Extended Hierarchy State Transition Matrix Design Method

— Version 2.0 —

Masahiko Watanabe

● Author's biography

Early 1980s: Engaged in VOSIII database language compiler and CAI.
Late 1980s: Engaged in the real-time monitor control system, RTOS.
Early 1990s: Engaged in ZIPC

Published papers and books

- Nikkei Electronics, "Real-time CASE Tools for Supporting Basic Design," January 6, 1992.
- CQ Shuppan Interface, "ZIPC for Real-Time Applications and System Development," May, July, August, 1996.
- Electronics Development Institute Press, "CASE for Real-time Control"
- Electronics Development Institute Press, "Implementing CASE in the Real-time Environment"
- 53rd Convention of Japan Information Processing Society, "ZIPC: An Environment for Developing a Microcomputer Embedded System Using the State Transition Matrix"
- 53rd Convention of Japan Information Processing Society, "Definition of State Transition Matrix Types: The Basic Concept"

The Extended-Hierarchy State-Transition Matrix Design Method, Version 2.0

February 12, 1997
Author: Masahiko Watanabe

1. Introduction

The original version of the Extended Hierarchy State Transition Matrix Design Method was published in 1992. Five years have passed since that time.

Changes in "The Extended Hierarchy State Transition Matrix Design Method version 2.0" are as follows:

(1) States can be concurrent.

(2) State hierarchy is allowed.

(3) Functions and interrupts can be described as events.

The range of applications that can be designed using the state transition matrix has been extended, owing to these additions.

The trend is moving from structured analysis and design to object-oriented analysis and design. The embedded system engineers would ask, "Can it be used for real-time controlled software in the embedded, one-chip microprocessor?" as they did when structured analysis and design were introduced. The answers from methodologists are always, "Yes, it can," but embedded system engineers are skeptical about such an answer. Information processing engineers never ask whether they can use it.

Why are their reactions different? Probably the focus of each party is a little different. The most important information concerning the structured analysis and design method is "data flow," while the "object" is the most important for object-oriented analysis and design. It is natural that these methodologies, which come from the information processing field, focus on "data flow" and "object," and that the information processing engineers have no doubt about it.

The focuses of embedded system engineers, however, are on "actual time" and "control." To our regret, even state transition does not capture "actual time." Maybe "actual time" is a kind of information that can be captured only by activation (simulation) such as computer graphics.

In order for software to realize "control," it is necessary to manage various states within the software, switch actions depending on the state or received event, then change the state further. "Event," "state," "action" and "transition" are important items of information. If a class is designed without considering the importance of such information, the class will not have "control," and we wonder what happens to the software built using the class.

The basic concept of the state transition matrix design method is to design software with consideration for "events," "states," "actions" and "transitions." Therefore, embedded system engineers do not become skeptical. However, the information about "object" should be added to it.

It is not that the "state transition" information is added to "object." "Object-oriented" in the embedded world means to consider "when," "where" and "what" the class executes before considering what the class exactly does (method and attribute). It is necessary to design objects that handle control, not the objects that handle data.

⓵Introduction

The state transition diagram is used to design state transitions. Mealy, Moore, Harel and others have rapidly improved the notational conventions. On the other hand, the state transition matrix has been treated as a supplement to the state transition diagram, so very little has been improved. I have no idea why the state transition diagram has been improved but the state transition matrix has not.

I have challenged this issue because I was curious about the reason. I have always preferred the matrix because it is more organized and easier to spot missing items. As for the diagram, even though deletions could be handled somehow, we did not like the troublesome insertion operations, which required moving other shapes and connecting the transition destinations. Another reason I prefer the state transition matrix is that if it is used for designing, it is not necessary to create a separate checklist for testing. This document is a result of our efforts to realize what the most up-to-date state transition diagrams can represent by using the state transition matrix. I have been designing control systems using the state transition matrix for a long time. The software created thus far is still used actively. The method of design, using the state transfer matrix, did not change even once from the age of assembly language well into the age of C. I believe the importance of the state transition matrix will further expand in the future.

2. Format of the state transition matrix (STM)

The state transition matrix (STM) consists of the following four categories of information:

(1) Event

(2) State

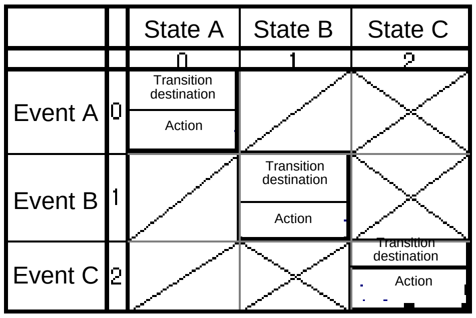
(3) Action

(4) Transition

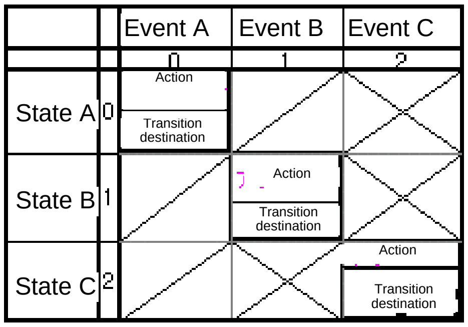
Events (or states) are written in the rows and states (or events) are written in the columns.

Actions and transition destinations are written in the cells where the rows and columns cross.

The STM can clearly and precisely describe when (event), where (state), and what (action).



		State A	State B	State C
		0	1	2
Event A	0	Transition destination Action		
Event B	1		Transition destination Action	
Event C	2			Transition destination Action

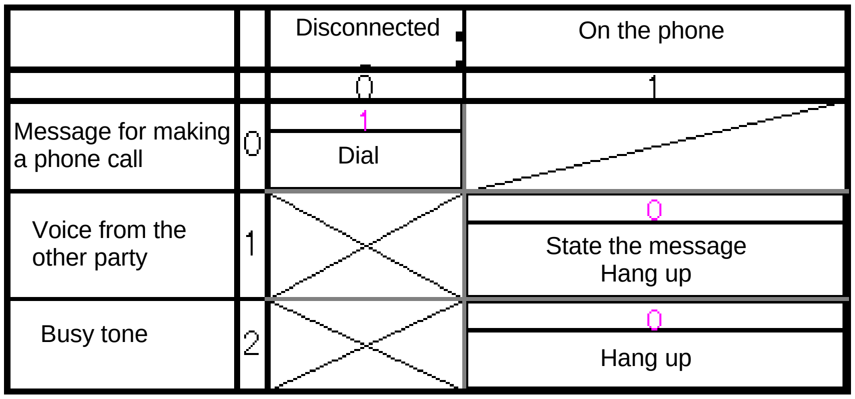


		Event A	Event B	Event C
		0	1	2
State A	0	Action Transition destination		
State B	1		Action Transition destination	
State C	2			Action Transition destination

Figure 2- 1 Format of the state transition matrix

A simple example is described using the STM. This STM shows someone's actions when making a phone call.

In this example, events are described in the rows, states in columns, transition destinations in the upper half of the cells and actions in the lower half of the cells.



		Disconnected	On the phone
		0	1
Message for making a phone call	0	1 Dial	
Voice from the other party	1		0 State the message Hang up
Busy tone	2		0 Hang up

Figure 2- 2 STM describing actions when making a phone call

When the "message for making a phone call" event occurs while in the "disconnected" state, the "dial" action occurs and the state transfers to "busy." The active state is "busy" at this stage, "state the message" and "hang up" are activated when the "voice from the other party" event occurs, and the transition is made to the "disconnected" state. At this stage, the active state is "disconnected."

The STM in this document uses a format in which events are written in rows, states in columns, actions at the bottom and transitions at the top of the cells.

3. Format of the extended hierarchy state transition matrix (EHSTM)

Table 3-1 shows the Extended Hierarchy State Transition Matrix (EHSTM). Details are explained in the following chapters.

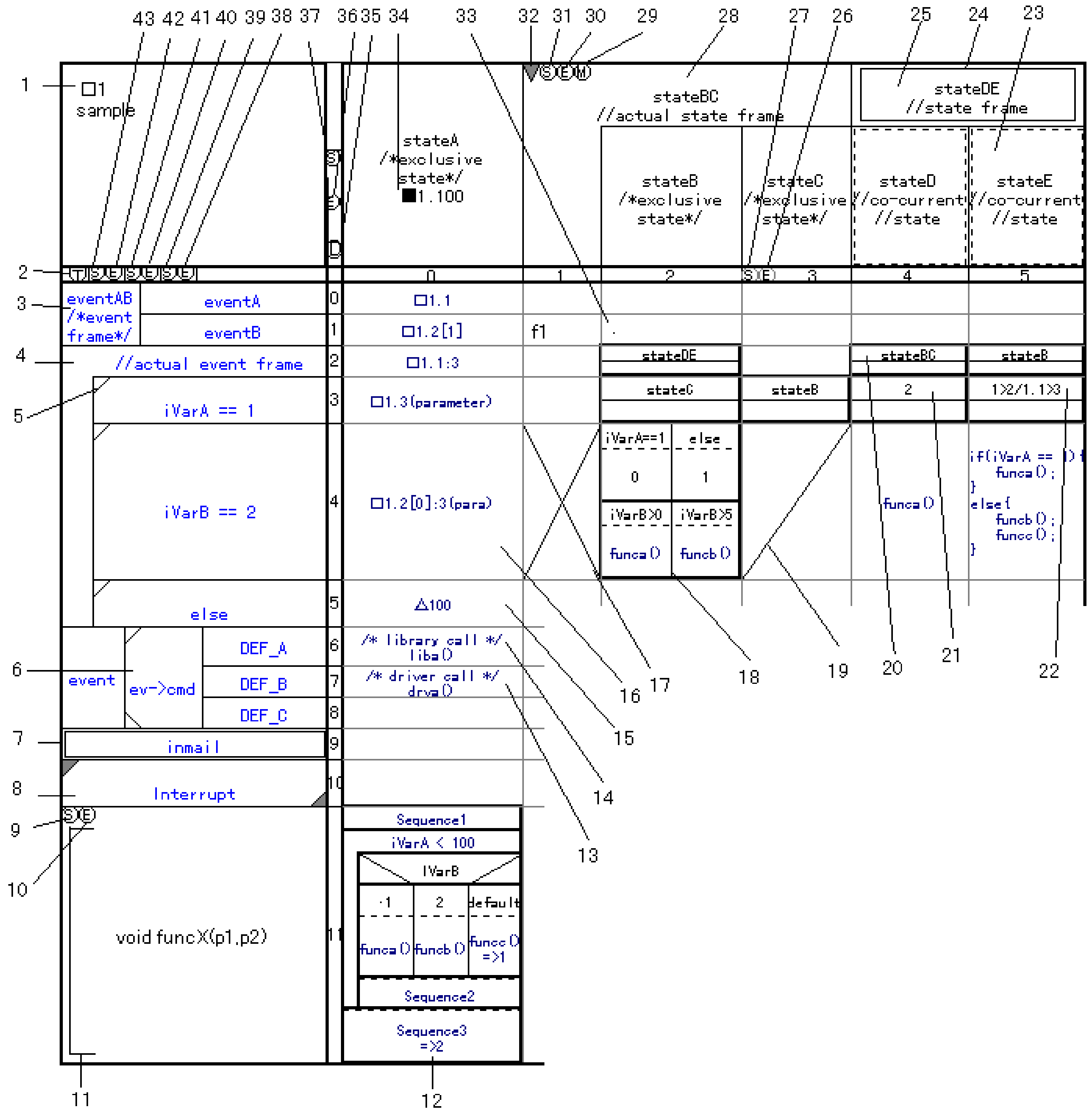


Figure 3- 1 Format of the extended hierarchy state transition matrix

🕒Format of the extended hierarchy state transition matrix

1: STM definition
2: Trigger activity
3: Event virtual frame
4: Event actual frame
5: If condition
6: Switch condition
7: In-mail
8: Interrupt event
9: Event-hit start activity
10: Event-hit end activity
11: Function-call event
12: Nassi-Shneiderman chart
13: Driver STM call
14: Library STM call
15: Subroutine STM call
16: Event-hierarchy STM call
17: Invalid
18: Divided action cell
19: No use
20: Name transition
21: Local transition
22: Global transition
23: Concurrent state
24: Synchronized state
25: State virtual frame
26: Event-analyzer end activity: State-driven type only
27: Event-analyzer start activity: State-driven type only
28: State actual frame
29: State mode activity
30: State end activity
31: State start activity
32: Default state
33: Don't care
34: State hierarchy STM call

35: Dispatch activity
36: STM main end activity
37: STM main start activity
38: Flag analyzer end activity
39: Flag analyzer start activity
40: In-mail analyzer end activity
41: In-mail analyzer start activity
42: Event-analyzer end activity
43: Event-analyzer start activity

Notes: Notations in Japanese and English

Number	Japanese	English	Meaning
Number	□	ZEH_	Event hierarchy STM
Number	■	ZSH_	State hierarchy STM
Number	△	ZES_	Event-driven subroutine STM
Number	▲	ZSS_	State-driven subroutine STM
Number	○	ZEL_	Event-driven library STM
Number	●	ZSL_	State-driven library STM
Number	◎	ZTR_	Trigger event

🕒Format of the extended hierarchy state transition matrix

4. Event

An event is an external stimulus. The state transition matrix design is used to determine to respond to the stimuli. The first step in the state transition design would be to list events. In the example phone call in Chapter 2, there are three events: "message for making a phone call," "voice from the other party" and "busy tone." The events handled in the analysis stage tend to be very abstract. The events handled in the design stage, however, need to be conscious of the implementation, even though they are abstract. For this reason, events in the design stage have to be clearly defined. A clearly defined event has a specific type. The event types are explained below.

4.1 Event type

In the extended hierarchy state transition matrix design method (hereinafter referred to as the EHSTM design method), four types of events are defined:

(1) Variable memory

(2) Interrupt

(3) In-mail

(4) Function call

4.1.1 Variable memory event

The variable memory event considers the change of a variable value as an event. The most general event a program handles is the change in variable value. When a variable value changes from 0 to 1, the program recognizes the occurrence of an event.

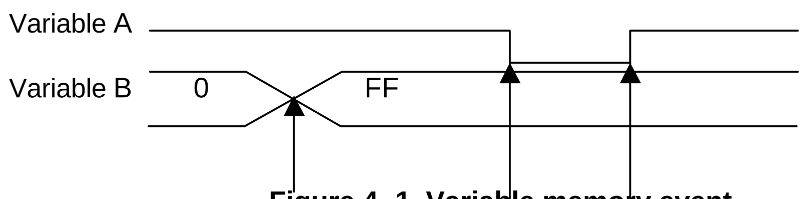


Figure 4- 1 Variable memory event

🕮: Refer to 4.2, "How to access variables," 9.1, "Variable event analyzer," and 9.2, "Event analyzer and variable access method" for more information on variable memory events.

4.1.2 Interrupt event

The interrupt event is used by the CPU to directly notify the program of an external stimulus. When the Real-Time Operating System (RTOS) is used, an interrupt handler provided by the RTOS is called. When the RTOS is not used, the user creates an interrupt routine that stores or "pushes" information from the interrupted program and retrieves or "pops" the information when the interrupt processing is complete before going back to the program. Interrupts can be prevented when a DI (disable interrupt) command is explicitly issued in the program. The interrupt prevention is released by the EI (enable interrupt) command.

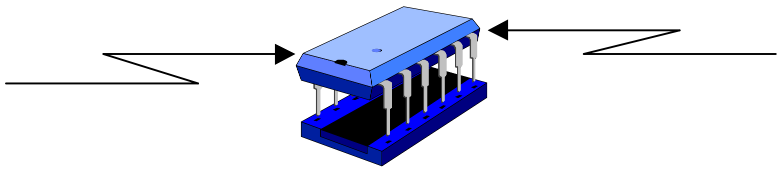


Figure 4- 2 Interrupt event

An interrupt can occur even during the STM operation. Therefore, note that the following two behaviors may occur when an interrupt event is used:

(1) If the STM does not have the reentrant structure, the state transition via the interrupt event may take priority.

(2) If the STM has the reentrant structure, the state transition via the interrupt event may be ignored.

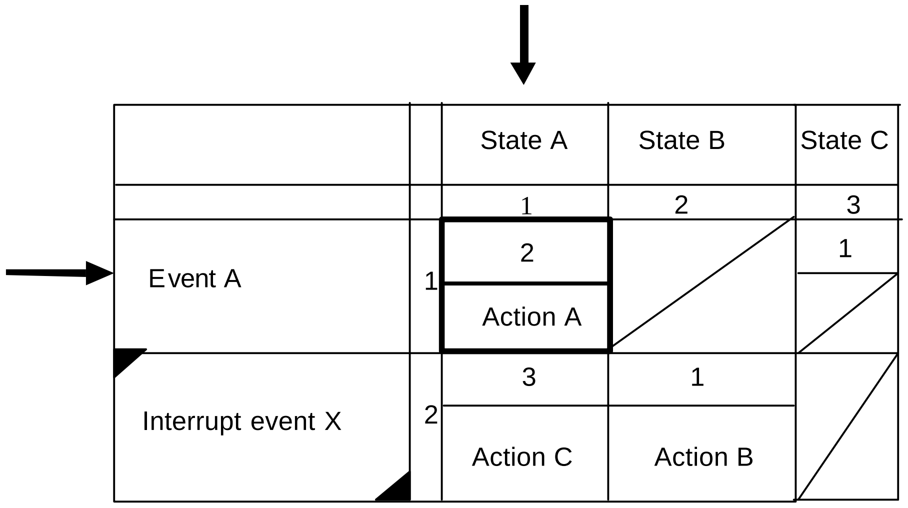


Figure 4- 3 Interrupt event

Assume action A was being executed because event A occurred in state A. When the interrupt event X occurs in this instance, the PC (program counter) of the interrupted action A is pushed

in the TCB (task-control block) if the RTOS is used. If the RTOS is not used, the PC is pushed in the supervisor stack. Then, event analysis is performed during interrupt processing, and action C, where the interrupt event and state A cross, is executed as event number 2. Once action C is complete, a transition is made to state C, whose state number is 3. At this time, the control returns to the interrupted program. If the RTOS is used, it may return after other tasks have been operated. When the execution of the rest of action A is complete, an attempt is made to retrieve the state-transition destination. At this time, **if the STM does not have a reentrant structure, the state-transition destination will be state A, which has the state number 1** because the current state is state C, which has state number 3, due to the state transition by the interrupt event X. **If the STM has a reentrant structure**, a state transition is made to state B having state number 2, because the state number returns to the one at the time of the interrupt. In this case, **the result is equivalent in that no state transition has occurred via the interrupt event X.**
Interrupts can explicitly be prohibited or enabled via the state transition matrix by writing zdi() or zei() to prohibit or enable the interrupt, respectively, in actions or activities. zdi() and zei() are the system calls of the EHSTM design method.

🕮: Refer to 4.4.6, "Interrupt event," 9.4, "Interrupt event analyzer," 12.6, "Interrupt handler STM" and 17.6, "zdi/zei/zdil/zeil," for more about interrupt events.

4.1.3 In-mail event

In-mail events are the events that can be exchanged between STM hierarchies within the same STM tree. In other words, the in-mail function exchanges messages between STMs in the same way as the RTOS's message communication function between tasks. The in-mail function can be used regardless of whether the RTOS is implemented or not. The in-mail is for notify other STMs of internal events occurring in the action cell. A usage example is as follows: When an error is detected during the processing of an action cell, an in-mail indicating an abnormality is transmitted to the STM controlling another equipment in order to perform error recovery.

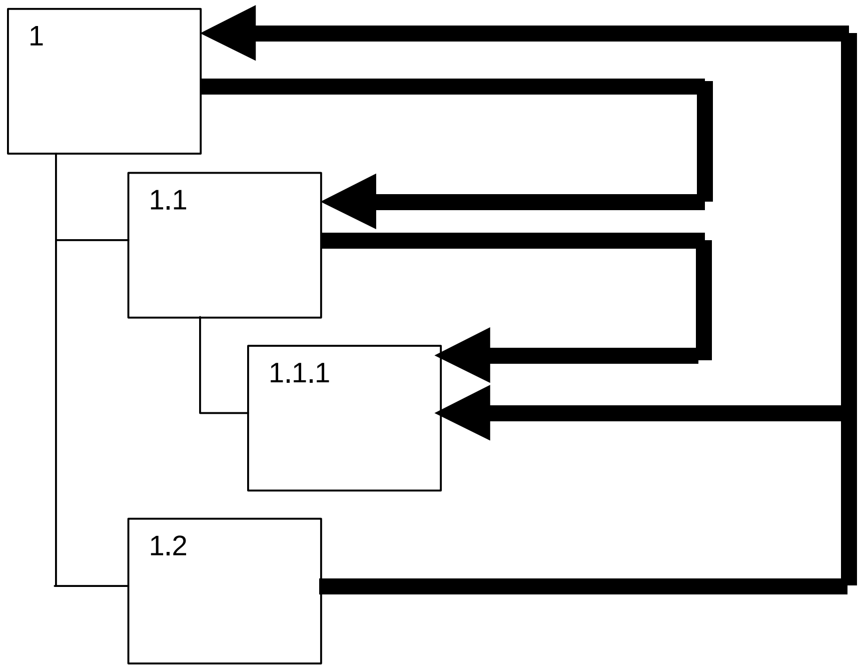
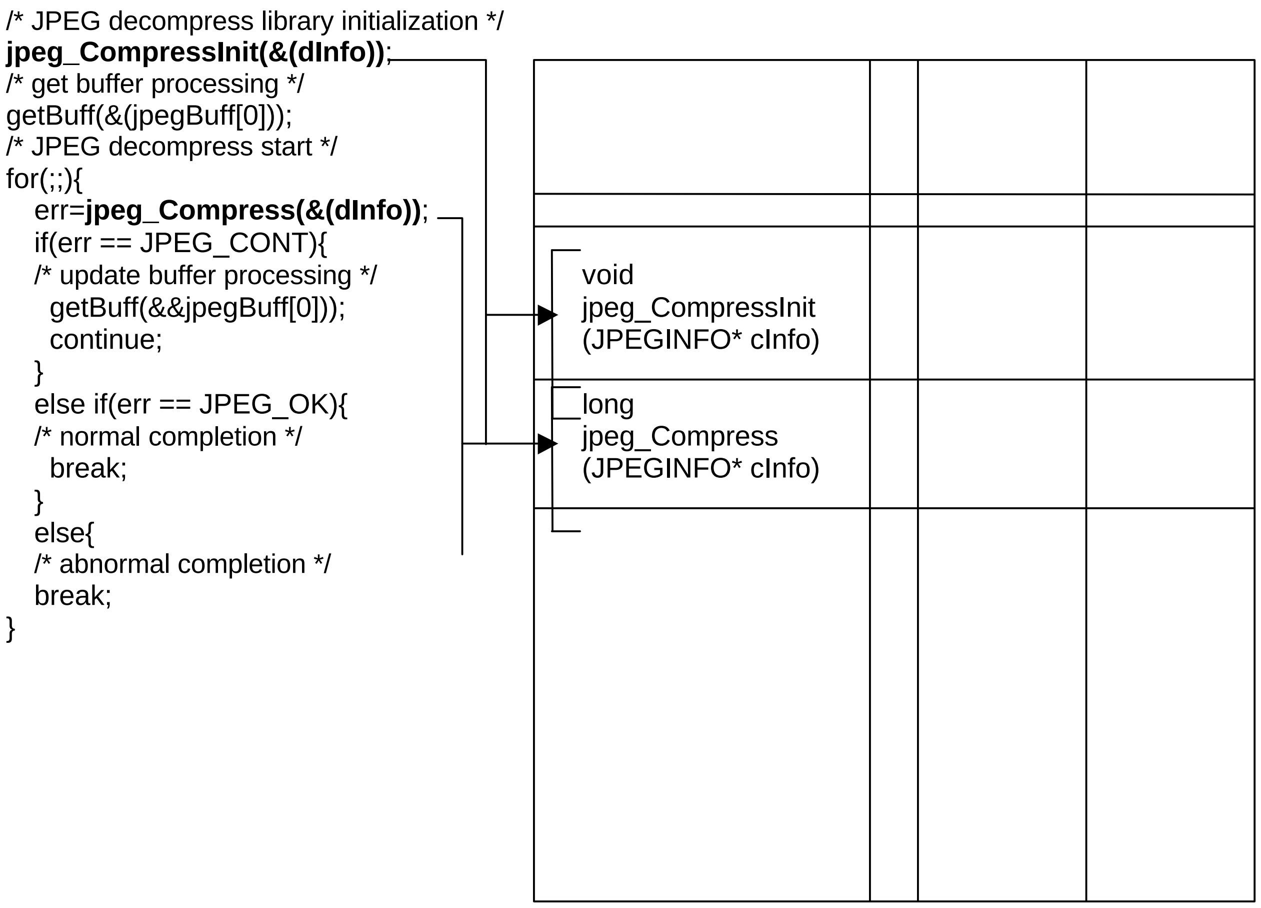


Figure 4- 4 In-mail event

🕮: Refer to 4.4.5, "In-mail event," 9.3, "In-mail analyzer," and 17.1, "In-mail," for the in-mail events.

4.1.4 Function-call event

A function-call event considers a function call as an event. Therefore, the STM that can handle function-call events has to be in the library section.



```
/* JPEG decompress library initialization */
jpeg_CompressInit(&(dInfo));
/* get buffer processing */
getBuff(&(jpegBuff[0]));
/* JPEG decompress start */
for(;;){
   err=jpeg_Compress(&(dInfo));
   if(err == JPEG_CONT){
   /* update buffer processing */
     getBuff(&&jpegBuff[0]));
     continue;
   }
   else if(err == JPEG_OK){
   /* normal completion */
     break;
   }
   else{
   /* abnormal completion */
   break;
}
```

void jpeg_CompressInit (JPEGINFO* cInfo)
long jpeg_Compress (JPEGINFO* cInfo)

Figure 4- 5 Function-call event

An STM that has been called as a function returns to the caller when the zret() system call is issued.

📖: Refer to 9.5, "Function-call event analyzer" and 12.4, "Library STM" for the function-call events.

4.2. How to access variables

There are two access methods for variables:

(1) Message type

(2) Flag type

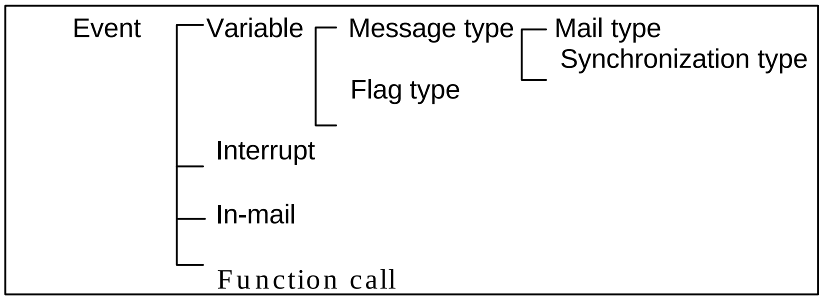


Figure 4- 6 Event types

The access method is closely related to the timing when the event is detected (retrieved).

4.2.1 Message event

Message events can be divided into communication-type events and synchronization-type events. The RTOS is necessary in order to handle the message-type events. Mail-type events and synchronization-type events do not simply access the variables in the program, but they do so by executing mail communication and synchronization using the RTOS system calls. The program is called a task or process because it operates under the control of RTOS.

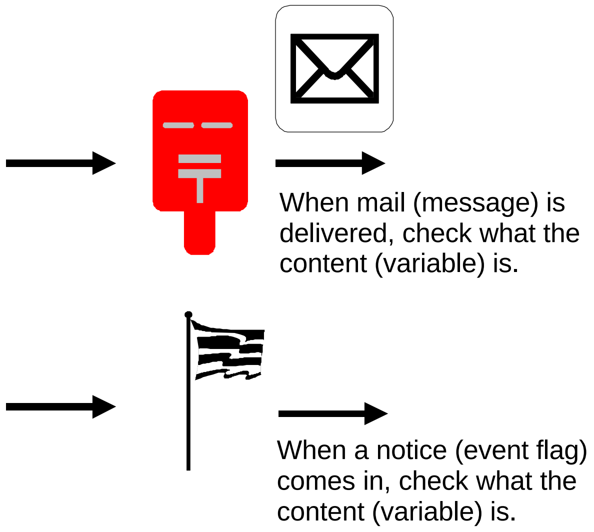

When mail (message) is delivered, check what the content (variable) is.
When a notice (event flag) comes in, check what the content (variable) is.

Figure 4- 7 Message event

4.2.2 Flag event

With flag events, the program simply accesses variables. It does not matter whether the RTOS exists when handling flag events.

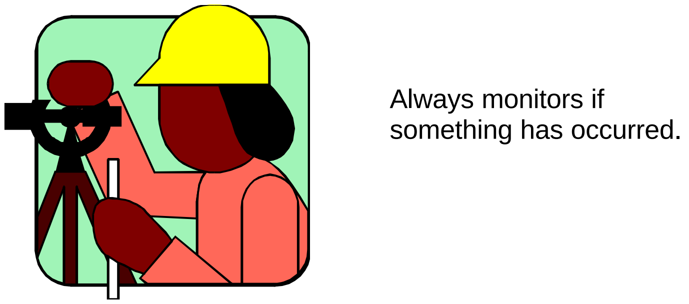


Figure 4- 8 Flag event

A flag event is represented by an "if" or "switch" condition in terms of its format.
The condition of the highest level in the event cell is considered a flag event. Events with condition symbols following a message event are considered regular condition statements and are not subject to monitoring by polling as flag events are.

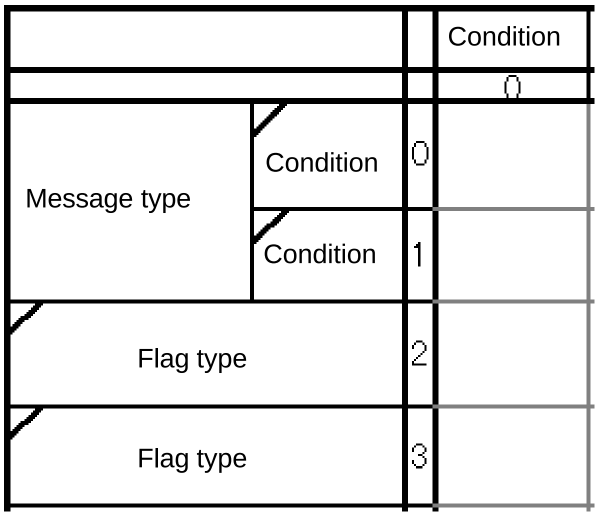


Figure 4- 9 Flag events and conditions

: Refer to 9.1, "Variable event analyzer" and 9.2, "Event analyzer and variable access method," for information on accessing variables.

4.3. Mixed events

Events of different types can coexist in a single STM. However, function-call events must belong to the library or device-driver section.

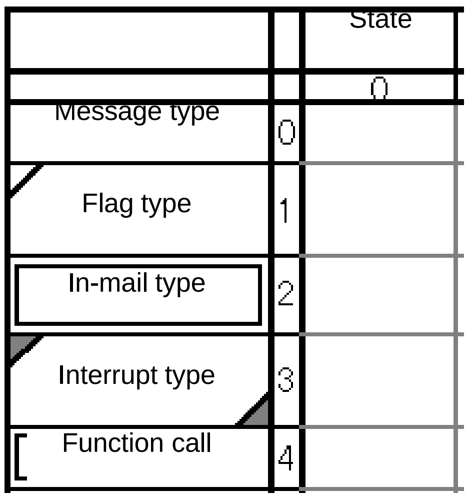


Figure 4- 10 Mixed events

In the above example, five event analyzers are generated in one STM. They are the message analyzer for the message event, flag-sensing analyzer for the flag event, in-mail analyzer for the in-mail event, interrupt handler for the interrupt event, and the function for the function-call event.

The default operations of mixed events are:

🕓 message analyzer

🕓 flag analyzer

🕓 in-mail analyzer

Further, the interrupt handler is executed when the interrupt occurs, and the function-call event occurs with the function call. "Event analyzer" is a generic term for the above three.

When communication-type and synchronization-type message events coexist, a problem occurs in the task execution control, such as when use of the synchronization type is disabled the communication type enters the wait state, and vice versa.

When the highest-level event is a flag type or in-mail type, the event symbols for its children can be omitted.

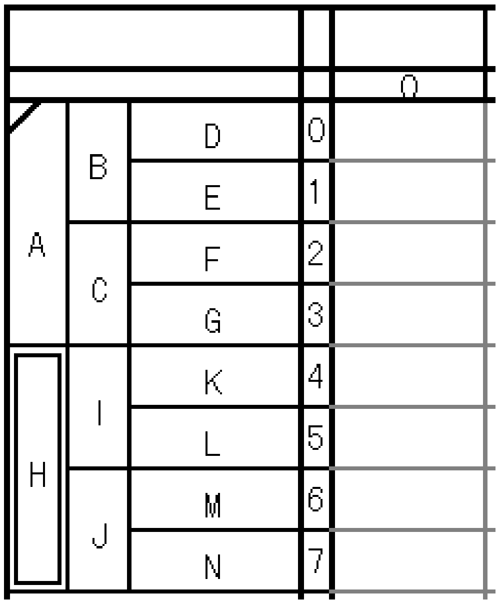


Figure 4- 11 Nested events

In Figure 4-11, events "B" through "G" are not considered message events but flag events, which are the same event type as parent event "A." In other words, a message event cannot be placed after a flag event. Similarly, events "I" through "N" are not considered message events but in-mail events, which are the same event type as parent event "H." In other words, the parent of a message event has to be a message event.

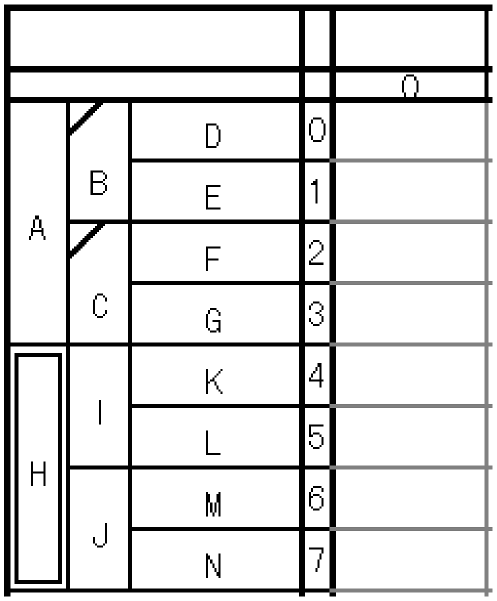


Figure 4- 12 Flag events and conditions

In Figure 4-12, event "A" is a message event. Events "B" and "C" are not flag events but condition statements. Events "D" through "G" are message events, which are the same event type as parent event "A."

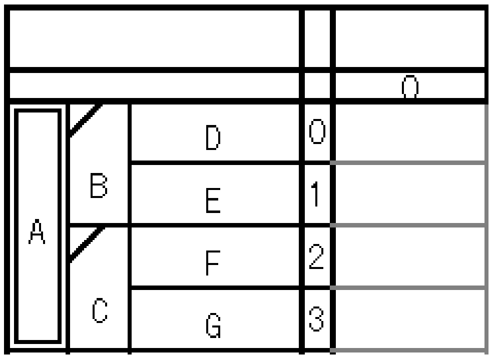


Figure 4- 13 In-mail events and conditions

In Figure 4-13, event "A" is an in-mail event. Events "B" and "C" are not flag event but condition statements. Events "D" through "G" are in-mail events, which are the same event type as parent event "A." They are not interpreted as message events.

When the head is a message event, in-mail events cannot be placed thereafter.

🕮: Refer to 9, "Event analyzer," for details on the event analyzer.

4.4. Format of event cell

The following information can be described in the event cells:

1: Event

2: Event frame

3: Message event

4: Flag event

5: In-mail event

6: Interrupt event

7: Function-call event

8: Trigger event

9: “else” event

10: Event virtual frame

11: Event actual frame

12: “if” condition

13: “switch” condition

14: Event-analyzer start activity

15: Event-analyzer end activity

16: Event-hit start activity

17: Event-hit end activity

18: Default/else (no action default/else)

19: Analysis sequence number

4.4.1 Event

"Event" is a generic team for message events, flag events, in-mail events, interrupt events and function-call events. An event is converted to an event number by the event analyzer. An event normally has an event number, but sometimes it refers to an event and event frame collectively.

📖: Refer to 9, "Event analyzer," for details on the event analyzer.

4.4.2 Event frame

An event frame is a bundle of events or event frames. The event frame allows the creation of a tree structure of events or conditional branches. There are two kinds of event frames: event virtual frames and event actual frames.

4.4.3 Message event

There are two kinds of message events -- mail events and synchronization events -- but they use the same notations.

4.4.4 Flag event

A triangle is attached to the top-left corner of a flag event.

4.4.5 In-mail event

In-mail performs communication between STMs within the same tree. The reception of an in-mail is represented by a double square. The issuance of in-mail is represented by “in-mail (STM level number of the notification destination, in-mail name)” in ZIPC.

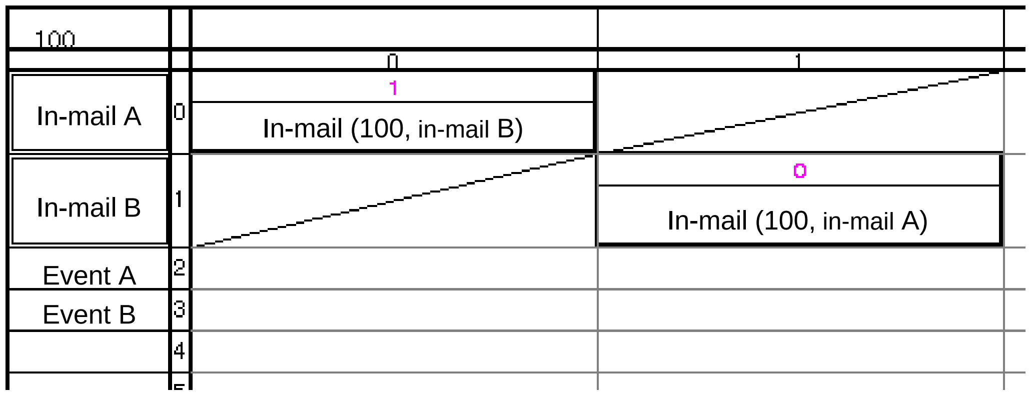
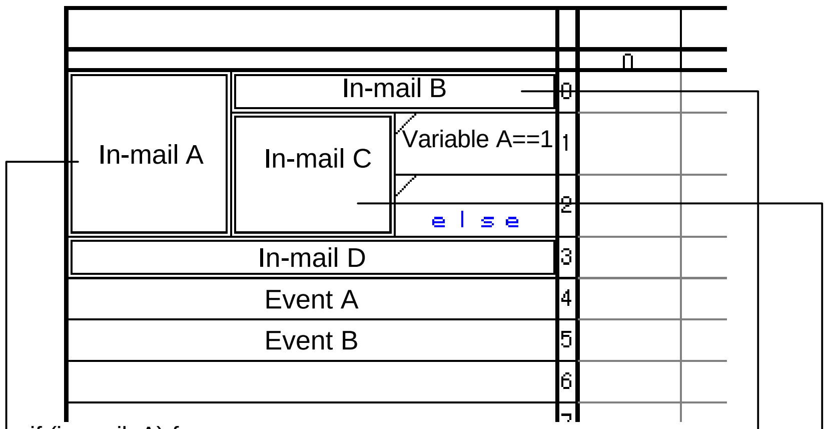


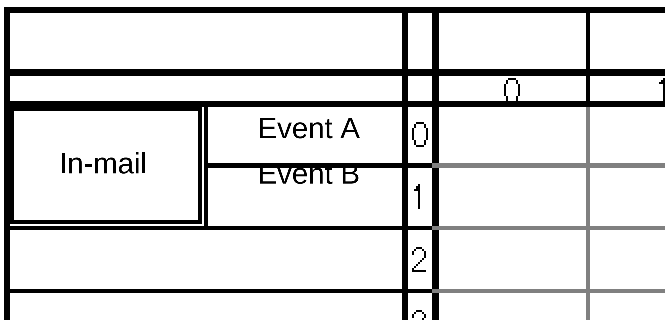
Figure 4- 14 Sample in-mail STM (1)

There is a possibility that an in-mail can form an infinite loop. In the above STM, the transmission and reception of in-mail are repeated infinitely when in-mail A occurs during state number 0. The "if" and "switch" conditions can be stated after in-mail. An event-hit activity can be defined for an in-mail. No events can be stated following an in-mail.



```
if (in-mail_A) {
          if (in-mail_B) {
                    // event number 0
          }
          else if (in-mail_C) {
                    if (variable A = 1) {
                              // event number 1
                    }
                    else {
                              // event number 2
                    }
          }
}
else if (in-mail_D) {
                    // event number 3
}
```

Figure 4- 15 Sample in-mail STM (2)



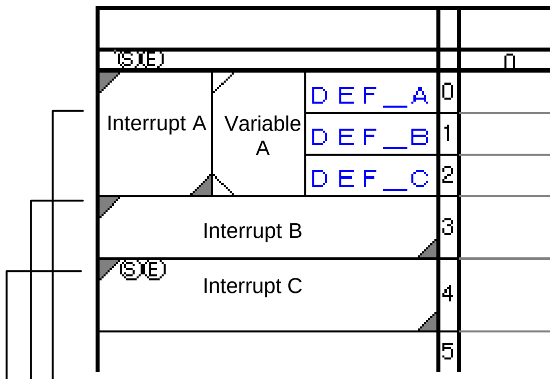
The in-mail analyzer and event analyzer are separate functions.

Figure 4- 16 Sample of unallowed in-mail STM

📖: Refer to 9, "Event analyzer," for the in-mail and event analyzers.

4.4.6 Interrupt event

The interrupt event analyzes events within the interrupt handler and manipulates the STM. The "if" and "switch" conditions can be stated after the interrupt. An event detection activity can be defined for an interrupt event. No events can be stated after an interrupt event.

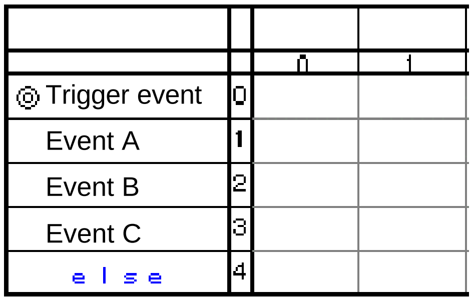


```
interrupt void interrupt_A (void) {
            switch (variable_A) {
            case DEF_A:
                            // event number 0
                            break;
            case DEF_B:
                            // event number 1
                            break;
            case DEF_C:
                            // event number 2
                            break;
            }
}

interrupt void interrupt_B (void) {
            // event number 3
}

interrupt void interrupt_C (void) {
            // event-hit start activity
            // event number 4
            // event-hit end activity
}
```

Figure 4- 17 Sample interrupt event STM

4.4.7 Function-call event
A function-call event is indicated by "[" on the left.

4.4.8 Trigger event
Only one trigger event can be defined in an STM department (STM tree). However, it cannot be used in the subroutine, library or driver section. In other words, one trigger event can be defined in a task or module. A trigger event is executed only once at the beginning of a task or module execution. **In order not to increase the number of cells, trigger events are not written in the event column but as trigger activities in EHSTM version 2 or later. In EHSTM version 2, "●" and "•" are not treated as keywords for trigger events. If a trigger event needs to be specified anyway, use “ .”** The trigger event is executed only in the left-most action cell (state number 0), even if it is a concurrent state. In a trigger event, the specification of the default state position with the default symbol ( ) is ignored, and the state with the youngest number is selected instead.



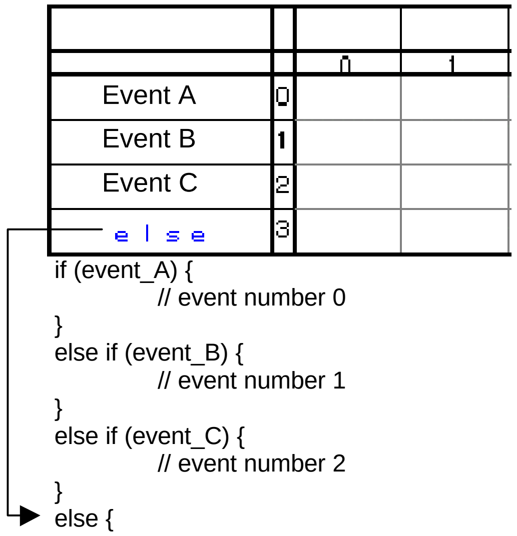
		0	1
◎Trigger event	0		
Event A	1		
Event B	2		
Event C	3		
else	4		

Figure 4- 18 Trigger event

4.4.9 “else” event

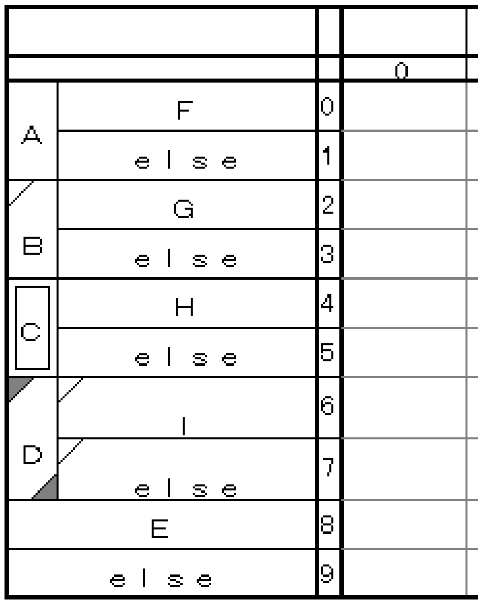
One "else" event can be defined in each frame. This is the event not classified into any event.

