ITRON Debugging Interface Specifications

Version 1.00.00

TRON Association ITRON Committee
ITRON Debugging Interface Specification Working Group

ITRON Debugging Interface Specifications (Ver. 1.00.00)

All inquiries about this specifications and its contents should be addressed to the following:

TRON Association ITRON Committee
5th floor, Katsuta Bldg., 1-3-39, Mita, Minato-ku, Tokyo 108-0073, Japan
Phone: +81-3-3454-3191 Fax: +81-3-3454-3224

2002, 15 February

- TRON is an abbreviation for "The Real-time Operating system Nucleus".
- ITRON is an abbreviation for "Industrial TRON".
- μITRON is an abbreviation for "Micro Industrial TRON".
- BTRON is an abbreviation for "Business TRON".
- CTRON is an abbreviation for "Central and Communication TRON".
- TRON, ITRON, μITRON, BTRON, and CTRON are not the names of specific products or product groups.

I. CONTENTS

II. Table of Contents

III. Fig of Contents

IV. Function of Contents

1. Formats in This Document

1.1 Notation

In this document, entries that must be written as shown are indicated as follows (bold, italic, Gothic). Command names, structure names, and constant names are indicated in this manner.

% ***command, T_RSBRK, rif_***xxx_yyy, etc.

In this document, program codes are described as follows:

Program source

```
Program code
```

Program source

Service calls are described as follows:

Name	Overview of Functions	[Category] mark

Prototype

Argument type Argument name

Meaning of argument

(Return value) Return value type Name representing meaning of return value if not ER

Meaning of return value

Explanation

Explanation of functions

Supplementary explanation

Supplementary explanation of parameters and service call

Flag

Flag name Explanation of flag

Flag name Explanation of default flag **(default)**

Extension

The explanations of extended functions are provided with these upper and lower banners. For more details of 'TIF', see ***Section 2.4***. In the explanation of an essential function [R] that is a component of TIF, these banners indicate the portion that is handled as an extended function.

Extension

Key

Key name Meaning (See "format description of information acquisition key codes" described later.)

Error

Error constant name Error description

Arrays and array members are described as follows:

Array type name {
  Type name Name : Explanation
  Type name <u>Name</u> : Explanation (variable whose value may be rewritten at execution)
}

RIFs are classified into [OBJ], [CTX], [SVC], [BRK], [CND], and [LOG] on an individual function basis. TIFs are classified into [R] and [E] depending on the required level. A callback for each interface is also described within a category entry in [xxx:callback] form. For details of categories, see ***Section 4.2***.

The ○ mark indicates that the function must be implemented on the RIM side. The □ mark indicates that the function must be implemented on the debugging tool side.

Key codes of getting information are entered as follows. For details of key codes, see ***Section 3.6***.

First key Value [type]
  Explanation of information that key can get
.Second key Value [type]
  Explanation of information that key can get
.Third key Value [type]
  Explanation of information that key can get
.Fourth key Value [type]
  Explanation of information that key can get

The key codes are entered with the following symbols:

Table 1: Symbols and Key Code Types

Symbol	Type
W	32-bit signed integer
S	Character string
T	Structure or other special type
1	Boolean value (FALSE → 0, TRUE → other than 0) (32-bit signed integer [W] in reality)

1.2 Naming Rules

1.2.1 Variable name/Argument name

Structure internal variable names and function argument names used within functions included in the debugging interface are named according to the following rules:

Variables are named as shown below. Their names consist of lowercase letters only.

Variable name := [*prefix "_"] ((supplementary explanation *explanation [*suffix]) | (unique name))

Constants are named as shown below. Their names consist of uppercase letters only.

Constant name := *(type "_") <character string representing meaning>

The following types are used:

ACS	Flag for access method setup
FLG	Flag in common use to plural functions
OPT	Option constant for giving hint to Function
OBJ	Flag to specify object type
BRK	Break-related constant
E	Error code
ET	Error code on target
DSP	Dispatcher-related constant
EV	Event code
LOG	Log

The structure is named as shown below. Their names constant of uppercase letters only.

Name of the structure = “T_” ([interface]<the first character of the function name xxx-yyy> explanation) | <recognizable character string>

The structure used as a member of another structure (nested structure) is written as follows:

Name of the structure=<name of structure that contains structure indicated at left>

<uppercase name assigned to member>

Further, the structure is named in lowercase letters as structure tag name. Specifically, the tag name for the structure ***T_ROSEM*** is ***t-rosem***.

