne\_flag(V) = \text{false} \ \ resto\_con(R,H) = (R',H) \end{array}}{instr_{exce}(R,V,H,et=1) \Rightarrow (R',V,H,et=0)}, \tag{10}$$

$$\frac{\begin{array}{c} disable\_exce(et) = (0 \rightsquigarrow et) \ \ ne\_flag(V) = \text{true} \\ sw\_flag(V) = \text{false} \ \ resto\_con(R,H) = (R',H) \end{array}}{instr_{exce}(R,V,H,et=1) \Rightarrow (R',V,H,et=0)}, \tag{11}$$

$$\frac{\begin{array}{c} disable\_exce(et) = (0 \rightsquigarrow et) \ \ ne\_flag(V) = \text{true} \\ sw\_flag(V) = \text{true} \ \ save\_sw(R,B,H) = (R',B',H') \\ resto\_sp(R',V) = (R'',V) \\ resto\_con(R'',H') = (R''',H'') \end{array}}{instr_{exce}(R,B,V,H,et=1) \Rightarrow (R''',B',V,H'',et=0)}. \tag{12}$$

## 4.6 Exiting the Exception Management

Exiting is the last phase of the exception management, and the control flow is about to jump out of the exception management. In phase 1, the processor rotates the register window to the right. Therefore, the processor needs to restore the position of the windows in this phase. The return status of the function *winuf_flag* shown in Fig.14 indicates whether a window underflow occurs. The processor stores the values of the *cwp* and *wim* fields in the $o5$ and $o4$ registers respectively. If $cwp+1$ is equal to *wim*, the return status is true; otherwise, the return status is false. The function *instr_uf* (see Fig.14) represents window underflow processing. First, the processor rotates the window to the left, then saves the data in the stack to the corresponding register, and adjusts the value of *wim*. Finally it restores the window position.

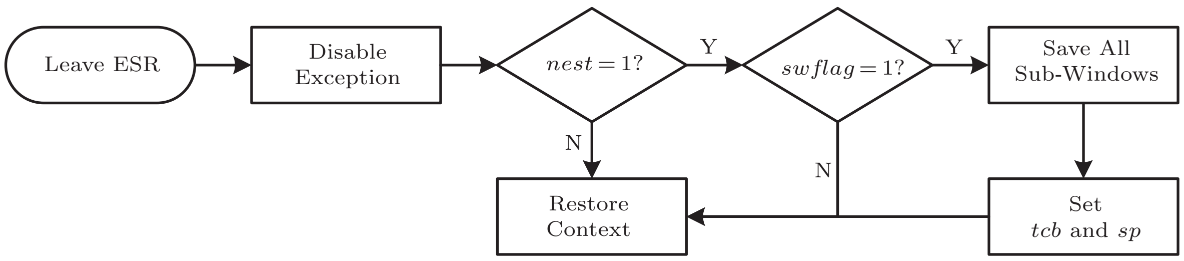

Fig.15. State transition of phase 5.

Each time when the processor responds to an exception event, *nest* increases by 1. After the ESR program is executed, nesting layers need to be readjusted. This action is defined as the function $Nem$. In the first stage, the processor has adjusted the system state in response to exception events. When exiting the exception, the processor needs to restore the state of the system. In the function *enter_exce*, the processor makes a backup of the tables *psr*, *pc*, and *npc*. Now the processor can restore the state before the system enters the exception management through $l0$, and then store the contents of $l1$, $l2$ in *pc, npc* respectively, and finally direct the system control flow out of the exception management represented by the function *quit_exce*.

We use (13) and (14) to express the state transition process at this phase. The corresponding transition diagram is shown in Fig.16.

$$\frac{\begin{array}{c} winuf\_flag(R)=\text{true}\ \ instr\_uf(R,B,H)=(R',B',H') \\ Nem(R',V)=(R'',V)\ \ enable\_exce(et)=(1\rightsquigarrow et) \\ quit\_exce(R'',B')=(R''',B'') \end{array}}{instr_{exce}(R,B,V,H,et=0)\Rightarrow usr(R''',B'',V,H',et=1)}, \tag{13}$$

$$\frac{\begin{array}{c} winuf\_flag(R)=\text{false}\ \ instr\_uf(R,B,H)=\perp \\ Nem(R,V)=(R',V)\ \ enable\_exce(et)=(1\rightsquigarrow et) \\ quit\_exce(R',B)=(R'',B') \end{array}}{instr_{exce}(R,B,V,H,et=0)\Rightarrow usr(R'',B',V,H',et=1)}. \tag{14}$$

The processor first judges whether a window underflow occurs. If the status of *winuf_flag* is true, the process executes the window underflow processing function *intsr_uf*. Otherwise, it subtracts the number of nesting levels ($Nem$), and restores the system state when the control flow enters the exception management. It is worth noting that if exception nesting occurs, the system is still in privileged mode after executing the function *quit_exce*. If it does not happen, the processor will switch the system state to user mode and finally set the field *et* to 1, and thus the system can respond to exception events that meet the requirements.