Figure 4-19 represents the else event in a format similar to the C programming language.



```
if (event_A) {
            // event number 0
}
else if (event_B) {
            // event number 1
}
else if (event_C) {
            // event number 2
}
else {
            // event number 3
}
```

Figure 4- 19 “else” event



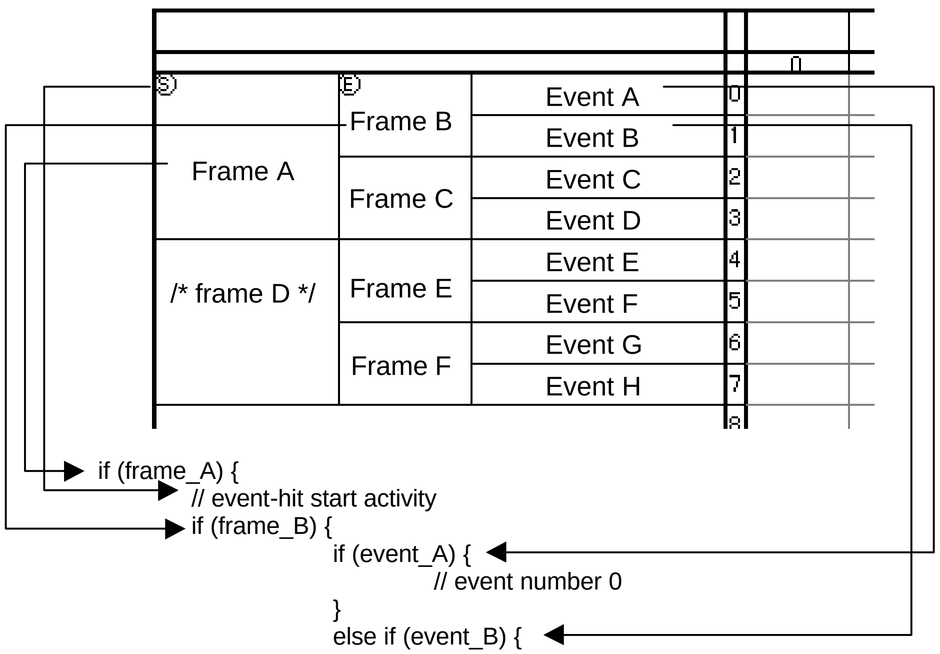
The else event can be placed in each frame as shown in Figure 4-20.

Figure 4- 20 “else” event in each frame

4.4.10 Event virtual frame

The event virtual frame is used for analyzing events, and can execute event-hit activities. An event virtual frame can be a comment frame that will not be executed, by enclosing the entire frame with comment symbols. In this case, the activity does not run, either. Event virtual frames do not have event numbers.

Take a look at Figure 4-14. Frame A is the parent frame of frames "B" and "C," and possesses an event-hit start activity. Frame B is the parent frame of events "A" and "B," and possesses an event-hit end activity. Frame D is a comment. A comment is an abstract event that does not have to be analyzed by the event analyzer.

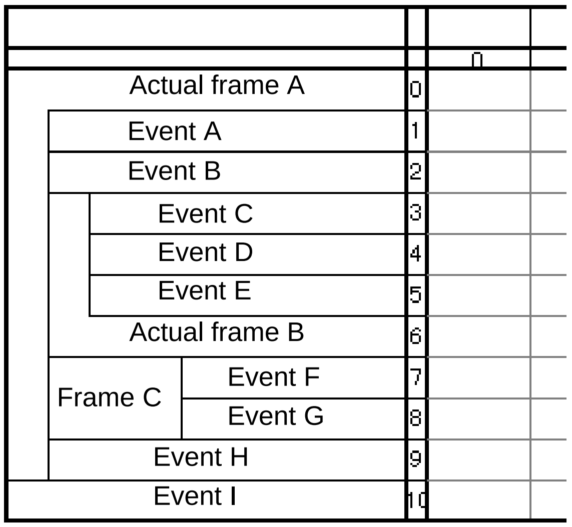


```
if (frame_A) {
        // event-hit start activity
        if (frame_B) {
                    if (event_A) {
                            // event number 0
                    }
                    else if (event_B) {
                            // event number 1
                    }
                    // event-hit end activity
        }
        else if (frame_C) {
                    if (event_C) {
                            // event number 2
                    }
                    else if (event_D) {
                            // event number 3
                    }
        }
}
else if (frame_E) { // frame D
        if (event_E) {
                // event number 4
        }
        else if (event_F) {
                // event number 5
        }
}
else if (frame_F) {
        if (event_G) {
                // event number 5
        }
        else if (event_H) {
                // event number 6
        }
}
```

Figure 4- 21 Event virtual frame sample STM

4.4.11 Event actual frame

The event actual frame can be described as an event virtual frame that has an event number and drives the STM.



```
if (actualframe_A) {
          // event number 0
          if (event_A) {
                    // event number 1
          }
          else if (event_B) {
                   // event number 2
          }
          else if (actualframe_B) {
                    if (event_C) {
                              // event number 3
                    }
                    else if (event_D) {
                              // event number 4
                    }
                    else if (frame_E) {
                              // event number 5
                    }
                    // event number 6
          }
          else if (frame_C) {
                    if (event_F {
                              // event number 7
                    }
                    else if (event_G) {
                              // event number 8
                    }
          }
```

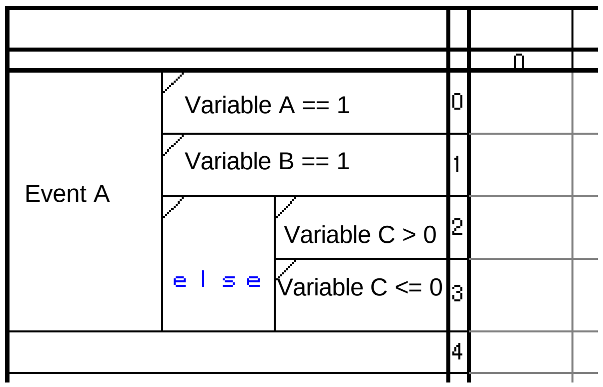
```
        else if (event_H) {
                // event number 9
        }
}
else if (event_I) {
        // event number 10
}
```

Figure 4- 22 Event actual frame sample STM

Event actual frames can also define activities the same way event virtual frames can.

4.4.12 "if" condition

The "if" condition classifies events based on a specific condition. In the traditional STM, "if" conditions are written in condition cells. However, it is not practical to write a retry counter as a state, for example, because it will lead to a huge increase in the number of states. Be careful, since the use of too many "if" conditions will increase flags, which may ruin the STM design. Remember, the essence of the STM design is to reduce flags as much as possible. The purpose of the STM design will be lost if too many "if" conditions are used in the state transition matrix.



```
if (event_A) {
        if (variable_A = 1) {
                // event number 0
        }
        else if (variable_B = 1) {
                // event number 1
        }
        else {
                if (variable_C > 0) {
                        // event number 2
                }
                else if (variable_C <= 0) {
                        // event number 3
                }
        }
}
```

Figure 4- 23 "if" condition sample STM (1)

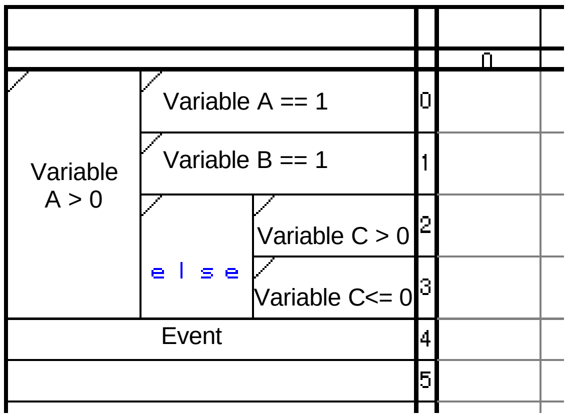


Figure 4- 24 "if" condition sample STM (2)

Conditions and events can be mixed, as shown in Figure 4-24. When the event at the highest level is a flag or in-mail type, the child events are interpreted as the same type as that of the parent event.

In Figure 4-25, the event is interpreted as a flag type instead of a message type.

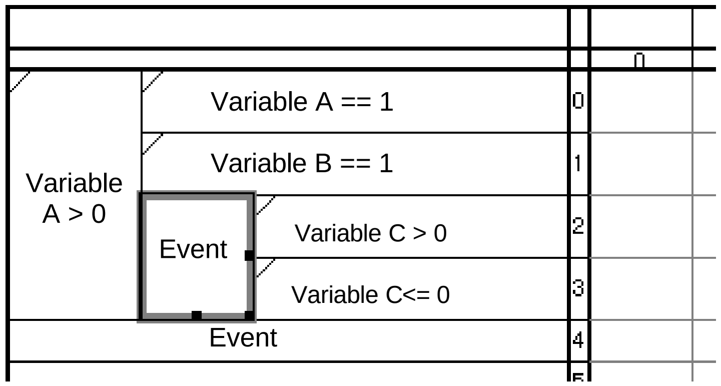


Figure 4- 25 "if" condition sample STM (3)

A function call (differing from the function-call event) can be written in the "if" condition. For example, a function (not a variable) can also be written as follows:

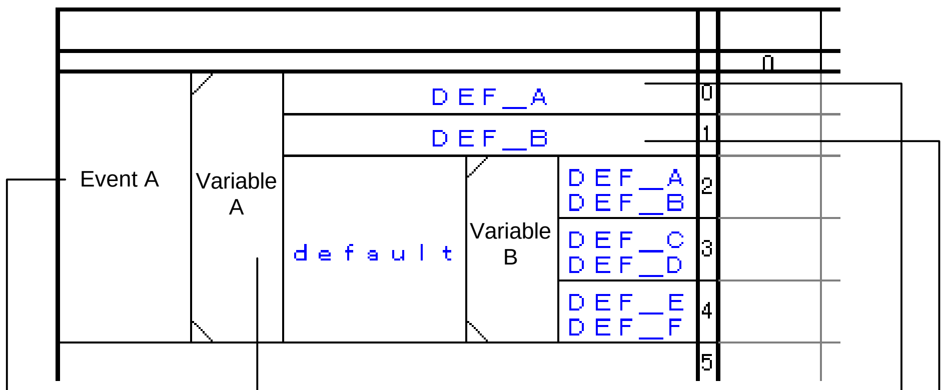
GetChar() = OK

This can also be written using a "switch" condition or flag-type event.

📖: Refer to 4.3, "Mixed events," for the interpretation when events and conditions are mixed.

4.4.13 "switch" condition

"Switch" conditions work in the same way as "if" conditions. The difference is the same as between the "if" statement and "switch" statement in the C programming language. A "case" statement follows immediately after the "switch" condition, and no events are allowed.



```
if (event_A) {
          switch (variable_A) {
          case DEF_A:
                        // event number 0
                        break;
          case DEF_B:
                        // event number 0
                        break;
          default:
                        switch (variable_B) {
                        case DEF_A:
                        case DEF_B:
                                      // event number 2
                                      break;
                        case DEF_C:
                        case DEF_D:
                                      // event number 3
                                      break;
                        case DEF_E:
                        case DEF_F:
                                      // event number 4
                        }
          }
}
```

Figure 4- 26 "switch" condition sample STM

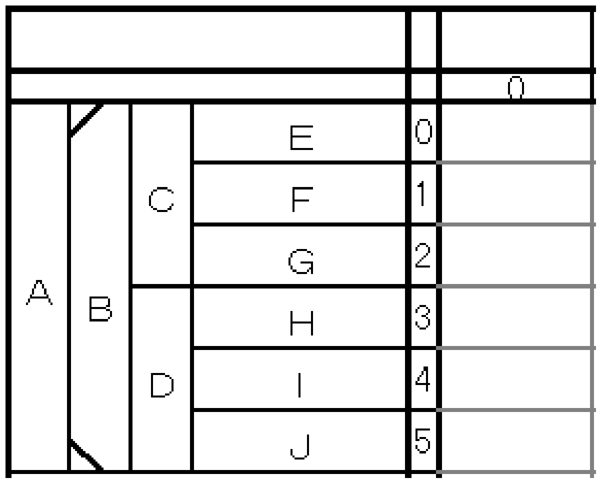


Figure 4- 27 Coexistence of switch condition and message event

In Figure 4-27, "A" is interpreted as a message event, "B" as a switch statement, "C" and "D" as case statements, and "E" to "J" as message events.

4.4.14 Event-analyzer start activity

The processing that is called immediately prior to starting the event analyzer is described. Three types can be set in one state transition matrix: message-event analyzer, flag-event analyzer and in-mail event analyzer. An "S" is attached to the state transition matrix to indicate that an event-analyzer start activity exists. Refer to the interrupt-event sample STM for the format.

📖: The activity will be explained in 8, "Activity."

4.4.15 Event-analyzer end activity

The processing that is called immediately prior to the end of the event analyzer is described. Three types can be set in one state transition matrix: message-event analyzer, flag-event analyzer and in-mail event analyzer. An "E" is attached to the state transition matrix to indicate that an event-analyzer end activity exists. Refer to the interrupt event sample STM for the format.

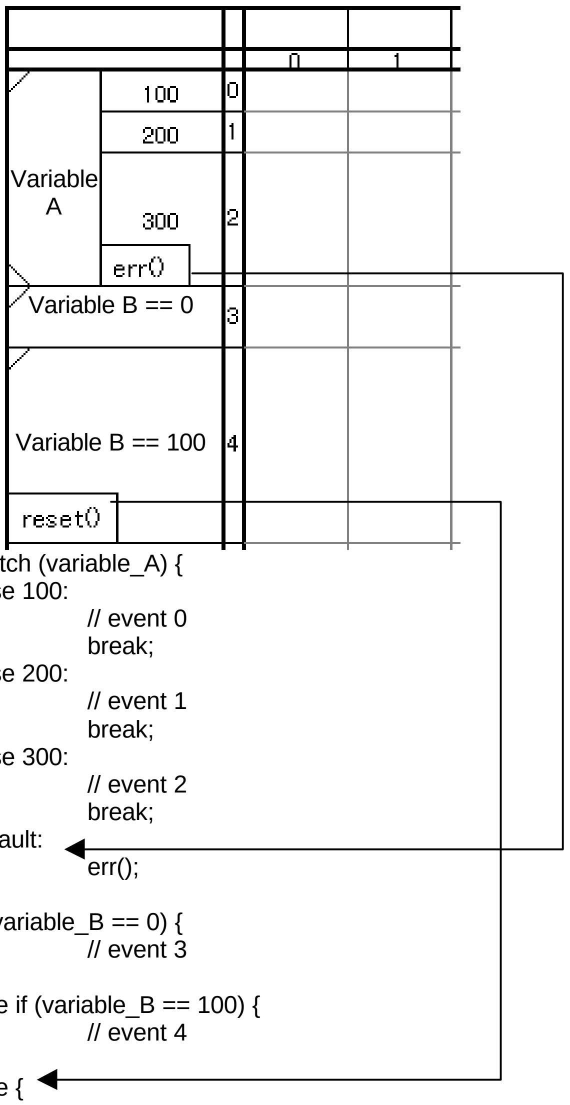
4.4.16 Event-hit start activity

This is the processing that is executed immediately after an applicable event is detected. It can be set in four types of events: variable type, interrupt type, in-mail type and function-call type. An "S" is attached when there is activity. Refer to the interrupt event sample STM for the appropriate format.

4.4.17 Event-hit end activity

This process is executed when the action and transition processing are finished after an applicable event is detected. It can be set in four types of events: variable type, interrupt type, in-mail type and function-call type. An "E" is attached when there is activity. Refer to the interrupt-event sample STM for the format.

4.4.18 No action default/else



```
switch (variable_A) {
case 100:
            // event 0
            break;
case 200:
            // event 1
            break;
case 300:
            // event 2
            break;
default:
            err();
}
if (variable_B == 0) {
            // event 3
}
else if (variable_B == 100) {
            // event 4
}
else {
            reset ()
}
```

Figure 4- 28 No action default/else

When processing is required for the "default" of a switch statement or "else" of an "if" statement, regardless of the state, the increase in the number of cells can be suppressed by specifying no action default/else.

4.4.19 Analysis sequence number

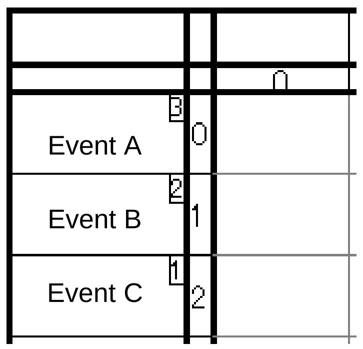


Figure 4- 29 Analysis sequence number

```
if (event_C) {
        // event number 2
}
else if (event_B) {
        // event number 1
}
else if {event_A} {
        // event number 0
}
```

The order of the analysis can be set by specifying the analysis sequence number.

5. State

A state can be described as a shelf for storing events that have occurred in the past. In the example of the telephone in Chapter 2, there were two states: "disconnected" and "busy." A state has two attributes: one is the exclusive attribute, and the other is the concurrent attribute. The STM created only with the exclusive attribute is called the exclusive STM, and the others are called concurrent STMs.

5.1. Exclusive state

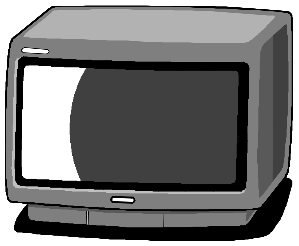
If only one of the states of the design target is active at a time, it is in the exclusive state. The telephone example in Chapter 2 is in the exclusive state. The exclusive state means that either the "disconnected" or "busy" state can be active.

5.2. Concurrent state

When multiple states are active in the design target, it is in the concurrent state. In the example of the concurrent state shown below, the action of watching TV is added to the telephone example from Chapter 2.

		TELEPHONE		TV		
		DISCONNECTED	ON THE PHONE	OFF	ON	
					NORMAL	MUTE
		0	1	2	3	4
Message for making a phone call	0	1 Dial	/	/	/	/
Voice from the other party	1	×	0 State the message Hang up	/	4 Mute	/
Busy tone	2	×	0 Hang up	/	/	/
Interested program	3	/	/	3 Turn TV on	/	/
End of program	4	/	/	×	2 Turn TV off	2 Turn TV off

Figure 5- 1 Concurrent state STM

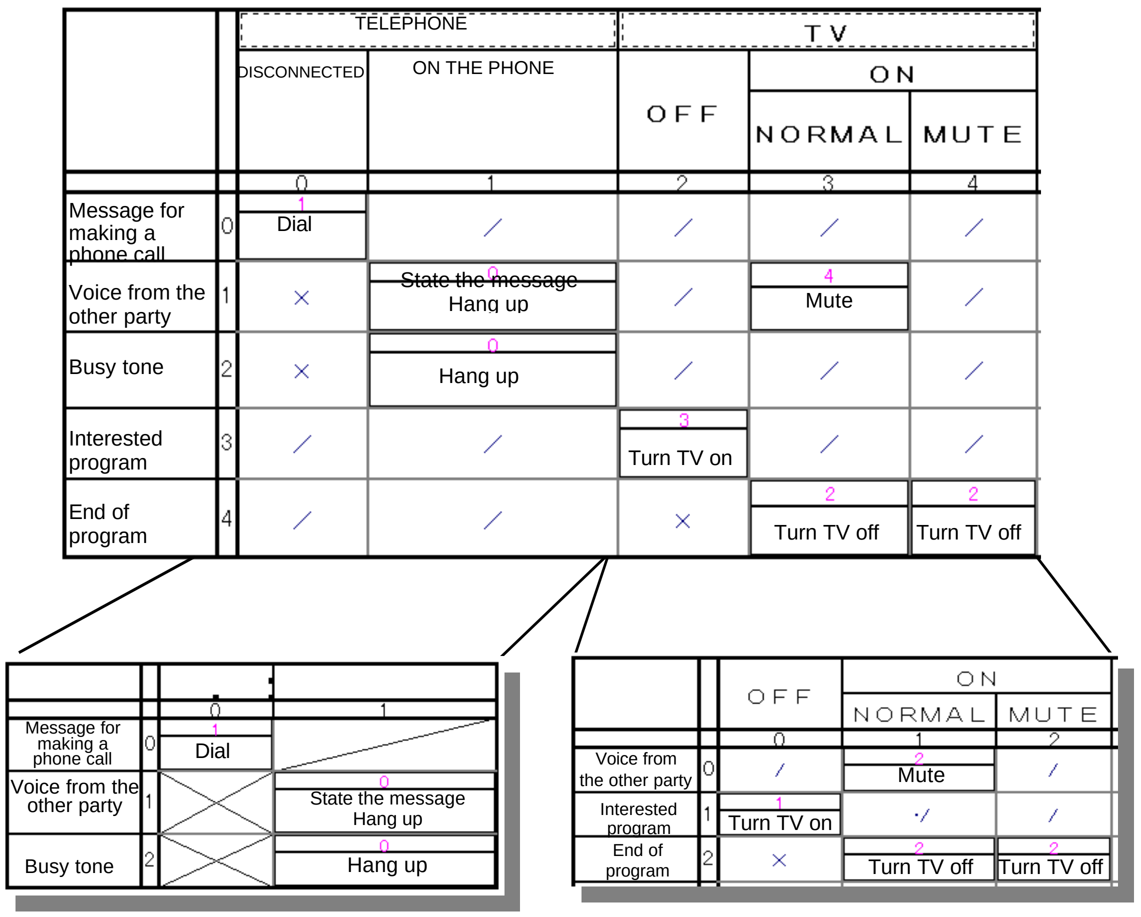


This is an STM for the action to make a phone call while watching TV. When making a phone call while watching TV, the sound of the TV is muted when the other party answers. The telephone and TV can operate concurrently.

A concurrent state is surrounded by dotted lines. The "telephone" and "TV" state frames have concurrent attributes, and both of them could be active simultaneously. The "telephone" state frame has two child state frames ("disconnected" and "busy"), both of which are exclusive attributes. The "TV" state frame has one exclusive-attribute state ("OFF") and one exclusive-attribute state frame ("ON"). In addition, the "ON" state frame has two exclusive-attribute states ("NORMAL" and "MUTE").

A concurrent state is a candidate for task splitting. Concurrent states mean operations are executed concurrently. Therefore, concurrent actions can be achieved through multitasking by considering each concurrent state as a task. In this case, the STM that represents one of these tasks is in the exclusive state.

[System specifications]



		TELEPHONE		TV		
		DISCONNECTED	ON THE PHONE	OFF	ON	
					NORMAL	MUTE
		0	1	2	3	4
Message for making a phone call	0	1 Dial	/	/	/	/
Voice from the other party	1	×	0 State the message Hang up	/	4 Mute	/
Busy tone	2	×	0 Hang up	/	/	/
Interested program	3	/	/	3 Turn TV on	/	/
End of program	4	/	/	×	2 Turn TV off	2 Turn TV off

		0	1
Message for making a phone call	0	1 Dial	
Voice from the other party	1	×	0 State the message Hang up
Busy tone	2	×	0 Hang up

		OFF	ON	
			NORMAL	MUTE
		0	1	2
Voice from the other party	0	/	2 Mute	/
Interested program	1	1 Turn TV on	/	/
End of program	2	×	2 Turn TV off	2 Turn TV off

[Telephone task] [TV task]

Figure 5- 2 Concurrent states and exclusive state tasks

5.3. Format of state cell

The following information can be described in the state cells:

1: State

2: State frame

3: Exclusive state

4: Concurrent state

5: State virtual frame

6: State actual frame

7: State start activity

8: State mode activity

9: State end activity

10: Default state

11: State hierarchy STM call

12: Synchronized state

5.3.1 State

State is a generic term for the exclusive state and concurrent state. In some cases, it includes state frames.

5.3.2 State frame

A state frame is a bundle of states or state frames. A tree structure of states can be created via the state frame. State frames have two types: state virtual frame and state actual frame.

5.3.3 Exclusive state

An exclusive state is represented by a square.

5.3.4 Concurrent state

A concurrent state is represented by a dotted square inside a solid square.

5.3.5 State virtual frame

A state frame is a parent state that bundles child states. A state frame itself may become a child. Similarly to the states, state frames have attributes of either exclusive or concurrent.

Rules in the parent-and-child relationship of state attributes

( ) Exclusive	( ) Exclusive	( ) Concurrent	( ) Concurrent
|	|	|	|
Exclusive	Concurrent	Exclusive	Concurrent

Rules in the brotherhood relationship of state attributes

( ) Exclusive - Exclusive ( ) Concurrent - Concurrent (×) Exclusive - Concurrent

Exclusive attribute means only one of the brothers becomes active at a time. In other words, all brothers have to be exclusive. Therefore, exclusive and concurrent states cannot be brothers.

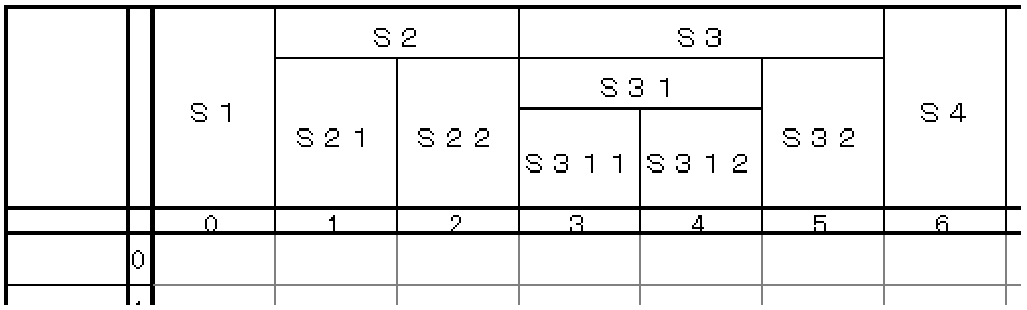


Figure 5- 3 Exclusive state

The parent-and-child and brotherhood relationships of the above STM can be represented by the following state tree.

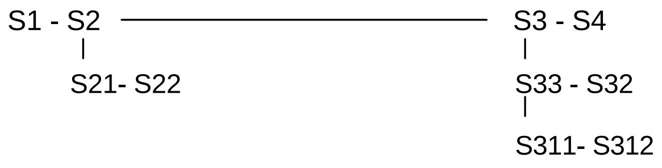


Figure 5- 4 State tree

There are four errors in the following STM:

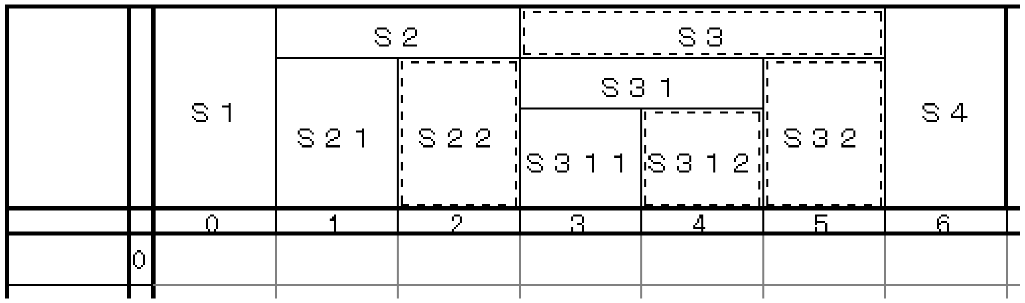


Figure 5- 5 Erroneous states

(1) S3 is concurrent. S3 should be exclusive, or S1, S2 and S4 should be concurrent.

(2) S22 is concurrent. S22 should be exclusive or S21 concurrent.

(3) S32 is concurrent. S32 should be exclusive or S31 concurrent.

(4) S312 is concurrent. S312 should be exclusive or S311 concurrent.

5.3.6 State actual frame

The state actual frame is a state frame that can drive actions by events and make transitions. State frames cannot drive actions by events. The state virtual frames (S2, S3 and S31) described in 5.3.5 are unrelated to the execution of actions. Only the states (i.e., the ones having state numbers) are related to actions.

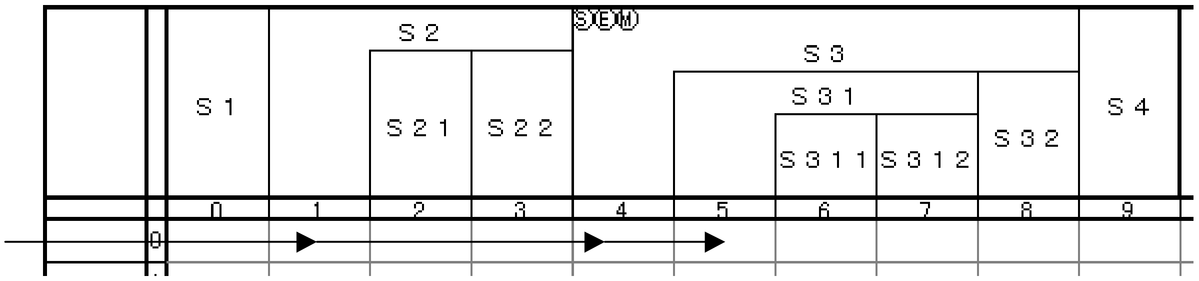


Figure 5- 6 State actual frame

S2, S3 and S31 are state actual frames. Since S21 and S22, which are the child states of S2, are exclusive brother states, only one of them is executed. The rule that determines which one is executed is explained in Chapter 7, "Transition." Since S31 and S32, which are the child states of S3, are exclusive brother states, only one of them is executed. When S31 is executed, one of the child states, S311 or S312, is executed.

5.3.7 State start activity

This process is executed when a transition is made to the applicable state (when the applicable state becomes active). An "S" is attached to indicate that activity exists. See S3 in Figure 5-6 for the format.

5.3.8 State mode activity

This process is executed as long as the applicable state is active. An "M" is attached to indicate that activity exists. See S3 in Figure 5-6 for the format.

5.3.9 State end activity

This process is executed when a transition occurs from the applicable state (when the applicable state becomes inactive). An "E" is attached to indicate that activity exists. See S3 in Figure 5-6 for the format.

5.3.10 Default state

The default state specifies which state of the brother states becomes active. When there is no default mark (τ), the state with the lowest state number becomes the default state. The default state of the STM in Figure 5-6 is S1 (state number: 0). The state that becomes active first shall be S1. When the transition to S2 occurs, of the child states of S2 (S21, S22), the S21 becomes active by default.

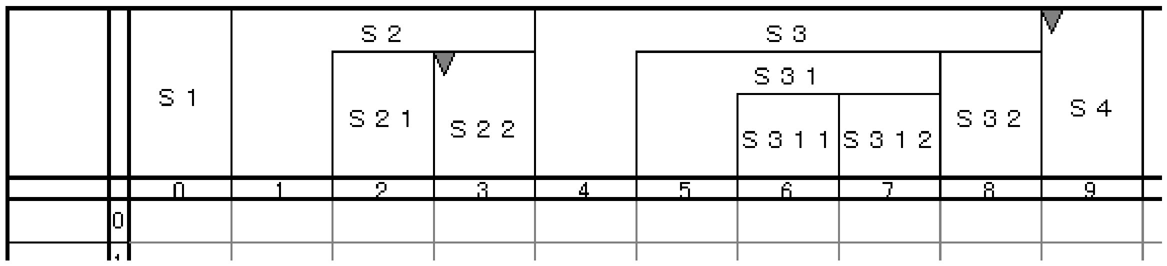


Figure 5- 7 Default state

🕮: Several types can be specified for transitions. Depending on the type, the default may be valid or invalid. Refer to 7, "Transition," for details.

5.3.11 State hierarchy STM call

The state hierarchy is described in the state cell as follows:

STM level number

STM name

The state hierarchy STM can be called only from a state having a state number. Others cannot call this. State frames and state actual frames cannot call it, either. The hierarchy states are implemented via the state tree. The state tree shown in Figure 5-4 is described below using the state hierarchy.

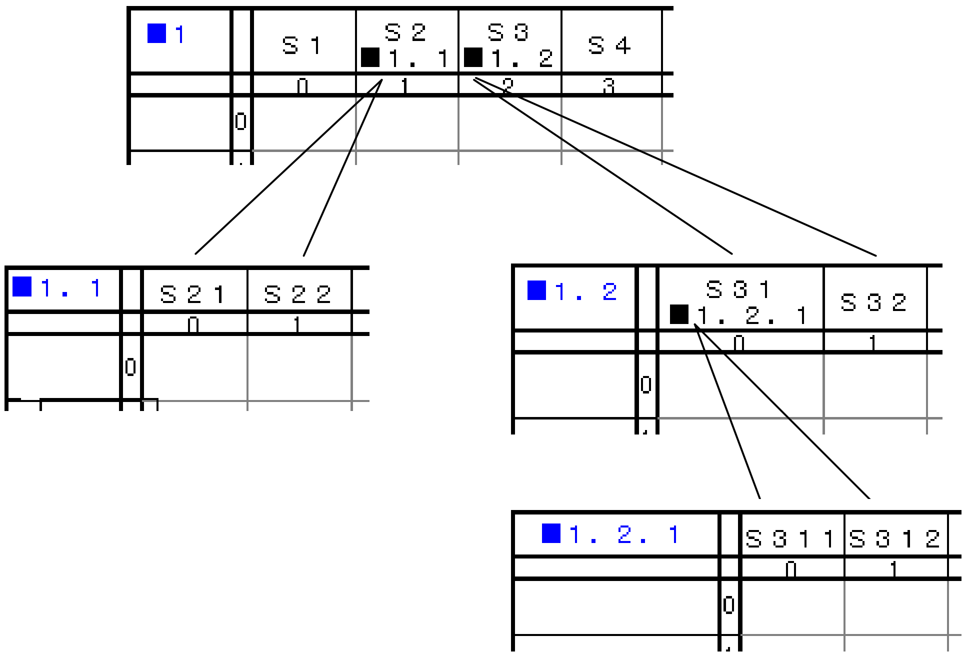
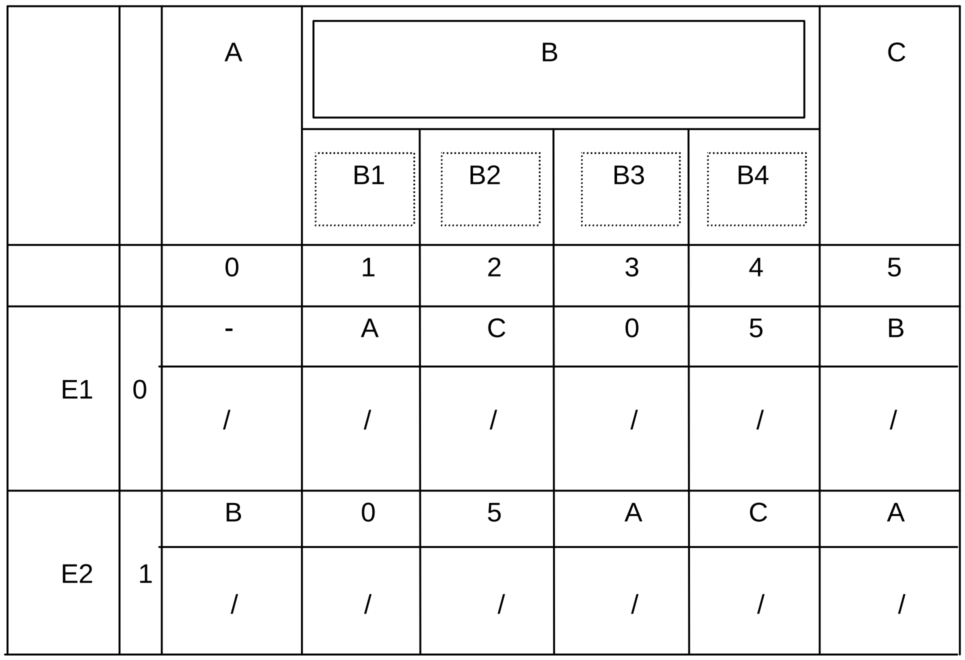


Figure 5- 8 State hierarchy STM

: The details of hierarchy are explained in 15, "Hierarchy."

5.3.12 Synchronized state

The synchronized state is a concurrent state that performs transition synchronous to the transition from a concurrent state. A synchronized state is represented by an exclusive parent state frame surrounded by solid lines.



		A	B				C
			B1	B2	B3	B4	
		0	1	2	3	4	5
E1	0	-	A	C	0	5	B
		/	/	/	/	/	/
E2	1	B	0	5	A	C	A
		/	/	/	/	/	/

Figure 5- 9 Synchronized state

The exclusive state "B" is a synchronized state. The "B1" synchronized state makes a state transition to "A" via event "E1." Normally, a transition occurs to the "A" state at this time. However, since the synchronized state is specified as the parent's exclusive frame, "B1" waits until all of the brother concurrent states "B2/B3/B4" make transitions to state "A" (state number: 0). What happens if "B2" accepts "E1" in this state? The action will be executed but the transition to state "C" will be ignored, because "B1," which is a synchronized state, is already waiting for the synchronization to state "A." State "B1" will keep waiting until the states "B2," "B3" and "B4" make transitions to state "A," resulting in a deadlock since "B2" and "B4" will have no transitions to state "A." The deadlock can be released by specifying a forced transition. This kind of deadlock should be statically detected by a CASE tool that supports the EHSTM.

The priority of the synchronized transition destination is determined on a first-come, first-serve basis.

📖: Synchronized transition and forced transitions are explained in 7.3, "Synchronized transition."

6. Action

An action is a function or process that is executed when the specific event occurs during the specific state. The action cell is a cell in which the event cell and the state cell cross. While the "what" (function) is normally described in the action, "how" (processing) can also be described. A function is described as the action in the example phone call in Chapter 2. It is also allowed to describe how to process the function of "making a phone call" in the action cell.

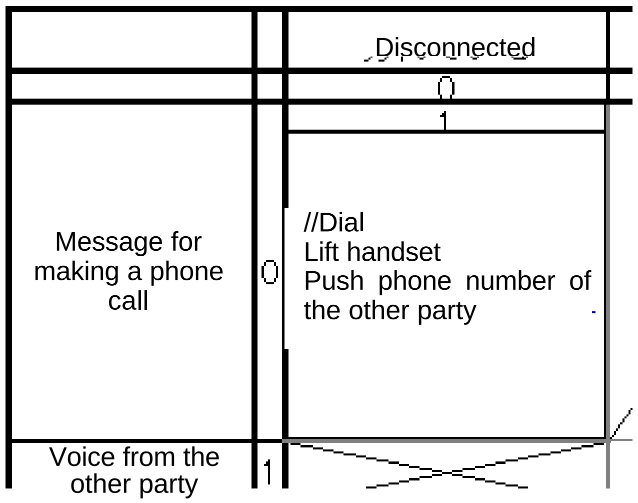

Disconnected
0
1
Message for making a phone call
0
//Dial
Lift handset
Push phone number of the other party
Voice from the other party
1

Figure 6- 1 Action that describes processing

6.1. Format of action cell

The following information can be described in the action cells:

1: Human language statement
2: Computer language statement
3: NS chart
4: Invalid
5: No use
6: Event hierarchy STM call
7: Subroutine STM call
8: Library STM call
9: Divided action cell
10: EHSTM system call
11: Transition symbol
12: Don’t care

6.1.1. Human language statement

A function or process is described in human language in the action cell. The range of usable Actionhuman language statements is not specified in this document, but depends on the specifications of the CASE (Computer-Aided Software Engineering) tool that supports the EHSTM design method.

6.1.2. Computer language statement

A function or process is described using computer language in the action cell. The range of usable computer language statements is not specified in this document, but depends on the specifications of the CASE tool that supports the EHSTM design method.

6.1.3. NS chart

The NS chart can be described in the action cell. The NS chart is a documentation technique for the structured approach. In the NS chart, three basic logical structures (sequence, if then else, do while) are represented by the following symbols. [1]The range of usable NS charts is not defined in this document. It depends on the specifications of the CASE tool that supports the EHSTM design method.

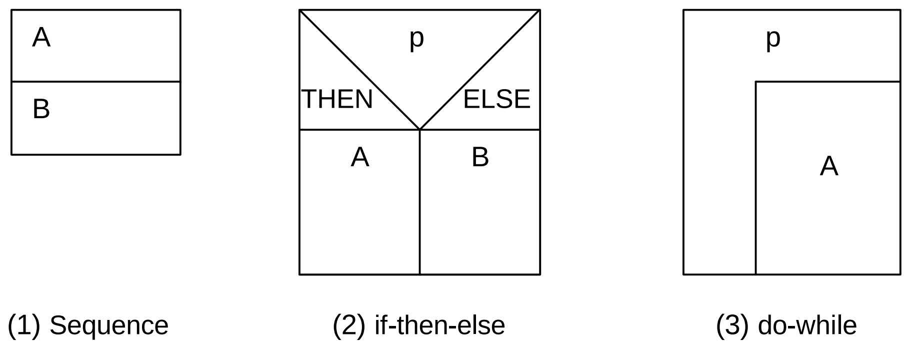


Figure 6- 2 Basic symbols of NS chart (A and B: functions; p: test condition)

6.1.4. Invalid

“Invalid” is represented by an “X" mark. The invalid indicates an event and state combination that can never occur. Therefore, a debugging process such as halting the system needs to be implemented in case the invalid action cell is called. In the example of a phone call in Figure 2, the "voice from the other party" and "busy tone" events never occur in the "disconnected" state. Therefore, invalid (X) is written in the action cell.

[1] Kunitomo, Yoshihisa, "Introduction to the structured programming," Ohm-sha press

6.1.5. No use

No use is indicated by the "/" symbol. No use means the event and state combination can exist but no processing is executed. In the example phone call in Figure 2-2 in Chapter 2, even if a "Message for making a phone call" occurs under the "Busy" state, it is ignored (/).

6.1.6. Event hierarchy STM call

The event hierarchy STM call is described as follows.

STM level number <option>
STM name <option>

Unlike the state hierarchy STM call, the event hierarchy STM call can call multiple child event hierarchy STMs. The STM of a phone call in Figure 2-2 is shown as an event hierarchy.

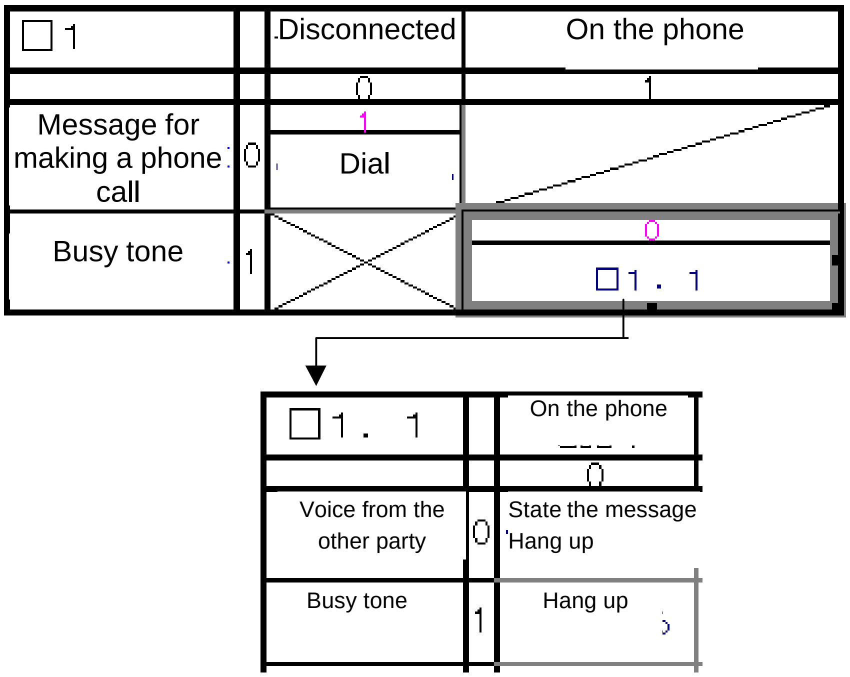


Figure 6- 3 Event-hierarchy STM call

The event hierarchy constructs a tree of events. The event tree of Figure 6-3 is shown as follows, and can be represented not only by hierarchy but also by an event frame.

Sound heard from the phone - Voice from the other party
└ Busy tone

Figure 6- 4 Event tree

Three items can be specified for the option: argument, clone number and event number.

(1)Argument
By specifying arguments, information can be exchanged between the calling STM and the called STM.

```
Tel(&para1, para2)
```

The description is the same as that of the arguments in the C programming language. When the argument type is necessary, declare it in the STM on the called side.

```
Tel(int *ip1, int i2)
```

(2)Clone number
The clone STM is treated as an array in C. The clone number is an array index. Specify an index, assuming the STM as an array.

```
1.1[2]
```

The array index starts from 0. The array size for the STM is declared on the called side.

```
1.1[3]
```

(3)Event number
By specifying the event number, a call can be made so that the specified event number is driven without going through the event analyzer of the called STM. This is called the direct event number call. Insert a colon (:) after the name or level number of the STM to be called, then specify the event number after that.

```
tel:0
```

No declaration is necessary on the called side.
The event number starts with 0 or 1.
(Use of STMs starting with 0 and those starting with 1 mixed in the same project may easily lead to confusion.)

The composite format for options is described in the order of (1) clone number, (2) event number and (3) argument.

```
1.1[0]:0(p1,p2,p3)
```

🕮: The hierarchy is detailed in 15, "Hierarchy." The details of the clone are explained in 13, "Clone STM." The details of the event analyzer are explained in 9, "Event analyzer."