1.2.2 Prefixes

When the following prefixes are followed by a variable, it indicates the variable structure or usage.

Table 2: Prefixes

Character(s)	Meaning
p	The variable with this prefix is altered when storing a value.
pk	Entity of structure
str	Null-terminated character string

1.2.3 Supplementary explanation

Supplementary explanation characters prefixing a name supplement the meaning of the target variable.

Table 3: Supplementary Explanation

Character(s)	Meaning
w	Wait state
s	Send
r	Receive
f	Free
c	Call (rendezvous port)
a	Acceptance (rendezvous port)
run	Running

1.2.4 Explanation

The following characters are used to indicate the meaning of a variable. The abbreviations in ***Table 8***, Notation of xxx and yyy, may also be used.

Table 4: Explanation

Characters	Meaning
id	ID number
blk	Block
stat	Status
pri	Priority
obj	μITRON object
sem	Semaphore
tsk	Task
type	Type information flag
opt	Optional item
ptn	Bit pattern
dtq	Data queue
msg	Message
mbf	Message buffer
sz	Size
fn	Functional code
prm	Parameter
ptr	Pointer

len and ***sz*** indicate the length. They have different units. The ***len*** unit is the size of an item element. ***sz*** is indicated in bytes.

1.2.4.1 Suffix

The following suffixes of variables have usage and data in itself.

Table 5: Suffixes

Characters	Meaning
adr	Address
cnt	Stores count
lst	Stores list
ptr	Pointer storing information
ofs	Offset
len	Length

The difference between the suffix ***adr*** and suffix ***ptr*** lies in the meaning of the target variable. When a variable has the suffix ***adr***, it is attached to an item whose address is meaningful. A typical example is a break point (***brkadr***). On the other hand, the suffix ***ptr*** is attached to the name of a variable that is attached to an item when the information indicated by its address is meaningful. A typical example is the buffer pointer (***bufptr***).

The suffixes ***cnt*** and ***lst*** have a special function for the function ***rif_ref_obj***. For details, see ***Section 5.2***.

1.2.4.2 Unique name

The following unique names indicate that the variable has a unique meaning.

Table 6: Unique Names

Characters	Meaning
result	Stores result
storage	Data storage area, etc. (mainly for write)
param	Parameter
flags	Flag variable/argument
name	Name
length	Length (when structure contains only one variable)

The following interface identification characters are used to identify the interface with the structure.

Table 7: Interface Identification Characters

Character	Meaning
R	RTOS access interface
T	Target access interface

However, the interface identification characters are omitted only in the following situations:

- **Common structure for both interfaces**
- **Independent structure from both interfaces**

1.2.5 Function names

All the functions included in the debugging interface take of the form www_xxx_yyy (software conponents naming standard). A www is specified according to each interface ("***rif***" for a function on the RTOS access interface or "***tif***" for a function on the target access interface). ***dbg*** is used for functions that do not come under the RIF or TIF category.
For the xxx and yyy portions, see the table below:

Table 8: Notation of xxx and yyy

Abbreviation	Complete form	Meaning
alc	allocate	Allocation
brk	break	Break
cal	call	Call
can	cancel	Cancel
cfg	configure	Configuration
fin	finalize	Finalization
fre	free	Freeing
get	get	Getting
hok	hook	Hook function registration
ini	initialize	Initialization
pol	poll	Polling
ref	refer (forward)	Reference
req	request	Request
rep	report	Report (callback included)
rrf	refer (backward)	Backward reference
rst	reset	Reset

Table 8: Notation of xxx and yyy

Abbreviation	Complete form	Meaning
set	set	Setup
sta	start	Start
stp	stop	Stop
bls	block set	A set of memory blocks
brk	break point	Break point
cfg	configuration	Configuration information
cnd	condition	Condition
ctx	context	Context
dgb	debug tool	Debugging tool
fnc	function	Function
log	trace log	Trace log
mbh	memory block on host	Memory block on the host side
mbt	memory block on target	Memory block on the target side
mem	memory on target	Memory on the target
rdt	register set description table	Register set description table
reg	register	Register
rim	RTOS interface module	RTOS interface module
stp	stop by break point	Stop by break point
svc	service call	Service call
sym	symbol	Symbol
tgt	target	Target

www***_rep_***yyy has a special meaning. It is handled as a callback function for the interface www.