## 5 Practical Verification

In this section, we use the EMS framework to verify the exception management of SpaceOS using the SPARCv8 processor. We describe the verification process through the two examples below. Fig.17 shows the mission operation process of the Beidou-3 central computer, with the red line segments indicating the flow direction of the system control flow. The horizontal coordinate indicates the control cycle time, and the vertical coordinate indicates the priority level. The blue part indicates the control task, the green part indicates the ESR procedure, and the green shaded part indicates the exception management procedure.

Example 1 shows the control flow direction of basic exceptions, and task 1 is interrupted by an exception event E-A during runtime. The processor suspends task 1 and guides the control flow to enter the exception management, then waits for the execution of the corre-

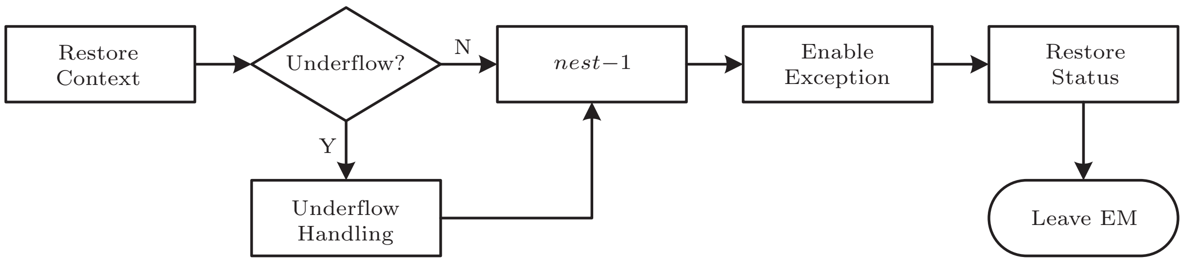

Fig.16. State transition of phase 6. EM means the control flow of exception management.

sponding ESR program, and finally restores the context of task 1. Example 2 shows the control flow direction of nested exceptions. Task 1 is interrupted by the exception event E-B, and the processor executes the corresponding ESR program. High priority exception event E-C interrupts the executing of this ESR program, and thus the processor guides the control flow to enter the exception management again.

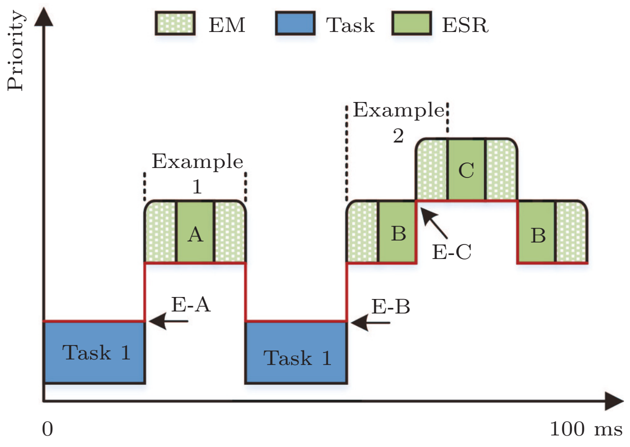

Fig.17. Beidou-3 mission operation illustration.

Program logic is used to describe and demonstrate the behavior of the program. Program and logic have an essential connection. Let us suppose the program is regarded as an execution process. In that case, the basic method of the program logic is first to establish the connection between the program and the logic, then establish the program logic system, and finally study various properties of the program in this system. Here we give the mathematical specification $\{P\}S\{Q\}$ for the logic of the exception management program.

For the convenience of explanation, we use $\mathbb{S}_1$–$\mathbb{S}_{14}$ to represent (1)–(14) in the following content. $S = \{\mathbb{S}_1, ..., \mathbb{S}_{14}\}$ stands for program logic, which is composed of the 14 transition formulas described in Section 4. $P$ and $Q$ are logical expressions related to program variables. $\{P\}$ is called the pre-condition, which indicates the nature of the program variable before executing the program logic, and can be regarded as a prerequisite for the correct execution of the program. $\{Q\}$ is called the post-condition, which describes the state of the system when the statement ends, and can be regarded as the logical result that S shall realize.

*Example* 1 (*Verify Basic Exception*). We take the processing of exception event E-A as the first example, and its control flow is shown in Fig.17. We first give its pre-conditions, that is, the state of the OS before the control flow enters the exception manager. This example focuses on register fields that reflect certain system states, window registers, and memory variables that represent flag information.

$$basic\_statu\_bef \overset{\text{def}}{=\!=} SRF_{\text{b}} \wedge MV_{\text{b}} \wedge WR_{\text{b}}.$$

$SRF_{\text{b}} \overset{\text{def}}{=\!=} \{(su = 0) \wedge (et = 1) \wedge (\tau = 1) \wedge (cwp = 7) \wedge (wim = 4) \wedge (tbr = \text{A}) \wedge (pc = \text{a}) \wedge (npc = \text{b})\}$.