6.1.7. Subroutine STM call

Subroutine STM calls are written as the following:

ΔSubroutine STM level number <option>
ΔSubroutine STM name <option>

Δ100[0]:0(p1,p2,p3)

For the specifications of options, refer to Section 6.1.6, "Event hierarchy STM call," since they are the same.

🕮: The details of the subroutine STM will be explained in 16, "Subroutine STM."

6.1.8. Library STM call

The library STM calls are written as the following:

library STM level number (function name <option>)
library STM name (function name <option>)
(*)The event number cannot be specified via a library call.
A function name is written instead of the event number.
JPEG[0]:jpeg_Compress(p1,p2,p3)

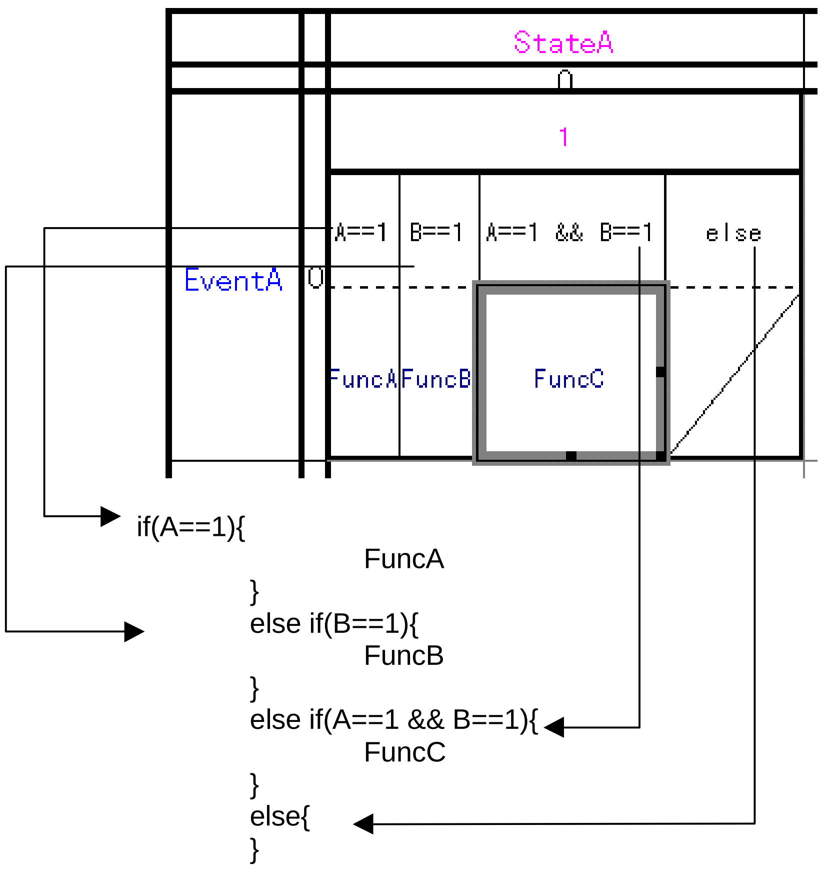
The STM level number and name can be omitted only when the call is made with the argument options only.

jpeg_Compress(p1,p2,p3)

🕮: The details of the library STM are explained in 12.4, "Library STM."

6.1.9. Divided action cell

The divided action cell is an action cell split into multiple cells, according to the condition.



```
if(A==1){
                    FuncA
          }
          else if(B==1){
                    FuncB
          }
          else if(A==1 && B==1){
                    FuncC
          }
          else{
          }
```

Figure 6- 5 Divided action cell

The rule for determining the condition of the divided action cell is the "if-else" method. In the example above, FuncC will never be executed. To change to the "if-if" method, describe it using the NS chart or directly write the C code.

6.1.10. EHSTM system call

The EHSTM system calls are not represented by symbols but by writing specific actions by words (sentences) such as “system call.”
The EHSTM system calls will be explained again in 16, "EHSTM system call." Refer to that section for details.

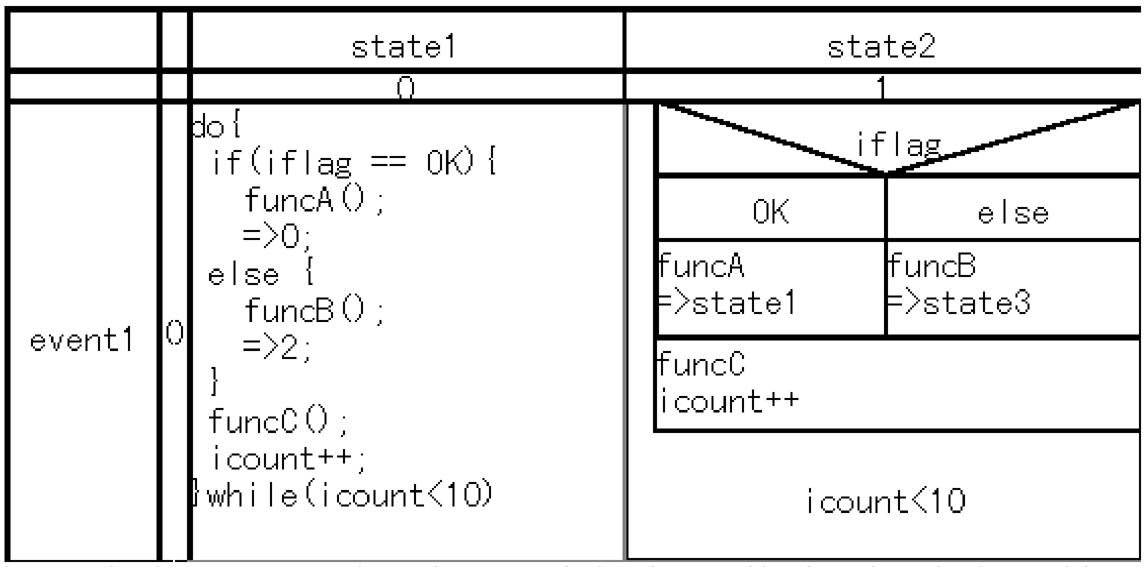
6.1.11. Transition symbol

By using the transition symbol (=>), the transition destination can be specified without writing in the transition cell. When the transition symbol is found, actions written after the transition symbol are executed after the activity (end and start) is operated.

=> transition destination

A local or global transition can be specified as the transition destination.

Figure 6- 6 Transition Symbol



```
do{
 if(iflag == OK){
   funcA();
   =>0;
 else {
   funcB();
   =>2;
 }
 funcC();
 icount++;
}while(icount<10)
```

In the STM in Figure 6-6, every funcC is executed after the transition is activated. If you wish not to execute the rest of the process after the transition is performed, it is necessary to write a return statement in the action cell of the STM.

: The details of the transition are explained in 7, "Transition."

6.1.12. Don't care

Don't care is represented by a dot (.). The meaning is the same as no use (/).

7. Transition

“Transition” means a transfer from one state to another state. In the example of a phone call in Figure 2-2, the transition from the "disconnected" state to the "busy" state occurs via the "message for making a phone call" event after executing the "dial" action.

7.1. Transition types

In the EHSTM design method, the transitions are defined in three types.[1]

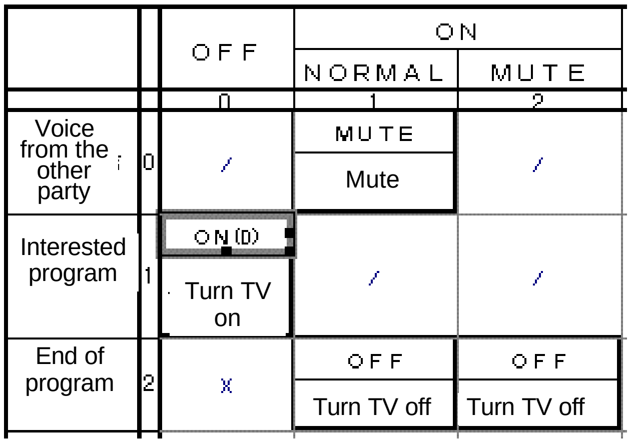
(1) Fixed
(2) Memorized
(3) Deep-Memorized

The transition type specifies which child state to activate when its parent state is specified as the transition destination. If a childless state is specified as the transition destination, there is no difference in the action when any type is specified. A state name or state number is specified as the transition destination. The specified name has to be a unique name.
When the transition type is omitted, the deep memorized is used.

7.1.1 Fixed transition

The fixed transition is written in the following format:

Transition destination state name (D)
Transition destination state number (D)

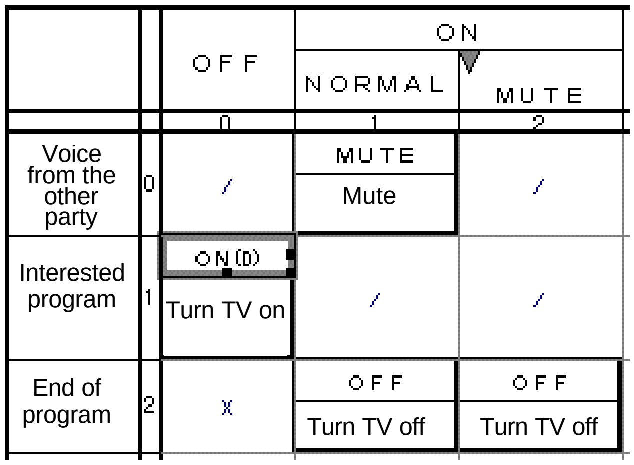


		OFF	ON	
			NORMAL	MUTE
		0	1	2
Voice from the other party	0	/	MUTE Mute	/
Interested program	1	ON(D) Turn TV on	/	/
End of program	2	X	OFF Turn TV off	OFF Turn TV off

Figure 7- 1 Fixed transition (1)

In the "OFF" state upon "Interested program" event, transition to the "ON(D)" state will occur after the "Turn TV on" action. This means that the default state of "NORMAL" or "MUTE" will be selected after the transition to the "ON" state occurs. In other words, since there is no default mark (τ), in this situation "NORMAL" will become the active state via the priority given to the smaller number.

[1] Rapid Simulation & Training



		OFF	ON	
			NORMAL	▼ MUTE
		0	1	2
Voice from the other party	0	/	MUTE Mute	/
Interested program	1	ON (D) Turn TV on	/	/
End of program	2	X	OFF Turn TV off	OFF Turn TV off

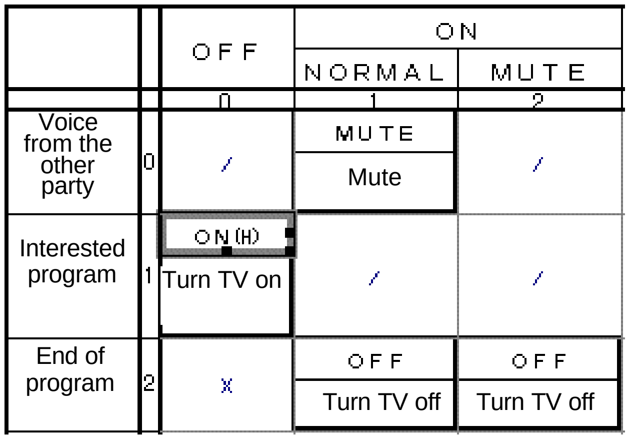
Figure 7- 2 Fixed transition (2)

If there is a default mark on "MUTE," the active state will be "MUTE" after transition to the "ON" state.

7.1.2 Memorized transition

The memorized transition is written in the following format:

Transition destination status name (H)
Transition destination status number (H)



		OFF	ON	
			NORMAL	MUTE
		0	1	2
Voice from the other party	0	/	MUTE Mute	/
Interested program	1	ON (H) Turn TV on	/	/
End of program	2	X	OFF Turn TV off	OFF Turn TV off

Figure 7- 3 Memorized transition (1)

The default state is ignored by the memorized transition. The child state, which was active in the past, will be activated. The default is used only when no state has been active.

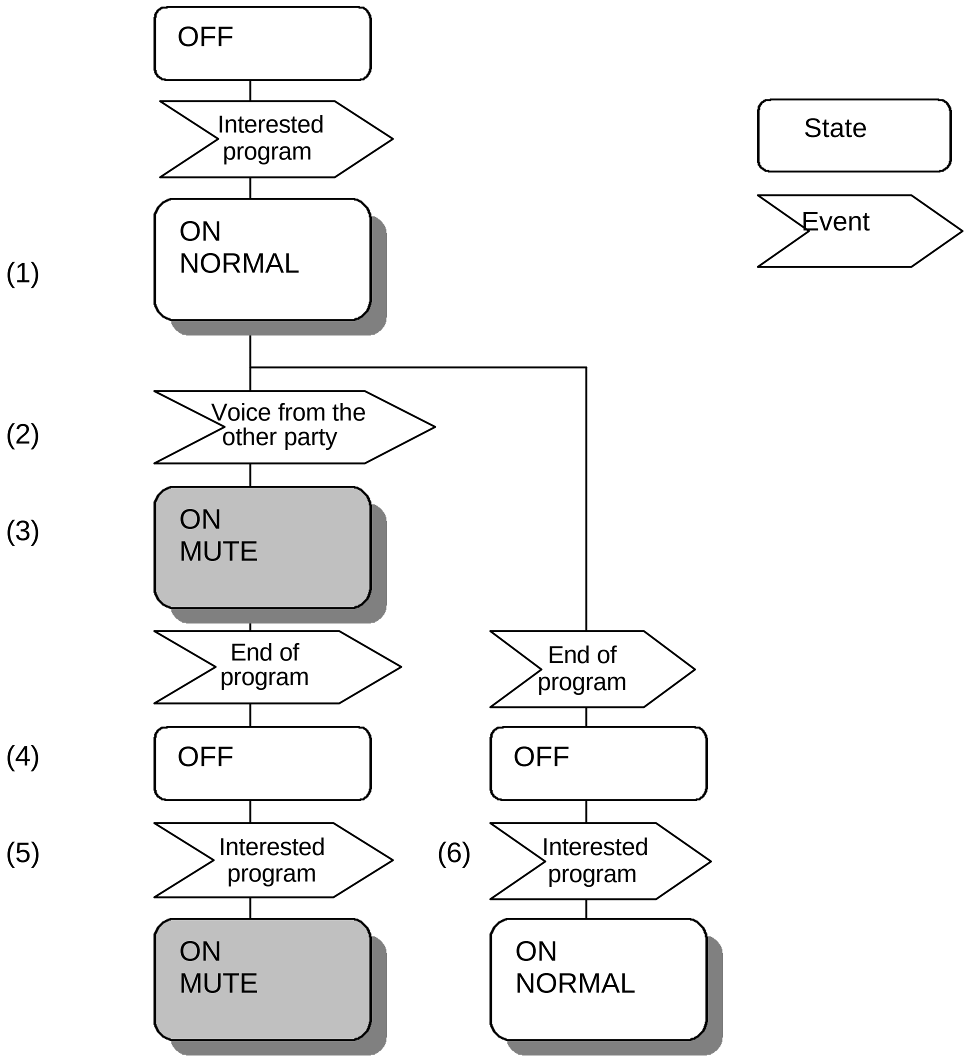


Figure 7- 4 Memorized transition SDL

(Functional Specification and Description Language)

(1) The "NORMAL" state is activated by default at the beginning.
(2) When the "Voice from the other party" event occurs, the "MUTE" state is activated.
(3) At this stage, the "ON" parent state will change the history of the child state to "MUTE."
(4) Again, the "Interested program" event occurs.
(5) Because the transition destination type is the memorized type (H), a transition occurs to "MUTE," which is the recorded history.
(6) When the "Voice from the other party" event does not occur, the “ON” parent state keeps the child state history at "NORMAL," so the transition is made to "NORMAL."

This requires only a change of the transition type to the memorized type, so there is no need for the transition requester side to be conscious about what was active last time. Also, if transition by default is desired, all you need to do is change the transition type to the fixed type.

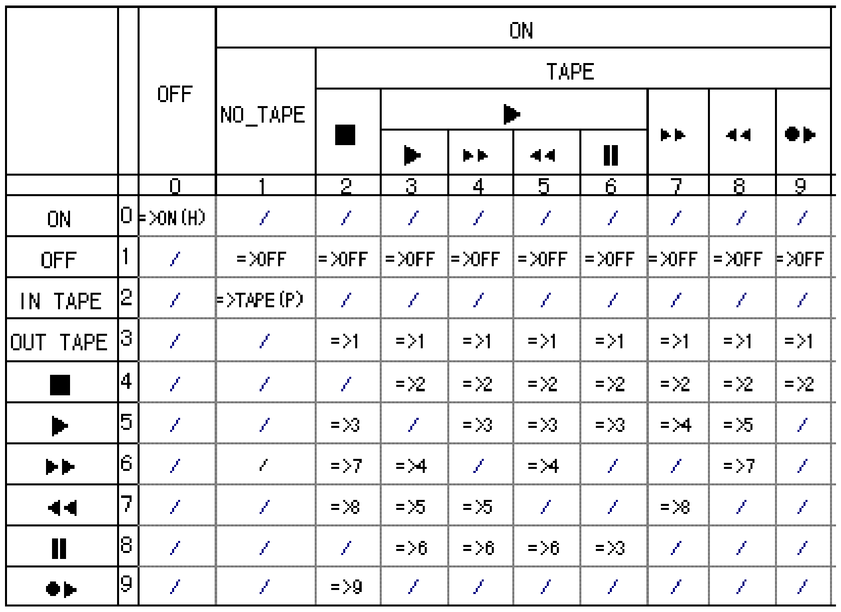
7.1.3 Deep-memorized transition

The deep-memorized transition is written in the following format:

Transition destination state name (P)

Transition destination state number (P)

The deep-memorized transition is applied when the transition type is not explicitly specified.
(*) Note that deep-memorized transition is applied, not fixed transition, when the transition type is omitted.



		OFF	ON								
			NO_TAPE	TAPE							
				■	▶				▶▶	◀◀	●▶
					▶	▶▶	◀◀	❚❚			
		0	1	2	3	4	5	6	7	8	9
ON	0	=>ON(H)	/	/	/	/	/	/	/	/	/
OFF	1	/	=>OFF	=>OFF	=>OFF	=>OFF	=>OFF	=>OFF	=>OFF	=>OFF	=>OFF
IN TAPE	2	/	=>TAPE(P)	/	/	/	/	/	/	/	/
OUT TAPE	3	/	/	=>1	=>1	=>1	=>1	=>1	=>1	=>1	=>1
■	4	/	/	/	=>2	=>2	=>2	=>2	=>2	=>2	=>2
▶	5	/	/	=>3	/	=>3	=>3	=>3	=>4	=>5	/
▶▶	6	/	/	=>7	=>4	/	=>4	/	/	=>7	/
◀◀	7	/	/	=>8	=>5	=>5	/	/	=>8	/	/
❚❚	8	/	/	/	=>6	=>6	=>6	=>3	/	/	/
●▶	9	/	/	=>9	/	/	/	/	/	/	/

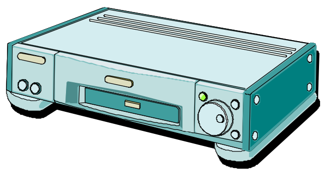


Figure 7- 5 STM for a videocassette recorder (1)

Through deep-memorized transition, the parent state activates (makes a transition to) the descendant states based on the history of all descendant states. In the memorized transition, the parent state executes a transition based on the history of its own child only, while the fixed transition is applied to the grandchildren and their descendants.
The STM for a videocassette recorder in Figure 7-5 is explained below as an example.

1 Initially "OFF" as the default state.
2 "ON" occurs.
3 Memorized transition (H) occurs to the "ON" state.
4 Since there is no history yet (none has been activated), transition is made to "NO_TAPE," which is the default state.
5 "IN TAPE" occurs.
6 Deep-memorized transition (H) occurs to the "TAPE" state.
7 Since there is no history yet, transition is made to the "■" (STOP) state, which is the default.

8 " ▶ " (PLAY) occurs.
9 Transition to state number 3.
10 "▶▶" (FAST FORWARD) occurs.
11 Transition to state number 4.
12 "OUT TAPE" occurs.
13 Transition to state number 1.
14 "IN TAPE" occurs.
15 Deep-memorized transition (P) occurs toward the "TAPE" state.
16 The history of states whose parents are "TAPE" will look like the following diagram at this time. The states indicated by ⊙ are the ones that were active previously.

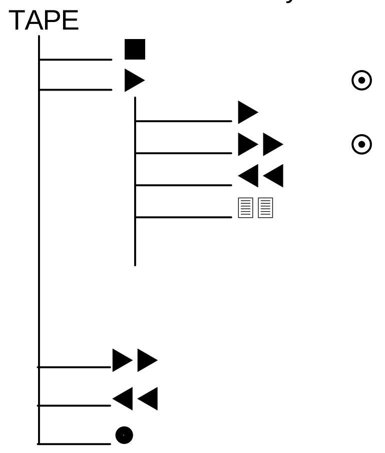
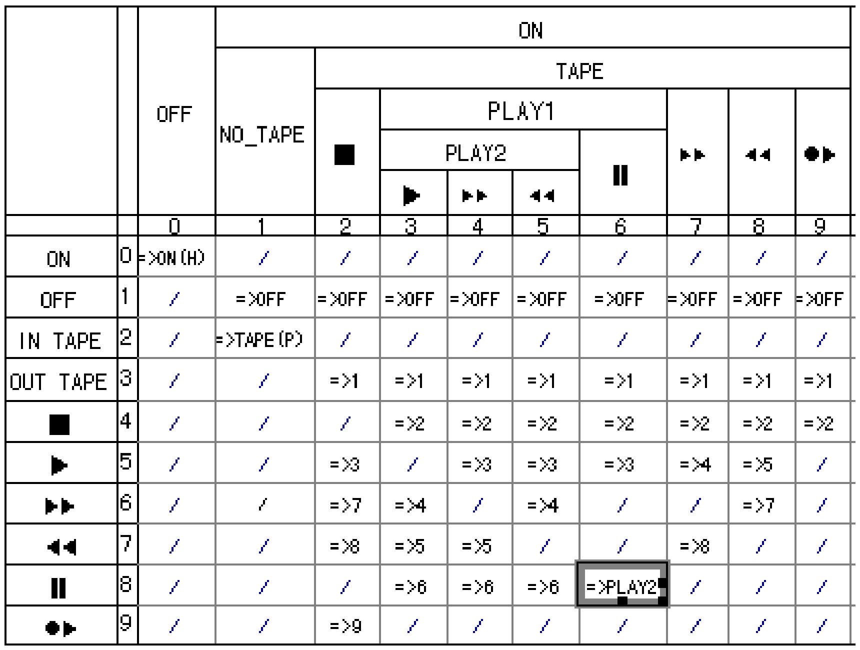


Figure 7- 6 State tree

17 " ▶▶ " of state number 4 becomes active.

To avoid the continuation of the previous operation when the tape is inserted, change
=>TAPE(P)
to
=>TAPE(D)
and it will always start from the default "■" (STOP) state.

In the current design, the PAUSE(▤▤) will be executed during PLAY, FAST FORWARD or REWIND, but it will return only to the PLAY state when it is released. In order to return to the previous state, add a virtual state frame and execute a deep-memorized transition for the state frame.



		OFF	ON								
			NO_TAPE	TAPE							
				■	PLAY1				▶▶	◀◀	●▶
					PLAY2			❚❚			
					▶	▶▶	◀◀				
		0	1	2	3	4	5	6	7	8	9
ON	0	=>ON(H)	/	/	/	/	/	/	/	/	/
OFF	1	/	=>OFF	=>OFF	=>OFF	=>OFF	=>OFF	=>OFF	=>OFF	=>OFF	=>OFF
IN TAPE	2	/	=>TAPE(P)	/	/	/	/	/	/	/	/
OUT TAPE	3	/	/	=>1	=>1	=>1	=>1	=>1	=>1	=>1	=>1
■	4	/	/	/	=>2	=>2	=>2	=>2	=>2	=>2	=>2
▶	5	/	/	=>3	/	=>3	=>3	=>3	=>4	=>5	/
▶▶	6	/	/	=>7	=>4	/	=>4	/	/	=>7	/
◀◀	7	/	/	=>8	=>5	=>5	/	/	=>8	/	/
❚❚	8	/	/	/	=>6	=>6	=>6	=>PLAY2	/	/	/
●▶	9	/	/	=>9	/	/	/	/	/	/	/

Figure 7- 7 STM for a videocassette recorder (2)

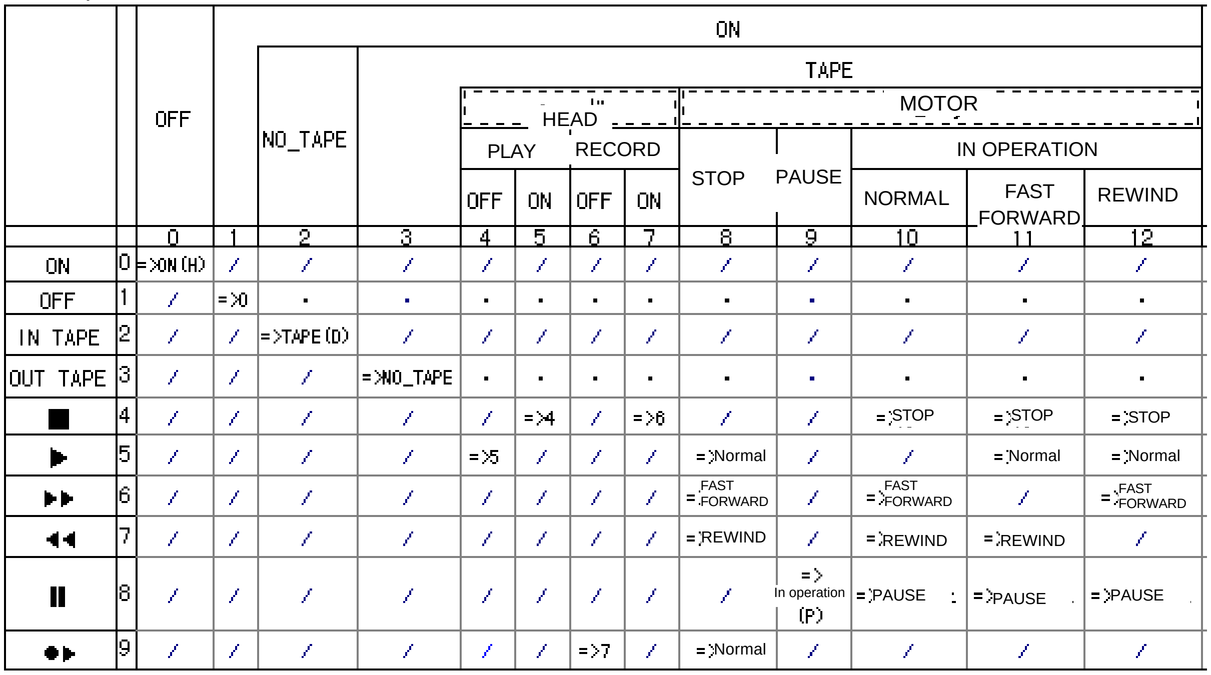
When the parents have only one generation of children (when there are no grandchildren), the deep-memorized transition and memorized transition have the same meaning.

7.2. Transition and state type

Any states or state frames except for the following two can be specified as transition destinations.
(1) Concurrent state
(2) Child states of concurrent brothers
The transition can be synchronized for concurrent states.

7.2.1 No transition to concurrent state

Transition to a concurrent state (including state frames) is not allowed. The concurrent state is a state that can always be active along with another concurrent state. Making a transition means to activate the transition destination.
Let's take the concurrent state STM in Figure 5-1 as an example. "Telephone" and "TV" are concurrent states, and it is not that only one of them can be active; both of them are active at the same time. Therefore, a transition to one of "Telephone" or "TV" (activate) cannot happen.
The STM in Figure 7-8 was created from the viewpoint of the videocassette recorder's manufacturer. The STMs in Figures 7-5 and 7-7 were created from the viewpoint of the user who operates the videocassette recorder.



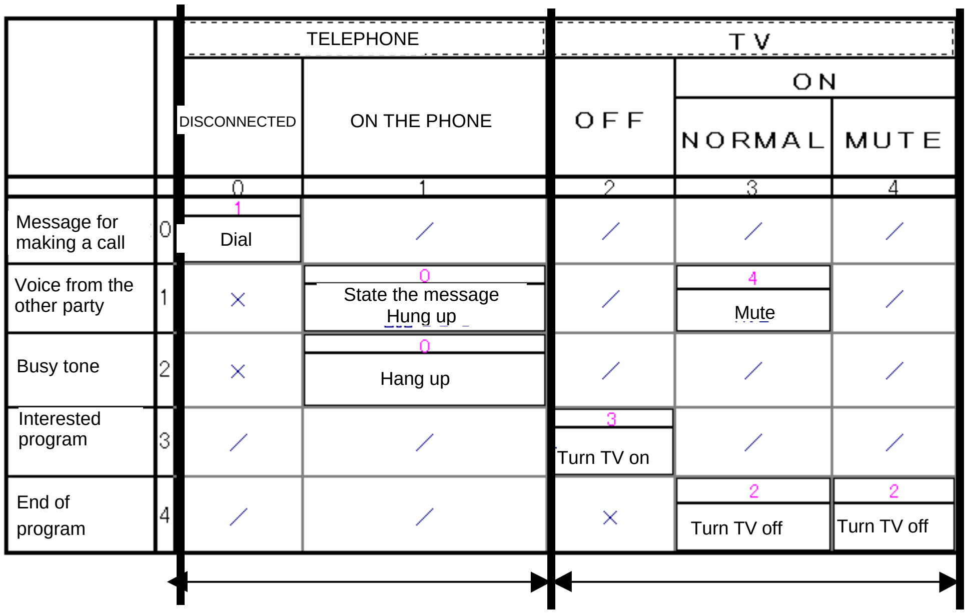
		OFF	ON											
				NO_TAPE	TAPE									
						HEAD				MOTOR				
						PLAY		RECORD		STOP	PAUSE	IN OPERATION		
						OFF	ON	OFF	ON			NORMAL	FAST FORWARD	REWIND
		0	1	2	3	4	5	6	7	8	9	10	11	12
ON	0	=>ON(H)	/	/	/	/	/	/	/	/	/	/	/	/
OFF	1	/	=>0	·	·	·	·	·	·	·	·	·	·	·
IN TAPE	2	/	/	=>TAPE(D)	/	/	/	/	/	/	/	/	/	/
OUT TAPE	3	/	/	/	=>NO_TAPE	·	·	·	·	·	·	·	·	·
■	4	/	/	/	/	/	=>4	/	=>6	/	/	=>STOP	=>STOP	=>STOP
▶	5	/	/	/	/	=>5	/	/	/	=>Normal	/	/	=>Normal	=>Normal
▶▶	6	/	/	/	/	/	/	/	/	=>FAST FORWARD	/	=>FAST FORWARD	/	=>FAST FORWARD
◀◀	7	/	/	/	/	/	/	/	/	=>REWIND	/	=>REWIND	=>REWIND	/
❚❚	8	/	/	/	/	/	/	/	/	/	=> In operation (P)	=>PAUSE	=>PAUSE	=>PAUSE
●▶	9	/	/	/	/	/	/	=>7	/	=>Normal	/	/	/	/

Figure 7- 8 STM for a videocassette recorder (3)

The "HEAD" and "MOTOR" are concurrent states and are activated concurrently, so neither of them can alone be specified as the transition destination. However, transition to the "TAPE" status, which is the parent of "HEAD" and "MOTOR," is allowed because it is an exclusive state. When a transition to "TAPE" occurs, "HEAD" and "MOTOR" are activated regardless of the transition type for "TAPE."

7.2.2 No transition to concurrent brother state

Concurrent brothers cannot make transitions to a child of the others.



		TELEPHONE		T V		
					O N	
		DISCONNECTED	ON THE PHONE	O F F	N O R M A L	M U T E
		0	1	2	3	4
Message for making a call	0	1 Dial	/	/	/	/
Voice from the other party	1	×	0 State the message Hung up	/	4 Mute	/
Busy tone	2	×	0 Hang up	/	/	/
Interested program	3	/	/	3 Turn TV on	/	/
End of program	4	/	/	×	2 Turn TV off	2 Turn TV off

Figure 7- 9 No transition to concurrent brother state

"Hang up after telling the message" when the "Voice from the other party" is heard while "On the phone."
"Mute the TV sound " when the "Voice from the other party" is heard while the "TV sound is turned on."
These two actions are processed concurrently. The "TV" state is ignored during the "Telephone" state, while the "Telephone" state is ignored during the "TV" state. If one has to worry about the other, they are not concurrent states.
In the example of the videocassette recorder in Figure 7-8, "HEAD" and "MOTOR" do not interfere with each other in the transition. This is because the same event can be obtained in concurrent states.
Note that it does not mean the concurrent state cannot be exited. Transition to the exclusive state, other than the child states of the concurrent brothers, is allowed.

7.3. Synchronized transition

A synchronized transition is also allowed from the concurrent state within a synchronized state. However, the synchronized state synchronizes to all the concurrent states.

		A	B				C
			B1	B2	B3	B4	
		0	1	2	3	4	5
E1	0	/	=>A	=>C	=0	=>5	=>B
E2	1	B	=>0	=>5	=>A	=>C	=>A

Figure 7- 10 Deadlock

A deadlock occurs in Figure 7-10 because the transition does not occur unless "B1/B2/B3/B4" are all synchronized. Therefore, the synchronized transition is used. The synchronized transition and synchronized state cannot be used at the same time.

7.3.1 Synchronization number

		A	B				C
			B1	B2	B3	B4	
		0	1	2	3	4	5
E1		/	=>A(*1)	=>C(*2)	=>0 (*1)	=>5(*2)	=>B
E2		B	=>0 (*1)	=>5(*2)	=>A (*1)	=>C(*2)	=>A

Figure 7- 11 Synchronized transition

The synchronized transition is indicated by (*n). The n is called a synchronization number. Synchronization works only with states having the same synchronization number. Therefore, when a transition of "B1" to "A" occurs after accepting "E1," a transition of "B3," which has the same synchronization number, to the "A" state will be waited. During this wait, "B2" and "B4" will accept events and execute corresponding actions, but transitions will not occur. To disable the actions also, use the optional specifications of the CASE tool that supports the EHSTM.
The synchronized transition is performed on a first-come, first-served basis. When "B2" executes "E1" first, for example, it waits for the synchronization of the "C" state.
The contention of transition type is also solved on a first-come, first-served basis. If a wait for synchronization occurs in a fixed transition first, the transition type after synchronization is ignored even if it is a memorized type, and a fixed transition is performed.

7.3.2 Wildcard transition

A wildcard transition can be specified for the synchronized transition. A wildcard transition is represented by (**).

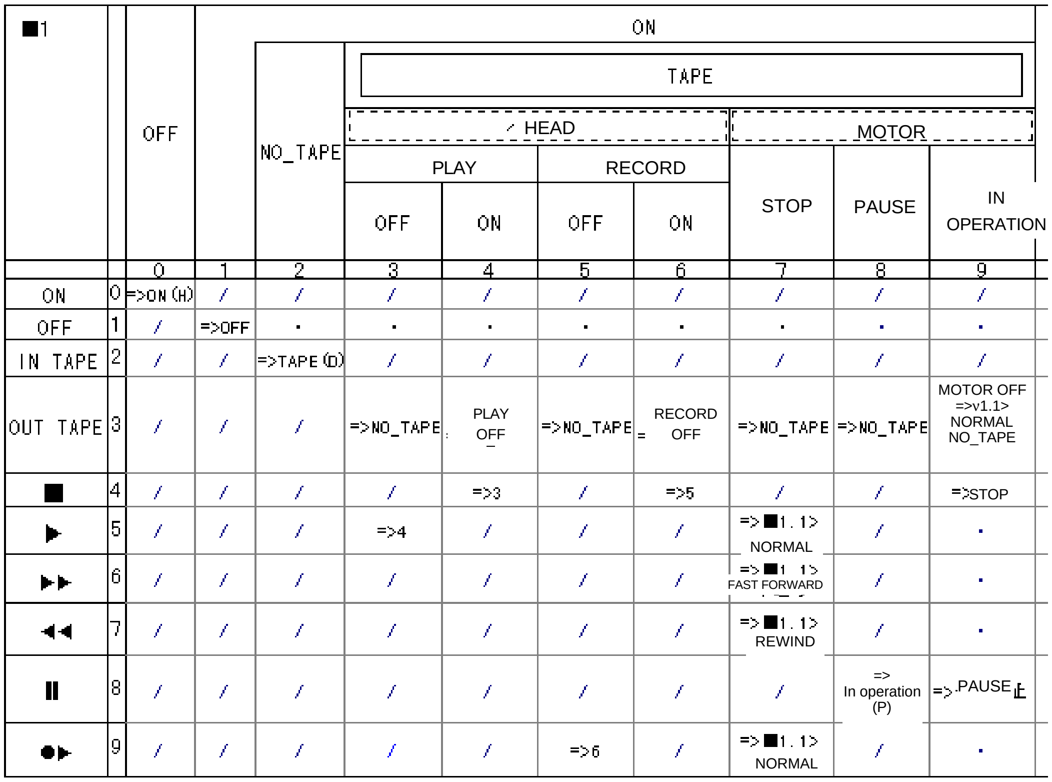
		A	B				C
			B1	B2	B3	B4	
		0	1	2	3	4	5
E1	0	/	=>A(*1)	=>C	=>0(*1)	=>5	=>B
E2	1	B	=>0(*2)	=>5	=>A(**)	=>C	=>A

Figure 7- 12 Wildcard transition

The wildcard transition is a special type of synchronized transition. It has the characteristic of adjusting to all synchronization numbers. When “E1” occurs first in “B1” and transfers to "A," "B3" can synchronize to either "E1" or "E2."
If "E1" occurs first for "B1," (*2) becomes the synchronization number. Therefore, "B3" waits for "E2" to occur at the wildcard transition. When "E1" occurs for "B3," the transition is ignored even though the action is executed. The wildcard becomes the object of synchronization.
If "E2" is activated first for "B3," the one executed next becomes the synchronization number. Even if a transition without a synchronization number occurs while waiting for synchronization at the wildcard transition, the transition is ignored, since it is ignored by the synchronized transition.

7.3.3 Synchronized state and synchronized transition

The synchronized state is explained in 5.3.12, "Synchronized state." The difference between the synchronized state and synchronized transition is that the synchronized transition can specify the transition destination for the synchronization in more detail.



■1		OFF	ON								
				NO_TAPE	TAPE						
					HEAD				MOTOR		
					PLAY		RECORD		STOP	PAUSE	IN OPERATION
					OFF	ON	OFF	ON			
		0	1	2	3	4	5	6	7	8	9
ON	0	=>ON (H)	/	/	/	/	/	/	/	/	/
OFF	1	/	=>OFF	·	·	·	·	·	·	·	·
IN TAPE	2	/	/	=>TAPE (D)	/	/	/	/	/	/	/
OUT TAPE	3	/	/	/	=>NO_TAPE	PLAY OFF	=>NO_TAPE	RECORD OFF	=>NO_TAPE	=>NO_TAPE	MOTOR OFF =>v1.1> NORMAL NO_TAPE
■	4	/	/	/	/	=>3	/	=>5	/	/	=>STOP
▶	5	/	/	/	=>4	/	/	/	=> ■1. 1> NORMAL	/	·
▶▶	6	/	/	/	/	/	/	/	=> ■1. 1> FAST FORWARD	/	·
◀◀	7	/	/	/	/	/	/	/	=> ■1. 1> REWIND	/	·
❚❚	8	/	/	/	/	/	/	/	/	=> In operation (P)	=> PAUSE
●▶	9	/	/	/	/	/	=>6	/	=> ■1. 1> NORMAL	/	·

Figure 7- 13 Synchronized state example

Let's consider a practical example. See Figure 7-13. The state transitions of "HEAD" and "MOTOR," which are the concurrent child states, will be synchronized because "TAPE" is a synchronized state. Transitions within "HEAD" and "MOTOR" are unrelated to synchronization. The transitions affected by the synchronization are the ones that exit "TAPE," which is the parent state of "HEAD" and "MOTOR." In this example, =>NO_TAPE will be synchronized. Even if a transition from "HEAD" to the "NO_TAPE" state occurs first, the "HEAD" will wait until a transition of "MOTOR" to the "NO_TAPE" state occurs.

■1		OFF	ON								
				NO_TAPE	TAPE						
					HEAD				MOTOR		
					PLAY		RECORD		STOP	PAUSE	IN OPERATION ■1. 1
					OFF	ON	OFF	ON			
		0	1	2	3	4	5	6	7	8	9
ON	0	=>ON (H)	/	/	/	/	/	/	/	/	/
OFF	1	/	=>OFF	·	·	·	·	·	·	·	·
IN TAPE	2	/	/	=>TAPE (D)	/	/	/	/	/	/	/
OUT TAPE	3	/	/	/	=>NO_TAPE	PLAY OFF =>NO_TAPE (*1)	=>NO TAPE	RECORD OFF =>NO_TAPE (*1)	=>NO_TAPE	=>NO_TAPE (*1)	MOTOR OFF =>v1.1> NORMAL NO_TAPE
■	4	/	/	/	/	=>3	/	=>5	/	/	=>STOP
▶	5	/	/	/	=>4	/	/	/	=> ■1. 1> NORMAL	/	·
▶▶	6	/	/	/	/	/	/	/	=> ■1. 1> FAST FORWARD	/	·
◀◀	7	/	/	/	/	/	/	/	=> ■1. 1> REWIND	/	·
⏸	8	/	/	/	/	/	/	/	/	=> In operation (P)	=>PAUSE
●▶	9	/	/	/	/	/	=>6	/	=> ■1. 1> NORMAL	/	·

Figure 7- 14 Synchronized transition example

The synchronized transition synchronizes the specified transitions. In Figure 7-14, (*1) is specified in four locations. The synchronized transition (*1) is specified for synchronizing a transition to "NO_TAPE" while "HEAD" and "MOTOR" are in operation.
However, when "HEAD" is "OFF" or "MOTOR" is "STOP," caution must be taken when the synchronized transition and normal transition are mixed because no special synchronization is necessary. When "HEAD" is "OFF" the "PLAY" and "MOTOR" is "PAUSE," a transition to "NO_TAPE" will occur independently when "HEAD" accepts an "OUT TAPE" event, even if "MOTOR" wants to synchronize. As is shown in the example, the wildcard transition can be specified when the other end wants to synchronize even if the own side does not necessarily require synchronization.

■1		OFF	ON								
				NO_TAPE	TAPE						
					HEAD				.MOTOR		
					PLAY		RECORD		STOP	PAUSE	IN OPERATION ■1. 1
					OFF	ON	OFF	ON			
		0	1	2	3	4	5	6	7	8	9
ON	0	=>ON(H)	/	/	/	/	/	/	/	/	/
OFF	1	/	=>OFF	.	.	.	.	.	.	.	.
IN TAPE	2	/	/	=>TAPE(D)	/	/	/	/	/	/	/
OUT TAPE	3	/	/	/	=>NO_TAPE (**)	PLAY OFF =>NO_TAPE (*1)	=>NO_TAPE (**)	RECORD OFF =>NO_TAPE (*1)	=>NO_TAPE (**)	=>NO_TAPE (*1)	MOTOR OFF =>v1.1> NORMAL NO_TAPE (*1)
■	4	/	/	/	/	=>3	/	=>5	/	/	=>STOP
▶	5	/	/	/	=>4	/	/	/	=> ■1. 1> NORMAL	/	.
									–		

Figure 7- 15 Wildcard example

Specifying a wildcard transition can match with the synchronization of the other. When a normal transition occurs during a WAIT of the wildcard transition, the normal transition is ignored. When a synchronized transition (*1) occurs during a WAIT of the wildcard transition, the synchronized transition (*1) is used. The wildcard transition cannot coexist with the synchronization state, since the synchronized transition cannot.
A "-" symbol is used just like -(**) for executing synchronization only, according to the transition destination of the concurrent brother state when the transition destination cannot be decided by its own state.

7.3.4 Forced transition

The forced transition is used to make a transition happen forcibly. The forced transition can be used with the synchronized state. A forced transition to state number 0 occurs when 0(*) is used. To use a forced transition to make a forcible transition to the transition destination currently waiting for synchronization, write as -(*), as in the wildcard transition.

		A	B				C
			B1	B2	B3	B4	
		0	1	2	3	4	5
E1	0	/	=>A	=>C	=>0	=>5	=>B
E2	1	B	=>0	=>5	=>A	=>C	=>A
Timeout	2	/	=>-(*)	=>A(*)	=>0(*)	=>-(*)	/

Figure 7- 16 Forced transition

7.4. Transition range

Making a transition within its own STM (a sheet of STM) is called a local transition. A transition of another STM is called a global transition. Therefore, all transitions without hierarchies shall be local transitions. There is one transition destination for each STM. Even though multiple STM transitions can be specified according to the hierarchy, only one transition destination can be specified for each STM.
An example of the state hierarchy for the videocassette recorder in Figure 7-5 is shown in Figure 7-17.