When functions added uniquely by an implementer or undefined in this specification are used with this specification, the prefix ***"v"*** should be attached to xxx to indicate its uniqueness (as with μITRON 4.0) (e.g., ***tif_vcal_svc***).

1.3 Terms and Definitions

The following terms are used in this specifications.

Table 9: List of Terms

Term	Meaning
Target	Program to be debugged or hardware to store such target program
Debugging tool	Hardware/software used for debugging (e.g., host computer, probe, and debugging applications)
Guideline	Non-mandatory standards that should be complied with
Agent	Support program introduced for specific purpose
Implement dependant	An unique specification that is determined by an implementer at adoption
Implement definition	An implement dependant which should be declared to TRON association

1.4 Abbreviated Names

In this document, the following abbreviations are used to represent long names or frequently used names:

Table 10: Abbreviations

Abbreviation	Meaning
RIF	RTOS access interface
TIF	Target access interface
RIM	RTOS interface module
Register table	Register set description table

2. Overview

2.1 Background

Computers are now being used for various purposes. In embedded applications, which account for the majority of applications, the number of associated products is increasing and the software scale is growing gradually to implement more advanced functions. Meanwhile, the time to market (interval between product development and coming on the market) is falling and large-scale applications need to be created quickly.
To complete development of large-scale software quickly, it is necessary to improve the development environment. Improvement of the debugging environment is particularly important. It is not easy to accurately determine the time required for software testing/debugging, which accounts for the greater part of the overall development process. The time spent on debugging depends largely on the performance of tools and debugging personnel's experience.
When an application uses an OS, the OS support provided by debugging tools is an important factor. If the displayed OS internal code for stepping-in or task status are irrelevant to the currently targeted codes which debugging personnel uses, productivity may be decreased.
In the field of embedded applications, the Real-Time Operating System (RTOS), which focuses on real-time capabilities, is widely used in addition to the common OS function. The results of a 1999 survey of RTOS market share are shown in ***Table 11***. In Japan, the share of ITRON Specification compliant OSes accounts for more than 30% of the total.

Table 11: OSes Used for Recently Developed Embedded Device

Category	Share
Commercially-available ITRON Specification compliant OSes	18.8%
In-house ITRON Specification compliant OSes	12.0%
CTRON Specification complaint OSes	1.0%
Other commercially-available unique specification compliant OSes	40.4%
Not used because of OS problems	3.5%
Not used because no OS needed	24.3%

It is not so difficult to create debugging tools that support only one ITRON specification-compliant OS. Such debugging tools already exist. However, it is not easy to provide support for all ITRON specification-compliant OSes. The reason is that the internal structure varies with the respective OS installation method as the ITRON Specification states the API specifications. Regarding debugging tools dependent on the internal structure, RTOS-related modules might have to be rewritten whenever a new ITRON specification-compliant OS is released.
The development environment of ITRON specification-compliant OS has another problem. That is, ITRON specification-compliant OS is provided by the manufacturer of the chip to be embedded while OS debugging tools are provided by the tool vender dedicated to creating tools.
It causes difficulty in keeping adjustability of tool and OS.

There would be no problem if everything from the OS to debugging tools is supplied by one company. However, it would be difficult for two divisions of different companies to cooperate with development. Therefore, it is difficult to maintain consistency between tools and OSes. Under these circumstances, the user may be afraid of possible debugging environment changes and reluctant to use the latest ITRON specification-compliant OS even when programs running on ITRON specification-compliant OSes with a high degree of portability. It has been difficult to continue supplying a standard debugging method for ITRON-compliant OSs due to the above-mentioned problem.
It was therefore pointed out that the development environment is inadequate for ITRON specification-compliant OSes. Although ITRON specification-compliant OSes have nearly 30% domestic share, the survey in 1999 revealed that more than 20% of engineers pointed out this problem (***Table 12***).