SRF (special register field) reflects the system's state; $su = 0$ means the system is in user mode at this time; $et = 1$ means the system can respond to the exception; $\tau = 1$ means there is an exception event waiting for a response from the processor; $cwp$ and $wim$ indicate register sub-windows used by task 1; $pc$ and $npc$ record the current instruction address "a" and the address of the next instruction "b" respectively; $tbr$ means the type of exception event.

$MV_{\text{b}} \overset{\text{def}}{=\!=} \{(swflag = 0) \wedge (nest = 0) \wedge (osregtt = 0)\}$.

MV (memory variables) indicates the flag information; $swflag = 0$ means the exception event does not lead to task switching, thereby there is no need to consider the backup of $curtcb$; $nest = 0$ means that no nested exceptions have occurred in the system yet. The memory variable $osregtt$ is used to record the exception type, and its value is zero in the initial state.

$WR_{\text{b}} \overset{\text{def}}{=\!=} \{(cwp - 1) \otimes (ins, local, global)\}$.

The symbol "$\otimes$" indicates which register types are used in the current window. WR (window registers) indicates the usage of windows registers. The exception event E-A does not trigger the task switching, thereby the register wheel only needs to rotate one frame to the right and then save the data of this sub-window into the corresponding stack.

We illustrate the data flow changes in the basic exception (example 1 in Fig.17) handler with Fig.18. Fig.18(a) shows the initial state of the system. In Fig.18(b), the processor responds to the exception event E-A and the sub-window pointed by $cwp$ is turned one frame to the right, but this sub-window overlaps with the window pointed by $wim$, thereby it is necessary to save all registers of window $wim$ into the corresponding stack, and then the value of $wim$ is subtracted by 1. In Fig.18(c), the contents of all registers in sub-window 6 are saved to the corresponding memory stack; $sp$ and $fp$ are equivalent to $o6$ and $i6$ of this sub-window. Then the processor starts to execute ESR A program. In EMS, we assume that all ESRs correct, thereby the execution process of ESR A is not shown here. In