■1		OFF	ON					
			NO_TAPE	TAPE				
				■	▶ ■1.1	▶▶	◀◀	●▶
		0	1	2	3	4	5	6
ON	0	=>ON(H)	/	/	/	/	/	/
OFF	1	/	=>OFF	=>OFF	=>OFF	=>OFF	=>OFF	=>OFF
IN TAPE	2	/	=>TAPE(P)	/	/	/	/	/
OUT TAPE	3	/	/	=>1	=>1	=>1	=>1	=>1
■	4	/	/	/	=>2	=>2	=>2	=>2
▶	5	/	/	/	·	=>3/■1.1>2	=>3/■1.1>3	/
▶▶	6	/	/	=>4	·	/	=>4	/
◀◀	7	/	/	=>5	·	=>5	/	/
❚❚	8	/	/	/	·	/	/	/
●▶	9	/	/	=>6	/	/	/	/

■1.1		▶	▶▶	◀◀	❚❚
		0	1	2	3
▶	0	/	=>0	=>0	=>0
▶▶	1	=>1	/	=>1	/
◀◀	2	=>2	=>2	/	/
❚❚	3	=>3	=>3	=>3	=>0

Figure 7- 17 STM for a videocassette recorder (4)

There are no synchronized or wildcard transitions for global transitions. This is because global transition is synonymous with forced transition, in the sense that the transition is forced. For global transition, write as 1.1>- in order to make a forced transition to the transition destination that is in a wait for synchronization.

7.5. Format of transition

Local transitions

Transition destination name (transition type specification *n)

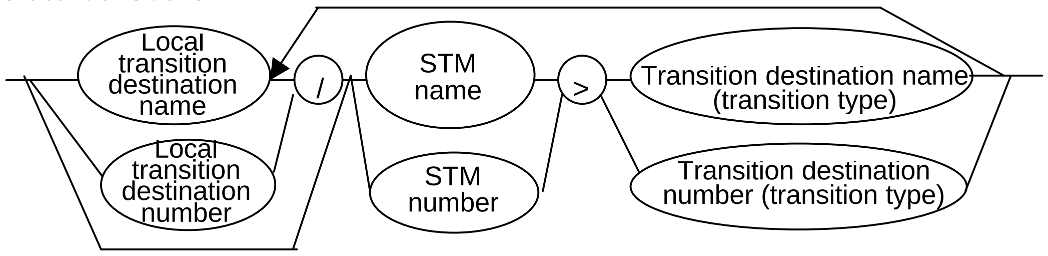
<u>The transition destination name</u> has to be a unique name.
Separate by a colon (:) if the name becomes unique in conjunction with the parent state.

state1:state2:state3

Transition destination number (transition type specification *n)

<u>Transition type specification</u>
D: Fixed type
H: Memorized type
P: Deep-memorized type
The deep-memorized type is used when omitted.
<u>***n**</u>
Forced transition by * only.
Wildcard transition when * is used for n.
n is the number that indicates the synchronization type.
Non-numeric characters are allowed in some CASE tools that support the EHSTM design method.

Global transitions


Local transition destination name
/
STM name
>
Transition destination name (transition type)
Local transition destination number
STM number
Transition destination number (transition type)

(*)The or attached to the head of STM names may be omitted.

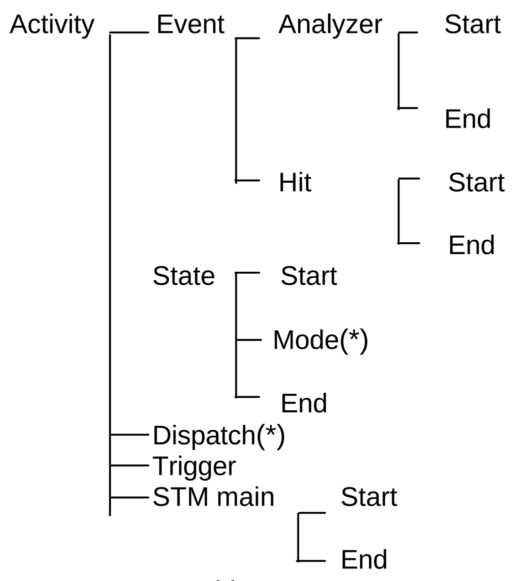
Figure 7- 18 Syntax of global transition

For transitions from the action area
=> local transition
=> global transition

Example
=>OFF
=>3/ 1.1>2

8. Activity

The activity is a function or process that is called under specific circumstances. The specific circumstances include the situation in which an event is detected or when a state transition occurs. A function or process called when the specific event occurs during the specific state is an action. Figure 8-1 shows the classification of activities.



(*)cannot be used for an event-driven STM.

Figure 8- 1 Classification of activities

8.1. Event activity

The event activities are used for imbedding regular processing related to event occurrences.

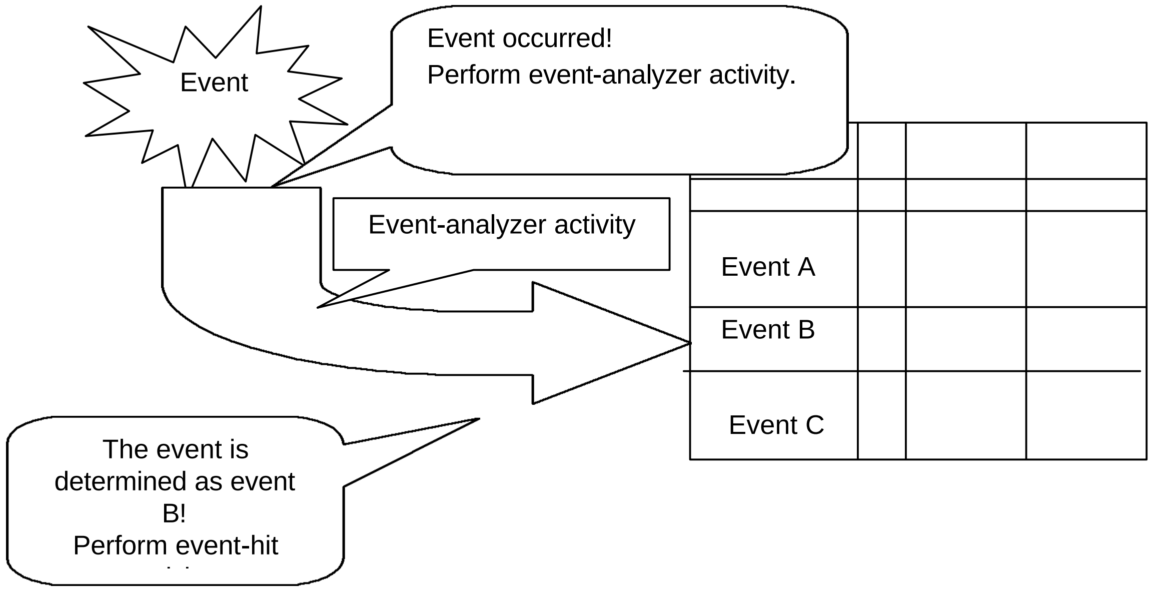
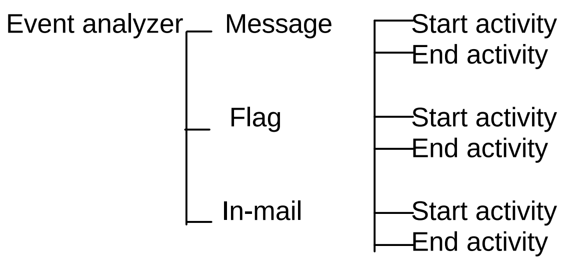
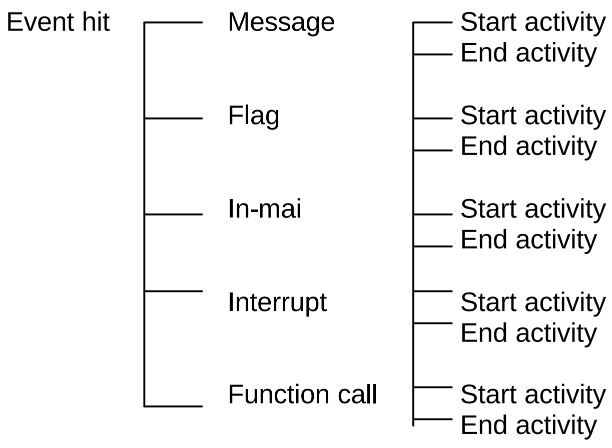


Figure 8- 2 Event activity

The event-analyzer activity begins the activity if one exists when an event has occurred or is considered to have occurred before event-analyzer processing is executed. The event-hit activity begins the activity if one exists under circumstances in which the specific event has occurred (after the event analyzer has been executed).
The event-analyzer and event-hit activities can be used together. There are three methods of event analysis, and these can be mixed. Refer to Section 4.3, "Mixed events," for more about mixed events.
There are six kinds of event-analyzer activities:



There are ten kinds of event-hit activities:

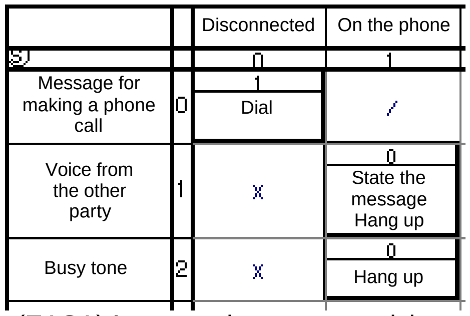


There are usable and unusable event activities, depending on the STM driven type.

📖: The details of the event analyzer and STM driven type are explained in 9, "Event analyzer," and 11, "Driven type," respectively.

8.1.1. Event-analyzer start activity

The Event Analyzer Start Activity (EASA) is executed before event analyzer processing is executed. It is always called for flag events and when the STMs driven type is the state-driven type.



		Disconnected	On the phone
S)		0	1
Message for making a phone call	0	1 Dial	/
Voice from the other party	1	X	0 State the message Hang up
Busy tone	2	X	0 Hang up

(EASA) Interrupt the current activity

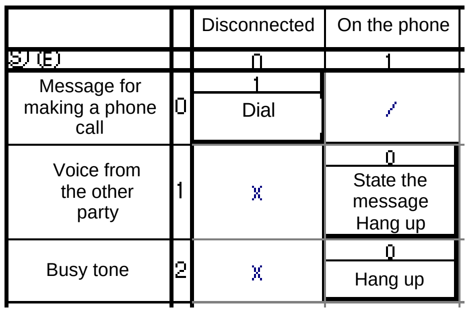
Figure 8- 3 Event-analyzer start activity

When an EASA is set as shown above, "Interrupt the current activity" processing is called when all events including "Message for making a phone call," "Voice from the other party" and "Busy tone" have occurred.

📖: The details of the STM driven type are explained in 11, "Driven type."

8.1.2. Event analyzer end activity

The Event-Analyzer End Activity (EAEA) is executed immediately before the wait for the next event after event-analyzer processing is executed, and actions and transitions are processed if any of those exist. It is always called for flag events and when the STM driven type is the state-driven type.



		Disconnected	On the phone
S) (E)		0	1
Message for making a phone call	0	1 Dial	/
Voice from the other party	1	X	0 State the message Hang up
Busy tone	2	X	0 Hang up

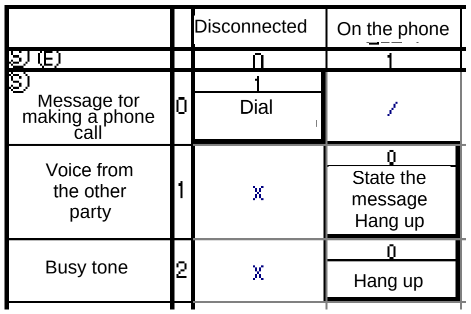
(EASA) Return to the interrupted activity

Figure 8- 4 Event-analyzer end activity

When the EAEA is set as shown above, "Return to the interrupted activity" processing is called which waiting for the next event after all events including "Message for making a phone call," "Voice from the other end" and "Busy tone" have occurred and the actions and transitions have been completed.

8.1.3. Event-hit start activity

The Event-Hit Start Activity (EHSA) is executed when the generated event is analyzed via the execution of event analyzer processing.



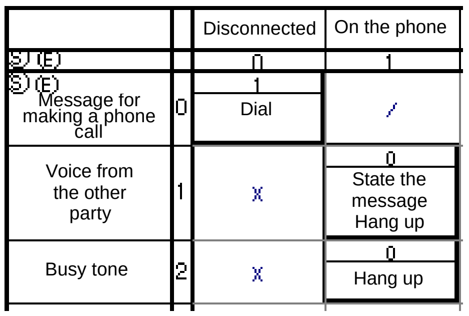
		Disconnected	On the phone
Ⓢ Ⓔ		0	1
Ⓢ Message for making a phone call	0	1 Dial	/
Voice from the other party	1	X	0 State the message Hang up
Busy tone	2	X	0 Hang up

Figure 8- 5 Event-hit start activity

When the "Message for making a phone call" event occurs, the "Take a note" activity is called.

8.1.4. Event-hit end activity

The Event-Hit End Activity (EHEA) is called when the actions and transitions have been finished after event analyzer processing is executed and the generated event is analyzed.

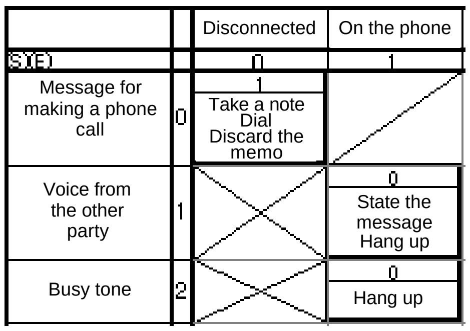


		Disconnected	On the phone
Ⓢ Ⓔ		0	1
Ⓢ Ⓔ Message for making a phone call	0	1 Dial	/
Voice from the other party	1	X	0 State the message Hang up
Busy tone	2	X	0 Hang up

Figure 8- 6 Event-hit end activity

When "Message for making a phone call" occurs during the "Disconnected" state, the "Dial" action is executed and a transition to "Busy" occurs. Then, the "Discard the note" activity is executed.

The example of the above event-analyzer activity has the same meaning as describing the actions as follows:



S\E		Disconnected	On the phone
		0	1
Message for making a phone call	0	1 Take a note Dial Discard the memo	
Voice from the other party	1		0 State the message Hang up
Busy tone	2		0 Hang up

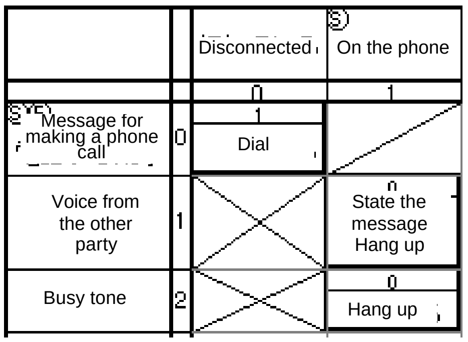
Figure 8- 7 Event-hit activity

8.2. State activity

The state activity is executed at intervals when the state becomes active or inactive.
: Refer to 10.5, "State scheduler and activity," for more about the activation timing of the state activity.

8.2.1. State start activity

The start activity (SSA) is executed when the applicable state becomes active as a result of a state transition.



S\E		Disconnected	On the phone
		0	1
Message for making a phone call	0	1 Dial	
Voice from the other party	1		0 State the message Hang up
Busy tone	2		0 Hang up

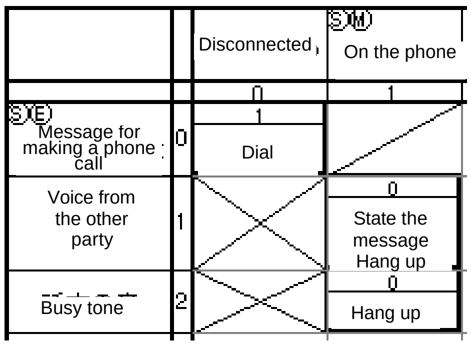
(SSA) Pick up a pen

Figure 8- 8 State start activity

Assume that a "Message for making a phone call" occurs during the "Disconnected" state. The "Dial" action is executed, and a transition occurs from the "Disconnected" state to the "Busy" state, thus activating the "Busy" state. The "Pick up a pen" activity is executed at this time.

8.2.2. State mode activity

The state mode activity (SMA) is executed at all times while the state is active. The SMA cannot be used if the STM-driven type is the event-driven type.



		Disconnected	SM On the phone
		0	1
SE Message for making a phone call	0	1 Dial	/
Voice from the other party	1	X	0 State the message Hang up
Busy tone	2	X	0 Hang up

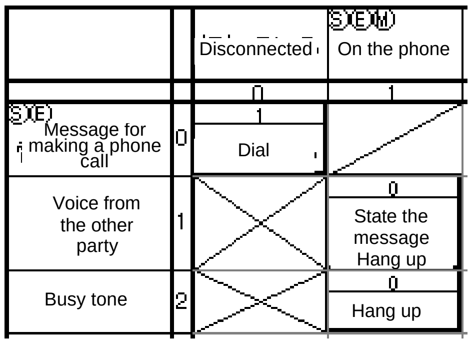
(SMA) Draw a circle

Figure 8- 9 State mode activity

The "Draw a circle" activity is called as long as the "On the phone" state remains active.

8.2.3. State end activity

The state end activity (SEA) is called when the state becomes inactive.



		Disconnected	SEM On the phone
		0	1
SE Message for making a phone call	0	1 Dial	/
Voice from the other party	1	X	0 State the message Hang up
Busy tone	2	X	0 Hang up

(SEA) Put the pen down

Figure 8- 10 State end activity

After the "Voice from the other party" or "Busy tone" event occurs during the "Busy" state and the "State the message, hang up" or "Hang up" action is executed, the "Put down the pen " activity is executed immediately before a transition from the "Busy" state to the "Disconnected" state occurs.
If the SSA is set in the "Disconnected" state, it is executed once the "Put down the pen " activity is complete.
🕮: The execution order of SSA, SSE and actions can be changed. This will be explained in 8.6, "Order of actions and activities."

8.3. Dispatch activity

The dispatch activity is activated each time a dispatch occurs. A dispatch is a switch of states in the state-driven STM. It is different from the state mode activity, in which the scheduled activity associated with a state is activated. Even if the function to be described as the dispatch activity is described as the state start activity for all functions, it cannot substitute the dispatch activity. This is because **the dispatch activity is also executed at the moment the ready state is changed to active in order to achieve the concurrent state, in addition to activation via the occurrence of a transition**. Rather, describing in the state mode activities of all states can substitute for dispatch activity. The state type of the state-driven STM is the same regardless of whether it is the exclusive or concurrent type. Normally, the dispatch activity is described when the system has to be monitored and controlled at all times.
The dispatch activity does not exist for the event-driven STM because there is no Dispatching of states. If the event is P-type in the event-driven STM, the event-hit activity substitutes the dispatch activity.

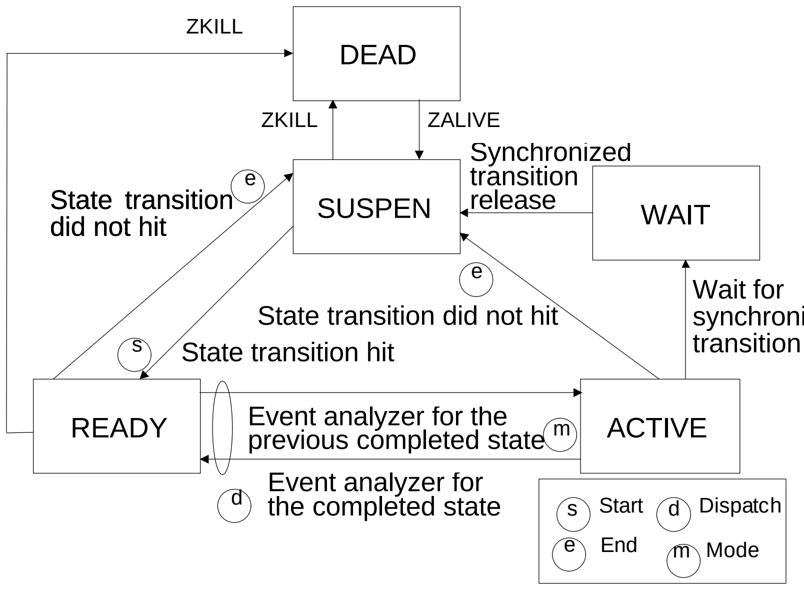
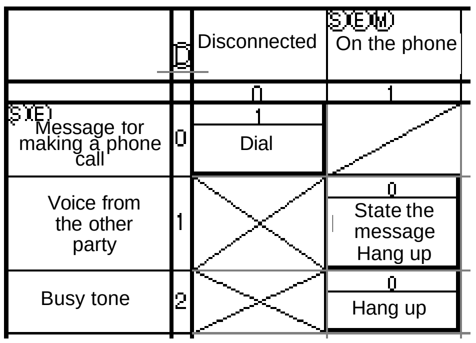

ZKILL
DEAD
ZKILL
ZALIVE
Synchronized transition release
State transition did not hit
e
SUSPEN
WAIT
e
State transition did not hit
Wait for synchronized transition
s
State transition hit
READY
Event analyzer for the previous completed state
m
ACTIVE
d
Event analyzer for the completed state
s Start
d Dispatch
e End
m Mode

Figure 8-11 State transition of state driven STM states



		Disconnected	On the phone
		0	1
Message for making a phone call	0	1 Dial	
Voice from the other party	1		0 State the message Hang up
Busy tone	2		0 Hang up

(D)"Monitor if birds come to eat the food"

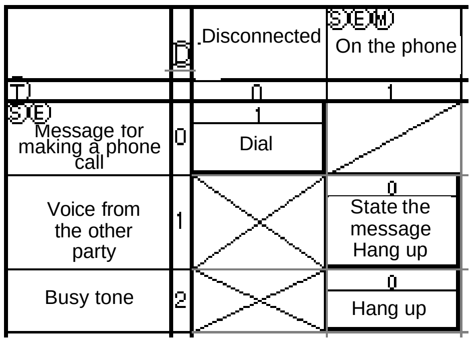
Figure 8- 12 Dispatch activity

When the above STM is defined as a state-driven type, "Monitor if birds come to eat the food" is always executed in any state.

: The STM driven type is explained in 11, "Driven type."

8.4. Trigger activity

The trigger activity has the same characteristics as the trigger event.



		Disconnected	On the phone
T		0	1
Message for making a phone call	0	1 Dial	
Voice from the other party	1		0 State the message Hang up
Busy tone	2		0 Hang up

(T)"Place the food"

Figure 8- 13 Trigger activity

"Place the food" processing is executed immediately after the program designed by the above STM starts the operation. Then, it waits for an event. Normally, the system's initial processing is described as the trigger activity.

: Refer to 4.4.8, "Trigger event," for the trigger event.

8.5. STM main activity

The STM main is explained in Chapter 14. The STM main controls the calls of the state transition matrix mechanism. The STM main's start activity is the activity called before a series of state transition matrix mechanisms starts. The STM main's end activity is the activity called once the operation of a series of state transition table mechanisms is complete. Note that these activities mark the start and end of the state transition matrix mechanism but not the start and end for the system. Such a delimiter (start and end) of the system is usually implemented by means of the state activity or event activity. The STM main usually has an infinite-loop structure. The STM main's start activity is called at the beginning of the infinite loop, which the STM main end activity is called at the end of the loop.

```
…
while(1){
  STM main start activity
   …
   …
   …
   STM main end activity
}
```

8.6. Order of actions and activities

The order of actions and activities is as follows:

Action → End activity → Start activity

Even though the following orders can be used, the choice as to whether or not to implement these orders depends on the specification of the tool supporting the EHSTM design method.

End activity → Action → Start activity

End activity → Start activity → Action

Specify '-' in the transition destination when executing the action only, without executing the activity. If the same state number or state name as that of the current state is explicitly specified, the state activity will run.

		state1	state2
		1	2
event1	1	=>-	=>2

Figure 8- 14 No transition specification

Because no transition or action is executed, a temporary operation can be processed by providing an in-mail as an event.

The order of actual activities differs, depending on the STM driven type.
(1)For the event-driven type
1 STM main start
2 Trigger (executed only once at the activation)
3 Event-analyzer start
4 Event-hit start (not executed when the event does not hit)
5 State end (not executed in the initial state or when the transition does not occur)
6 State start (not executed when the transition does not occur)
7 Event-hit end (not executed when the event does not hit)
8 Event-analyzer end
9 Event-analyzer end
When the concurrent-type STM is used, 5 and 6 are repeated for the number of concurrent states that are ready.
(2)For the state driven-type
1 STM main start
2 Trigger (executed only once at the activation)
3 Dispatch
4 State mode (not executed in the initial state)
5 Event-analyzer start (not executed in the initial state)
6 Event-hit start (not executed when the event does not hit)
7 State end (not executed in the initial state or when the transition does not occur)
8 State start (not executed when the transition does not occur)
9 Event-hit end (not executed when the event does not hit)
10 Event-analyzer end (not executed in the initial state)
11 STM main end
When the concurrent-type STM is used, → through ∝ are repeated for the number of concurrent states that are ready.

The order of event analyzer execution is (1) message, (2) flag and (3) in-mail.

8.7. Format of activity

Activities can describe mostly the same information as actions do.

1: Human-language statement
2: Computer-language statement
3: NS chart
4: Event hierarchy STM call
5: Subroutine STM call
6: Library STM call
7: Divided action cell
8: EHSTM system call
9: Transition symbol

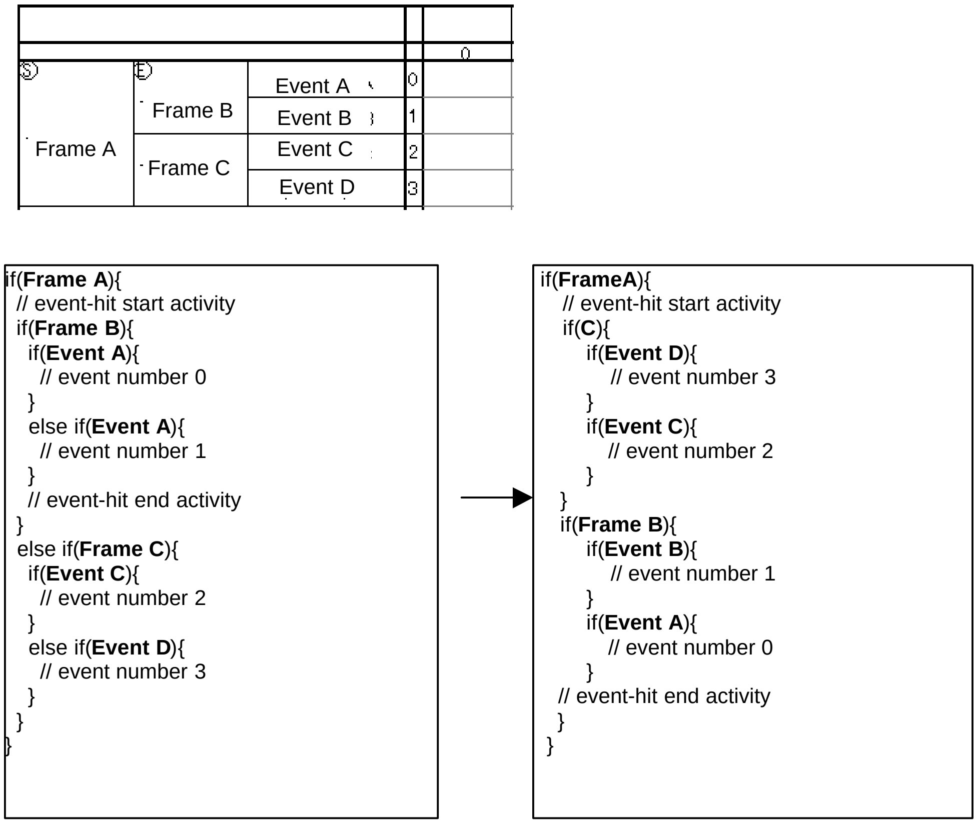
🕮: Refer to 6, "Action," for each action.

9. Event analyzer

The event analyzer is used to retrieve and analyze events, and for detecting the event number on the STM. Therefore, to be precise, there are two types of tasks, event retrieval and event analysis, and sometimes these are collectively called event analyzers. There are four types of events: variable, interrupt, in-mail and function call.

9.1. Variable event analyzer

To analyze the variable, comparisons are made using conditional statements ("if" and "switch" statements) to detect what the variable event is. Refer to the example in Chapter 4.4 showing what kind of event analyzer is generated from the event cell. The "if-else" rule is used as the default event analyzer for analyzing events in STM event cells, and the analysis is performed in the order from the event with the youngest event number first.



				0
Frame A	Frame B	Event A	0	
		Event B	1	
	Frame C	Event C	2	
		Event D	3	

```
if(Frame A){
 // event-hit start activity
 if(Frame B){
   if(Event A){
     // event number 0
   }
   else if(Event A){
     // event number 1
   }
   // event-hit end activity
 }
 else if(Frame C){
   if(Event C){
     // event number 2
   }
   else if(Event D){
     // event number 3
   }
 }
}
```

```
if(FrameA){
   // event-hit start activity
   if(C){
       if(Event D){
          // event number 3
       }
       if(Event C){
          // event number 2
       }
   }
   if(Frame B){
       if(Event B){
          // event number 1
       }
       if(Event A){
          // event number 0
       }
   // event-hit end activity
   }
 }
```

Figure 9- 1 Event analyzer

Events are analyzed in the order from event number 3 to number 0, and the "if-if" rule is used for the analysis rule. By using the "if-if" rule, a single event can drive the STM multiple times. There may be cases in which the variable becomes the event number. In these cases, the event analyzer is not executed and the STM is simply driven using the value of the variable.
📖: Refer to 4, "Event," for event types.

9.2. Event analyzer and variable access method

There are three access methods for variable events: the mail type, synchronized type and flag type. The access method specifies how to retrieve events regardless of the event analyzer. This is called an event acquisition function.

9.2.1. Flag variable access method

The flag variable access method reads only the value of the variable directly, so no event acquisition function is necessary. Access to the variable is achieved by executing the event analyzer immediately. The change of variables can be monitored at all times by calling the event analyzer from inside the infinite loop of the program.

9.2.2. Mail variable access method

If the event is set to the communication type using the RTOS, a procedure for retrieving messages and copying messages to the variable (copying may not be necessary), etc., will be necessary. This procedure is written as the event acquisition function.

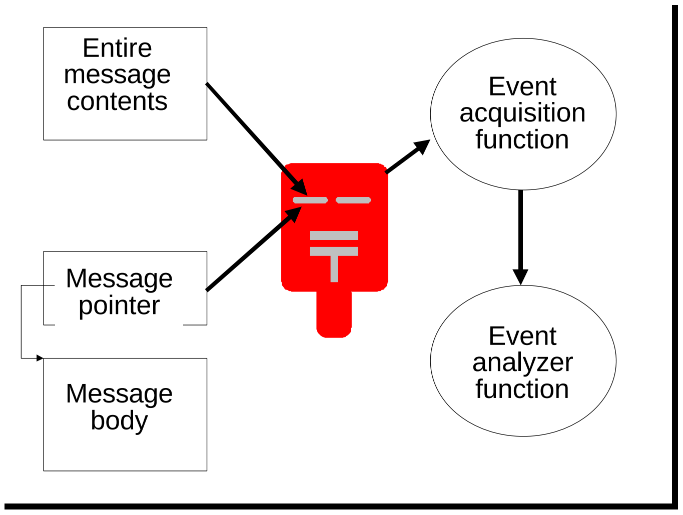


Figure 9- 2 Communication variable access method

The message is a mechanism existing almost only for the event-driven structure. The use of messages is recommended when designing a system using the event-driven exclusive-state STM.
A task has a message at a single location (RCV_MSG). While there is no message, "wait" is used so that the CPU is not wasted. When a message arrives, it is picked up by the RTOS and thrown into the event analyzer. The event analyzer examines the message, retrieves the event number and transfers the control to the STM drive mechanism.
Postal matters have several types, including postcards and envelopes. Likewise, messages

have various types. The entire content can be sent via a single message, or the body of the message is placed in the global memory and the pointer to the memory storage is notified as a message.

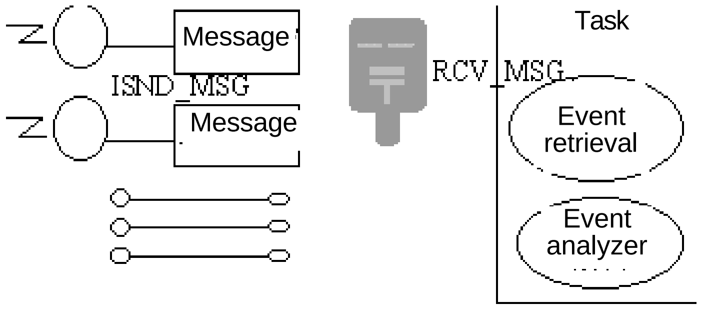


Figure 9- 3 Message and input port

The change in the input port cannot be known unless a message arrives, because the task does not start until the message is delivered. By changing RCV_MSG to PRCV_MSG, the input port can be monitored without shifting the task to "wait," even if there are no messages.

📖: Refer to 4.2, "Variable access method," for variable access methods.
Refer to the documents relating to the RTOS for detailed operations.

9.2.3. Synchronized variable access method

If the synchronized variable access method is realized using the RTOS event flag (WAI_FLG), the task does not occupy the CPU unnecessarily because the task waits until the event occurs.

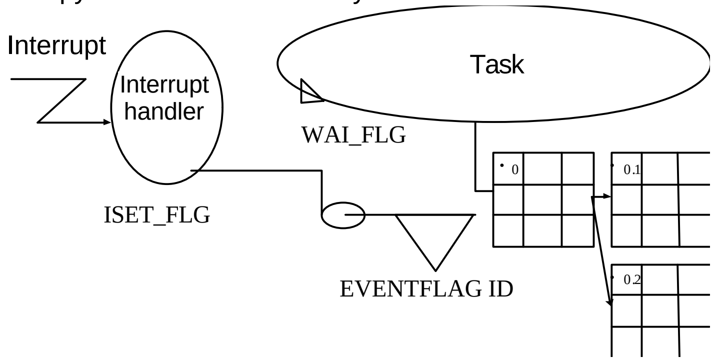


Figure 9- 4 Synchronized type waiting for a single event

However, the task enters the wait for the WAI_FLG event flag, other flags cannot be monitored (9-5). This is not desirable for the STM design that would need to wait for multiple events.

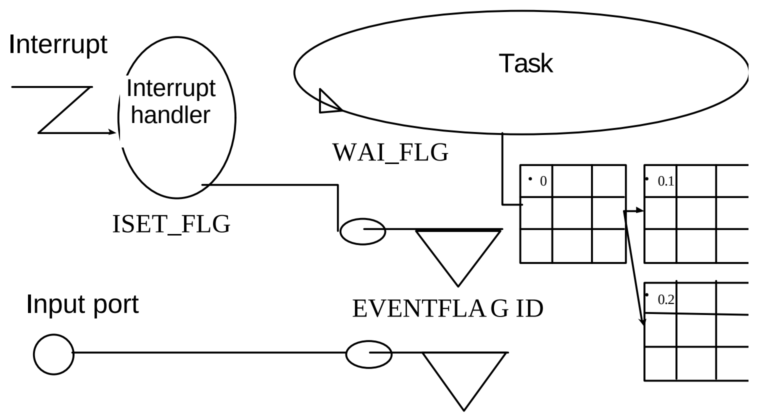


Figure 9- 5 Synchronized type waiting for multiple events

The task can monitor both flags in a single wait by connecting two handlers to EVENTFLAG (Figure 9-5). Even though the task could synchronize with handlers 1 and 2, it cannot know which handler the setting is from. In this case, provide an external variable that shows the type so that it can be known from which handler the setting comes. The event analyzer examines this external variable.

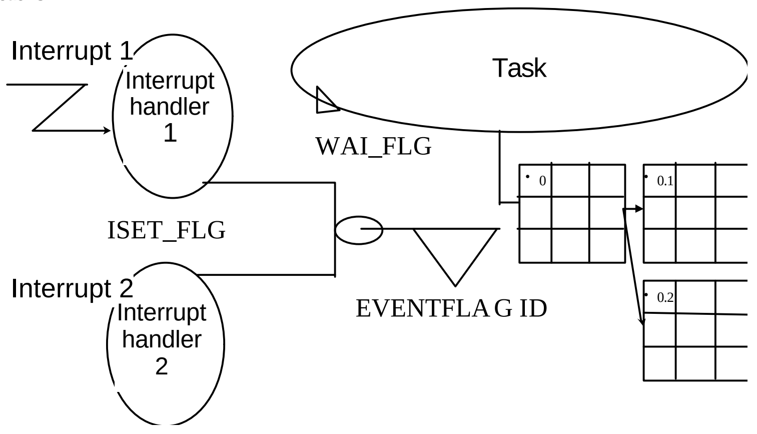


Figure 9- 6 Synchronized type waiting for multiple events with a single event flag

The WAI_FLG is convenient for achieving the concurrency of activation for multiple events awaited by a task using a single flag set.

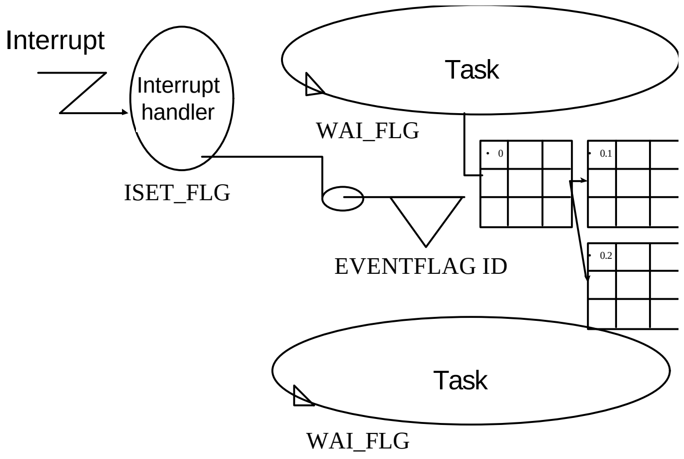


Figure 9- 7 Notifying multiple tasks simultaneously

9.3. In-mail analyzer

The in-mail analyzer analyzes the in-mail events. The in-mail does not care whether or not the RTOS is installed. The in-mail buffer defined by the STM is analyzed by the in-mail analyzer to drive the STM. The in-mail analyzer's default rule described in the event cell is the same as that of the event analyzer. The in-mail analyzer and event analyzer are individually prepared for each STM.

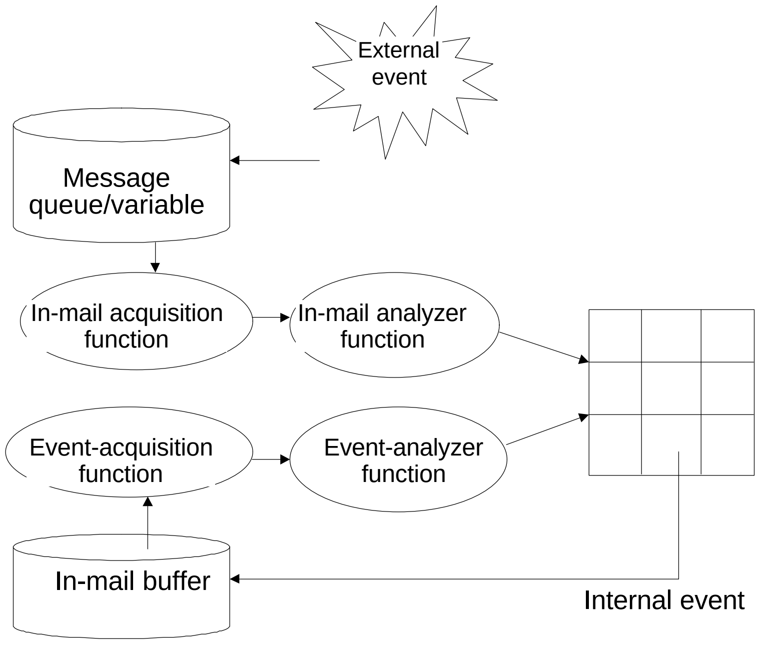


Figure 9- 8 In-mail

By default, the in-mail analyzer function is executed until the in-mail buffer becomes empty after the event-analyzer function is completed.

```
while(TRUE){
 // event-acquisition function
 // event-analyzer function
 // STM drive
 while (in-mail buffer becomes empty) {
   // in-mail acquisition function
   // in-mail analyzer function
   // STM drive
 }
}
```

9.4. Interrupt event analyzer

The interrupt event analyzer is created within the interrupt handler. Interrupts can be handled by setting a variable from the interrupt handler so that interrupts can be received as polling events, or by transmitting a message from the interrupt handler so that interrupts can be received as communication events, without writing in the STM directly as an interrupt type. If the event is described directly to the STM as the interrupt type, the interrupt handler will drive the STM. The interrupt type is separated from other events by drawing sloped lines at the top-left and bottom-right of the event column.

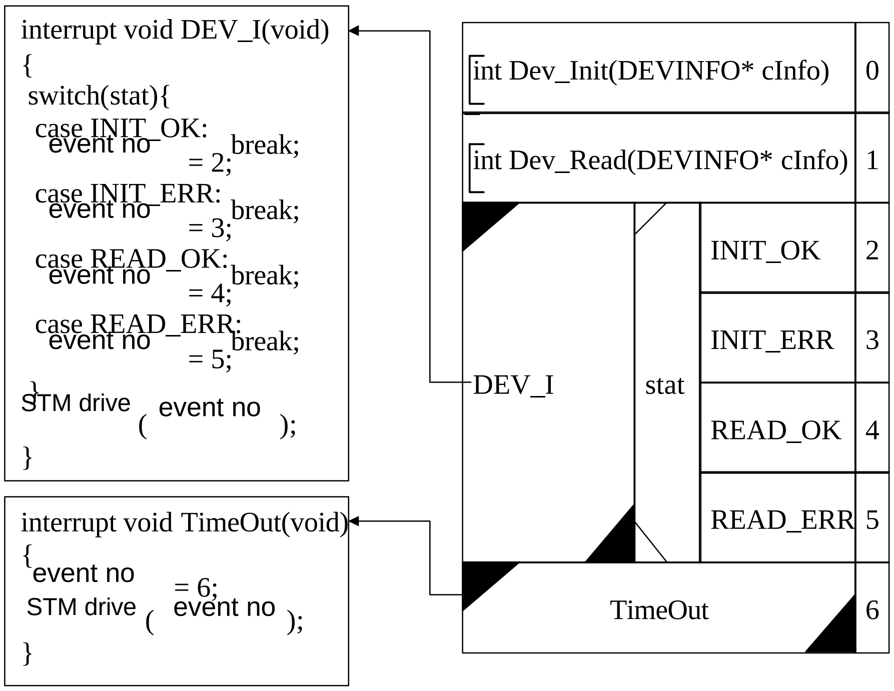

interrupt void DEV_I(void)
{
switch(stat){
case INIT_OK: event no = 2; break;
case INIT_ERR: event no = 3; break;
case READ_OK: event no = 4; break;
case READ_ERR: event no = 5; break;
}
STM drive (event no);
}
interrupt void TimeOut(void)
{
event no = 6;
STM drive (event no);
}
int Dev_Init(DEVINFO* cInfo)
0
int Dev_Read(DEVINFO* cInfo)
1
DEV_I
stat
INIT_OK
2
INIT_ERR
3
READ_OK
4
READ_ERR
5
TimeOut
6

Figure 9- 9 Interrupt type

The default rule of the event analyzer is the same as that of the variable type.

9.5. Function-call event analyzer

The function-call event analyzer has the same mechanism as the interrupt event analyzer. The event analyzer is executed in a function of the function-call type, and the STM is driven.

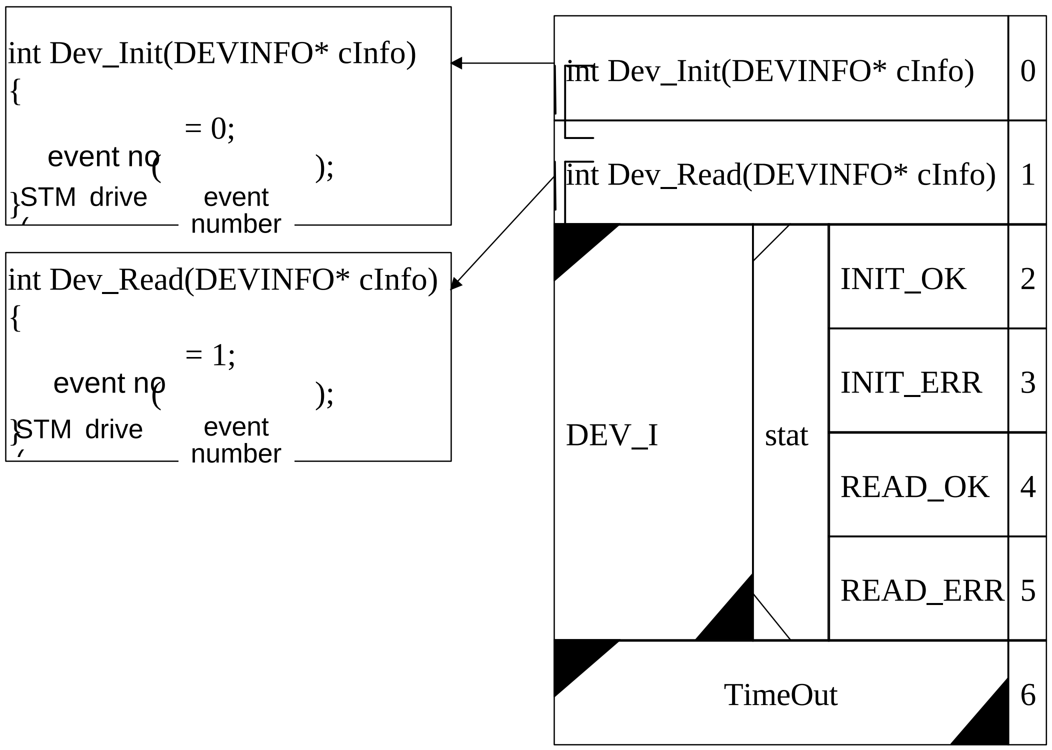


Figure 9- 10 Function-call type

10. State scheduler

The state scheduler schedules the state described in the STM. Even for the exclusive state STM in which all of the states are exclusive, a parent schedule occurs when an activity is installed in the state frame.