Table 12: Shortcomings of ITRON Specification-compliant OSes

Description	Percentage
Inadequate development environment and tools	22.9%
High dependency and poor portability	12.9%
Insufficient software components	11.5%
Insufficient number of engineers	7.8%
Insufficient functionality	4.4%
Excessive resource requirements of OS	4.4%
Other	18.9%
No significant deficiencies	17.2%

To solve the above-mentioned problem, it is necessary to standardize the interface between the RTOS and debugging tools. When the interface is standardized, it is possible to use any combination of debugging tools and ITRON specification-compliant OSs. As a result, it is possible to offer an RTOS level debugging environment with an increased degree of freedom.
The ITRON Debugging Interface Specification in this document was developed by the ITRON Debugging Interface Specification Working Group, which started in February 1999.

2.2 Standardization Objective

The main objective of the ITRON Debugging Interface Specification Working Group is to establish an interface for adding RTOS support functions to debuggers.
The significant items were defined by the Working Group to achieve the above objective as follows:

- **Furnish high degree of scalability**
  To handle processors ranging from 8-bit low-speed processors to 32-bit high-speed processors
- **Develop specifications for variety of debugging environments**
  To offer an interface that is commonly applicable to software Debuggers, ICE, JTAG Emulators, software Emulators, etc.
- **Create interface without limiting functionality to ITRON specification-compliant OSes**
  To offer an interface that is available for of debugging the other RTOS and software modules as well

2.3 Approaches to Standardization

To develop the interface specification, we conducted interface specification studies from various viewpoints. This section states the approach plans for interface specification studies, including their merits and demerits, as well as the adopted plan and reasons for adoption.

2.3.1 Approach plans

Approach 1: Fixing object information

This method uses a stronger binary level standard instead of the current name-only standardization level to bind a control block that retains the status of objects defined by the μITRON specification. It provides compatibility between OSes by uniquely determining the block storage site, alignment, etc.

- **Merits**
  Realizes ITRON debugging interface implementation without any modifications to debugging tools.
- **Demerits**
  Current commercial OSes mostly unsupported
  Dependent on CPU architecture
  Originality of each company lost

Approach 2: Implementing support function on target side

This method standardizes the differing information among RTOSes when it is acquired from the target. It can be classified into the following two types depending on the function implementation location.

Implementing support function as task

The support mechanism is introduced as a task. If, for example, there is a memory management unit (MMU) within the target, the OS internal information cannot be read because the support function is implemented by the task. However, other tasks are unlikely to be affected.

Implementing inside RTOS

The support mechanism is directly introduced into the RTOS. Detailed information can be obtained. The effect of the MMU is averted. However, other tasks are likely to be affected.

Expanding debug monitor

This method expands a debug monitor that is used for target debugging. The RTOS operations are likely to be affected.

- **Merits**
  Wide range of OSes, including existing ones, covered
- **Demerits**
  Burden on target (both CPU and memory resource)
  If the MMU or protective mechanism is located in the target, it is necessary to furnish the kernel with support functions, etc. As a result, the structure will be complicated.

Approach 3: Introducing support module within debugging tool

This method incorporates a module with the function for RTOS to get information into a debugging tool, and standardizes a series of associated functions.

- **Merits**
  Various OSes, including existing ones, covered
  Load on target minimized
- **Demerits**
  Flexibility of the module is required to be incorporated in debugging tool.

2.3.2 Approach selection and its reasons

The ITRON Debugging Interface Specification Working Group examined the above three approaches and adopted Approach 3 (Introducing support module within debugging tool). The prime reason is that it was easy to switch from the former debugger design to the design based on the ITRON Debugging Interface Specification.

In most previously created debugging environments, many RTOS level debugging support mechanisms are incorporated in the target (Approach 2) to permit RTOS level debugging.
Under these circumstances, support modules should be newly incorporated in debugging tools when Approach 3 is used. However, regarding debugging tools, it is just that the target functions are transferred to the host. For RTOS manufacturers, it is just that the write destination merely changes from the RTOS kernel interior or debugging support task to a support module. Therefore, debugging tool vendors and RTOS vendors can both switch to a new environment without wasting previous assets.

This approach does not conflict with the previously employed approach. Therefore, RTOS manufacturers can implement the above-mentioned mechanism as needed. If, for example, all functions cannot be provided by support modules alone or a high degree of scalability can be attained by introduction, the support mechanism will be provided within the target.
Even in this situation, Approach 3 is instrumental in reducing the amount of information transfer between the host and target. If Approach 2 is used, a problem arises because the information between the target and debugging tools needs to be standardized. However, when Approach 3 is used, it is just necessary that the information be standardized before and after support modules existing in the host. Therefore, the debugging support mechanism within the target merely exchanges the required minimum information with the debugging tools. Eventually, when the support module expands internal information and reshapes it to a standard type, the previous functions can be realized with minimum information transfer to the target computer and minimum load.