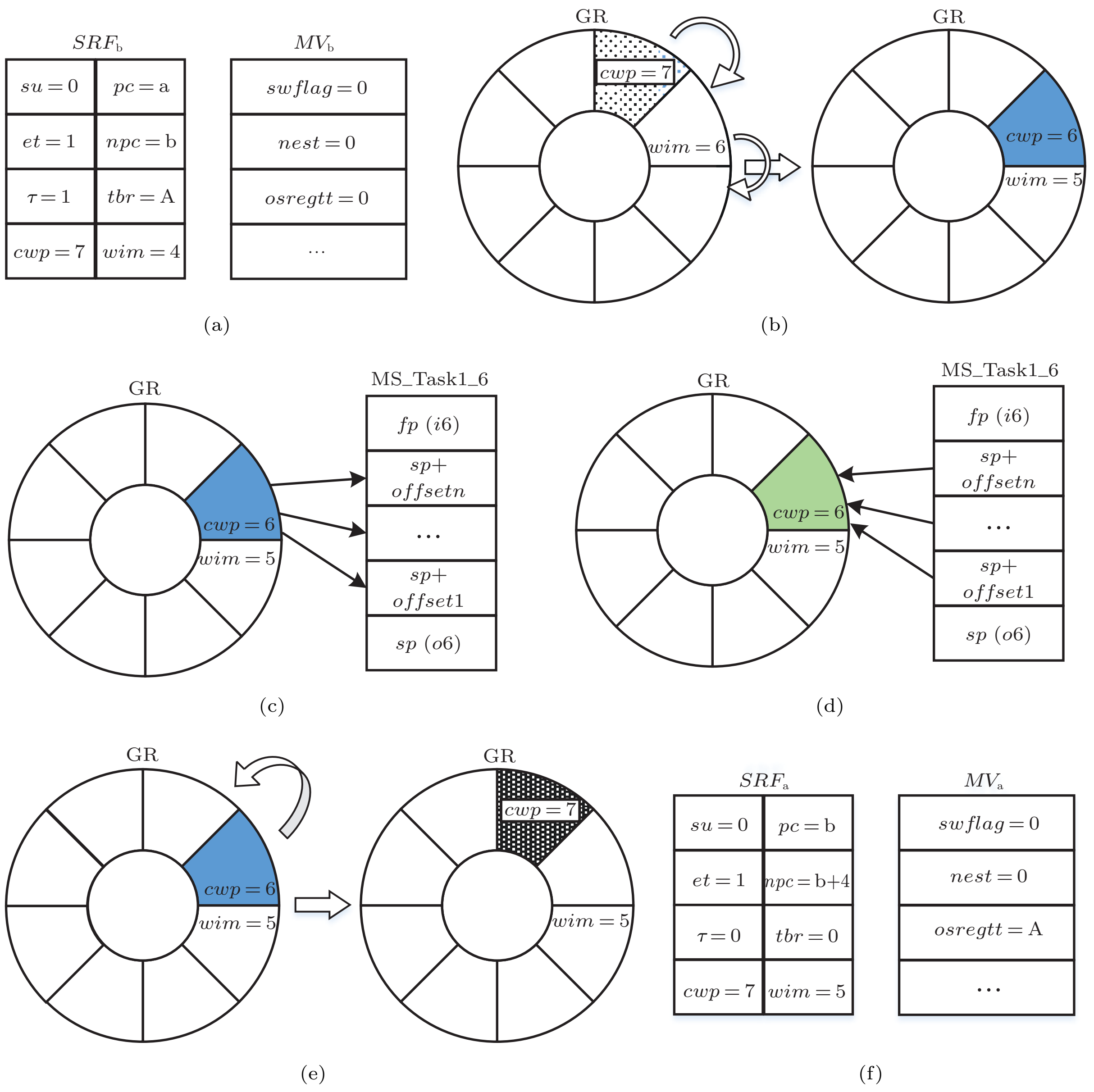

Fig.18. Migration process of data flow of basic exception. (a) Initial state. (b) Entering the exception management. (c) Protecting data. (d) Restoring data. (e) Leaving the exception management. (f) Termination state.

Fig.18(d), the ESR A program returns to the exception management after the execution is complete. The green sub-window 6 indicates that the contents of the *ins*, *local*, and *global* register groups have been changed, and the processor needs to restore the stack data to the corresponding registers. In Fig.18(e), the window rotation is rotated one frame to the left, and the value of *cwp* is restored. It is worth noting that the value of *wim* is not restored in the exception management. After the control flow exits the exception management in Fig.18(f), we show the status and define post-conditions as *basic_statu_aft*.

$$basic\_statu\_aft \overset{\text{def}}{=\!=} SRF_{\text{a}} \wedge MV_{\text{a}} \wedge WR_{\text{a}}.$$

$SRF_{\text{a}} \overset{\text{def}}{=\!=} \{(su = 0) \wedge (et = 1) \wedge (\tau = 0) \wedge (cwp = 7) \wedge (wim = 5) \wedge (tbr = 0) \wedge (pc = \text{b}) \wedge (npc = \text{b} + 4)\}$.

$\tau = 0$ indicates that there is no exception event in the system; the value of *cwp* is restored to 7; *tbr* is reset to 0; *pc* points to the address of *npc*; *npc* points to the address of the next instruction.

$MV_{\text{a}} \overset{\text{def}}{=\!=} \{(swflag = 0) \wedge (nest = 0) \wedge (osregtt = \text{b})\}$.

The memory variable *osregtt* records the type of the exception, which is the value "A" of *tbr*.

$WR_{\mathrm{a}} \overset{\mathrm{def}}{=\!=} \{(cwp-1)\otimes(ins, local, global)\}$.

During the execution of the program, the contents of the window registers are first saved to the memory stack and later restored from the stack, that is, the data of the window registers remain the same at the initial and termination states of the basic exception.

After giving the specification of the basic exception handler, we verify the handler's correctness by proving that it can be safely executed under the given pre-conditions. As shown in Theorem 1, it says, if the initial state satisfies the pre-condition, then we can safely reach a resulting state satisfying the post-conditions within $S_{\mathrm{be}}$ that consists of the formal formula $\{\mathbb{S}_1, \cdots, \mathbb{S}_{14}\}$. Here, we use $S_{\mathrm{be}}$ to represents the code heap of the basic exception handler.

**Theorem 1** (Correctness of the Basic Exception). *If $P$ := $basic\_statu\_bef$, and $Q$ := $basic\_statu\_aft$, then $P \overset{S_{\mathrm{be}}}{\Longrightarrow} Q$.*

*Example* 2 (*Verify Nested Exceptions*). We take the processing of exception events E-B and E-C as the second example, and the control flow is shown in Fig.17. Similar to the verification of basic exceptions, we first give the pre-conditions for nested exceptions:

$$nest\_statu\_bef \overset{\mathrm{def}}{=\!=} NSRF_{\mathrm{b}} \wedge NMV_{\mathrm{b}} \wedge NWR_{\mathrm{b}}.$$

$NSRF_{\mathrm{b}} \overset{\mathrm{def}}{=\!=} \{(su=0)\wedge(et=1)\wedge(\tau=1)\wedge(cwp=7)\wedge(wim=2)\wedge(tbr=\mathrm{B})\}$.