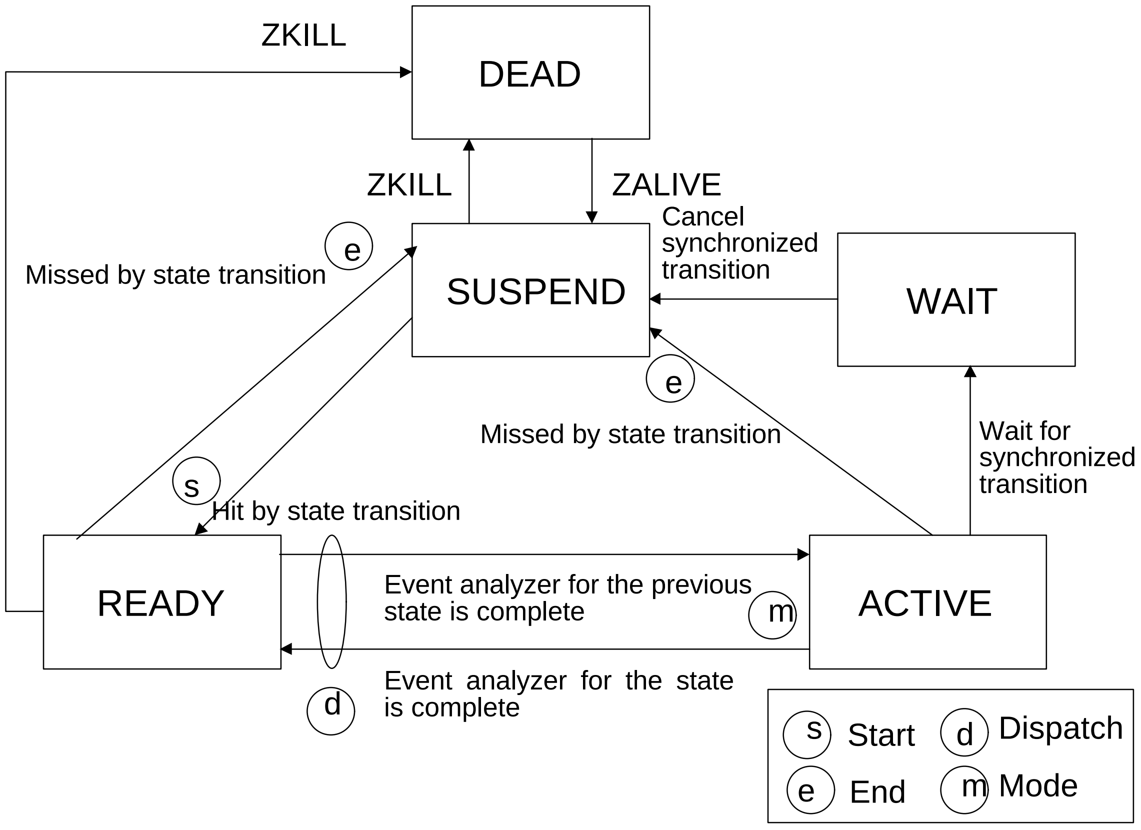

ZKILL
DEAD
ZKILL
ZALIVE
Cancel synchronized transition
e
Missed by state transition
SUSPEND
WAIT
e
Missed by state transition
Wait for synchronized transition
s
Hit by state transition
READY
Event analyzer for the previous state is complete
m
ACTIVE
d
Event analyzer for the state is complete
s Start
d Dispatch
e End
m Mode

Figure 10- 1 State scheduler

The following schedules occur:
(1)Parent-and-child schedule
(2)Concurrent schedule
(3)State tree schedule
There are occasions when you will wish to set priorities for the states, especially the concurrent states. In EHSTM Ver. 2, the state priority can only be defined by the scheduling as described below. The detailed definition of a state priority depends on the specification of the CASE tool supporting state schedulers. EHSTM Ver. 3 will have the priority definition of states and events.

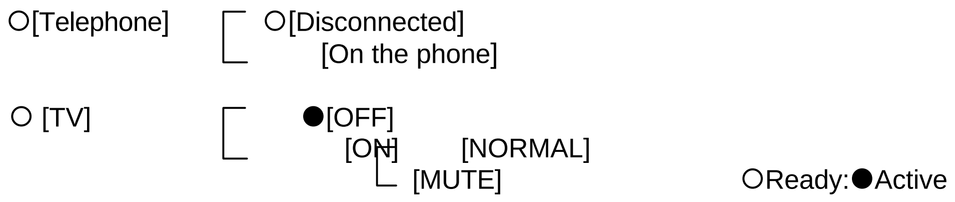
10.1. Parent-and-child schedule

When the child state is specified as the transition destination, its parent becomes ready first.

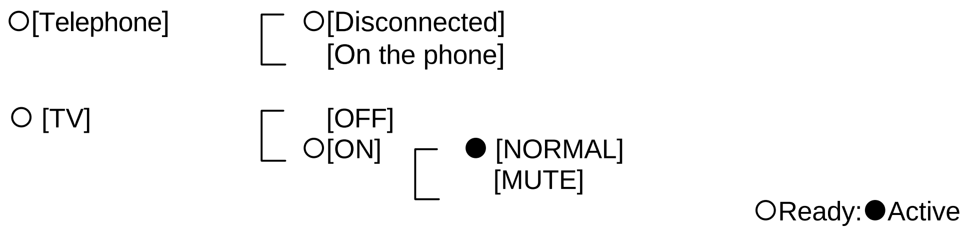
		Telephone		TV		
					ON	
		Disconnected	On the phone	OFF	NORMAL	MUTE
		0	1	2	3	4
Message for making a phone call	0	1 Dial	/	/	/	/
Voice from the other party	1	×	0 State the message Hang up	/	4 Mute	/
Busy tone	2	×	0 Hang up	/	/	/
Interested program	3	/	/	3 Turn TV on	/	/
End of program	4	/	/	×	2 Turn TV off	2 Turn TV off

Figure 10- 2 STM of telephone and TV

See the example for telephone and TV in Figure 10-2 .



Assume the "Interested program" event occurs during the "OFF" state. After the "Turn TV on" action is executed, a state transition to "NORMAL," which has state number 3, occurs. The order of readiness is "ON," then "NORMAL." When the child is the transition destination, the parent always becomes ready first. The order of switching from READY to SUSPEND is from child to parent.



When the "End of program" event occurs during the "NORMAL" state and upon transition to the "OFF" state, "ON" and "NORMAL," which have been READY or ACTIVE until now, become SUSPEND in the order of from "NORMAL" to "ON."

10.2. Concurrent schedule

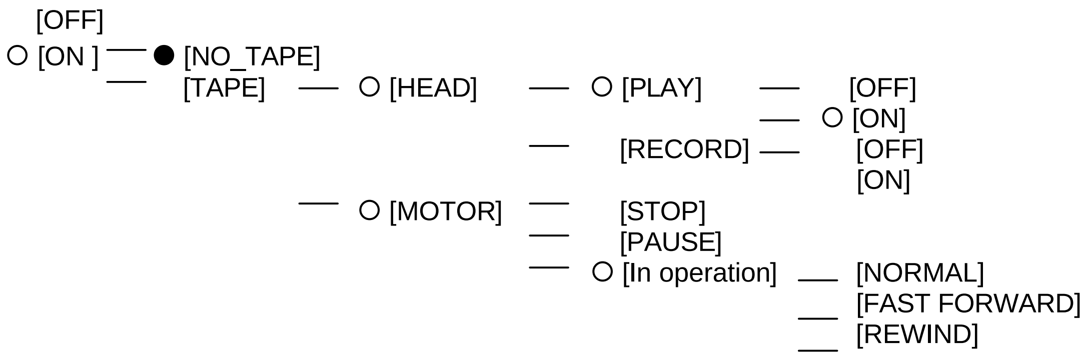
There are two types of concurrent schedules:
(1)Default type (D-type)
(2)Dispatch type (P-type)
By the default type, the priority of switching for the concurrent state to READY and ACTIVE is given to the default side. By the dispatch type, the entry to the right of the state that became READY first the previous time is switched to READY first. The priority returns to the HEAD if there is nothing to its right. In other words, the READY priority of the concurrent states circulates.

			ON											
		OFF		NO_TAPE	TAPE									
						HEAD				MOTOR				
						PLAY		RECORD		STOP	PAUSE	In operation		
						OFF	ON	OFF	ON			NORMAL	FAST FORWARD	REWIND
		0	1	2	3	4	5	6	7	8	9	10	11	12
ON	0	=>ON (H)	/	/	/	/	/	/	/	/	/	/	/	/
OFF	1	/	=>0	.	.	.	.	.	.	.	.	.	.	.
IN TAPE	2	/	/	=>TAPE	/	/	/	/	/	/	/	/	/	/
OUT TAPE	3	/	/	/	=>2	.	.	.	.	.	.	.	.	.
■	4	/	/	/	/	/	=>4	/	=>6	/	/	=>8	=>8	=>8
▶	5	/	/	/	/	=>5	/	/	/	=>10	/	/	=>10	=>10
▶▶	6	/	/	/	/	/	/	/	/	=>11	/	=>11	/	=>11
◀◀	7	/	/	/	/	/	/	/	/	=>12	/	=>12	=>12	/
❚❚	8	/	/	/	/	/	/	/	/	/	=> In operation (P)	=>9	=>9	=>9
●▶	9	/	/	/	/	/	/	=>7	/	=>10	/	/	/	/

Figure 10- 3 Videocassette recorder STM

```
   [OFF]
○ [ON]   —— ● [NO_TAPE]
         ——   [TAPE]  —— [HEAD]   —— [PLAY]          —— [OFF]
                                                     —— [ON]

                                  —— [RECORD]        —— [OFF]
                                                     —— [ON]
                      —— [MOTOR]  —— [STOP]
                                  —— [PAUSE]
                                  —— [In operation]  —— [NORMAL]
                                                     —— [FAST FORWARD]
                                                     —— [REWIND]
```

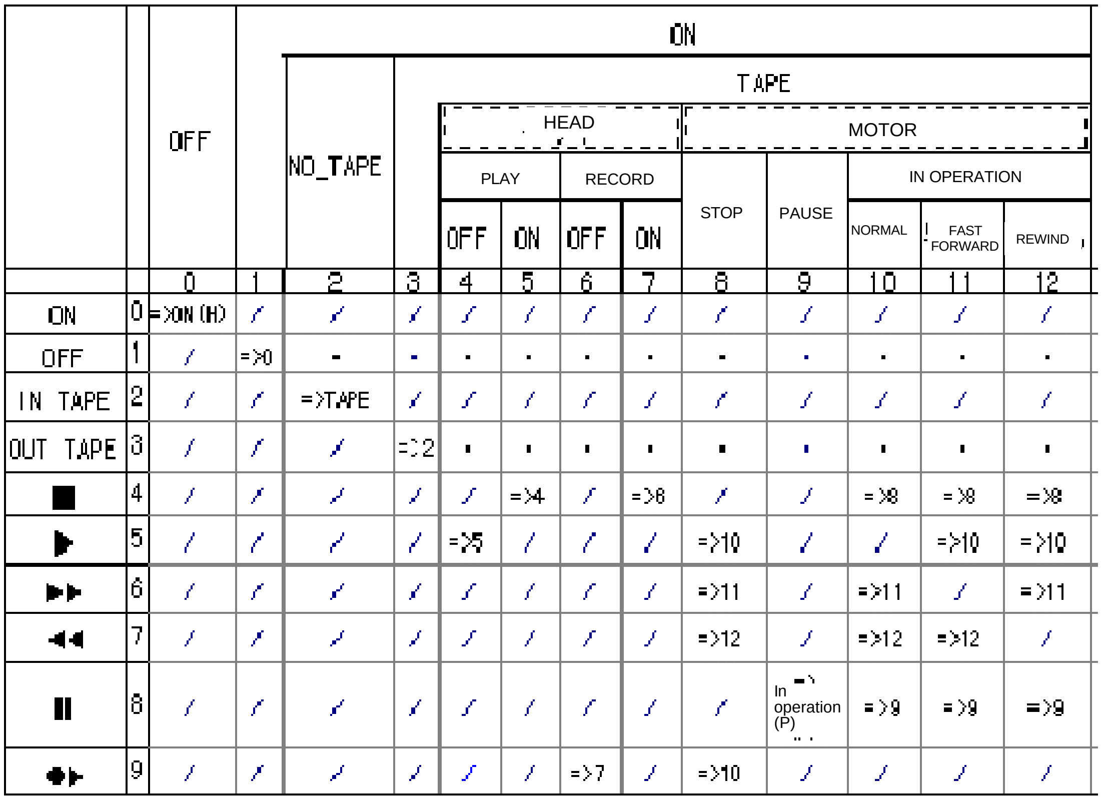
When the "IN TAPE" event occurs, a transition to "TAPE" occurs. Because "HEAD" and "MOTOR" are concurrent, "HEAD" always becomes READY before "MOTOR" if the default type is set for the concurrent schedule. If the dispatch type is selected for the concurrent type, the READY priority is given to “MOTOR” if “HEAD” took priority previously, and the priority circulates among concurrent brother states.



Assume a videocassette recorder as described above is in the “PLAY” state and an "OUT TAPE" event occurs when the "TAPE" state is active. The order of changing from READY to SUSPEND is from child to parent. This is the reverse of the order in which SUSPEND becomes READY, i.e., from parent to child. If the order of changing from SUSPEND to READY is
"TAPE" -> "HEAD" -> "PLAY" -> "ON" -> "MOTOR" -> "In operation" -> "NORMAL,"
the order of switching from READY to SUSPEND will be as follows:
"TAPE" <- "HEAD" <- "PLAY" <- "ON" <- "MOTOR" <- "In operation" <- "NORMAL"
Even if a transition occurs from the "ON" state and causes to suspend READY, the "ON" state does not become SUSPEND first.

10.3. Tree schedule

There are two types of tree schedules:
(1) Vertical type (V-type)
(2) Horizontal type (H-type)
The vertical tree schedule operates by giving priorities to parent-and-child relationships upon the change from SUSPEND to READY. The horizontal tree schedule operates by giving priorities to brother relationships upon the change from SUSPEND to READY.



		OFF	ON											
				NO_TAPE	TAPE									
						HEAD				MOTOR				
						PLAY		RECORD		STOP	PAUSE	IN OPERATION		
						OFF	ON	OFF	ON			NORMAL	FAST FORWARD	REWIND
		0	1	2	3	4	5	6	7	8	9	10	11	12
ON	0	=>ON (H)	/	/	/	/	/	/	/	/	/	/	/	/
OFF	1	/	=>0	·	·	·	·	·	·	·	·	·	·	·
IN TAPE	2	/	/	=>TAPE	/	/	/	/	/	/	/	/	/	/
OUT TAPE	3	/	/	/	=>2	·	·	·	·	·	·	·	·	·
■	4	/	/	/	/	/	=>4	/	=>6	/	/	=>8	=>8	=>8
▶	5	/	/	/	/	=>5	/	/	/	=>10	/	/	=>10	=>10
▶▶	6	/	/	/	/	/	/	/	/	=>11	/	=>11	/	=>11
◀◀	7	/	/	/	/	/	/	/	/	=>12	/	=>12	=>12	/
❚❚	8	/	/	/	/	/	/	/	/	/	=> In operation (P)	=>9	=>9	=>9
●▶	9	/	/	/	/	/	/	=>7	/	=>10	/	/	/	/

Figure 10- 4 Videocassette recorder STM

```
   [OFF]
○[ON] ——●[NO_TAPE]
      —— [TAPE]   —— [HEAD]    —— [PLAY]              —— [OFF]
                                                      —— [ON]
                  —— [RECORD]                         —— [OFF]
                                                      —— [ON]
                  —— [MOTOR]   —— [STOP]
                               —— [PAUSE]
                               —— [IN OPERATION]      —— [NORMAL]
                                                      —— [FAST FORWARD]
                                                      —— [REWIND]
```

"NO_TAPE" makes a transition to the "TAPE" state when it accepts an "IN TAPE" event during the active state. At this time, changes from SUSPEND to READY occur in the following order if vertical (parent type) scheduling is specified:
"TAPE" → "HEAD" → "PLAY" → "OFF" → "MOTOR" → "STOP"
When the horizontal (brother type) scheduling is specified, the order is:

"TAPE" → "HEAD" → "MOTOR" → "PLAY" → "STOP" → "OFF"
Changes from READY to SUSPEND will follow the above order in reverse.

10.4. State scheduler and transition

The three transition types (default transition, history transition and deep history transition) do not affect the concurrent scheduler types.
For example, if the transition "=>TAPE" described in the action cell for state number [2] of event number [2] is changed to "=>TAPE(D)," the child state of "HEAD" is simply changed to "PLAY" and "OFF" is selected as the child state of "PLAY." This is not for specifying the priority of the schedule. If "=>TAPE(H)" is stated, the one that was READY last time is simply selected as the child state of "HEAD" and the default is specified for the grandchild, but the priority of the schedule is not specified. If "=>TAPE(P)" is stated, the child state of "HEAD" is simply selected to the one that was READY previously, and the one that was READY last time is specified for the grandchild, also. Specifying the priority of the schedule is independent of this operation.

10.5. State scheduler and activity

The event activities and trigger activities are not related to the state scheduler. On the other hand, the state activity and dispatch activity are related to the state scheduler. This relationship is shown in Figure 10-5.
The state start activity is executed when SUSPEND changes to READY. The state end activity is executed when READY changes to SUSPEND. The state mode activity is executed as long as it is active. A dispatch activity is executed each time a dispatch occurs.

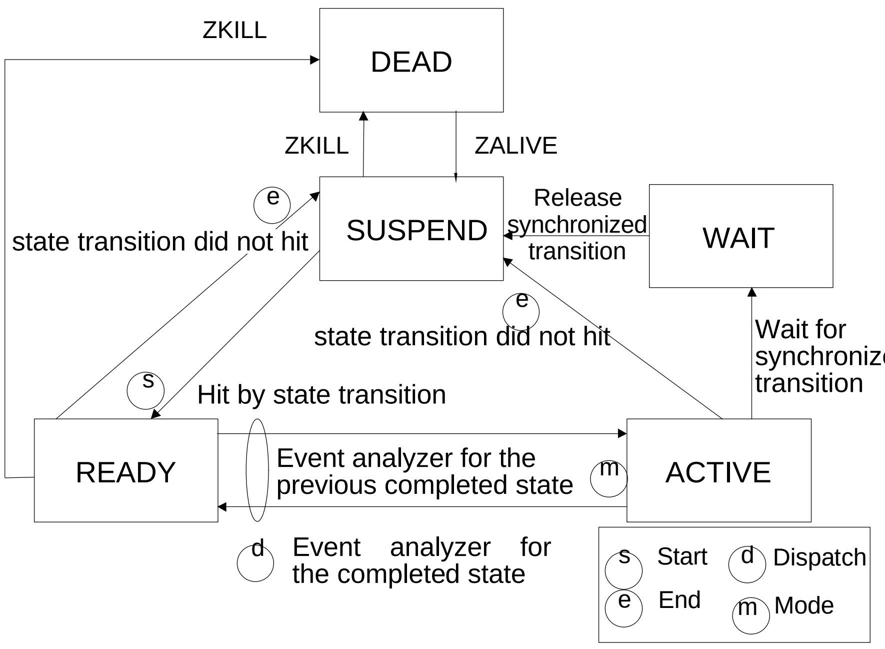

ZKILL
DEAD
ZKILL
ZALIVE
e
state transition did not hit
SUSPEND
Release synchronized transition
WAIT
e
state transition did not hit
Wait for synchronized transition
s
Hit by state transition
READY
Event analyzer for the previous completed state
m
ACTIVE
d
Event analyzer for the completed state
s Start
d Dispatch
e End
m Mode

Figure 10- 5 State scheduler STD

		SUSPEND	READY	ACTIVE	DEAD	WAIT
		0	1	2	3	4
Hit by state transition	0	ⓢ =>READY	/	/ else ⓔ ⓢ	/	/
Missed by state transition	1	/	ⓔ =>SUSPEND	ⓔ =>SUSPEND	/	/
Event analyzer for the previous state is complete	2	×	ⓓ ⓜ =>ACTIVE	×	×	/
Event analyzer for the state is complete	3	×	×	ⓓ =>READY	×	/
ZKILL	4	=>DEAD	=>DEAD	×	/	/
ZALIVE	5	/	/	/	=>SUSPEND	/
Wait for the synchronized transition	6	×	:	=>WAIT	×	/
Release synchronized transition	7	×	×	/	×	=>SUSPEND

Figure 10- 6 State scheduler STM

10.6. State scheduler and interrupt event

All interrupt events perform event-driven actions regardless of whether the driven type is defined as event-driven or state-driven. In other words, all action cells that cross with READY concurrent states are called.

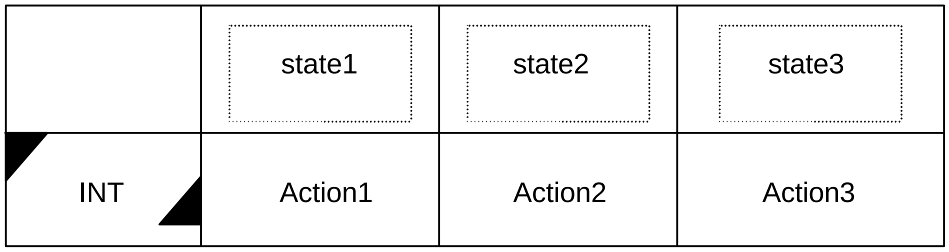


Figure 10- 7 Interrupt event for concurrent states

In Figure 4-24, Action1, Action2 and Action3 are called when the "INT" occurs. The order of the calls depends on the state scheduler. The function that keeps the same schedule order by initializing the READY authority upon interrupt is not specified in this document, but depends on the specification of the tool supporting the EHSTM.

🕮: Refer to 8, "Activity," 11, "Driven type," 5, "State" and 4, "Event," for the activity, driven type, concurrent state and interrupt event, respectively.

10.7. Format of state scheduler

The state scheduler is not specified on the STM. It is usually set as a property of the STM by the tool that supports the EHSTM design method.

11. Driven type

There are two driven types of STMs: the event-driven type (E-type) and state-driven type (S-type).

11.1. Event-driven type

The event-driven type waits for events at one location in the program. For the wait method the RTOS structure can be used if it is installed, or sensing of the flag (so-called polling) by the program may also be used. The important thing is to wait for an event at a single location, making it possible to receive any event under any state. The received event undergoes the event analysis and the event number assigned within the STM is calculated, then the action at which the event and current state number cross is called.

Event-driven type

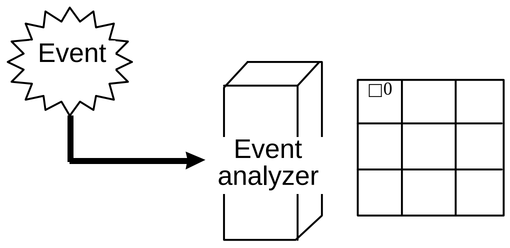


Figure 11- 1 Event-driven type

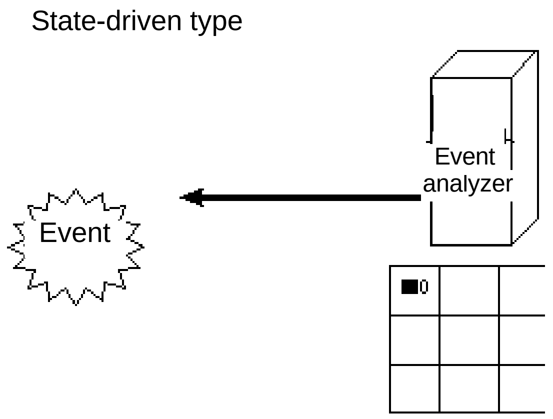
		TELEPHONE		TV		
					ON	
		DISCONNECTE	ON THE PHONE	OFF	NORMAL	MUTE
		0	1	2	3	4
Message for making a phone call	0	1 Dial	/	/	/	/
Voice from the other party	1	×	0 State the message Hang up	/	4 Mute	/
Busy tone	2	×	0 Hang up	/	/	/
Interested program	3	/	/	3 Turn TV on	/	/
End of program	4	/	/	×	2 Turn TV off	2 Turn TV off

Figure 11- 2 STM of telephone and TV

We'll use the example of a telephone and TV in Figure 11-2. When the currently ready states are "Telephone"-"Disconnected" and "TV"-"OFF," there is no scheduling of a state in the event-driven type. Instead, events are awaited at a single location. When an event occurs after a while, the ready states are activated in sequence. Vertical scheduling and horizontal scheduling are the types used for specifying the sequence. In addition, default scheduling and dispatch scheduling are used as the priority scheduling for concurrent states. In this way, the event-driven type activates the state only after an event occurs.
The event-driven STM is declared in the highest-level STM via the ☐ symbol.

11.2. State-driven type

The state-driven type monitors events for each state. It does not wait for (or does not analyze, to be more accurate) events pertaining to no use or invalidity in the matrix. The "cyclic executive" is easily performed for the S-type because it accepts events for each state and the events to be analyzed for the state are predetermined.



Let's use the example of the telephone and TV in Figure 11-2. In the state-driven type, when the states currently ready are "Telephone"-"Disconnected" and "TV"-"OFF," the event-analyzer function associated with each state is initiated and whether or not an event has occurred is checked each time one of the "Telephone," "Disconnected," "TV" or "OFF" states is activated. Even if no event has occurred, one of the states is always active.
The state-driven STM is declared in the highest-level STM using the ν symbol.

11.3. Driven type and event

Caution should be exercised concerning the types of handled events depending on the driven type.

	RTOS not required (it can exist)		RTOS required	
	Polling type	Interrupt type	Synchronized type	Message
Event-driven type	CPU resource problem		Queue management problem	
State-driven type			Task wait problem	Task wait problem

Figure 11- 4 Driven types and event types

In the event-driven STM, events are awaited at a single location within the program. If polling-type events are handled by the event-driven type, the CPU resource is always used. This is not a problem, however, when the RTOS is not installed or when there is only one task, even if it is installed. When synchronized events are handled by the event-driven type, one event flag is shared. Then, a variable is necessary to identify the specific event from the shared event flag. When a variable is used, it is necessary to pay attention if it is overwritten. If the variable is overwritten, it has to be in the form of a queue.

The state-driven type has a structure that accepts events in each state. When synchronized events are handled by the state-driven type, the task waits for synchronization within the event analyzer associated with a single state. Therefore, even in the concurrent state, the task itself will be in the wait state and concurrent processing may not be possible. This also applies to message events.

11.4. Driven type and state

In the event-driven type, a state is activated at the time the event is captured. In the state-driven type, the state scheduler activates a state via dispatch.

State type / Driven type	Exclusive	Concurrent
Event-driven type	All events are awaited at a single location. Call actions from a single state.	All events are awaited at a single location. Call actions from multiple states.
State-driven type	A single state checks whether the specific event has occurred.	Multiple states check whether the specific event has occurred.

Figure 11- 5 Driven type and state type

11.5. Driven type and action

In either driven type, the action cell is executed where the event occurred and the currently active state cross.

11.6. Driven type and transition

Transitions are executed in the same way for either driven type.

11.7. Driven type and activity

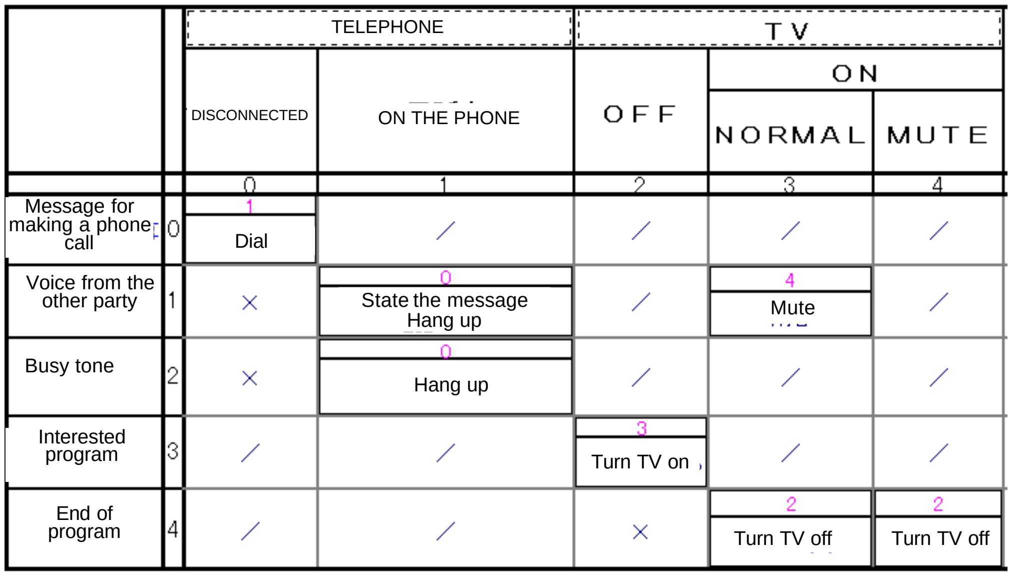
In the event-driven type, the dispatch activities and state-mode activities cannot be used because the state scheduler is not operating all the time but is activated when an event is received.

	State start	State end	State mode	Dispatch	Event Analyzer start	Event Analyzer end	Event hit start	Event hit end	Trigger
Event-driven type	○	○	×	×	○	○	○	○	○
State-driven type	○	○	○	○	○	○	○	○	○

Figure 11- 6 Driven type and activities

11.8. Driven type and event analyzer

There is no close relationship between the driven type and the event analyzer. There are three kinds of event analyzers: the message (mail or synchronized) event analyzer, flag event analyzer and in-mail event analyzer. These event analyzers can be allocated to each STM or state.



		TELEPHONE		T V		
					O N	
		DISCONNECTED	ON THE PHONE	O F F	N O R M A L	M U T E
		0	1	2	3	4
Message for making a phone call	0	1 Dial	/	/	/	/
Voice from the other party	1	×	0 State the message Hang up	/	4 Mute	/
Busy tone	2	×	0 Hang up	/	/	/
Interested program	3	/	/	3 Turn TV on	/	/
End of program	4	/	/	×	2 Turn TV off	2 Turn TV off

Figure 11- 7 STM of telephone and TV

The following shows the event analyzer of Figure 11-7 created for each STM.

```
if(message for making a phone call){
   // event number 0
}
else if(voice from the other party){
   // event number 1
}
else if(busy tone){
   // event number 2
}
else if(interested program){
   // event number 3
}
else if(end of the program){
   // event number 4
}
```

The following shows the event analyzer of Figure 11-7 created for each state.
(1)Event analyzer associated with the "Disconnected" state

```
if(message for making a phone call){
   // event number 0
}
```

(2)Event analyzer associated with the "Busy" state

```
if(voice from the other party){
  // event number 1
}
else if(busy tone){
   // event number 2
}
```

(3)Event analyzer associated with the "OFF" state

```
if(interested program){
  // event number 3
}
```

(4)Event analyzer associated with the "NORMAL" state

```
if(voice from the other party){
  // event number 1
}
else if(end of the program){
   // event number 4
}
```

(5)Event analyzer associated with the "MUTE" state

```
if(end of the program){
  // event number 4
}
```

The event analyzer defined for each STM can be used in the event-driven or state-driven program structure.

12. Department

The department is a framework that defines the target of the STM design.
The following departments can be designed by the STM in the implementation environment for which no RTOS is installed:

(1) Main section
(2) Library section
(3) Interrupt-handler section
(4) Device-register section

The following departments can be designed by the STM in the implementation environment for which the RTOS is installed:

(1) Task section
(2) Library section
(3) Interrupt-handler section
(4) Device-driver section
(5) Device-register section

12.1. Department under non-RTOS

The main section enters an infinite loop, the event analyzer of the main STM is executed, and the variable event is captured. The device-register STM becomes the external environment from the CPU's viewpoint and generates interrupts. The interrupt is captured by the interrupt-handler STM, then the result is written to the variable and passed to the main section. The library section is called as a function-call event from the action or activity of the main section. The device section is an STM that describes how to respond to command registers such as various LSIs. This STM does not usually become a program code, but is used for simulations conducted without the target or code.

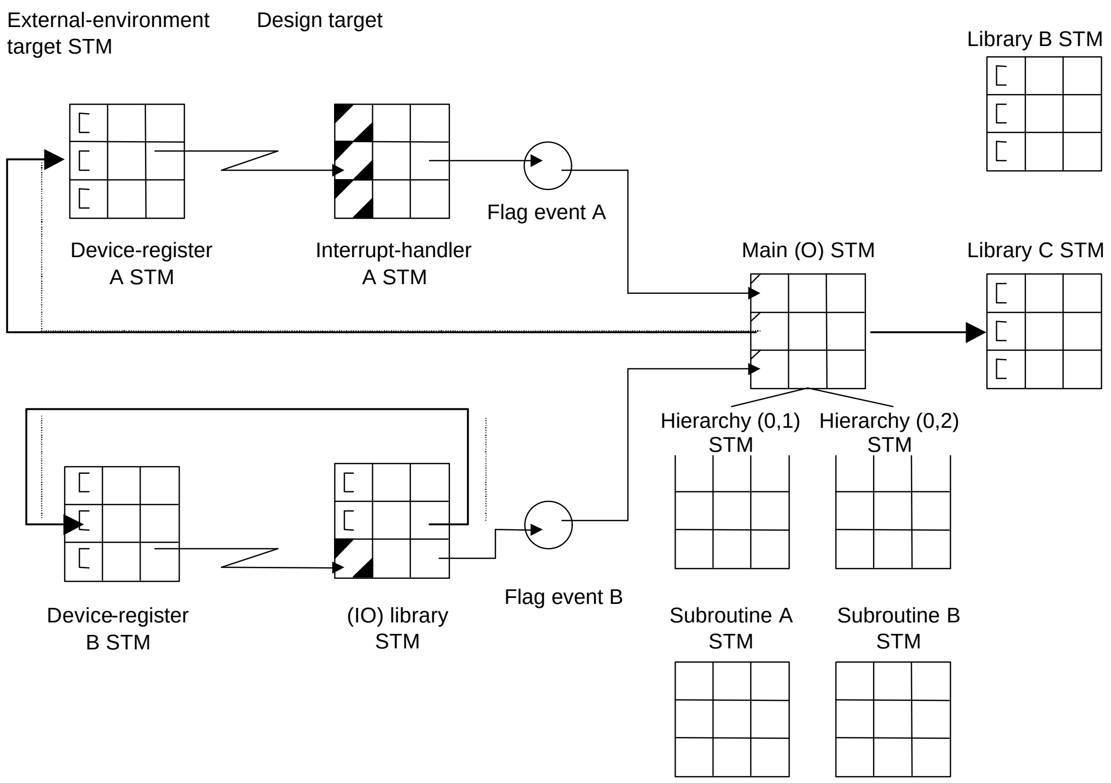


Figure 12- 1 RTOS Departments under non-RTOS

There are cases in which an interrupt event is incorporated in the main STM but the STM for the interrupt-handler section is not created. When the main section needs to be conscious of interrupts, design so that the main section handles interrupt events. When state management via the interrupt handler is necessary, create the interrupt-handler STM.

12.2. Department under RTOS

The RTOS implements the multitask structure instead of the single-path program structure. A task STM is designed for each task. The library section is called from the task section via a function-call event. Exclusive control when sharing the library among tasks is performed using the semaphore mechanism of the RTOS within the action of the task section. It is convenient when message-type events are used in the task section, because events from device drivers and other tasks can be accepted. The events can be used as synchronized events for analyzing variables, or as flag events. Refer to Chapter 9, "Event Analyzer." Interrupts are captured by the interrupt-handler STM. The device-driver section is called from a task as call, capturing interrupt events.

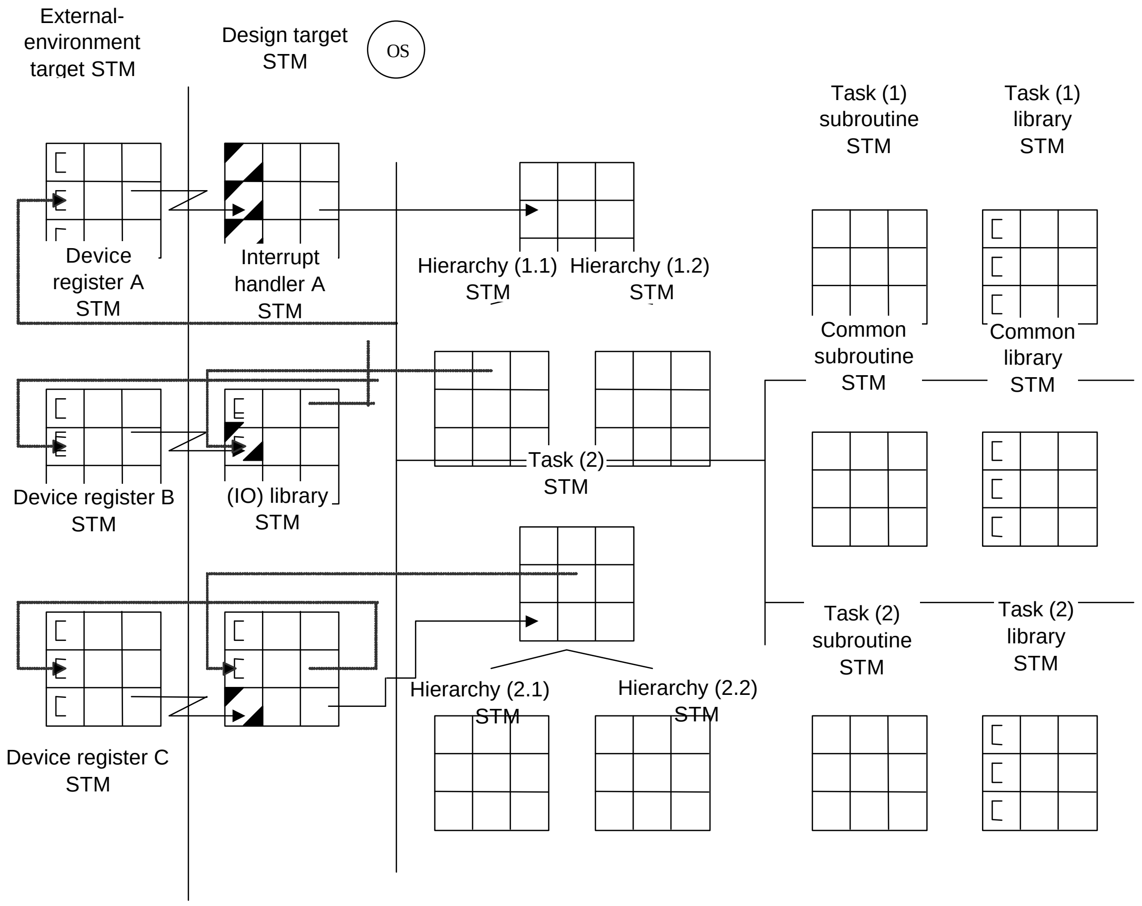


Figure 12- 2 Departments under RTOS

In general, when the RTOS is installed, the interrupt handler and device driver are placed under RTOS control or manipulated by each manager layer of the RTOS. The basic STM mechanism is the control of calling and returning of functions. A CASE tool that simulates the state-transition matrix is required to support the operations of the following: call of the RTOS or manager layer, interrupt handler and device driver, return-operation control and system-call operation for each RTOS. This document does not cover these operations; it only defines the description method of the state transfer matrix.

12.3. Main/Task STM

The main STM and task STM are declared as follows:

(return-value type) (driven type) STM name (number of clones) (argument)

(1) Items in parentheses ( ) can be omitted.
(2) Driven type:□(event-driven or event hierarchy)
■(state-driven or state hierarchy)
(3) STM name: A number starting with 0 when a hierarchy exists.
(4) Number of clones: Write the number of clones in [ ].
Clones cannot be used in the state hierarchy.

Declaration example:int □sample[2](char *lpstat1, int icode)
Call example□1.1[0]:0(lp1,p2,p3)

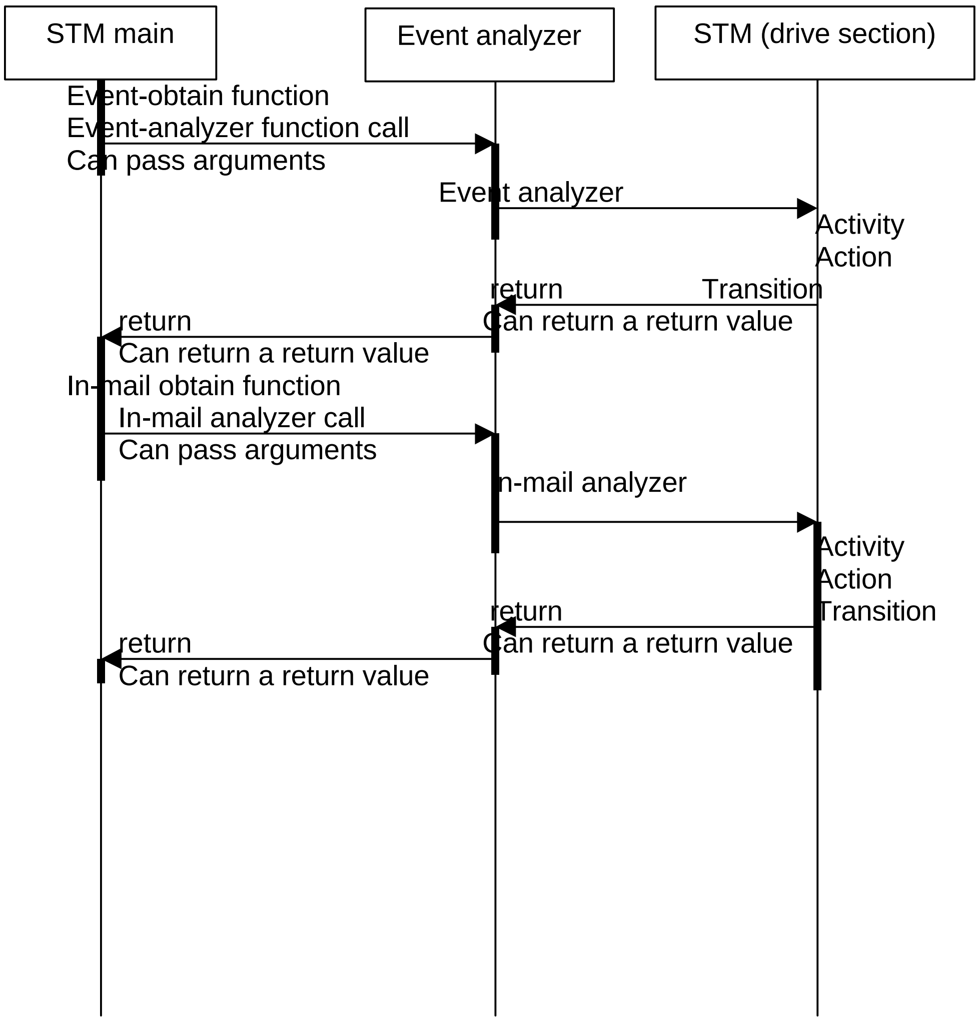


Figure 12- 3 Sequence diagram for main/task STM

: Refer to 13, "Clone STM," for the clone.
Refer to 14, "STM Main," for the STM main.

12.4. Library STM

The library STM is declared as follows:

(return-value type) (driven type) STM name (number of clones) (argument)

(1) Items in parentheses ( ) can be omitted.

(2) Driven type:○(Event-driven or event hierarchy)

●(State-driven or state hierarchy)

(3) STM name: A number starting with 0 when a hierarchy exists.

(4) Number of clones: Write the number of clones in [ ].

Clones cannot be used in the state hierarchy.