For these reasons, the ITRON Debugging Interface Specification Working Group selected Approach 3.

2.4 Concept

The ITRON Debugging Interface Specification is developed to improve the debugging environment for applications that use a μITRON specification-compliant OS.
The figure below shows the debugging interface concept diagram:

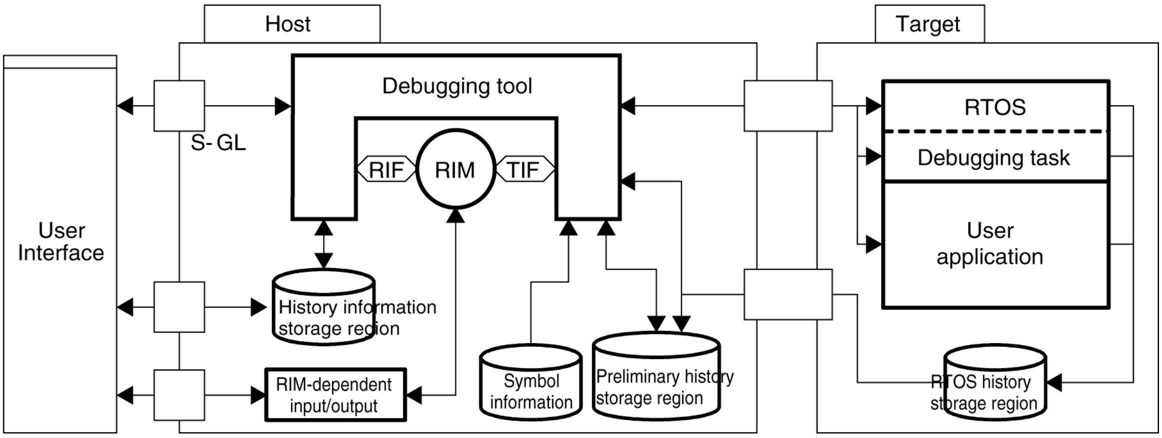

Figure 1: ITRON Debugging Interface Specification Concept Diagram

To enable the host to get the RTOS-dependent information on the target, the ITRON Debugging Interface Specification has the concept of one module and the definitions of two associated interfaces and one guideline.

- **Support function guideline**
  This guideline determines the functions related to the RTOS support functions that are to be implemented in the debugging tools and their details. These guidelines make it enables standardization of the terms and similar functions among the debugging tools that support the ITRON Debugging Interface Specification, and assure the minimum functions for the user. These guidelines are also used to define the two interfaces (RIF and TIF) described later.
- **RTOS interface module (RIM)**
  This module notifies a debugging tool of the RTOS internal information and translates RTOS-dependent instructions that are not understandable to the debugging tool into understandable instructions. It is provided and incorporated into a debugging tool by an RTOS manufacturer. (Typical providing means are C language source program and Windows DLL.) This module is the core of this specification.
- **RTOS access interface (RIF)**
  When a debugging tool performs an RTOS-dependent debugging operation with the RIM function, it uses the RIF as the interface. It provides a debugging tool with a means of knowing the RTOS current status. It consists of a total of 21 functions (callback functions included) that are defined in C language API format. It offers funcitons, including getting RTOS object details and context.
- **Target access interface (TIF)**
  To answer a request issued by a debugging tool via the RIF, the RIM needs to access the target and RTOS with the debugging tool function. The target access interface, which consists of 31 functions and callback functions that define the basic debugging tool functions, provides the RIM with debugging tool functions to cope with such a situation. This interface offers memory read/write, run/break, and other functions.

The other modules are described below:

- **Previous history storage region**
  This region is used to temporarily store log information while getting a trace log.
- **Standard information storage region**
  This region is used to store a trace log, etc., in the standard format for the ITRON Debugging Interface Specification. The stored information can be viewed with a standard format compliant viewer instead of support of debuging tool.
- **RIM-dependent input/output**
  When the RTOS has advanced debugging options or handles unique implement-dependent information, the standard information input with a debugging tool may be insufficient. In such a situation, the RIM-dependent input/output is used. This RIM-independent input/output standardizes a part of a debugging tool user interface and permits the RIM to be interactive with the user. (This function is not supported by the current specification.)