$su=0$ means the system is in user state; $et=1$ means the system can respond to exception events; $\tau=1$ means the system has exception events waiting for a response by the processor; *cwp* points to the sub_window that task 1 is using at this time; $wim=2$ means part of task 1's data is also stored in sub-window 0 and sub-window 7.

$NMV_{\mathrm{b}} \overset{\mathrm{def}}{=\!=} \{(swflag=0)\wedge(nest=0)\wedge(osregtt=\mathrm{A})\}$.

This example does not involve a task switching, thereby *swflag* is 0. The exception event has not yet occurred, the nesting count is 0, and *osregtt* still records the type of exception event E-A that occurred last time.

$NWR_{\mathrm{b}} \overset{\mathrm{def}}{=\!=} \{(cwp-1)\otimes(ins, local, global)\}$.

The content to be saved is the data in *ins*, *local*, and *global* registers of window 6.

We illustrate the change process for nested exceptions (example 2 in Fig.17) data streams in Fig.19. Fig.19(a) represents the initial state of the system, which is the pre-condition. Figs.19(b) and 19(c) are the same as the basic exception. Fig.19(d) indicates that ESR B is interrupted by a higher priority exception event E-C. The processor suspends ESR B and subsequently enters the nested exceptions. The window register wheel is rotated to the right, *cwp* points to window 5, and the processor saves this window's register data to the corresponding memory stack. It shall be noted that both *fp* and *sp* of the memory stack in Fig.19(c) belong to sub_window 6, while *fp* and *sp* of the memory stack in Fig.19(f) belong to window 5. Finally, Fig.19(g) shows the post-conditions:

$$nest\_statu\_aft \overset{\mathrm{def}}{=\!=} NSRF_{\mathrm{a}} \wedge NMV_{\mathrm{a}} \wedge NWR_{\mathrm{a}}.$$

$NSRF_{\mathrm{a}} \overset{\mathrm{def}}{=\!=} \{(su=1)\wedge(et=0)\wedge(\tau=0)\wedge(cwp=5)\wedge(wim=2)\wedge(tbr=\mathrm{C})\}$.

The control flow has not yet jumped from the exception manager to the ESR C program, and thus the system is in a privileged state at this point: *su* has changed from 0 to 1. So as to prevent data loss, the system blocks all exception events, *tbr* holds the flag information for the exception event E-C; *cwp* points to window 5; *wim* has not changed.

$NMV_{\mathrm{a}} \overset{\mathrm{def}}{=\!=} \{(swflag=0)\wedge(nest=2)\wedge(osregtt=\mathrm{B})\}$.

When the exception event E-B occurs, the number of nested levels is 1. Meanwhile, exception event E-C preempts the processor, the number of nested levels is added by 1 again, and thus *nest* is 2. The flag information for exception event E-B is recorded in *osregtt*.

$NWR_{\mathrm{a}} \overset{\mathrm{def}}{=\!=} \{(cwp=6)\otimes(ins, local, global), (cwp=5)\otimes(ins, local, global)\}$.

During execution handling, the processor saves the data of sub_windows 6 and 5 into the corresponding memory stack.

We use $S_{\mathrm{ne}}$ to represent the code heap of nested exceptions. Unlike example 1, the code heap of example 2 contains phases 1–4 involved in exception event E-A and phases 1–3 involved in exception event E-B. In other words, $S_{\mathrm{ne}}$ consists of $\{\mathbb{S}_1, \cdots, \mathbb{S}_9\}$ of E-A and $\{\mathbb{S}_1, \cdots, \mathbb{S}_7\}$ of E-B. As Theorem 2 says, if the initial state satisfies the pre-condition, then we can safely reach a resulting state satisfying the post-conditions within $S_{\mathrm{ne}}$.

**Theorem 2** (Correctness of the Nested Exceptions). *If $P$ := $nest\_statu\_bef$, and $Q$ := $nest\_statu\_aft$, then $P \overset{S_{\mathrm{ne}}}{\Longrightarrow} Q$.*