Declaration example: ○100

Call example: func(lpara1,para2)

○100:func(lpara1,para2)

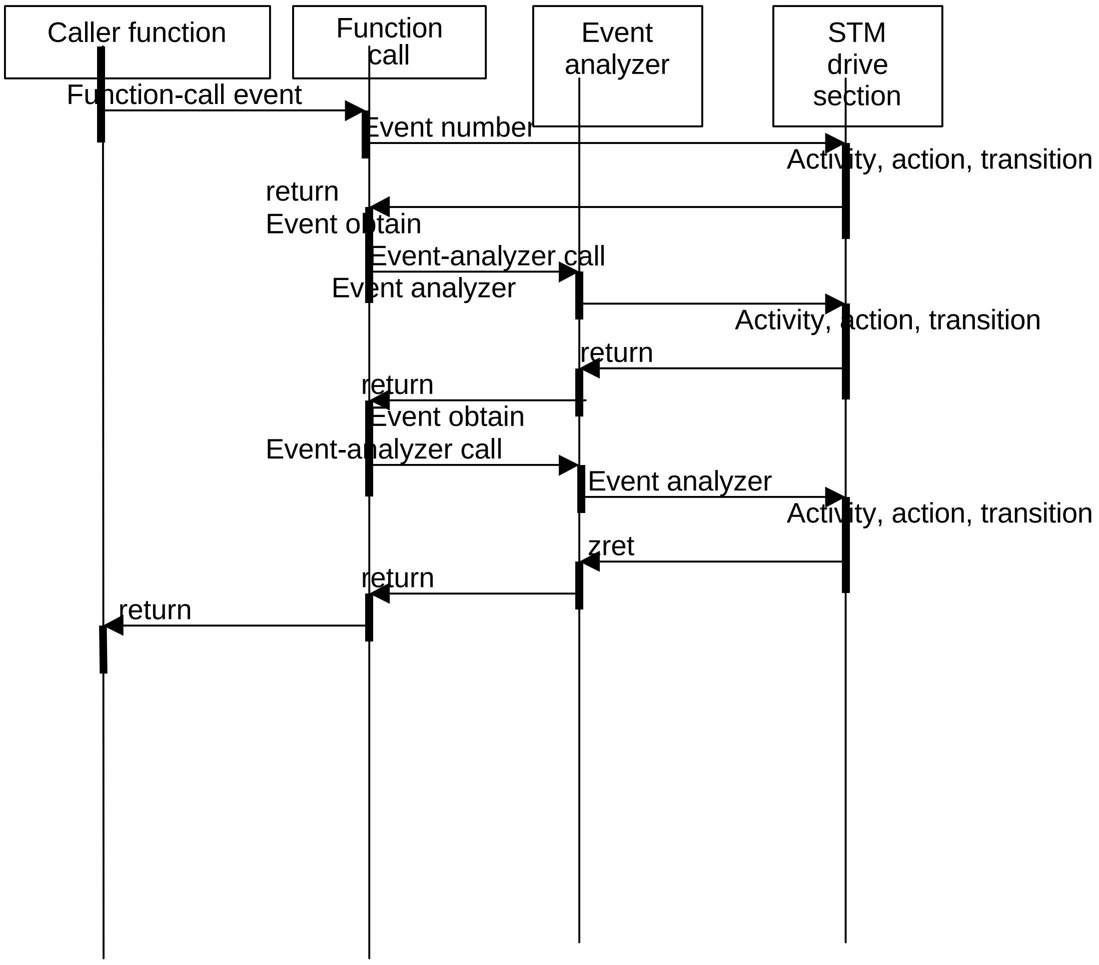
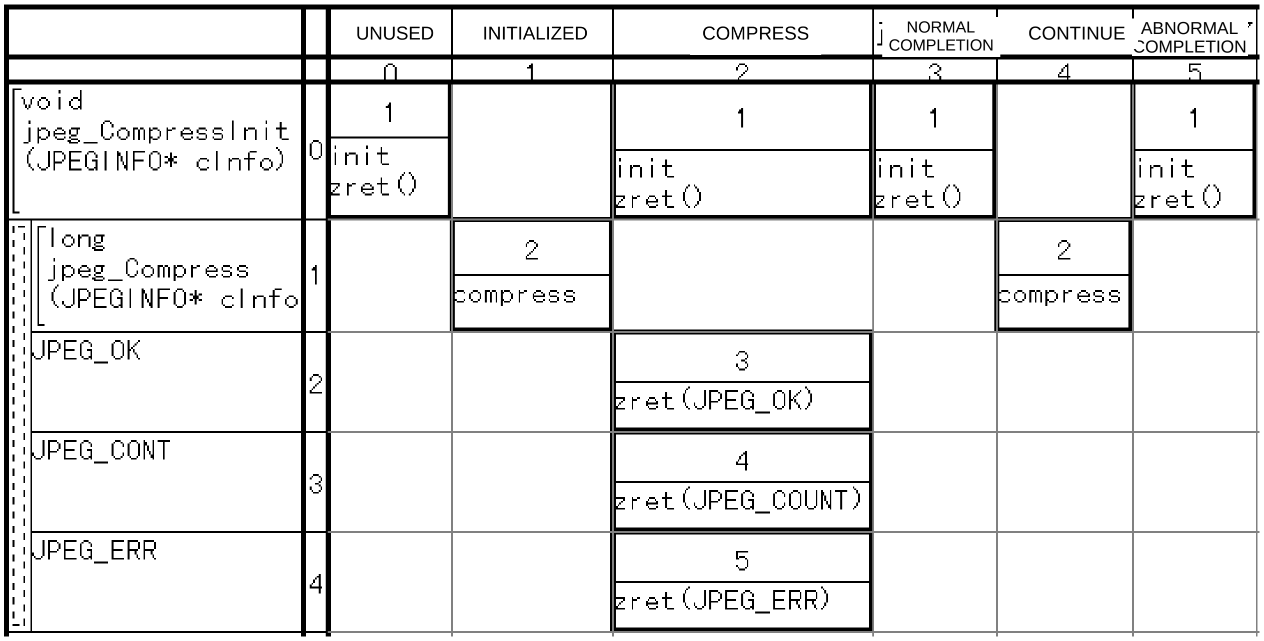


Figure 12- 4 Library STM operation sequence

The control is not returned to the caller STM unless "zret" is executed, because the library STM is called as a function from the caller. The middleware STM[1] in Figure 12-5 is used as an example for the explanation.



		UNUSED	INITIALIZED	COMPRESS	NORMAL COMPLETION	CONTINUE	ABNORMAL COMPLETION
		0	1	2	3	4	5
void jpeg_CompressInit (JPEGINFO* cInfo)	0	1 init zret()		1 init zret()	1 init zret()		1 init zret()
long jpeg_Compress (JPEGINFO* cInfo	1		2 compress			2 compress	
JPEG_OK	2			3 zret(JPEG_OK)			
JPEG_CONT	3			4 zret(JPEG_COUNT)			
JPEG_ERR	4			5 zret(JPEG_ERR)			

Figure 12- 5 Middleware STM

```
initial JPEG decompress library    /
 jpeg_CompressInit(&(dInfo));
 /get buffer processing      */
 getBuff(&(jpegBuff[0]));
begin JPEG decompression */
 for(;;){
   err = jpeg_Compress(&(dInfo));
   if(err == JPEG_CONT){
   update buffer processing */
     getBuff(&(jpegBuff[0]));
     continue;
   }
   else if(err == JPEG_OK){
  normal completion */
   break;
  }else{
  abnormal completion */
   break;
  }
```

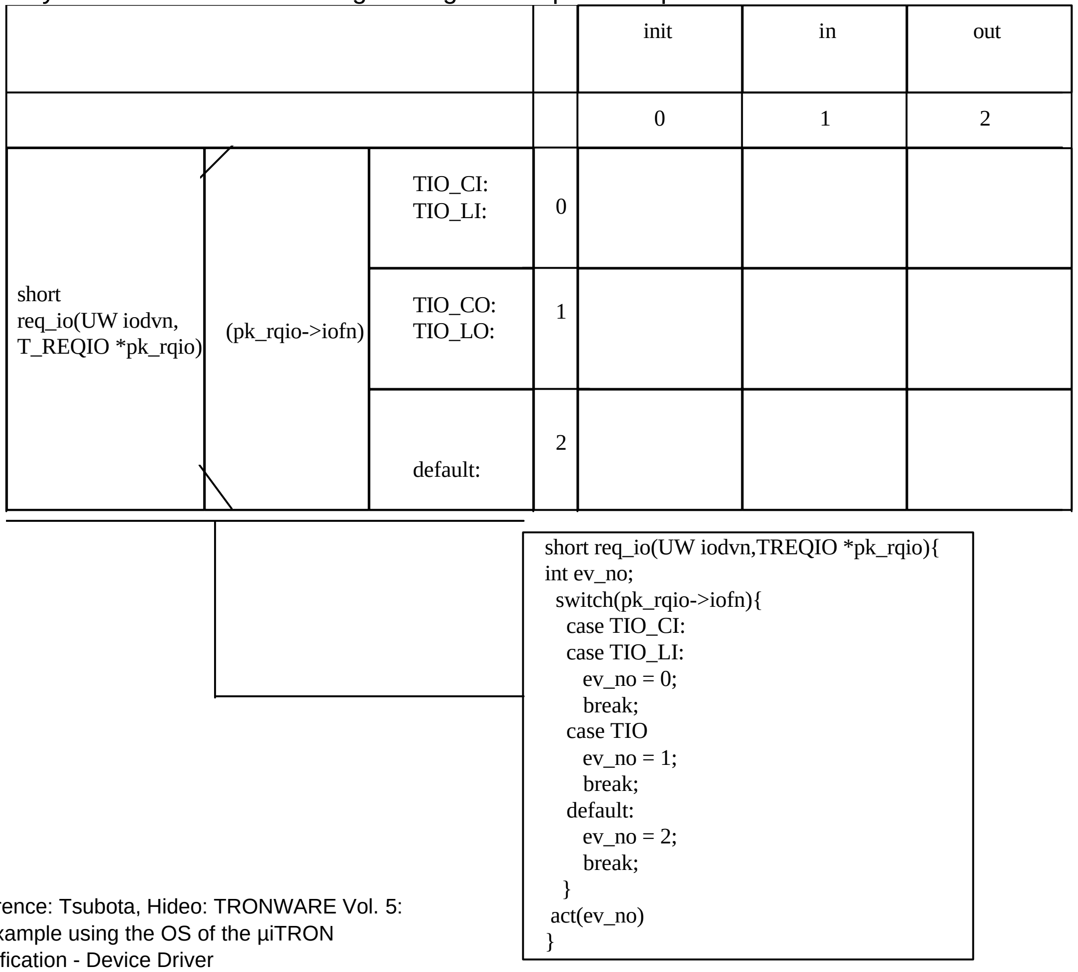
Figure 12- 6 Middleware call

When jpeg_CompressInit() is called, the control is transferred to the STM and is not returned to the caller until "zret" is executed. When jpeg_Compress() is called, the control is not returned to

[1] NEC: µSAP703000-B03JPEG middleware (tentative)

the caller until the library STM executes "zret()" in response to JPEG_OK, JPEG_ERR or JPEG_CONT. During this period, the event analyzer of the library STM is being processed. The execution method for this event O(E-type) or ●(S-type)) is declared in the called library STM. All of the P, C and M types can be used for the event type. Care should be taken when the M type is used, since the same message queue as the one used by the caller is used for the functions of the RTOS message mechanism. Normally, the P type is used for the event type of the library STM.

The analyzer can be executed using the arguments passed upon the function call as conditions.



				init	in	out
				0	1	2
short req_io(UW iodvn, T_REQIO *pk_rqio)	(pk_rqio->iofn)	TIO_CI: TIO_LI:	0			
		TIO_CO: TIO_LO:	1			
		default:	2			

```
short req_io(UW iodvn,TREQIO *pk_rqio){
int ev_no;
  switch(pk_rqio->iofn){
    case TIO_CI:
    case TIO_LI:
      ev_no = 0;
      break;
    case TIO
      ev_no = 1;
      break;
    default:
      ev_no = 2;
      break;
   }
 act(ev_no)
}
```

Reference: Tsubota, Hideo: TRONWARE Vol. 5: An example using the OS of the μiTRON specification - Device Driver

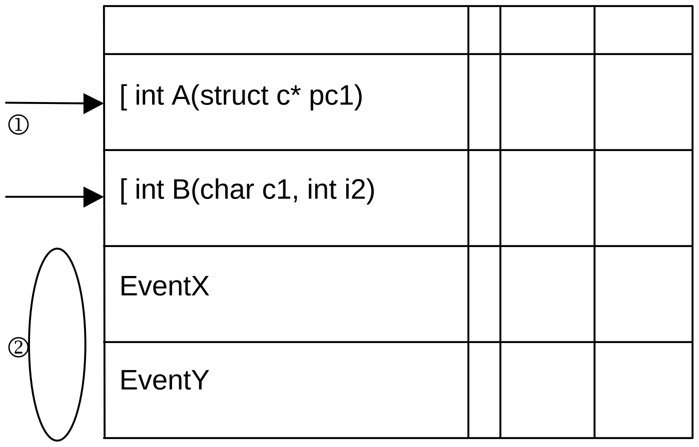
Figure 12- 7 Function call arguments [2]

[2] Reference: Tsubota, Hideo, “TRONWARE Vol.5: An example using the OS of the μiTRON specification - Device Driver –“

Let's take a closer look at the operations of the library STM.

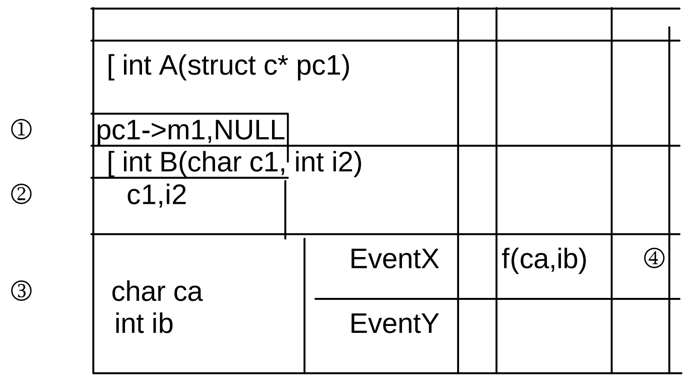
[int A(struct c* pc1)			
[int B(char c1, int i2)			
EventX			
EventY			

Figure xx is used as an example. The operations in this case are shown in Figure xx.

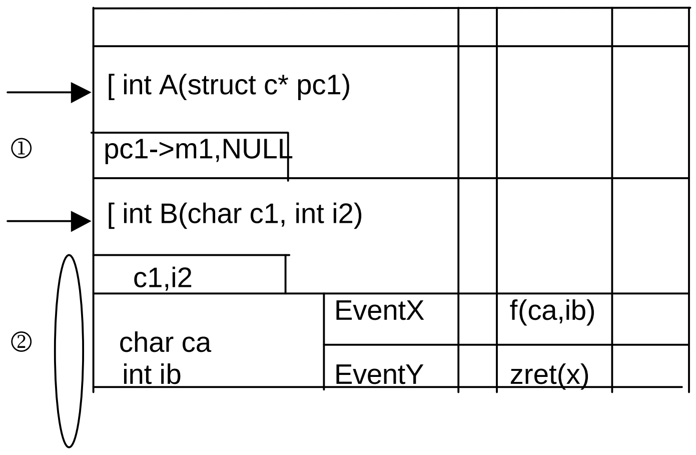


[int A(struct c* pc1)			
[int B(char c1, int i2)			
EventX			
EventY			

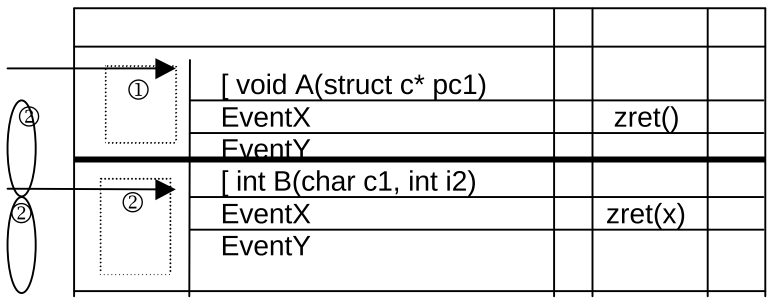
The library STM is called by the function-call event (①). If zret() does not exist, it enters a loop that waits for an event (②). Now there is an issue of how to pass the arguments upon entry to this loop. Functions A and B have different numbers and types of arguments. The arguments are declared using the frame upon entering the loop, as shown in Figure xx.



③ is the declaration of arguments for the function comprising the loop, while ①and ② are the declarations of the arguments to be passed. With this method the arguments can be passed to the function ④ described in the action cell.



The control returns to the caller when "zret" is executed at . An int-type value is returned in this case, since the return values from functions A and B are the same type. A problem occurs when the types of return values are different between functions A and B. In such cases, the difference of return values should be solved through the use of different frames.



The type of return value from zret() can be altered by adding a frame for each return-value type. The argument declaration can be omitted by adding frames. In this case, the function-event argument (①) is passed to the loop function (②).
To avoid wasting the CPU with the loop, RTOS system calls waiting for events can be written as event-analyzer start activities in ① and ② so that polling is not used and the CPU can be used efficiently.

12.5. Device-driver STM

The device-driver STM is declared the same way as the library STM.
Declaration example: O100[3](char* cpdata)
Call example: read(lpara1,para2)
O100:read(lpara1,para2)
The existence of an RTOS mechanism is assumed for the operation of a device-driver STM.
When the RTOS is not installed, the device-driver STM is not used but the interrupt-handler STM is used instead.

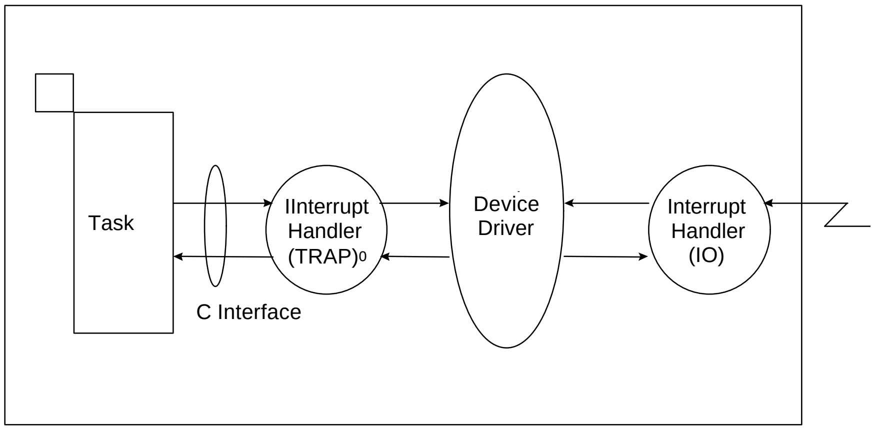


Figure 12- 8 Positioning of device-driver STM

The device-driver STM has exactly the same description format as the library STM mentioned earlier. In most cases, however, the operations of the actual device driver and library are different. Moreover, the operation varies depending on the implemented RTOS type. However, when a device driver is designed using the state-transition matrix, it is no different from creating a library or task. The device driver is called from the higher-level application by a system call (also called an IO system call or a driver system call). This can be considered a function-call event or message event. Or, if the system call is an internal interrupt such as a TRAP, there is no problem in considering it an interrupt event and proceeding with the design.
The notes explained hereinafter are for creating a simulator tool that supports this methodology. For actual simulations, the philosophy will change depending on which RTOS is simulated. A library STM is a task from the RTOS's viewpoint. A device-driver STM is controlled by the RTOS, separate from the task. System calls from a task to a device driver are forwarded to the device driver as software interrupts (TRAP). If the call is asynchronous (calling return), the device driver returns the control to the task as soon as it receives the command. If it is synchronous (complete return), the called task stays in the IO wait state until the device driver completes the processing, and the task does not operate until the device driver completes the processing. This is caused not by the STM operation but the RTOS operation. The device

driver does not return to the task that has called it. The relationship between them is not the simple call and return of a program, but the device driver is called from the task via RTOS and the control is returned. With this method, the RTOS can even give priority to the execution of another task. Here, zretset() is tentatively provided to express the intention of returning control to the RTOS. From the state-transition design viewpoint, there is no problem whether writing the zretset() function or writing codes that returns to the device driver of the RTOS to be used. Whether the code is understood or not depends on the specification of the CASE tool that supports the EHSTM.

Figure 12- 9

				UNUSED	INITIALIZED	INITIALIZATION COMPLETE	READ	NORMAL COMPLETION	ABNORMAL COMPLETION
				0	1	2	3	4	5
int Dev_init (DEVINFO* cInfo)			0	init(); tskid= cInfo->tskid; isus_tsk(tskid); =>初 Initializing	zretset(cInfo->tskid, CALL_ERR);	init(); Initializing cInfo->tskid; isus_tsk(tskid); =>初期化中	init(); Initializing cInfo->tskid; isus_tsk(tskid); =>初期化中	init(); Initializing cInfo->tskid; isus_tsk(tskid); =>初期化中	init(); Initializing cInfo->tskid; isus_tsk(tskid); =>初期化中
int Dev_read (JPEGINFO* cInfo)			1	zretset(cInfo->tskid, CALL_ERR);	zretset(cInfo->tskid, C Initialization complete	Read read(); zretset(cInfo->tskid, CALL_OK); =>リード	zretset(cInfo->tskid, CALL_ERR);	zretset(cInfo->tskid, CALL_ERR);	zretset(cInfo->tskid, CALL_ERR);
DEV_I	stat	INIT_OK	2	/	zretset(tskid, Abnormal completion IN…; irsm_tsk(tskid) =>初期化完了	/	/ Normal completion	/	/
		INIT_ERR	3	/	zretset(tskid, INIT_ERR); irsm_tsk(tskid) =>異常終了	/	Abnormal completion /	/	/
		READ_OK	4	/	/	/	MB=READ_OK; isnd_msg(MB1); =>正常終了	/	/
		READ_ERR	5	/	/	/	MB=READ_ERR; isnd_msg(MB1); =>異常終了	/	/

```
                s = DevRead(&dInfo);
                if(s == READ_OK){
                            ...
                }
                else{
                            ...
                }
    }
```

Figure 12- 10 Driver call

The operation upon calling Dev_Init is basically the same as that of the library-call STM. In other words, it is a synchronized (complete return) type. The synchronized type here means that the control is not returned to the caller until processing finishes completely. Not returning the control to the caller means to make the caller task WAIT, since the device driver STM uses the RTOS mechanism.
Dev_Init() becomes a TRAP command in C Inter, and is caught by the interrupt handler. The interrupt handler that receives this system call (Dev_init) saves the PC and SP of the task at the time the system call is issued, to the TCB of the task. It then notifies the device driver of the request from the task.
If the device driver that has received the request as a function-call event is an RTOS, which has to process the execution control of request tasks, describe the processing in the action or activity of the device-driver STM.
For iTRON[3], the request task is transferred to the forced wait state by isus_tsk(tskid). In order to resume the task in a forced wait, stateirsm_tsk(tskid) is issued. In the case of the RTOS[4] that does not support isus_tsk(), the event flag is executed using wait_flag(flgid) on the request-task side, and is set within the device driver using iset_flag(flgid).
The request task needs to obtain the task ID using get_tid(p_tskid) and pass it to the device driver as an argument.

[3] iTRON for NEC-made V850 (RX850) is used as an example.
[4] iTRON for NEC-made 78K/IV (RX78k/IV), etc.

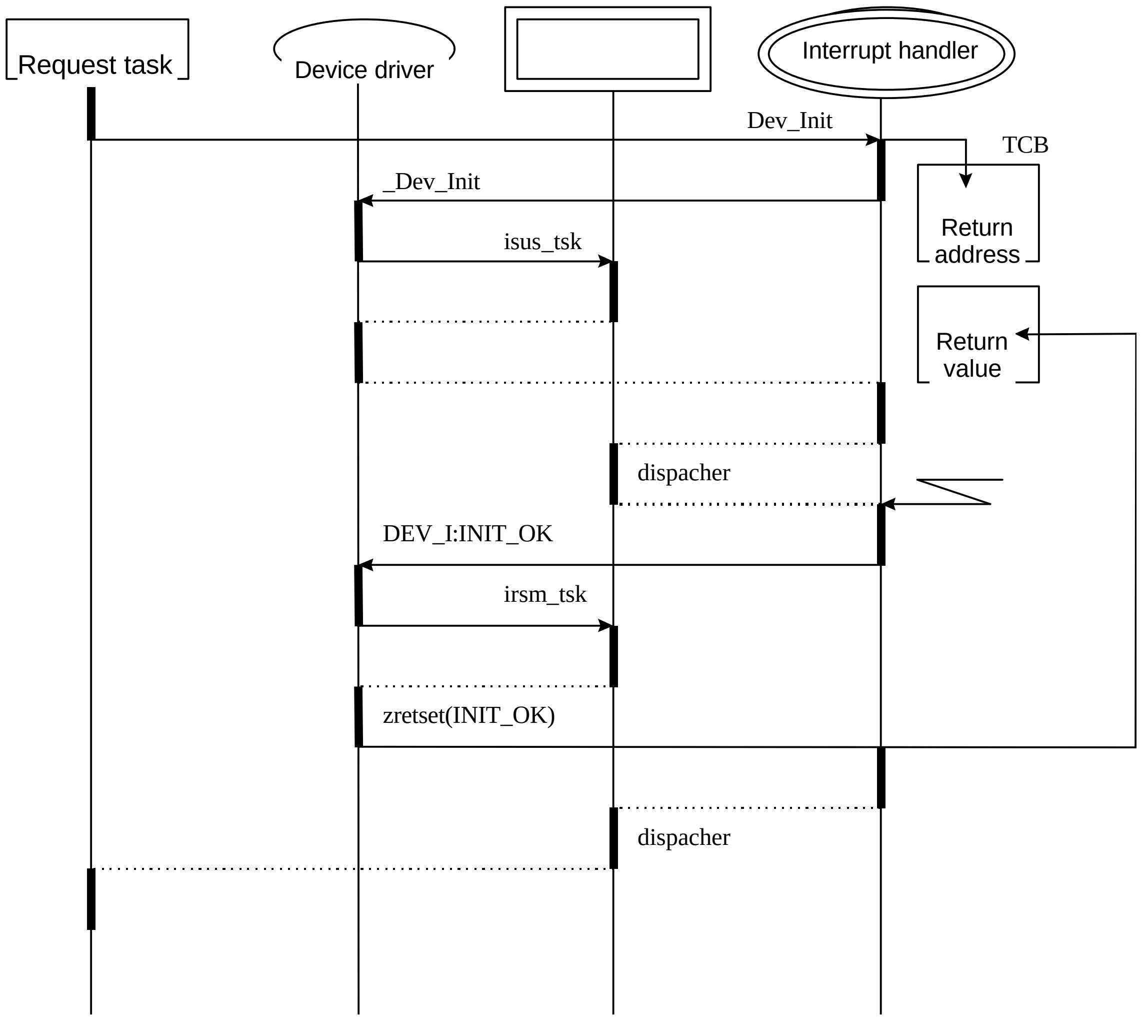


Figure 12- 11 Synchronous device driver sequence

In the device-driver STM example, DEV_I is set as an interrupt. This causes the creation of another loop that waits for an interrupt in addition to the interrupt handler. The device-driver STM transfers control to the created loop. The CPU resource, which would be used for the loop that just waits, can be saved by describing the RTOS synchronized event flag WAI_FLG(EV1) in the cell. To return the execution privilege to the request task after the device driver has received the specific end interrupt, zretset(ret_value) is called. This command sets the return value (ret_value) from the device driver to the TCB of the request task and switches the IO wait state held by the request task back to READY.
Scheduling then sends a request to the RTOS.

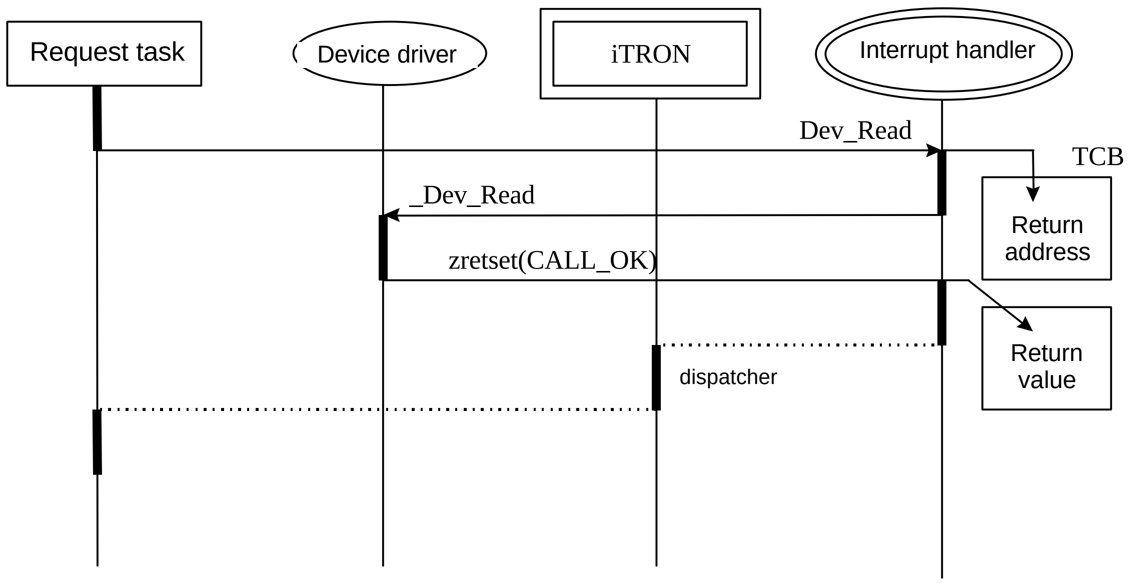


Figure 12- 12 Asynchronous device driver sequence

Dev_Read() is an asynchronous (calling return) type. The asynchronous type immediately returns the control to the caller after receiving the system call, even if processing has not been completed. Naturally, returning the control means making a request to the RTOS scheduler, so whether the control is actually returned or not depends on the circumstances of the task controlled by the RTOS at that time.
In the case of an asynchronous task, it will not know whether the read has been completed until a response has been received. Write it as the following in order to receive the response as a message:
rcv_msg(MB1);
The device-driver STM is a loop process that waits for an interrupt, since it is independent from tasks. Interrupt is disabled while the device-driver STM is in operation.

(*)zretset() is not a system call of EHSTM. The system call equivalent of zretset() depends on the specifications for the CASE tool that supports the EHSTM, because it varies depending on the RTOS type supported by the tool.

12.6. Interrupt-handler STM

The interrupt-handler STM is declared as follows:

STM name

Declaration example: Dev_1

Call example: Registered to the interrupt vector table

The interrupt-handler STM is designed so that an interrupt handler exists for each of the interrupt events and the handler is registered to the interrupt vector table of the CPU. The EHSTM design method does not cover such issues as the interrupt vector table, which must be shared because some CPUs have fewer interrupt vector table entries. In some RTOSs, the RTOS interrupt handler catches the interrupt first, then the individual interrupt handler is activated from the interrupt table provided within the RTOS. However, for its return, a mechanism similar to the one for the device-driver STM, such as zretset(), is necessary in the CASE tool.

In some cases, the STM exists as the interrupt-handler section. In other cases, the interrupt handler section is dispersed to the task STM or main STM. From the viewpoint of implementation, there are no problems in describing the interrupt handler (interrupt event) in the main section. If the interrupt handler is written in the task section, it is necessary to adjust it to the interrupt-handler mechanism of the implemented RTOS. When the interrupt-handler section (interrupt event) is described in the task section, it means "the task is always activated at the moment an interrupt occurs."

If the interrupt handler is created independently, the task or main section has to be notified of the occurrence of an interrupt. Normally, the task section uses an independent interrupt-handler section.

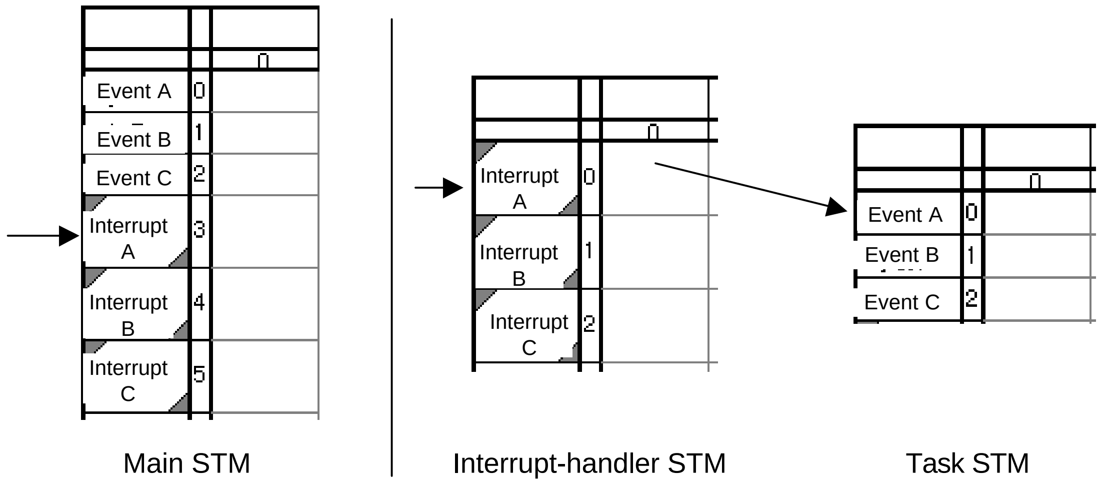


Figure 12- 13 Interrupt-handler section

:EHSTM Refer to 17, "EHSTM system call," for more about the EHSTM system call.

12.7. Device-register STM

The device-driver STM is declared the same way as the library STM.
Declaration example: 100[3](char* cpdata)
Call examples: read(lpara1,para2)
100: read(lpara1,para2)
During a system simulation the device-register STM is processed concurrently with the CPU, since it exists in the external environment.
If the RTOS is implemented, a device-driver STM is usually created, so it is not necessary to pay attention to the operation of the device itself. However, it is convenient to have the device's operations described in the form of an STM when the RTOS is not implemented, or for designers of device driver STMs or IO library (middleware) STMs.
Usually, communication with peripheral devices is performed via the command/status registers.

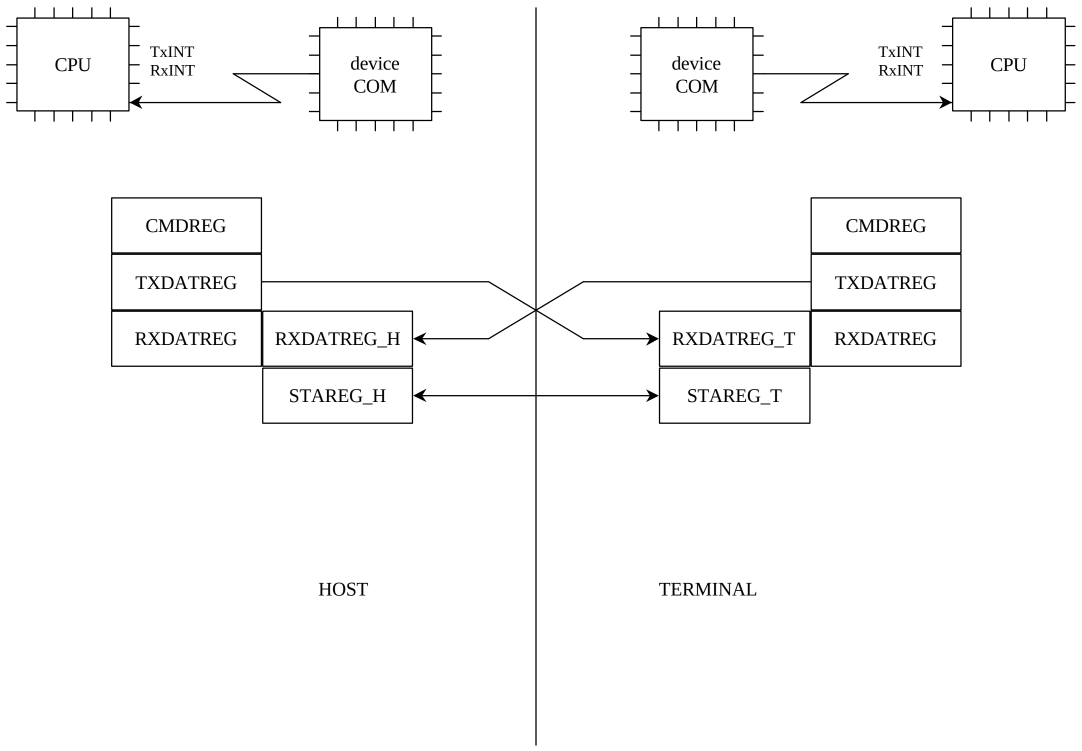


Figure 12- 14 An example of communication devices

Let's consider the example in Figure 12-14. There is a device COM, which is for communication. The CPU has access to CMDREG, TXDATREG, RXDATREG, and STAREG_H/T of COM. On the other hand, RXDATAREG_H/T cannot be accessed from the CPU. There must be another end, since it is a communication device. Assuming the other end also uses the same device, TXSATARREG - RXSATAREG_H/T and STATEREG_H - STATEREG_T are connected between devices. The connection here means to create an STM so that the registers of the other end can be referred to and set from both sides (HOST and TERMINAL).

ZEL_500 ComLSI				TX			RX	
				STOP	RUN			STOP	RUN
					b4800	b9600	b19200		
				0	1	2	3	4	5
setspeed (int ispeed)	ispeed	4800	0	=>b4800		=>b4800	=>b4800		
		9600	1	=>b9600	=>b9600		=>b9600		
		19200	2	=>b19200	=>b19200	=>b19200			
CMDREG | TXSTART /*activity*/ CMDREG &= TXCLR;			3	=>TX:RUN(P)	zsystim(T_O, 1666); iflag=OFF;	zsystim(T_O, 833); iflag=OFF;	zsystim(T_O, 416); iflag=OFF;		
CMDREG | TXSTOP /*activity*/ CMDREG &= TXCLR;			4		zsystim(T_O, 0); =>STOP	zsystim(T_O, 0); =>STOP	zsystim(T_O, 0); =>STOP		
CMDREG | RXSTART /*activity*/ CMDREG &= RXCLR;			5					=>RX:RUN	
CMDREG | RXSTOP /*activity*/ CMDREG &= RXCLR;			6						=>RX:STOP
STAREG_T == ON /*activity*/ STAREG_T = OFF;			7						RXDATREG = RXDATREG_T; INT(RxINT);
/*Tx TimeOut */ T_O			8		if(iflag==ON){ RXDATREG_H = TXDATREG; STAREG_H = ON; } else{ iflag = ON; } INT(TxINT);	if(iflag==ON){ RXDATREG_H = TXDATREG; STAREG_H = ON; } else{ iflag = ON; } INT(TxINT);	if(iflag==ON){ RXDATREG_H = TXDATREG; STAREG_H = ON; } else{ iflag = ON; } INT(TxINT);		

Figure 12- 15 Communication device register STM

According to the states, the transmission and reception (TX/RX) can operate concurrently in the COM. Unless setspeed() is executed, the transmission is not allowed. Event numbers 3 through 7 sense the CMDREG value as a flag event. If an arbitrary bit of the command register is on, the arbitrary processing is executed, and the bit is cleared by the event end activity when it is complete. Once the baud rate is set, it is not changed unless changed by setspeed(). This is known from the fact that the type of the transition from state number 1 by event number 3 is the deep history type. In order for COM to start transmission, it is also known from the matrix (state numbers 1 through 3 of event number 3) that the CPU side has to turn on the TXSTART bit of the CMDREG to on again and trigger an interrupt (zsystim) for transmission within the COM. The zsystim interrupt intervals are changed, depending on the baud-rate setting. When the first T_O (event number 8) interrupt occurs, it simply interrupts the CPU with an INT (TxINT), because the iflag is set to off. The CPU needs to set the transmission data to TXDATREG before the next interrupt is generated for the COM. The COM sets the EXDATREG value to the receive register RXDATREG_H of the opposite party (when its own side is the terminal side). Obviously, this is repeated as long as the transmission data exists. At the end of the

transmission, the TXSTOP bit has to be turned on in the CMDREG before the COM's internal transmission interrupt occurs.
The reception operation runs concurrently with the transmission operation. Reception will be on standby by turning on the RXSTART bit of CMDREG from the CPU. Then, when STAREG_T (when its own side is the terminal side) is turned ON, RXDATREG_T (when its own side is the terminal side) containing the value set by the opposite party is set in RXDATREG, and the reception interrupt INT (RxINT) is generated in the CPU. You can tell that the CPU side must read the value in RXDATREG before the next reception from COM occurs.

13. Clone STM

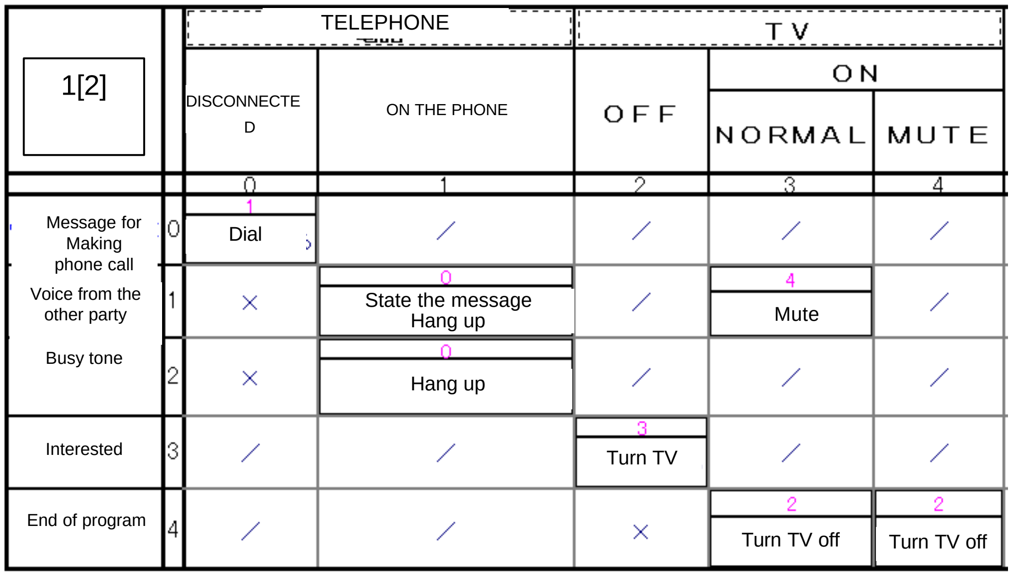
The clone STM generates multiple states from a single STM. “Multiple states” does not mean multiple concurrent states, but providing multiple sets of states to be controlled by the state scheduler.

The clone STM is declared as an array of the C programming language.
Declaration example:

```
int   1[3](lp1)
```

The meaning of the above declaration is, "The return value type is int; there are three clones for number 1 of the event driven STM; and receives argument lp1." To be precise, "three clones" means "one original and two clones." Clones are called by specifying the clone number. The clone number starts with 0. Clone number 0 is used to call the original. When the clone number is omitted, 0 is assumed.
Call example:

```
1[0](&var1)
```



1[2]		TELEPHONE		TV		
					ON	
		DISCONNECTED	ON THE PHONE	OFF	NORMAL	MUTE
		0	1	2	3	4
Message for Making phone call	0	1 Dial	/	/	/	/
Voice from the other party	1	×	0 State the message Hang up	/	4 Mute	/
Busy tone	2	×	0 Hang up	/	/	/
Interested	3	/	/	3 Turn TV	/	/
End of program	4	/	/	×	2 Turn TV off	2 Turn TV off

Figure 13- 1 The STM of the telephone and TV

When the STM of the telephone and TV declares the number of clone STMs as two, two sets of state-control variables are used.

Clone STM

```
State-control variable for Clone STM NO:0
          ■[Telephone]    ┌   ■[Disconnected]
                          └   ■[On the phone]
          ■[TV]
                          ┌   ■[OFF]
                          └   ■[ON]           ┌  ■[NORMAL]
                                              └  ■[MUTE]

State-control variable foe Clone STM NO:1
          ■[Telephone]    ┌   ■[Disconnected]
                          └   ■[On the phone]

          ■[TV]           ┌   ■[OFF]
                          └   ■[ON]           ┌  ■[NORMAL]
                                              └  ■[MUTE]

                          □:Dead ■:Suspended:○Ready:●Active
```

The clone STM can be used in all departments. The creation of clone STMs with respect to a task section does not produce multiple tasks. The same thing can be said of the device driver and library sections. Be especially careful with the device driver, since the device driver mechanism of the RTOS may be conducting the unit control. To realize exclusive control for clone STMs, the user must describe the contents in the action or activity. The EHSTM design method only provides the clone STM mechanism.

The clone STM is convenient when a communication-control protocol task controls channels or multiple partner stations by a single protocol.

The clone STM cannot use interrupt events.

14. STM main

An STM main is created for each main section and each task in the task section. The STM main calls the event-acquisition function and event-analyzer function. However, there is none in the library, device driver or interrupt-handler section. In these departments, the event-acquisition and event-analyzer functions are called within the functions generated by the function-call type or interrupt-type STM.
The STM main is closely related to the driven type.

14.1. STM main and event-driven type

The event-driven type has a program structure that waits for events at a single location. Therefore, the STM main acquires events and calls the event analyzer.