In the ITRON Debugging Interface, a debugging tool and the RTOS interface module (RIM) incorporated in the debugging tool transfer data with each other to realize debugging tools RTOS-compliance even when they do not support RTOS. The next section provides an example to explain the operation principles.

2.4.1 Operation

This section explains getting ID1 task status as an example.

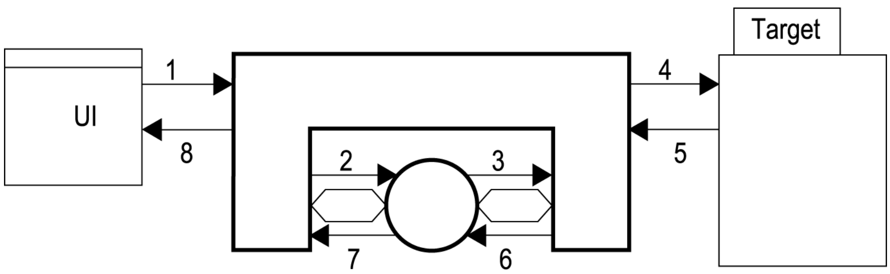

Figure 2: Getting ID1 Task Status

1. The user issues a request for getting the ID1 task status.
2. To refer the RTOS internal status, the debugging tool sends a request for an ID1 task status to the RIM.
3. The RIM refers to the symbol table, etc. for the address of task control block (TCB) that corresponds to the task ID1, and requests the debugging tool to read the associated memory on the target.
4. The debugging tool uses an existing function to read the specified memory on the target.
5. The read memory data is notified to the debugging tool.
6. The debugging tool sends the data read in step 3 to the RIM.
7. The RIM decodes the received data in accordance with the TCB data and then forwards it to the debugging tool in the standardized form.
8. The display screen shows the ID1 task status in accordance with the result.

When the RTOS and RIM are offered as a set, the debugging tool can access the RTOS-dependent information without knowing the details of the RTOS internal structure.

2.5 Characteristics

This section details the distinctive characteristics of the ITRON Debugging Interface Specification.

2.5.1 Two break methods with task ID

For the RTOS support function of a debugging tool, the break function is important as it breaks task at a specific operation performed by a task with a specific task ID. If this function is not provided for situations where two or more tasks share the same module, number of nonessential operations increases with an increase in the number of tasks. For example, execution resumption must be repeated manually until a break occurs at the task to be focused.
The ITRON Debugging Interface Specification implements two break functionalities for specific tasks in order to achieving one of the goals of covering plural debugging tool. Another objective of this break function is to utilize a highly functional debugging tool which fails its full expected performance due to use of function that is standardized in consideration of low-grade debugging tools.

Method 1: Break by callback routine based on RIM

Debugging tools that are incapable of getting a task ID and other RTOS-dependent information will be turned into an RTOS support debugging tool with RIM, as described earlier. This method performs a task ID dependent break with a break generation callback of RIM, which is the core of the interface.

It is assumed that the following types of debugging tools will use this function:

- **Having execution break function with respect to specific address**
- **Having no conditional break mechanism**
- **Having no RTOS-dependent mechanism**

This function is detailed below with reference to an example. In the example, it is assumed that a break occurs when a task with the task ID number 1 executes address 0x12345678.

1. The debugging tool requests the RIM to set an address 0x12345678 execution break for task ID1.
2. The RIM sets the address 0x12345678 execution break for the debugging tool.
3. The user executes a program.
4. The program executes address 0x12345678. The debugging tool performs a break.
5. The debugging tool uses the callback function to notify the RIM of the occurrence of a break.
6. The RIM checks the region for storage of the currently executed task ID to determine whether or not to perform a halt, and then notifies the debugging tool of the result.
7. When the RIM notifies the debugging tool that the conditions are satisfied, the debugging tool completes a break operation and notifies it to the user. If the debugging tool is notified that the conditions are not satisfied, it aborts the break operation and resumes program execution.

This break method has two characteristics. One is a task ID based break can be operated even when the debugging tool is not highly functional. The other characteristic is that nearly all RTOSes can be covered, because the RIM determines whether or not to operate a break.