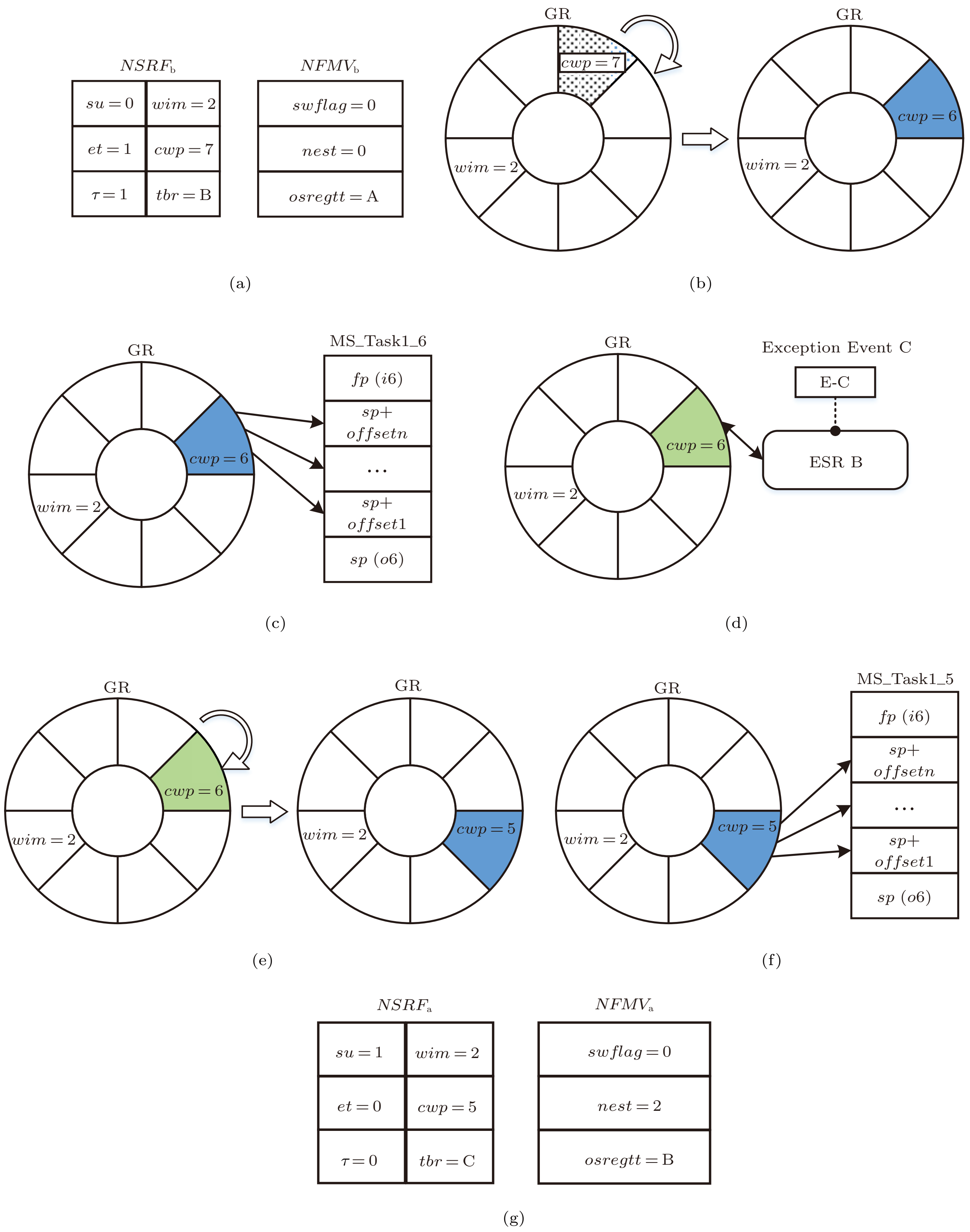

Fig.19. Migration process of data flow of nested exceptions. (a) Initialing state. (b) Entering the exception management. (c) Protecting data. (d) Nested exceptions. (e) Restoring data. (f) Leaving the exception management. (g) Termination state.

## 6 Related Work

Klein *et al.*[1] were the first to verify the correctness and security properties of a high-performance seL4 microkernel in a modern mechanized proof assistant Isabelle[15]. seL4 is a platform of unprecedented trustworthiness, which will allow the construction of highly secure and reliable systems on top. To simplify formal verification, they introduced an intermediate executable specification to hide C specifics. Gu *et al.*[2] built their certified mCertiKOS kernel (in Coq) by decomposing it into many abstraction layers. Such fine-grained layer decomposition led to significantly lower proof and development effort and also better extensibility. Bevier[16] showed how to formally certify Kit, an OS kernel implemented in machine code. Gargano *et al.*[17] showed a framework to certify OS kernel in the Verisoft project. Ni *et al.*[18] certified a non-preemptive thread implementation. However, these formal projects lack actual interruption models and therefore do not support reasoning about interruptible code.

Liu *et al.*[19] tried to use the method of multi-threaded system verification to prove the correctness of the exception management. Due to the lack of formal semantic support for interrupts, when using this method, the semantics of the interrupt was formalized with equivalent thread semantics. However, the subtle difference between interrupts and threads makes it difficult to describe the interruption process with thread semantics. Suenaga and Kobayashi[20] presented a type system to guarantee deadlock-freedom in a concurrent calculus with interrupts. Their calculus is an ML-style language with built-in support of threads, locks, and interrupts. Our EMS is at a lower abstraction level than theirs with no built-in locks. Their type system is also designed mainly for preventing deadlocks with automatic type inference, while our program logic supports verification of partial correctness.