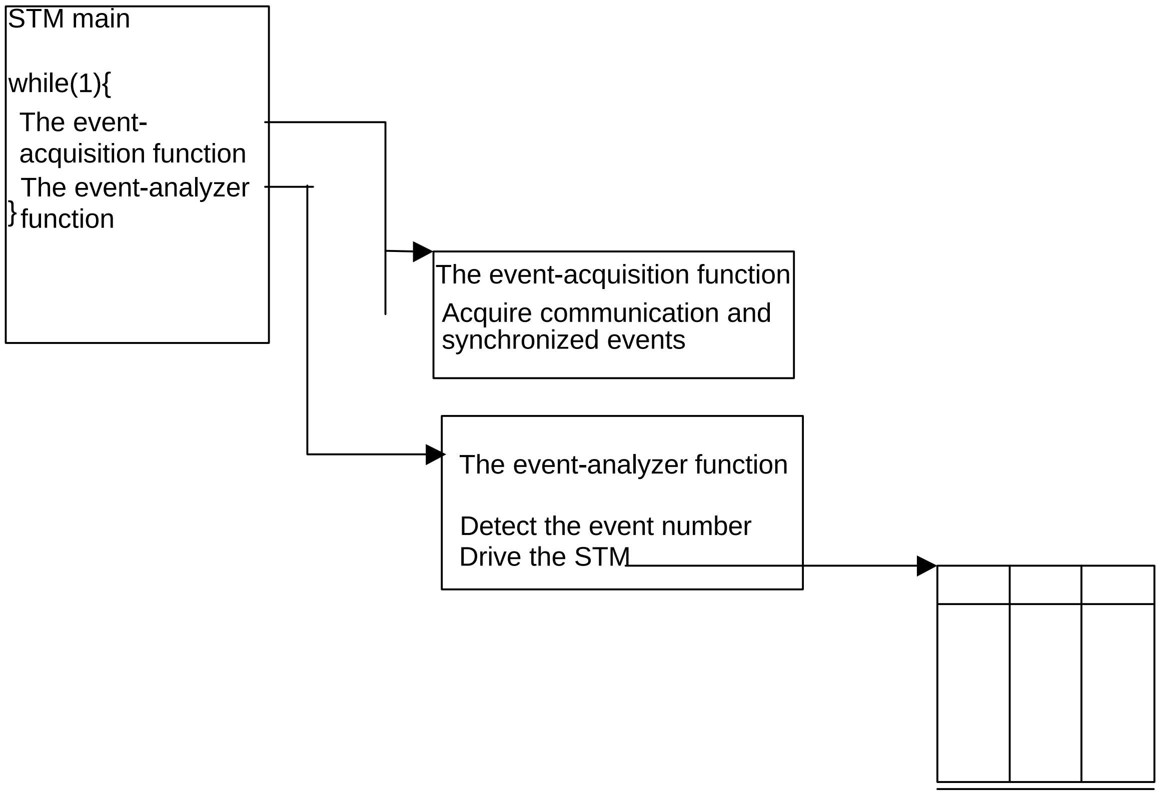

STM main
while(1){
The event-acquisition function
The event-analyzer function
}
The event-acquisition function
Acquire communication and synchronized events
The event-analyzer function
Detect the event number
Drive the STM

Figure 14- 1 STM main and event-driven type

14.2. STM main and state-driven type

The state-driven type has a structure that retrieves events at multiple locations in the program. Normally, polling-type events are used for the state-driven type. However, when the communication or synchronized type is used by the state-driven STM, events are acquired at a single location of the STM main by default. It is called "default" here because the tools that support the EHSTM design method will be likely to have specifications that allow selection from several event-acquisition locations. When polling is set as the event type, the event-acquisition function does not work. In the state-driven type, the state scheduler is called instead of the event-analyzer function from the STM main.

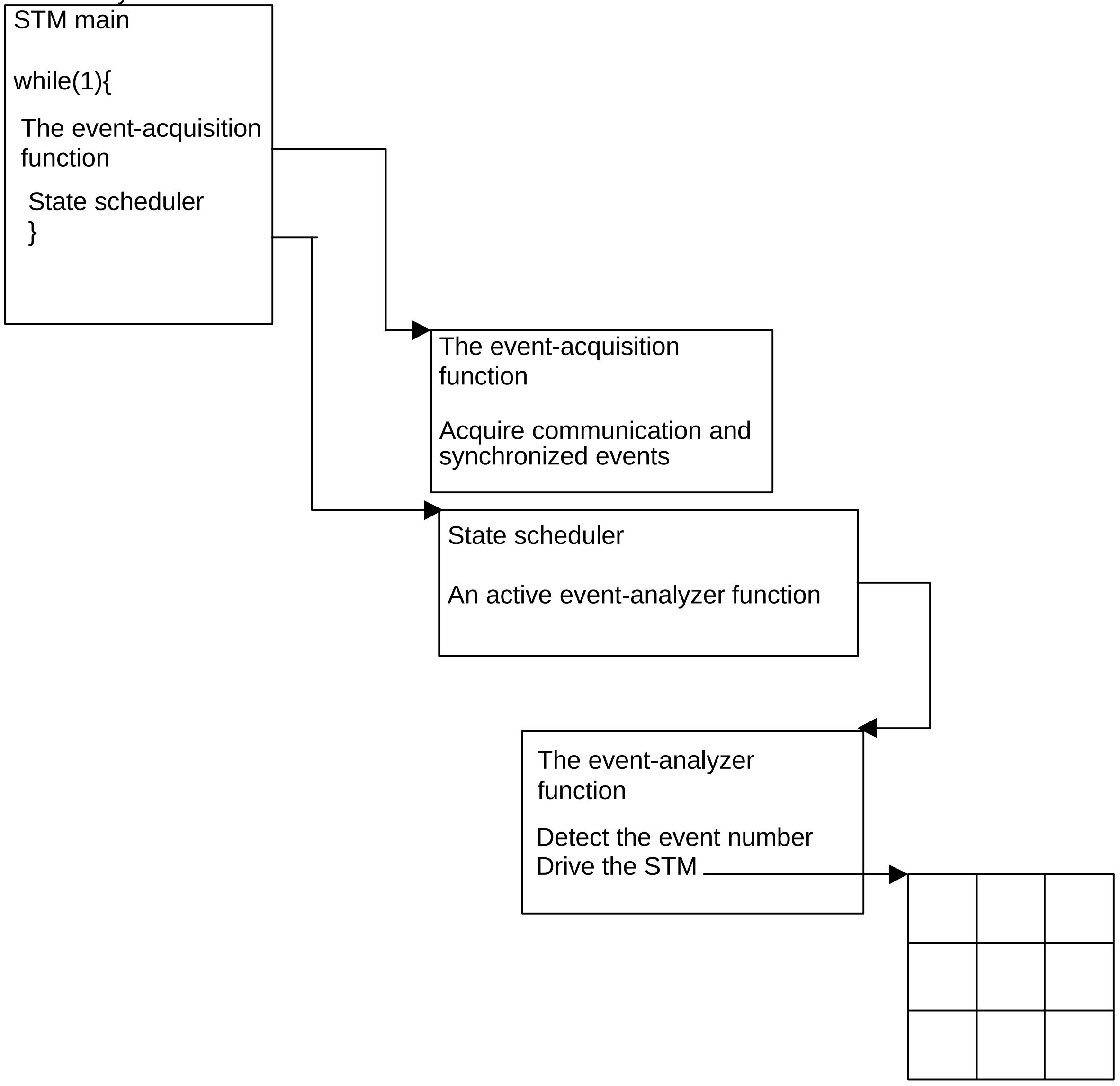

STM main
while(1){
The event-acquisition function
State scheduler
}
The event-acquisition function
Acquire communication and synchronized events
State scheduler
An active event-analyzer function
The event-analyzer function
Detect the event number
Drive the STM

Figure 14- 2 STM main and state-driven type

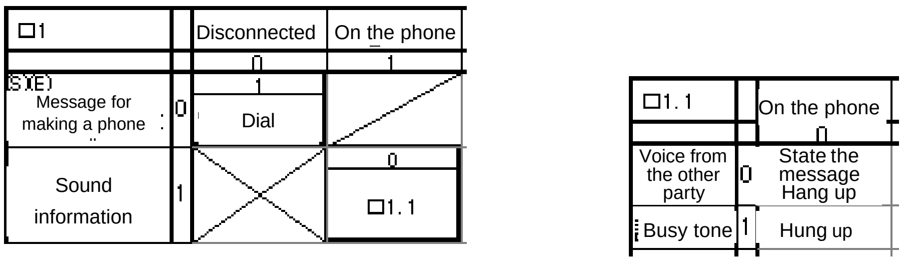
15. Hierarchy

The hierarchy describes the overview in the higher level and in more detail as the level becomes lower. Using the hierarchy a huge, complicated state transition can be represented in a compact, easy-to-understand structure. Normally, when the state transition is represented graphically -- in other words, in the case of the State Transition Diagram (STD) -- the hierarchy is implemented with respect to states. Harel's STD is the example. When the state transition is represented as a table -- in other words, in the case of the State Transition Matrix (STM) -- the hierarchy is implemented with respect to events. The EHSTM design method version 1 uses this method.
In the EHSTM design method version 2, the state hierarchy is included in the STM in addition to the event hierarchy.

🕮: Refer to 11, "Driven type," for the STM driven type.

15.1. Event hierarchy

In the event hierarchy, an overview event is captured in the higher-level STM, the higher-level STM notifies the lower-level STM of the event, and the lower-level STM captures it as a detailed event. Overview-detail relationships are constructed with respect to events, allowing the construction of a complicated large-scale system using compact, easy-to-understand STMs.
Let's take a look at an example of hierarchy for the telephone STM. We assume it is an event-driven STM.
Sound information



□1		Disconnected	On the phone
		0	1
(S)(E) Message for making a phone	0	1 Dial	
Sound information	1		0 □1.1

□1.1		On the phone
		0
Voice from the other party	0	State the message Hang up
Busy tone	1	Hung up

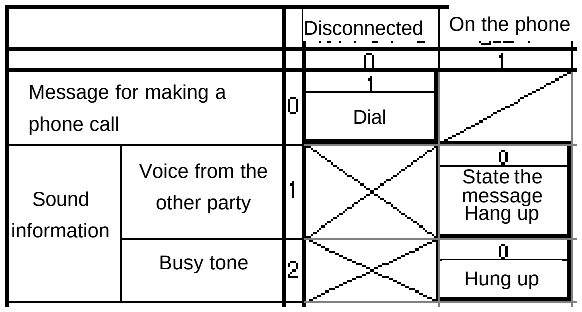
Figure 15- 1 Event hierarchy(1)

The "sound information" event and the "voice from the other party" and "busy tone" events have a hierarchical relationship.

"Sound information" ┌ "Voice from the other party"
└ "Busy tone"

Figure 15- 2 Event tree

This event tree structure can also be represented using the event virtual frame without using the hierarchy, and the user can choose the format to be used. Description in a single STM has the benefit of being able to confirm all combinations. The hierarchical structure has the benefits of reducing the number of "no use" and "invalid" cells, improving the ease of viewing and editing, and reducing the memory required when the STM is programmed in a table. (Usually, the total number of cells is reduced with the use of hierarchy, even though it is still six after implementing a hierarchy in the above example.)



			Disconnected	On the phone
			0	1
Message for making a phone call		0	1 Dial	
Sound information	Voice from the other party	1		0 State the message Hang up
	Busy tone	2		0 Hung up

Figure 15- 3 Event virtual frame

Let's take a look at another example of event hierarchy.
This is an STM that controls two drives (A and B).

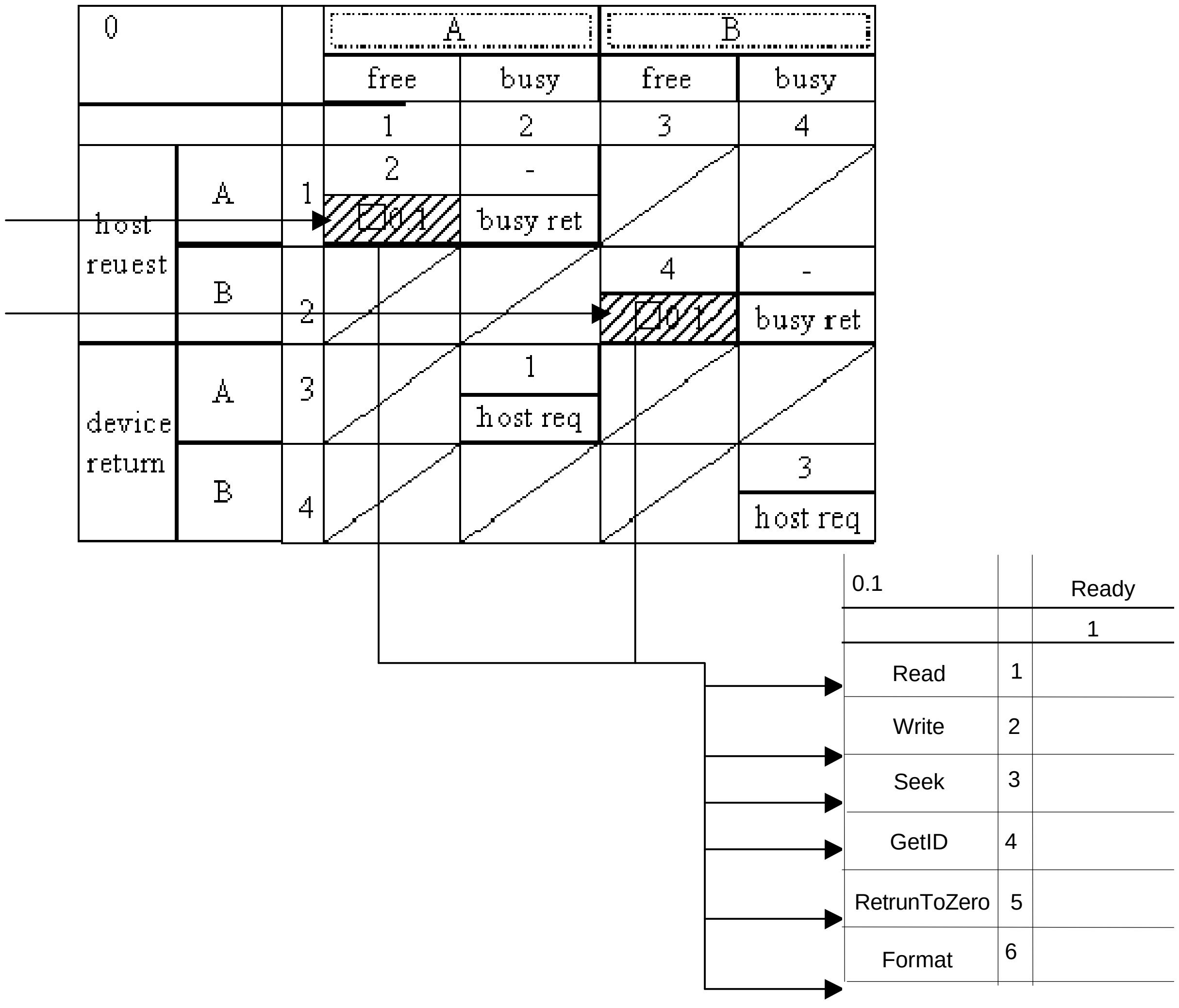


Figure 15- 4 Event hierarchy(2)

The parent STM(0) receives the "host request" by the event virtual frame, and "A:" and "B:" are treated as events. When the parent STM (0) is in the "free" state, the child STM (0.1) is called by inheriting the event received by the parent. The child STM classifies the event received by the parent STM into six detailed events.
The event hierarchy is represented using the STD, as follows:

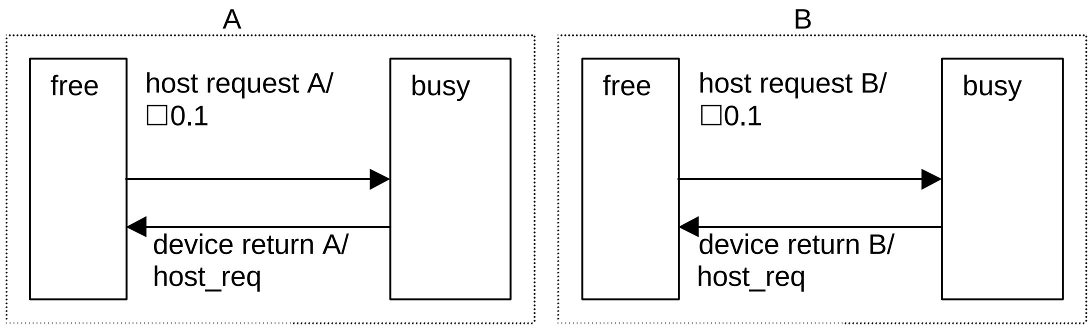


Figure 15- 5 Event hierarchy STD

The child STM in this case can also be written as a subroutine STM.

15.1.1. Call and return of event hierarchy

In the event hierarchy, the higher-level STM always receives events and passes these events on to the lower-level STM. The lower-level STM cannot receive events independently. If the parent cannot know all events of the child STM, it uses an "else" event.

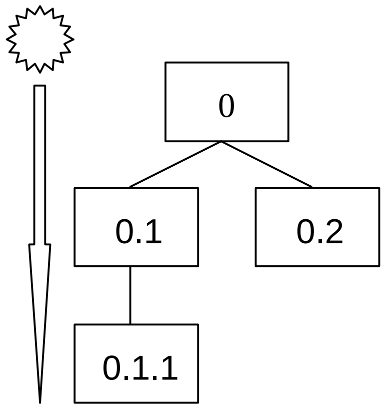


Figure 15- 6 Event hierarchy tree

The call of another event hierarchy is executed in the action cell. Therefore, an event-hierarchy STM can be called by passing arguments in the same way as a function call, and the called event-hierarchy STM can return with a value.

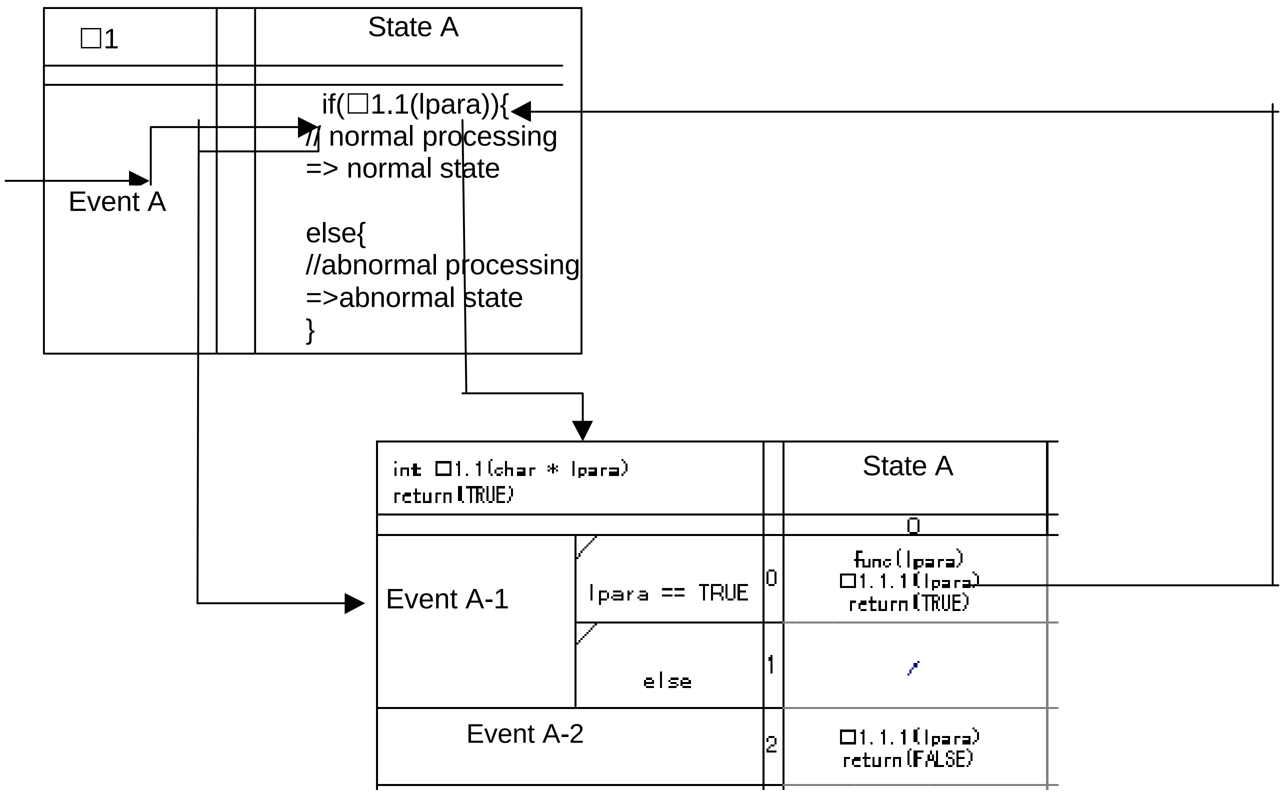


Figure 15- 7 Call and return of vent hierarchy

"Event A" is inherited from ☐1 to ☐1.1 as "Event A-1" and "Event A-2," and "lpara" is handed over when 1.1 is called. A "return" statement is executed when returning from ☐1.1 to ☐1. The return statement described in the STM declaration section is executed when no event hit has occurred.

The lower event number is determined by the higher-level STM, and the lower-level STM can be driven by the event number specified in the higher-level STM without executing the event analyzer of the lower level.

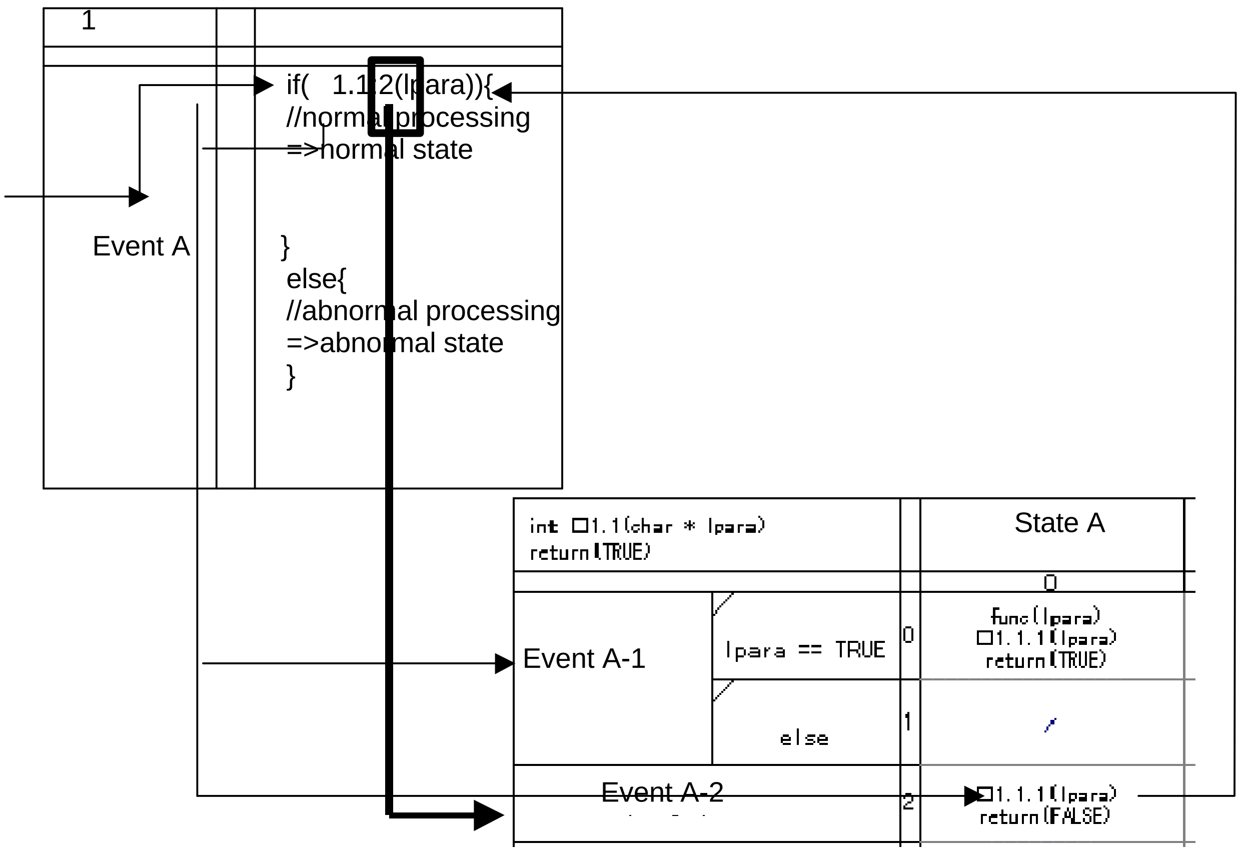


Figure 15- 8 Event number specification call

The child event (☐1.1) will be event number 2 ("Event A-2”), which is specified by the parent.
Event A is passed onto the grandchild ☐1.1.1).

🕮: Refer to 4.4.10, "Event virtual frame," for the event virtual frame.
Refer to 16, "Subroutine STM," for the subroutine STM.
Refer to 4.4.9, "ELSE event," for the else event.
Refer to 9, "Event analyzer," for the event analyzer.

15.2. State hierarchy

In the state hierarchy an overview state is captured in the higher-level STM, while detailed states are captured in the lower-level STM. Overview-detail relationships are constructed with respect to states, which allows the construction of a complicated large-scale system using compact, easy-to-understand STMs.

Let's take a look at a hierarchical STM for the telephone example. We assume it is an event-driven STM.

■1		Telephone ■1.1	TV ■1.2
		0	1

■1.1		Disconnected	On the phone
		0	1
Message for making a phone call	0	1 Dial	/
Busy tone	1	X	0 State the message Hang up
Voice from the other party	2	X	0 Hung up

■1.2		OFF	ON	
			NORMAL	MUTE
		0	1	2
Voice from the other	0	/	MUTE Mute	/
Interested program	1	ON (D) Turn TV on	/	/
End of program	2	X	OFF Turn TV off	OFF Turn TV off

Figure 15- 9 State hierarchy transition matrix (1)

1.2 in Figure 15-9 can set in a hierarchy with a deeper level.

■1.2		OFF	ON ■1. 2. 1
		0	1
Interested program	0	ON (D) Turn TV on	/
End of program	1	X	OFF Turn TV off

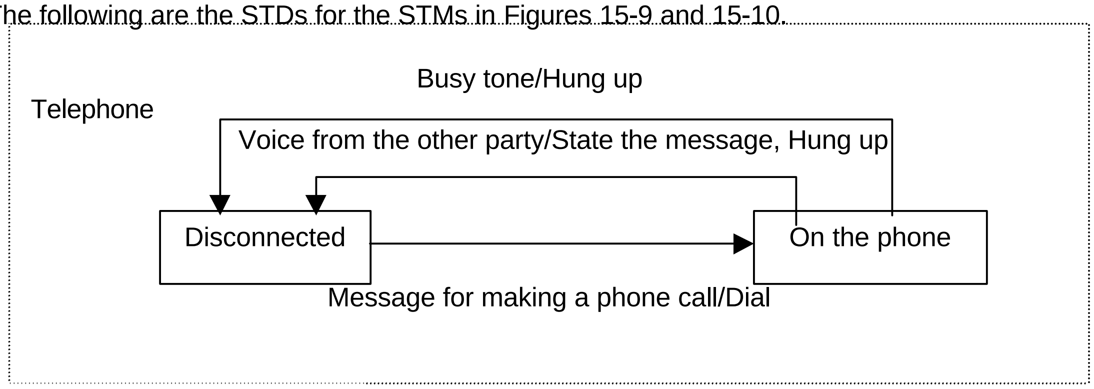
Figure 15- 10 State hierarchy transition matrix (2)

Compared with the 5x5=25 cells in the STM shown in Section 5.2 before implementing the hierarchy (Figure 15-10), the number of cells has been reduced to 2x0+2x3+3x3=15 in the state-hierarchy STM(1) and to 2x0+2x3+2x2+1x2=12 in the state-hierarchy STM(2). The decision whether or not to use state hierarchy is determined in the same manner as the event hierarchy. The state tree will be as below:

```
[Telephone]   ┌ [Disconnected]
              └ [On the phone]
[TV]       ┌ [OFF]
           └ [ON]      ┌ [NORMAL]
                       └ [MUTE]
```

Figure 15- 11 State tree

The following are the STDs for the STMs in Figures 15-9 and 15-10.


Busy tone/Hung up
Telephone
Voice from the other party/State the message, Hung up
Disconnected
On the phone
Message for making a phone call/Dial

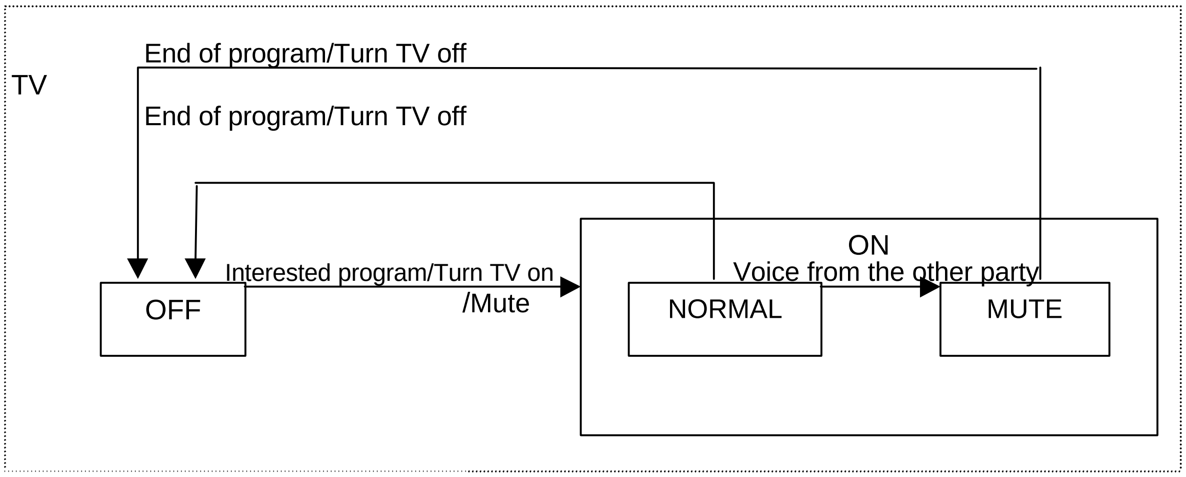

End of program/Turn TV off
TV
End of program/Turn TV off
ON
Interested program/Turn TV on
Voice from the other party
/Mute
OFF
NORMAL
MUTE

Figure 15- 12 State hierarchy STD (1)

The state hierarchy for those STMs is expressed the same way by the STD. In the STD for TV activity in Figure 15-12, when the transition from "ON" to "OFF" is written as shown in Figure 15-13, the STM uses the state actual frame as shown in Figure 15-14.

◷ Hierarchy

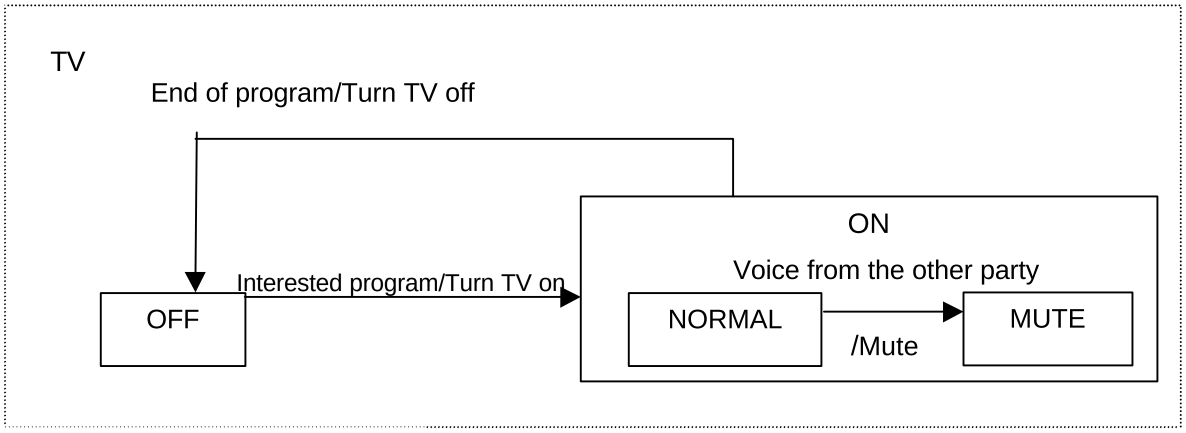
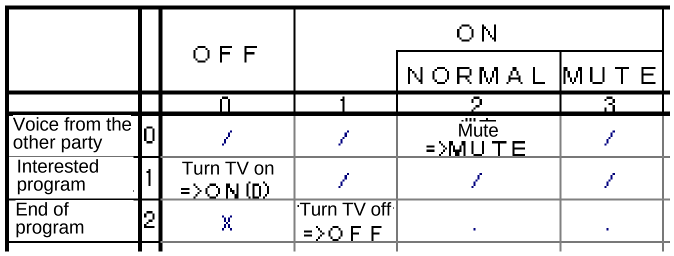

TV
End of program/Turn TV off
ON
Voice from the other party
Interested program/Turn TV on
OFF
NORMAL
MUTE
/Mute

Figure 15- 13 State hierarchy STD (2)



		OFF	ON		
				NORMAL	MUTE
		0	1	2	3
Voice from the other party	0	/	/	Mute =>MUTE	/
Interested program	1	Turn TV on =>ON(D)	/	/	/
End of program	2	X	Turn TV off =>OFF	.	.

Figure 15- 14 State actual frame (3)

🕮: Refer to 5.3.6, "State actual frame," for the state actual frame.

15.2.1. Call and return of state hierarchy

The call of a state-hierarchy STM is different from the call of an event-hierarchy STM. The state-hierarchy STM does not make a call at the interval at which an event occurs. A call is executed when a transition occurs and the state becomes active. Whether the called state-hierarchy STM inherits the event from the parent or acquires a new event by itself depends on the STM driven type. While the event-hierarchy STM is called and returns as if it was a function, there are no arguments or return values for the state-hierarchy STM because it is called as part of an active state.

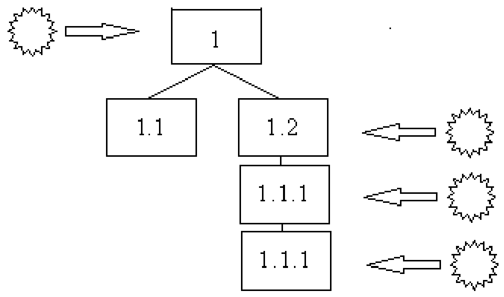


Figure 15- 15 Call and return of state hierarchy

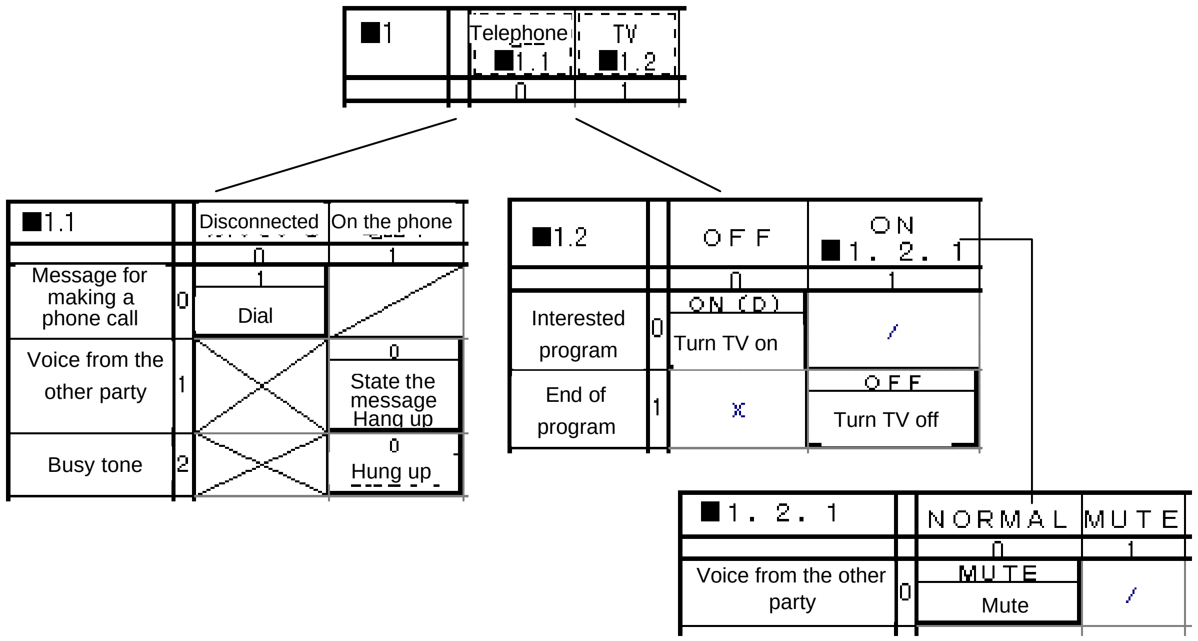


Figure 15- 16 State hierarchy STM (3)

Let's trace the actions in Figure 15-16. Assume the STM is a state-driven type. It is easier to understand if the state tree is used as a reference.

The first ready states are "Telephone" and "TV." "Disconnected" and "OFF," which are their respective child states, are in the READY state.

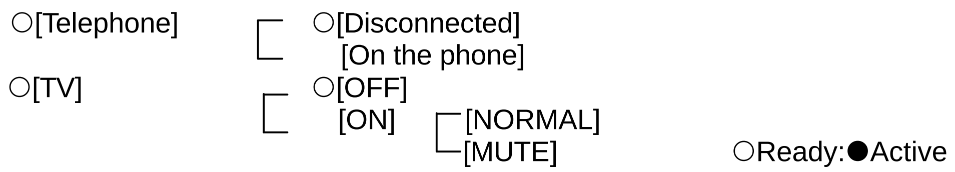


Figure 15- 17 State hierarchy scheduling (1)

Only one of the ready states becomes active. The active state is switched via a dispatch. Two specifications are available for scheduling of the dispatch:
(1) Vertical scheduling (V-type)
(2) Horizontal scheduling (H-type)

In concurrent states, two specifications are available for determining the priority:
(1) Default scheduling (D-type)
(2) Dispatch scheduling (P-type)

These scheduling types will be explained in the "STM driven type" section.

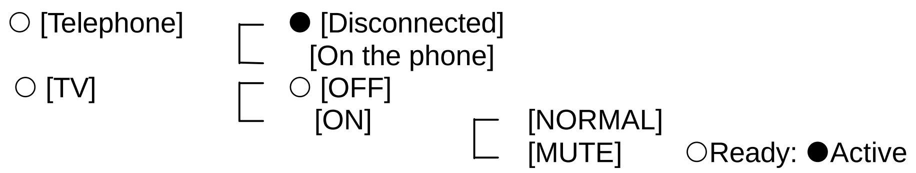


Figure 15- 18 State hierarchy scheduling (2)

Being active means that the event is accepted. Assume an "interested program" event occurs when the "Disconnected" state is active. This event is not known until the "OFF" state becomes active. This is characteristic of the state-driven type. If it is event-driven, the root STM is first initiated at the moment the event occurs, then each STM is called. In the state-driven type, whether or not the event has occurred is not known until the state becomes active because the active state looks for the occurrence of an event. Therefore, using the synchronized or communication-event type in the state-driven type will cause problems. This ia explained in the "STM driven type" section.

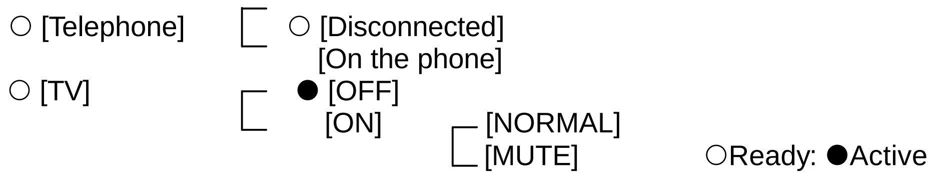


Figure 15- 19 State hierarchy scheduling (3)

When "OFF" becomes active and the "Interested program" event is captured, a transition to "ON(D)" occurs.

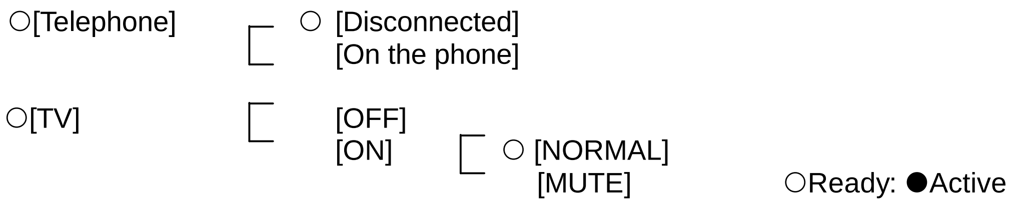


Figure 15- 20 State hierarchy scheduling (4)

Assume a "Message for making a phone call" event occurs when the "ON" state is active. The occurrence of this event will not be known until the "Telephone" state becomes active.

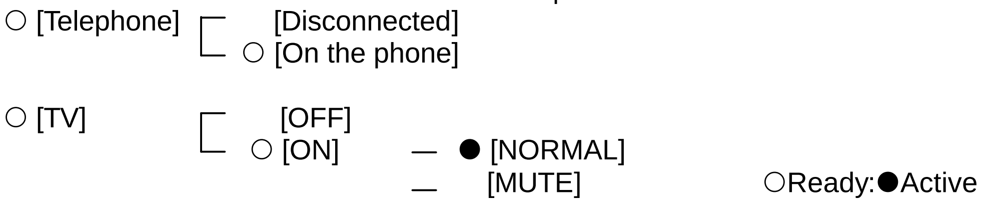


Figure 15- 21 State hierarchy scheduling (5)

Assume a "Voice from the other party" event occurs in the above state. This "Voice from the other party" event is described so that it is processed in the "Telephone" and "TV" concurrent states. Since they are designed as concurrent on the STM, they can be processed at the same time. Unless an RTOS supporting multiple CPUs is installed, however, in reality one of them is processed first. Which state should be processed first cannot be specified, but it is determined depending on the dispatch situation. To specify that one of them always has processing priority, the state hierarchy should not be used as in Figure 17 of Section 5.2, but the event-driven STM should be used. By using the default scheduling for concurrent states, the "Voice from the other party" event always accepts the "Telephone" state first in Table 17. Then the "State the message, hang up" action is executed and the "Voice from the other party" is accepted in the "TV" state. This is essentially the opposite of normal actions. In fact, a default mark (@) should be attached to the "TV" state because "Mute" should be executed first. The order of event acceptance is susceptible to change, due to the state scheduler. Even if a default mark is attached, this is only for specifying the order of scheduling but not for specifying the order of events to be processed. Therefore, nothing can be done further if a "Voice from the other party" event is detected while the "Busy" state is active. The driven type should be the event-driven type if the orders of events are important. Refer to Section 14.10 for more about this issue.
The "Voice from the other party" event is usually defined so that its occurrence made known by the change of a variable. Moreover, the lifespan of the "Voice from the other party" event is one of the most important items to design.

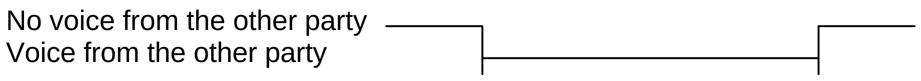


Figure 15- 22 Lifespan of an event

The occurrence of an event is captured by the event analyzer. The events are also cleared by the event analyzer. At this time, in the case of a concurrent state of the state-driven type, the events cannot be cleared by the analyzer independently because other states might also need it. In this case, processing such as a synchronized counter will be necessary for the event analyzer.

: Refer to 11, "Driven type," for more about the STM-driven type. Refer to 8.3, "Dispatch activity," for more about dispatch activity. Refer to 9, "Event analyzer," for more about the event analyzer.

15.3. Hierarchy and event

Events are inherited for communication events, synchronized events and in-mail events. Inheritance means that the child STM analyzes events acquired by the parent. In other words, the event inheritance deals with the location in which the event is acquired; namely, whether it is inherited or new events are handled. Normally, if the event-acquisition function is called from a single location of the STM main, the events acquired there will be inherited by the children of the event hierarchy. If the event acquisition is called from the state scheduler, events are not inherited and newly acquired ones are always used.

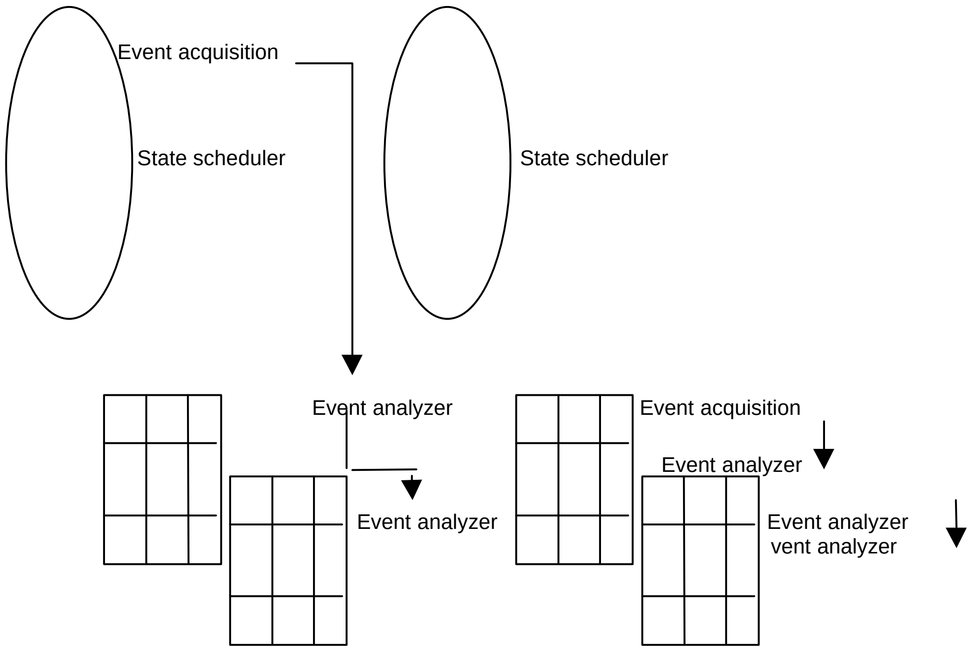


Figure 15- 23 Event inheritance

15.4. Hierarchy and state

There are two types of states: the exclusive type and concurrent type. The relationship of the hierarchy with regard to state type is the same as that of the parent-and-child relationship of the state frame.

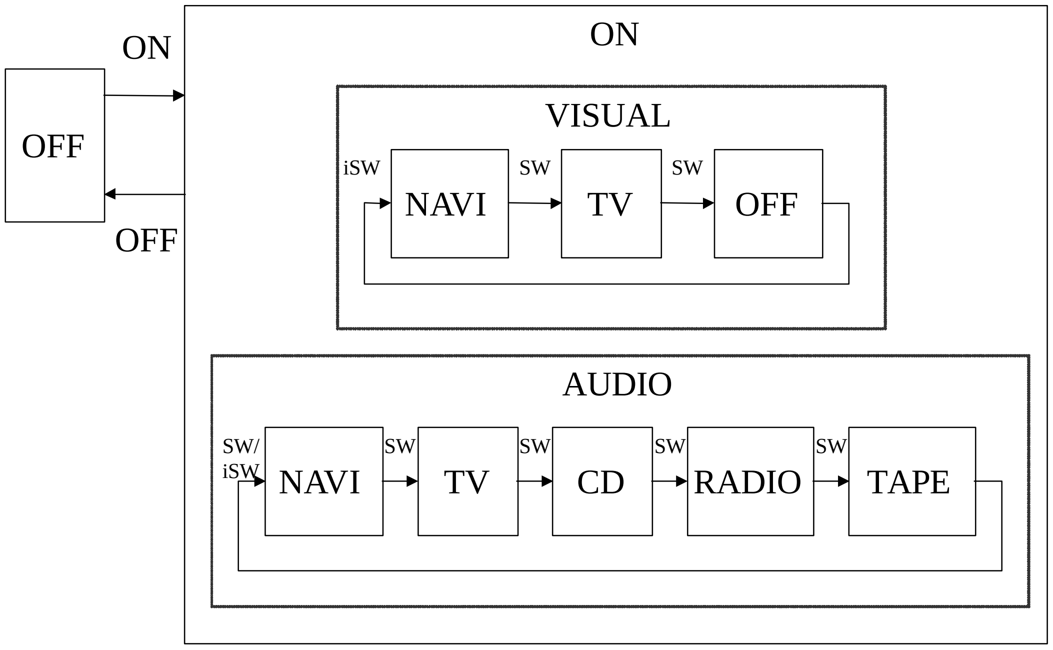
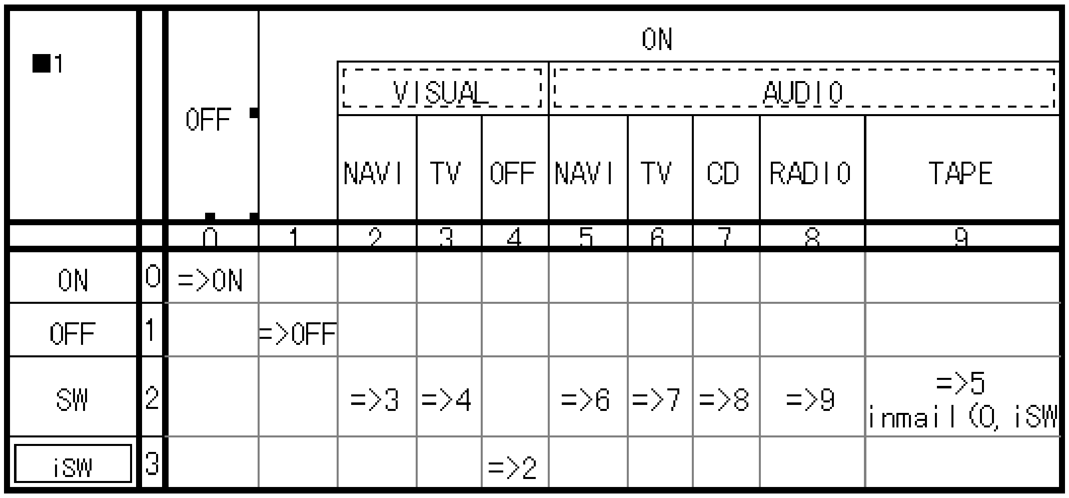


Figure 15- 24 Car audio STD

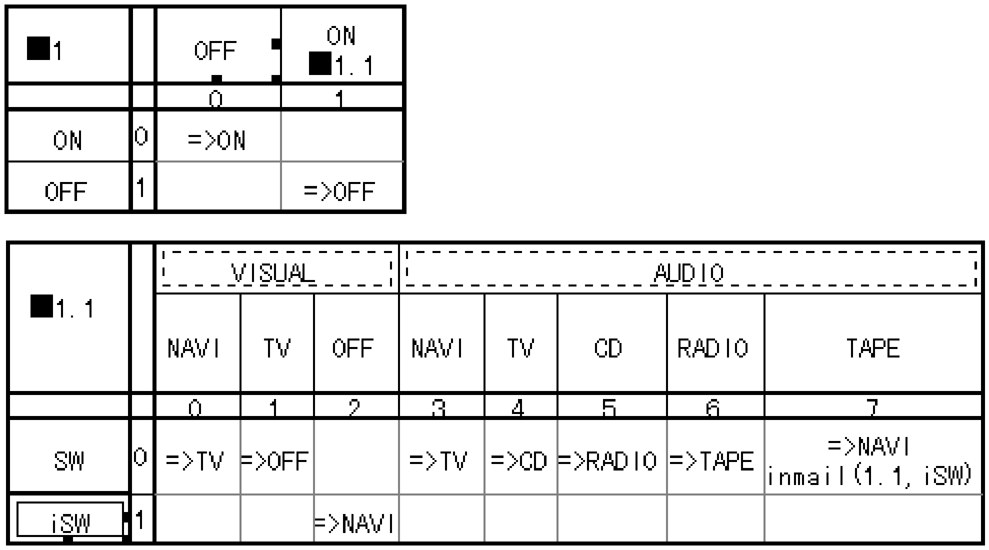
Assume a State Transition Diagram (STD) as shown in Figure 15-24. The state frame STM for this diagram will be as shown in Figure 15-25:



■1		OFF	ON	ON	ON	ON	ON	ON	ON	ON
				VISUAL			AUDIO			
				NAVI	TV	OFF	NAVI	TV	CD	RADIO	TAPE
		0	1	2	3	4	5	6	7	8	9
ON	0	=>ON									
OFF	1		=>OFF								
SW	2			=>3	=>4		=>6	=>7	=>8	=>9	=>5 inmail(0, iSW
iSW	3					=>2					

Figure 15- 25 State frame STM for a car audio system

The state-hierarchy STM without a state frame is shown in Figure 15-26.



■1		OFF	ON ■1.1
		0	1
ON	0	=>ON	
OFF	1		=>OFF

■1.1		VISUAL			AUDIO				
		NAVI	TV	OFF	NAVI	TV	CD	RADIO	TAPE
		0	1	2	3	4	5	6	7
SW	0	=>TV	=>OFF		=>TV	=>CD	=>RADIO	=>TAPE	=>NAVI inmail(1.1, iSW)
iSW	1			=>NAVI					

Figure 15- 26 State-hierarchy STM for a car audio system

🕮: Refer to 5.3.2, "State frame," for more about the state frame.

15.5. Hierarchy and action

A call for a state-hierarchy STM cannot be written in the action. Event-hierarchy STMs include the ones with hierarchical events and those with hierarchical actions. With hierarchical events, the event type of the child STM in the hierarchy is the same as the parent that initiated the call. With hierarchical actions, the timing of the call is after the parent receives the event, but the event types could be different. The child could be a flag type while the parent is a message type.

15.6. Hierarchy and transition

There are three types of transitions: the default type, history type and deep-history type. A local transition is a transition within its own STM. A global transition is a transition to an STM of another hierarchy within the same department. Transitions can be made to both the event hierarchy and state hierarchy in the same manner. No transition is allowed beyond the department. For example, a transition to the state in the library STM is not allowed from the task STM.
When a hierarchy is used, the transition symbol "-" indicates that priority is given to the transition within the child STM that has been called.

□1		S1	S2
		1	2
E1	1	1	
		□1.1	
E2	2	-	
		□1.1	

□1.1		SA
		1
EA	1	1>2/-

Figure 15- 27 Hierarchy and transition

When □1.1 is called by E1, the □1.1 has 1>2, causing a transition to 2 of the parent state □1. When it returns to the parent, the parent makes a transition to state 1 because 1 is explicitly specified.
When □1.1 is called by E1, the □1.1 has 1>2, causing a transition to 2 of the parent state □1. When it returns to the parent, the parent stays in state 2 because "-" is specified.

15.7. Hierarchy and activity

There are four types of activities: the event activity, state activity, dispatch activity and trigger activity. The event activity and state activity function in the same way for both the event hierarchy and state hierarchy. The dispatch activity and trigger activity function only for the root level. They do not exist in hierarchical STMs.

15.8. Hierarchy and event analyzer

Event analyzers are created for the number of hierarchical STMs in the case of event-driven STMs. For state-driven STMs, an event analyzer is provided for each state.
There are two processes in the event analyzer: event retrieval and analysis of the retrieved event. By default, the child uses the event retrieved by the parent for the communication type, synchronized type and in-mail type, as described in Section 14.3. If there are any problems because of this, they are solved by including the event retrieval process in the event analyzer. In this case, the event inheritance will not be the same as the table shown in Section 14.3.

15.9. Hierarchy and state scheduler

The state scheduler determines the schedule for the parent-and-child relationship among states. Since the event hierarchy and state hierarchy can coexist, the state scheduler functions in either type of hierarchy.

15.10. Hierarchy and driven type

The driven type is specified for the highest-level STM. The driven type cannot be specified for each hierarchical STM. The event-driven type has a structure that waits for events at a single location in the highest-level (root) STM, even if there are multiple hierarchical STMs. The state-driven type has a structure in which an event analyzer is provided for each hierarchy state, and the active event analyzer is executed by the state scheduler. When the event-hierarchy STM is called in the state-driven type, the called event-hierarchy STM is driven as the event-driven type. This is because the event-hierarchy STM is described in the action and behaves like a function. Conversely, when the parent calls a state-hierarchy STM in the event-driven type, the called state-hierarchy STM is the event-driven type. This is because the state hierarchy is a hierarchical state tree, and the crossed section of an event and one of the multiple ready states is simply called. There is no event analyzer for each state.
Let's discuss the state-driven type using the phone call example in Figure 15-1.

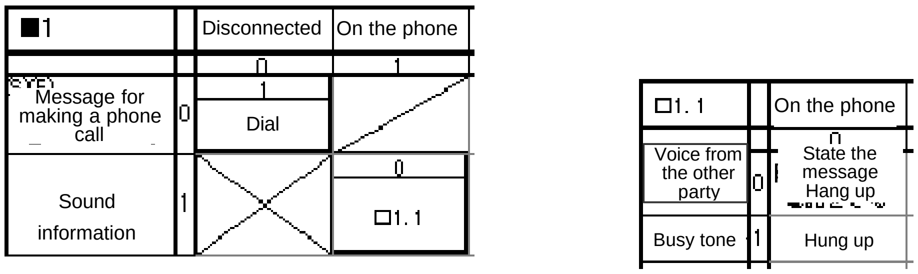


Figure 15- 28 State-driven telephone

Because the state is exclusive, the "Disconnected" state has been activated by the state scheduler at the beginning. Because it is a state-driven type, the event analyzer prepared for each state is called. In this example, the event analyzer is monitoring the occurrence of a "Message for making a phone call." When a "Message for making a phone call" occurs, "Dial" is executed and a transition to "Busy" occurs. The "Busy" state now becomes active and the "Sound information" event is monitored. When a "Sound information" event occurs, the child STM "   1.1" is called by inheriting the "Sound information" event. The called event-hierarchy STM of   1.1 is an event-driven type.

Next, let's look at the example of the telephone and TV represented by the event-hierarchy STM and state-hierarchy STM, to discuss the event-driven type.

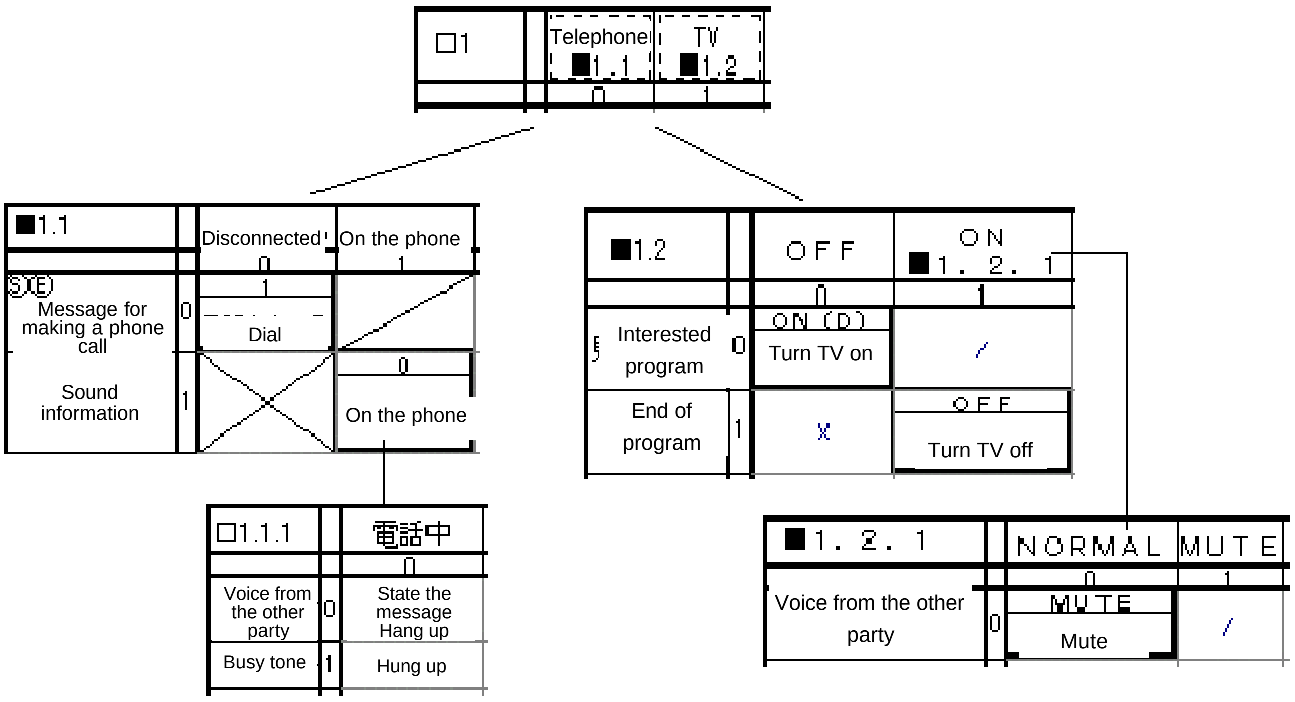


Figure 15- 29 Event-driven telephone and TV

In this example there are no events in the root STM. Even though the event-driven type seems to do nothing because it waits for events at the root, the event analyzers provided for each state-hierarchy STM are activated because of the state hierarchy. The event analyzer to be activated here requires that the parent calling be in the ready state. The event analyzer is not provided for each state. Assume that the state scheduler is managing the state as follows:

```
○[Telephone]   ┌  [Disconnected]
               └ ○[On the phone]
                  [On the phone]
○[TV]          ┌  [OFF]
               └ ○[ON]       ┌ ○[NORMAL]
                             └  [MUTE]        ○Ready:●Active
```

Figure 15- 30 State tree

The states of the event-hierarchy STM are not included in the tree by the state scheduler, but are managed as states of the STM itself. The states of the event-driven type become active only when the events are analyzed. The order of calling event analyzers follows the specifications for the parent schedule, concurrent schedule and tree schedule by the state scheduler. The states of the event-hierarchy STM are handled as states of the STM itself, and are processed in the order of action execution in which the event-hierarchy STM call is described.

Because it is event-driven, it waits for events at a single location of □1. Suppose that "Sound information - Voice from the other party" event occurs. Naturally, nothing happens because there is no event analyzer for □1. Next, the event analyzer of the state-hierarchy STM ■1.1 of the "Telephone," which is a ready state, is called (assuming the concurrent schedule is the default type). The "Sound information" is analyzed as event number 1. The action where the "Busy" event and the active state "Sound information" cross is called. □1.1.1 is called by inheriting the "Sound information - Voice from the other party" event. The "State the message - Hang up" action is executed, and "Disconnected" is changed to ready. The event analyzer of the state-hierarchy STM■1.2 of the "TV" state is called and "Sound information - Voice from the other party" is analyzed, but there is no hit. The event analyzer of the state-hierarchy STM■1.2.1 of "On," which is a ready state, hits "Sound information - Voice from the other party" as "Voice from the other party." The "Mute" action, located at the cross section of "NORMAL" and "Voice from the other party," is executed, and a state transition occurs to "MUTE." Now the control returns to the parent and waits for the next event. The difference from the state-driven type is that the event-driven type can specify the processing priority of events. Refer to Chapter 14.2.1.

In the event-driven type, an event hierarchy is called when the action is executed.
In the event-driven type, a state hierarchy is called when the parent's state is ready.
In the state-driven type, an event hierarchy is called when the action is executed.

In the state-driven type, a state hierarchy is called when the state is ready.

15.11. Hierarchy and department

The task section, main section, library section, device-driver section and device-register section can be represented by hierarchies. Subroutines, which are covered in Chapter 16, can also be represented by hierarchies.

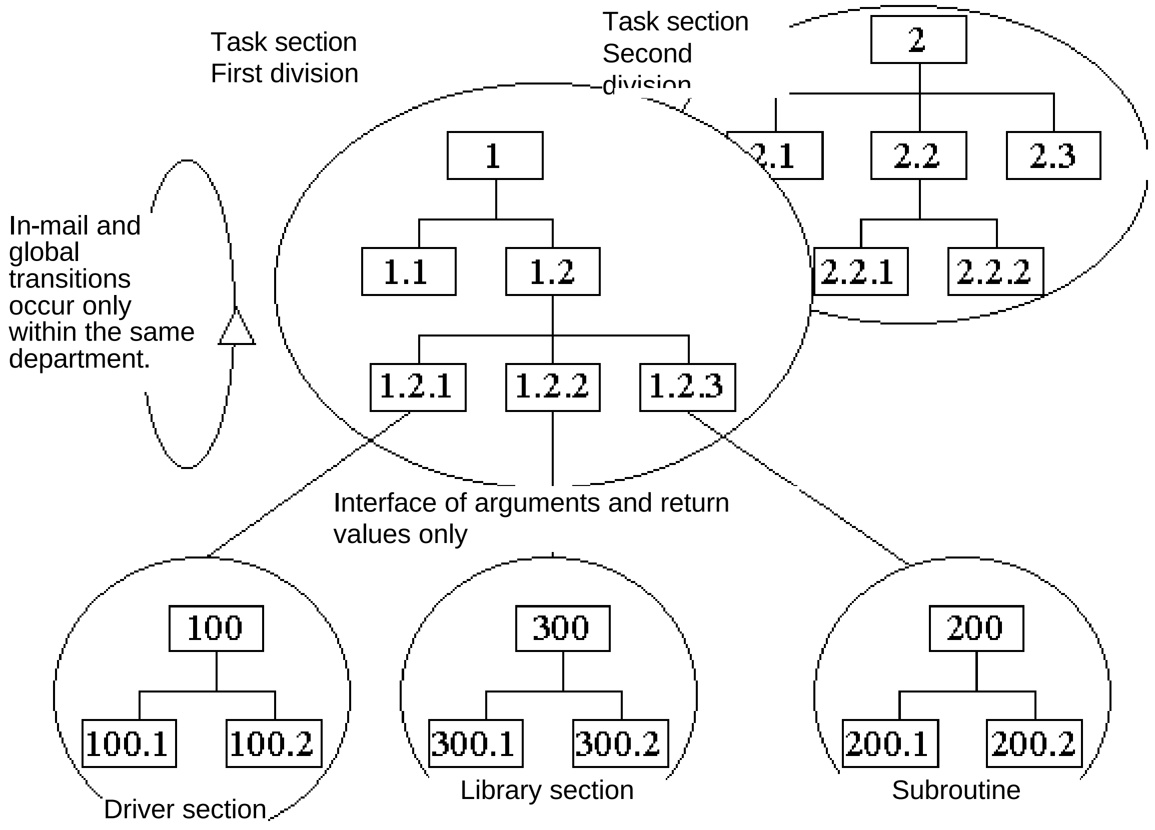
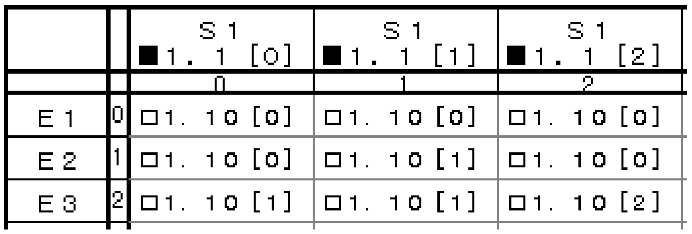


Figure 15- 31 Departments

The task STM and main STM are represented by □ or ■.
The library, device driver and device register sections are represented by ○or ●.
The subroutine section is represented by △ or ▲.
The □, ○and △at the root level indicate that it is event-driven and that the child level is an event hierarchy. However, they indicate that the subroutine section is an event hierarchy even at the root level. The subroutine takes over the driven type of the caller.
The ■, ● and ▲ at the root level indicate that it is state-driven and that the child level is a state hierarchy. However, they indicate that the subroutine section is a state hierarchy even at the root level. The subroutine takes over the driven type of the caller.

15.12. Hierarchy and clone STM

Both the event hierarchy and state hierarchy can be represented by a clone STM.



		S 1 ■1. 1 [0]	S 1 ■1. 1 [1]	S 1 ■1. 1 [2]
		0	1	2
E 1	0	□1. 10 [0]	□1. 10 [0]	□1. 10 [0]
E 2	1	□1. 10 [0]	□1. 10 [1]	□1. 10 [0]
E 3	2	□1. 10 [1]	□1. 10 [1]	□1. 10 [2]

Figure 15- 32 Hierarchical clone STM

A multiple of the same state hierarchy cannot be called, since a state hierarchy is part of the state tree and scheduled by the state scheduler.

15.13. Format of hierarchy

The event hierarchy is represented by □ and state hierarchy by ■. A call of an event hierarchy is described in the action cell and that of a state hierarchy in the state area. Multiple events can be called. The calling of multiple state hierarchies is not allowed.
The hierarchy level number is represented as follows:
parent number. child number
The parent number starts with 0. The child number starts with 1. The level numbers for the event hierarchy and state hierarchy cannot be the same. The child numbers need not necessarily be sequential, as long as they are unique.

🕮: Refer to 12, "Department," for the format of declaration and call of the hierarchy.

16. Subroutine STM

The subroutine STM is similar to the hierarchical STM. Each department (task section, main section, library section and device-driver section) can use subroutine STMs. The subroutine STM can be changed to a clone STM, and also to a hierarchy.
The hierarchical STM and subroutine STM are different in the following ways:
(1) While the hierarchy STM can only make a call through the hierarchy tree, subroutine STM can be called from anywhere. The state subroutine STM can be called from only one location, the same as the state hierarchy.
(2) The state control variable can be prepared by the caller side and passed to the subroutine STM as an argument. The subroutine STM for which the state-control variable is prepared by the caller side is called an external subroutine. On the other hand, the subroutine STM for which the state-control variable is prepared internally is called an internal subroutine.

16.1. Call and return of subroutine STM

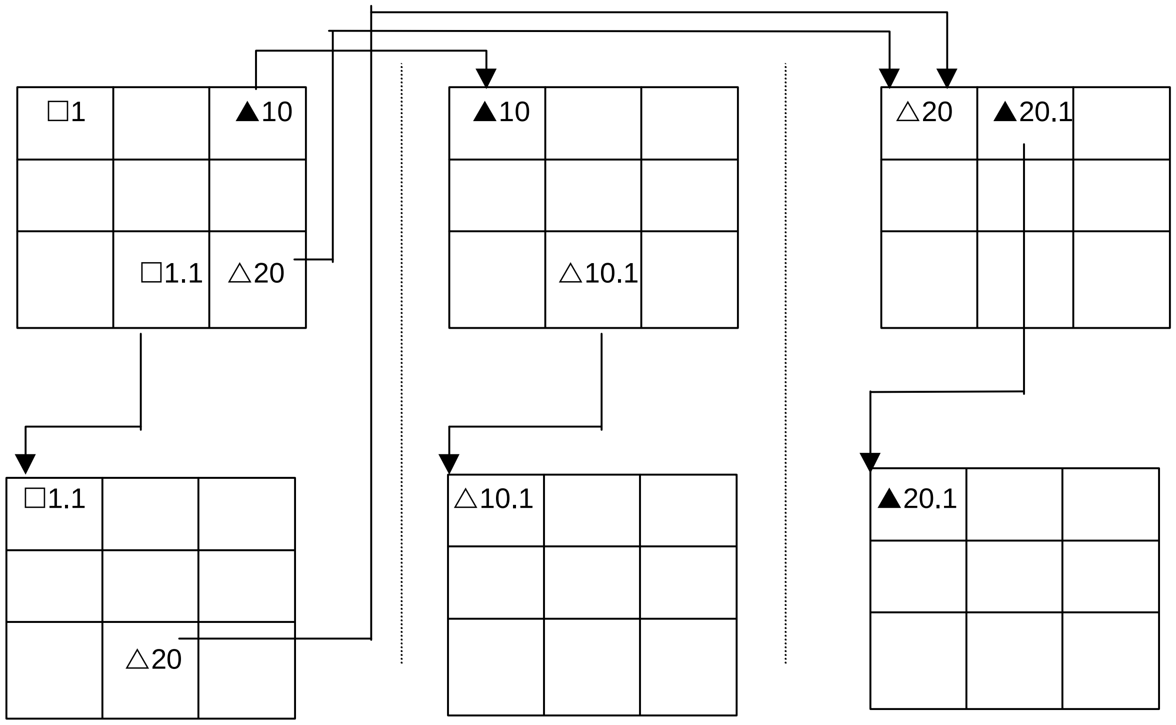


Figure 16- 1 Subroutine STM

The hierarchy STM can be called only by the parent-and-child relationship. On the other hand, subroutine STMs can be called from any location. However, like the hierarchy STM, the call is allowed from only one location when the state is a subroutine STM. In Figure 16-1, ▲10 can be called from only one location, while △20 can be called from multiple locations. The inheritance of events is the same as the table in Figure 15-24, in Chapter 15.3. Hierarchical subroutine STMs can only be called from the parent STM.

16.2. Internal subroutine STM

With the internal subroutine STM, the state scheduler can be used because the state-control variable is prepared internally. Therefore, all driven types, state types and hierarchies can be used.

16.3. External subroutine STM

With the external subroutine STM, the state scheduler cannot be used because the state-control variable is prepared externally. Accordingly, the state hierarchy is not allowed for the exclusive-state STM, which does not use the state scheduler. The event-driven type is used as the driven type.
Declaration example: int △100[2](char*cpstate, int*ipstate)
Call example: △100[1](p1, externalstate)

The state-control variable is defined both on the caller and receiver sides, and is passed as an argument.

16.4. Subroutine STM and multitask

Multiple tasks can call the same subroutine STM. The exclusive control for the functions called from the subroutine STM and accessed variables is executed via the user processing in the same way as is done by the library STM.

17. EHSTM system call

No.	System call	Meaning

1	inmail(stm_name,inmail)
	inmail(stm_level_no,inmail)
		Issues in mail.
		stm_name: STM name of in-mail destination
		stm_level_no: STM level number of in-mail destination
		inmail: Name of issued in mail

2	zkill(state_name)
	zkill(state_no)
		State transition of concurrent state to DEAD
		state_name: State name
		state_no: State number

3	zalive(state_name,mode)
	zalive(state_no,mode)
		State transition of concurrent state to READY
		state_name: State name
		state_no: State number
		mode: Mode specification when READY
		D: default/H: history/P: deep history
		(*) When the mode is omitted, the P specification is assumed.

4	zset(stm_name,state_name)
	zset(stm_level_no,state_no)
		Transition to the specified state (with activity)
		stm_name: Name of the desired STM for state transition
		stm_level_no: Level number of the desired STM for state transition
		state_name: Name of the state
		state_no: State number

5	zseth(stm_name,state_name)
	zseth(stm_level_no,state_no)
		Sets history to the specified state (without activity)
		stm_name: Name of the desired STM for setting history
		stm_level_no: Level number of the desired STM for setting history
		state_name: State name
		state_no: State number

6	bool
	zcheck(stm_name,state_name)
	zcheck(stm_level_no,state_no)
		Checks if the specified state is READY.
		stm_name: Name of the desired STM for setting history
		stm_level_no: Level number of the desired STM for setting history
		state_name: State name
		state_no: State number
		RETURN:TRUE(READY)
		:FALSE(not READY)

7	zret(p)	Returns from the library or interrupt-handler STM. p: Return value (*)p can be omitted.
8	event(task_name,event)	Issues an event. task_name: Name of the destination task of issued event event: Name of issued event
9	zdi()	Disables interrupt.
10	zei()	Enables interrupt.
11	zdil(level)	Disables level interrupt. level: Disables interrupts equal to or lower than the specified level. Level 0 is the highest and level 255 is the lowest.
12	zeil(level)	Enables level interrupt.

		level: Allows interrupts equal to or lower than the specified level. Level 0 is the highest and level 255 is the lowest.
13	ztrap(vector_no)	Generates a vector interrupt. vector_no: Generates the specified vector interrupt.

Figure 17- 1 List of EHSTM system calls

The EHSTM system calls from number 8 to 13 depend on the system implemented. Therefore, the original system calls can also be used. In this case, the CASE tool used to support the EHSTM design method needs to be compatible with the system call.

17.1. inmail

The inmail sets internal messages in the in-mail buffer. By default, one in-mail buffer is prepared for each department (in the root STM of the department, if the hierarchy is used). Also, subroutine STMs are prepared outside the departments. An in-mail buffer could be provided for each hierarchy STM, if it is not using the default. It is up to the specification of the CASE tool that supports the EHSTM design method.

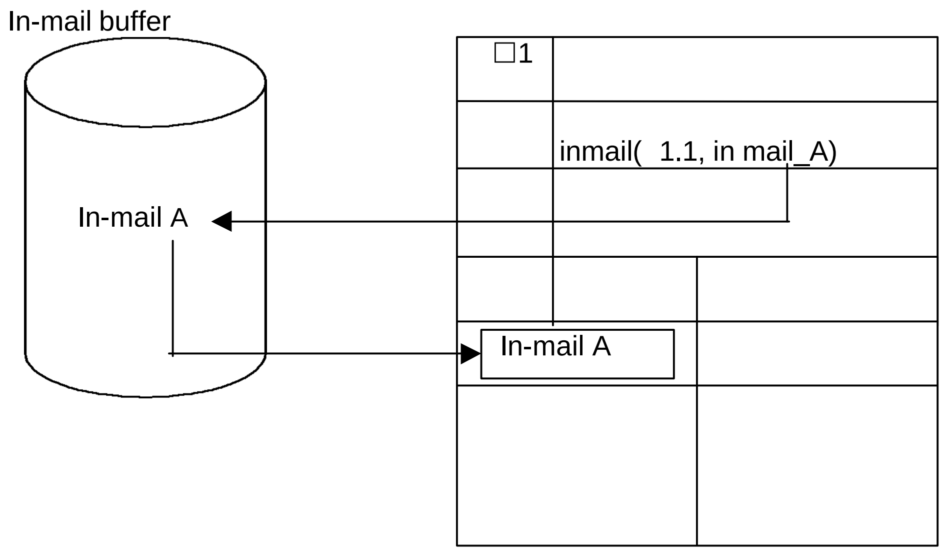


Figure 17- 2 inmail

The naming rule for in mails issued by “in mail” is the same as that for events. Refer to Section 17.6, "event."

17.2. zkill/zalive

□1			Telephone ・ T V				
			Telephone		T V		
						O N	
			Disconnected	On the phone	O F F	N O R M A L	M U T E
		0	1	2	3	4	5
Status information	0	□1.1					
Message for making a phone call	1	/	2 Dial	/	/	/	/
Voice from the other party	2	/	×	1 State the message Hang up	/	5 Mute	/
Busy tone	3	/	×	1 Hang up	/	/	/
Interested program	4	/	/	/	4 Turn TV on	/	/
End of program	5	/	/	/	×	3 Turn TV off	3 Turn TV off

			0
Telephone	Faulty	0	zkill (Telephone)
	Recover	1	zalive (Telephone)
T V	Faulty	2	zkill(T V)
	Recover	3	zalive(T V)

Figure 17- 3 zkill/zalive

The concurrent state specified by "zkill" (exclusive states cannot be specified) enters the DEAD state, which will not be scheduled by the state scheduler. The concurrent state specified by "zalive" enters the SUSPEND state, which can be scheduled by the state scheduler.

17.3. zset/zseth/zcheck

□1		OFF	ON				
				NO_TAPE	TAPE		
						HEAD	MOTOR ■1.2
		0	1	2	3	4	5
ON	0	zset(□1,ON)	/	/	/	/	.
OFF	1	/	zset(□1,OFF)	.	.	.	.
IN TAPE	2	/	/	=>TAPE(D) zseth(□1.1,PLAY:OFF)	/	/	.
OUT TAPE	3	/	/	/	=>NO_TAPE zseth(■1.2,STOP)	.	.
COMMAND	4	/	/	/	/	□1.1	.

□1.1		PLAY		RECORD	
		OFF	ON	OFF	ON
		0	1	2	3
■	0	/	=>PLAY: OFF	/	=>RECORD: OFF
▶	1	=>PLAY: ON	/	/	/
✚▶	2	/	/	=>RECORD: ON	/

■1.2		STOP	PAUSE	IN OPERATION		
				NORMAL	FAST FORWARD	REWIND
		0	1	2	3	4
■	0	/	/	=>STOP	=>STOP	=>STOP
▶	1	=>NORMAL	/	/	=>NORMAL	=>NORMAL
▶▶	2	=>FAST FORWARD	/	=>FAST FORWARD	/	=>FAST FORWARD
◀◀	3	=>REWIND	/	=>REWIND	=>REWIND	/
❚❚	4	/	=>In operation (P)	=>PAUSE	=>PAUSE	=>PAUSE
✚▶	5	=>NORMAL	/	/	/	/

Figure 17- 4 zset/zseth/zcheck

☐ "zset(☐1,ON)" is identical to "=>ON." When a transition type such as "=>ON(D)" is specified, it is specified in parentheses () after the transition destination name, such as "zset(☐1,ON(D))." See actions of "NO_TAPE" (state number: 2) and "IN TAPE" (event number: 2). The "=>TAPE(D)" means a transition to "TAPE" using the default type. In this case, the child state of "TAPE" also makes a transition using the default type. In other words, "STOP" becomes ready in ☐1.2. How about the states in ☐1.1? Since ☐1.1 is an event hierarchy, it is not included in the state tree. Therefore, "=>TAPE(D)" has no effect on ☐1.1. "zseth(☐1.1, PLAY: OFF)" changes the history of ☐1.1 to "PLAY: OFF." The difference between "zset" and "zseth" is that "zseth" modifies only the past state, or history, while the transition actually occurs by "zset" (the same as =>). This changes the READY position of the states when a history or deep-history transition occurs. The "zcheck" checks whether the specified state is currently

READY or not. TRUE is returned if it is ready, otherwise FALSE is returned.

17.4. zret

"zret" returns the control to the called source from the library STM or interrupt handler. But why is this return different from that of the task and main sections? The library STM is called from the call source by the function-call type. With the function-call type, the function described in an event is generated and the STM is called by the function. The "return" described in the library STM returns to this function. The called library STM might not return to the call source but wait for an event and process it by itself. To return to the call source, "zret" is used.

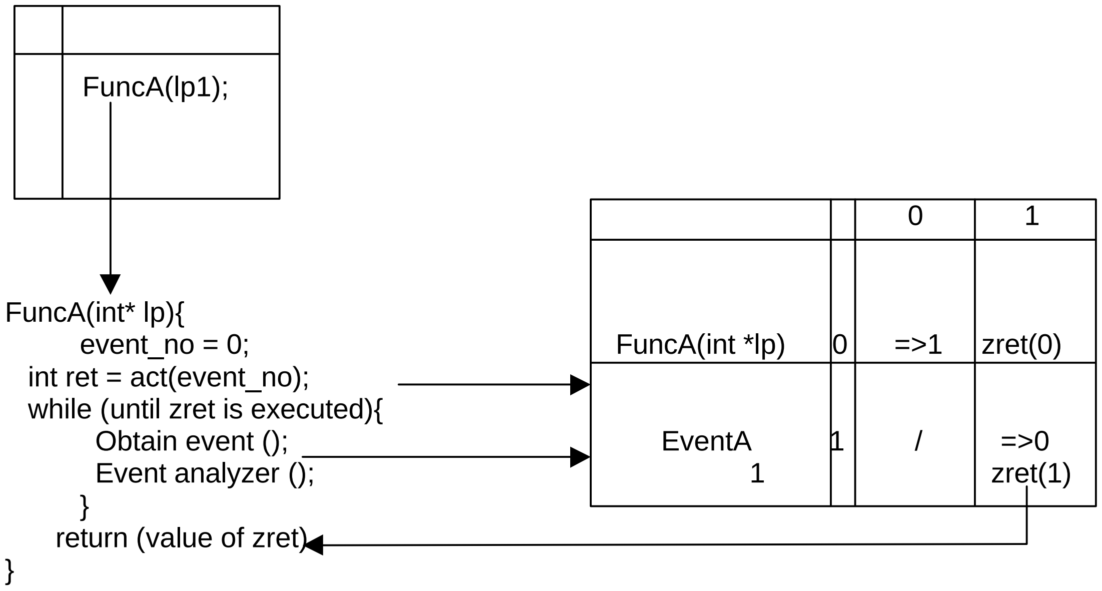
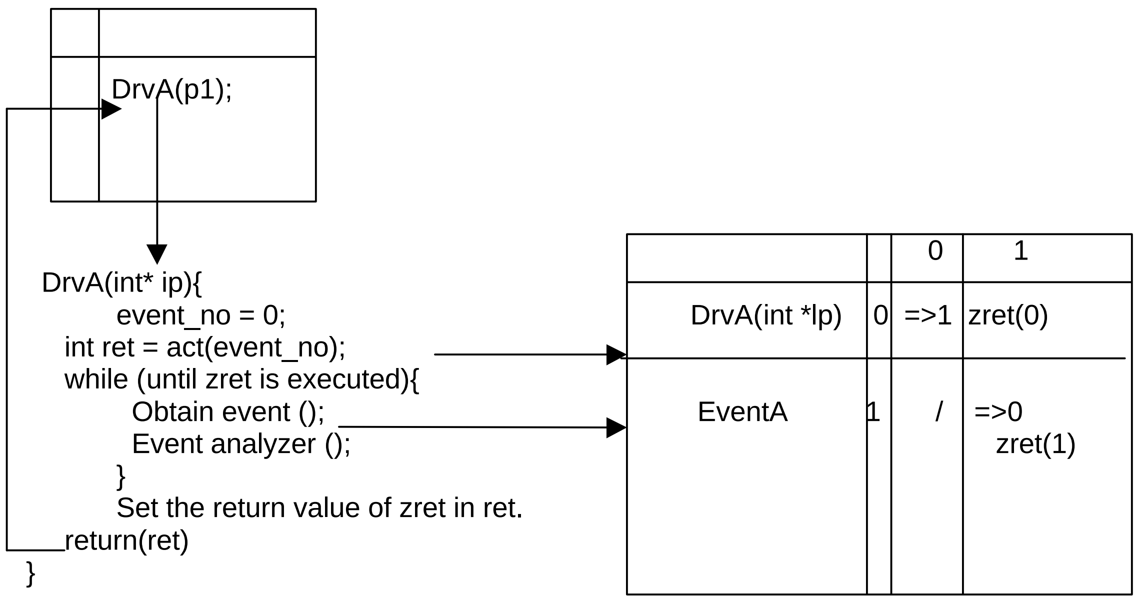


Figure 17- 5 zret

In the example above, the library STM called by FuncA does not return to the call source unless EventA occurs. When FuncA is called from multiple sources, the FuncA should be designed so it has a reentrant structure.

The following describes the general concept of device drivers and interrupt handlers. Please note that this is a general concept, and a philosophy separate from this is required for supporting individual RTOS.
Device drivers operate under the environment where an RTOS is installed. A device driver is called by the caller in the form of a device-driver call. When the called device driver returns to the caller, it returns to the RTOS scheduler. It never returns to the caller task. This is the difference from the concept of "zret" in the library STM. In the device driver, an event may have occurred to the task which has higher priority than that of the caller task. The same applies to the interrupt handler: "zret" is used if the return from the interrupt handler is simply to return the interrupted task.



```
DrvA(int* ip){
        event_no = 0;
  int ret = act(event_no);
  while (until zret is executed){
         Obtain event ();
         Event analyzer ();
        }
        Set the return value of zret in ret.
  return(ret)
}
```

		0	1
DrvA(int *lp)	0	=>1	zret(0)
EventA	1	/	=>0 zret(1)

Figure 17- 6 zretset

In the above example, the device-driver STM called by DrvA does not return to the caller unless EventA occurs. Also, the operation is the same as that of the library. The mechanism of putting the request task into the IO wait state when the driver call occurs is defined by the specification of the tool, and is not specified in this document.

17.5. event

This generates message events. Each event has a variable type (mail type, synchronized type or flag type), interrupt type, in-mail type and function-call type. The EHSTM supplies "ztrap" for generating the interrupt type. Other interrupts depend on the specification of each tool that supports the EHSTM design method. The in-mail type is generated by “inmail.” The function-call type is generated by calling a function. For the flag-variable type, writing to a variable generates an event. The synchronized event (event flag) is substituted by the mail-type event (message).

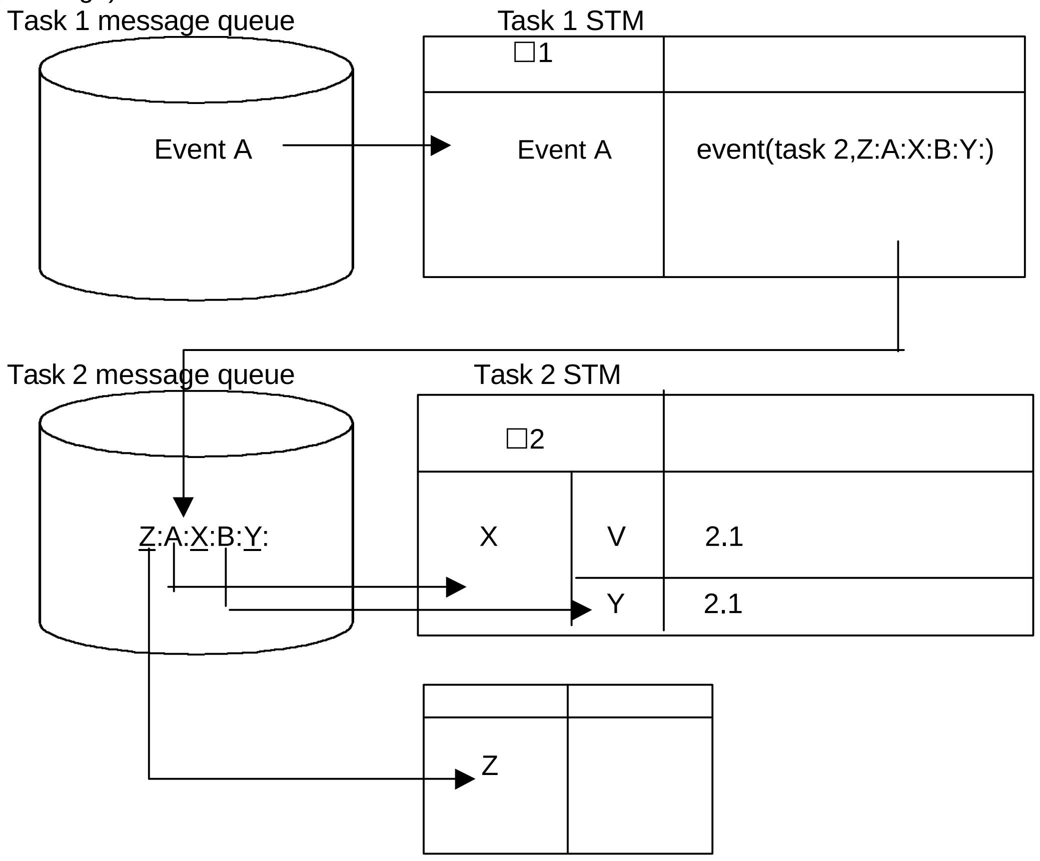


Figure 17- 7 event

An event tree (when a frame or hierarchy is used) is represented by a colon(:). The character string is inherited into the event hierarchy or event frame. The event hierarchy and event frame analyzes the string to see whether its own event exists.

17.6. zdi/zei/zdil/zeil

The disabling and enabling of interrupts is represented by "zdi()" and "zei()," respectively.
The level interrupt can be controlled in the form of "zdil(level)" or "zeil(level)."
"zdil(3)" prohibits interrupts of level 3 and below.
"zeil(3)" allows interrupts of level 3 and below.
Level number 0 is the highest interrupt level and 255 is the lowest level.
Operations of NMI (Non-Maskable Interrupt) and the level interrupt vary, depending on the CPU. Therefore, detailed specifications for zdi, zei, zdil and zeil follow the specification of the CASE tool that supports the EHSTM design method. In this case, the handling of the level number described above also changes.

17.8. ztrap

"ztrap(vector number)" generates vector interrupts.
The vector number ranges from 0 to 255.
The value of a vector number, as well as the combination of a vector number and a level number, vary according to the specification of the CASE tool that supports the EHSTM design method.