Duan and Regehr[21] verified the interrupt function with the Advanced RISC Machines (ARM) architecture. They created an abstract device model that was inserted into the instruction set of the ARM6 architecture, and their model also modeled the underlying details of the machine. The framework was subsequently ported to the ARMv7 architecture. Fox and Myreen[22] used the monadic specification, and standardized instruction decoding and operational semantics of ARMv7. Feng *et al.*[23] developed a novel Hoare-logic-like framework for certifying low-level system programs involving both hardware interrupts and preemptive threads based on x86 architecture. They showed that enabling and disabling interrupts can be formalized precisely using simple ownership-transfer semantics, and the same technique also extends to the concurrent setting. Kennedy et al. formalized the subset of x86 in Coq, and they used notations, type classes, and the mathematics library Ssreflect[24]. Zha *et al.*[8] and Wang *et al.*[9] formalized the semantics of SPARCv8 instruction set in Coq. They modeled the unique mechanism of SPARCv8, such as control transfer instruction, annulled delay instructions, register window, and delayed-write mechanism. But their work did not consider the impact of the OS functions on the system state.

## 7 Conclusions

This paper proposed a verification framework EMS based on Hoare-logic. The framework is used to prove the correctness of the exception management function of the real-time OS for the SPARCv8 processor architecture. It supports three different scenarios for handling exception events by real-time operating systems, including basic exceptions, nested exceptions, and task switching. We used a fine-grained modeling approach so that the framework can divide the OS kernel's exception management into six execution phases and verify the correctness for each phase. We used this framework to verify the correctness of the SpaceOS exception management module onboard Beidou-3, ensuring the safety of the space system.

In our current work, we assume that the ESR program corresponding to each exception event is correct. In future work, we will try to verify all ESR programs present in the OS. At the same time, we will try to abstract the assembly and C layers to the same level for complete verification by tasks or ESRs. We will try to modify the exception management code and then adapt the structure of the model based on the results returned from Coq. Finally, we will try to improve the efficiency of the verification effort by using the idea of a phased, hierarchical, and top-down formalization of the OS kernel while ensuring the correctness of the software throughout its development lifecycle.

## References

[1] Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, Sewell T, Tuch H, Winwood S. seL4: Formal verification of an OS Kernel. In *Proc. the 22nd ACM Symposium on Operating Systems Principles*, Oct. 2009, pp.207-220. DOI: 10.1145/1629575.1629596.

[2] Gu R H, Shao Z, Chen H, Wu X N, Kim J, Sjöberg V, Costanzo D. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proc. the 12th USENIX Conf. Operating Systems Design and Implementation*, Nov. 2016, pp.653-669.

[3] Chen H, Wu X N, Shao Z, Lockerman J, Gu R H. Toward compositional verification of interruptible OS kernels and device drivers. In *Proc. the 37th ACM SIGPLAN Conf. Programming Language Design and Implementation*, June 2016, pp.431-447. DOI: 10.1145/2908080.2908101.

[4] Xu F, Fu M, Feng X, Zhang X, Zhang H, Li Z. A practical verification framework for preemptive OS kernels. In *Proc. the 28th International Conference on Computer Aided Verification*, July 2016, pp.59-79. DOI: 10.1007/978-3-319-41540-6-4.

[5] Ma Z, Qiao L, Yang M F, Li S F. Verification of operating system exception management for SPARC processor architecture. *Journal of Software*, 2021, 32(6): 1631-1646. DOI: 10.13328/j.cnki.jos.006241. (in Chinese).

[6] Maruyama T, Yoshida T, Kan R *et al.* Sparc64 VIIIfx: A new-generation octocore processor for petascale computing. *IEEE Micro*, 2010, 30(2): 30-40. DOI: 10.1109/MM.2010.40.

[7] Qiao L, Yang M, Gu B, Yang H, Liu B. An embedded operating system design for the lunar exploration rover. In *Proc. the 5th International Conference on Secure Software Integration and Reliability Improvement*, June 2011, pp.160-165. DOI: 10.1109/SSIRI-C.2011.39.

[8] Zha J P, Feng X Y, Qiao L. Modular verification of SPARCv8 code. *Journal of Computer Science and Technology*, 2020, 35(6): 1382-1405. DOI: 10.1007/s11390-020-0536-9.

[9] Wang J, Fu M, Qiao L *et al.* Formalizing SPARCv8 instruction set architecture in Coq. *Science of Computer Programming*, 2019, 187: Article No. 102371. DOI: 10.1016/j.scico.2019.102371.

[10] Hoare C A. An axiomatic basis for computer programming. *Communications of the ACM*, 1969, 12(10): 576-580. DOI: 10.1145/363235.363259.

[11] Bryant R, O'Hallaron D. Computer Systems: A Programmer's Perspective (3rd edition). Pearson, 2016.

[12] Zapletal J, Merta M, Malý L. Boundary element quadrature schemes for multi- and many-core architectures. *Computers and Mathematics with Applications*, 2017, 74(1): 157-173. DOI: 10.1016/j.camwa.2017.01.018.

[13] Fang J B, Liao X K, Huang C *et al.* Performance evaluation of memory-centric ARMv8 many-core architectures: A case study with Phytium 2000+. *Journal of Computer Science and Technology*, 2021, 36(1): 33-43. DOI: 10.1007/s11390-020-0741-6.

[14] Patterson D A, Hennessy J L. Computer Organization and Design: The Hardware/Software Interface (ARM edition). Morgan Kaufmann, 2016.

[15] Paulson L C. Isabelle: A Generic Theorem Prover. Springer, 1994. DOI: 10.1007/BFb0030541.

[16] Bevier W R. Kit: A study in operating system verification. *IEEE Trans. Softw. Eng.*, 1989, 15(11): 1382-1396. DOI: 10.1109/32.41331.

[17] Gargano M, Hillebrand M A, Leinenbach D, Paul W J. On the correctness of operating system kernels. In *Proc. the 18th International Conference on Theorem Proving in Higher Order Logics*, August 2005, pp.1-16. DOI: 10.1007/11541868-1.

[18] Ni Z, Yu D, Shao Z. Using XCAP to certify realistic systems code: Machine context management. In *Proc. the 20th International Conference on Theorem Proving in Higher Order Logics*, Sept. 2007, pp.189-206. DOI: 10.1007/978-3-540-74591-4-15.

[19] Liu H, Zhang H, Jiang Y *et al.* iDola: Bridge modeling to verification and implementation of interrupt-driven systems. In *Proc. the 2014 Theoretical Aspects of Software Engineering Conf.*, Sept. 2014, pp.193-200. DOI: 10.1109/TASE.2014.33.

[20] Suenaga K, Kobayashi N. Type based analysis of deadlock for a concurrent calculus with interrupts. In *Proc. the 16th European Symposium on Programming*, March 24–April 1, 2007, pp.490-504. DOI: 10.1007/978-3-540-71316-6-33.

[21] Duan J, Regehr J. Correctness proofs for device drivers in embedded systems. In *Proc. the 5th International Conference on Systems Software Verification*, Oct. 2010.

[22] Fox A, Myreen M. A trustworthy monadic formalization of the ARMv7 instruction set architecture. In *Proc. the 1st International Conference on Interactive Theorem Proving*, July 2010, pp.243-258. DOI: 10.1007/978-3-642-14052-5-18.

[23] Feng X, Shao Z, Vaynberg A, Xiang S, Ni Z. Modular verification of assembly code with stack-based control abstractions. In *Proc. the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2006, pp.401-414. DOI: 10.1145/1133981.1134028.

[24] Kennedy A. Benton N, Jensen J B, Dagand P E. Coq: The world's best macro assembler? In *Proc. the 15th Symposium on Principles and Practice of Declarative Programming*, Sept. 2013, pp.13-24. DOI: 10.1145/2505879.2505897.

**Zhi Ma** is a Ph.D. candidate in the Department of Computer Science and Technology at Beijing Institute of Control Engineering, Beijing. He received his M.S. degree in computer science from the Waseda University of Japan, Tokyo, in 2018, and his B.S. degree in software engineering from University of Electronic Science and Technology of China, Chengdu, in 2016. His research interests are in operating system and formal methods.

**Lei Qiao** is a professor in the Center of On-Board Computer and Electronics at Beijing Institute of Control Engineering, Beijing. He received his Ph.D. degree in computer science from the University of Science and Technology of China, Hefei, in 2007. His research interests are in operating system design and formal verification. He is a member of CCF.

**Meng-Fei Yang** is a professor in the China Academy of Space Technology, Beijing. He received his Ph.D. degree in the Department of Computer Science and Technology from Tsinghua University, Beijing, in 2005. His research interests are in operating system, formal verification and artificial intelligence.

**Shao-Feng Li** is a Ph.D. candidate in the School of Computer Science and Technology at Xidian University, Xi'an. He received his M.S. degree in computer science at Xidian University, Xi'an, in 2018, and his B.S. degree in computer science at Jiangnan University, Wuxi, in 2014. His research interests are in embedded system, memory management and formal verification.

**Jin-Kun Zhang** is an engineer at Beijing Institute of Control Engineering (BICE), Beijing. He received his M.S. degree in computer science from BICE, Beijing, in 2018, and his B.S. degree in software engineering from Beihang University of China, Beijing, in 2016. His research interests are in operating system and formal methods.