However, this break method has a disadvantage in that the number of callbacks and the load on the host increases if a large number of breaks are set. When remote debugging is conducted via a serial port in particular, the task ID is checked at each break, considerably increasing the overhead.
Even if the RIM does not decide on performing a halt in the break notification sequence, the target remains stopped during the decision-making period. For a program with severe time limitations, such an unnecessary break could cause a malfunction or an inability to detect errors to be debugged.

Method 2: Conditional break by debugging tool after getting break condition

This method is used in case that a conditional break mechanism is already implemented in a debugging tool. In response to a request, the RIM notifies equivalent conditions to a debugging tool.

It is assumed that the following types of debugging tools will use this function:

- **Having execution break function with respect to specific address**
- **Having conditional break mechanism**
- **Having no RTOS-dependent mechanism**

When this method is used, the RIM does not set a break point itself. In the method 1, the RIM would merely generate a conditional expression for a conditional break equivalent to a conditional judgment formed by the RIM, and return the generated conditional expression to the debugging tool. The debugging tool adds a conditional expression as needed to the obtained condition, and then sets a conditional break directly.
In marked contrast from the description in the method 1, this method does not perform a callback even at a break hit. Therefore, when the debugging tool has an extremely advanced conditional break mechanism, a break mechanism dependent on RTOS information can be established without generating unnecessary overhead.

The following conditions are now applicable to this function.

- **Memory address**
- **Data length (in bytes)**
- **Value**
- **Condition (equal to, greater than, less than, or not equal to)**

Figure 3 shows two types of breaks and their difference in program flow. Parts indicated by solid lines represent the paths between different programs (between the RIM and debugging tool or between the target and debugging tool). Clearly an increase in the number of solid lines increases the program overhead. Parts indicated by dotted lines represent regions within the same program. Numbered arrow marks respectively represent a flow for setting a break point, a flow for a program for determining whether or not to stop the operation, and a flow for notifying a break hit to the user after the decision of a break hit.

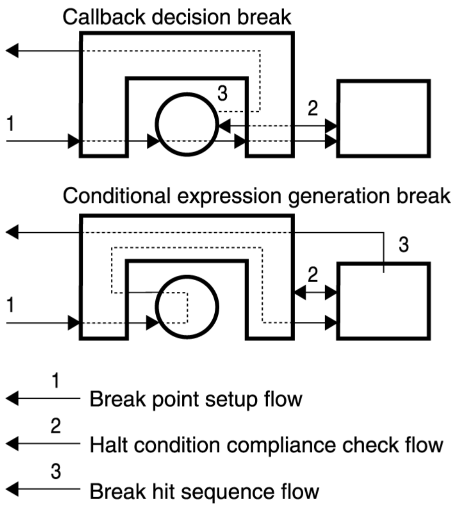

Figure 3: Two Break Methods and Difference in Operations Flow

2.5.2 Scalable debugging environment

The field of embedded applications is characterized by the fact that many bugs that should be detected are not encountered depending on the situation. For example, a bug may occur only in situations where a time-critical task is executed at a specific timing. A typical problem would be I/O read wait negligence after an I/O write. When a break point is set immediately before an I/O read that is performed stepwise, no error occurs in debugging operation because an adequate wait is taken by a break before the start of the I/O read. However, if execution is performed in the same manner as in an actual environment, an error occurs.

Regarding bugs that are timing-dependent, the full function of a debugging tool may cause unfavorable results as stated below. When simplified RIM implementation is completed for the aforementioned break support function of the ITRON Debugging Interface, timing-dependent bugs may not always occur depending on the time of the round trip between the RIM and target for making a decision.

For the architecture of the ITRON Debugging Interface, on the other hand, the RIM and RTOS are both supplied by an RTOS manufacturer. It is therefore possible to supply two or more sets of the RIM and RTOS depending on the situation to enable the user to select the best combination for the user environment.

Depending on whether RIM or RTOS has a larger number of debugging support functions, the following characteristics can be provided even if the same function is offered.

- **Implementation with greater importance to RIM side**
  When a larger number of functions are implemented in the RIM, applications can be debugged in an environment that is very close to the one for the release time. As a result, the load on the target can be reduced.
- **Implementation with greater importance to RTOS side**
  When a larger number of functions are implemented in the RTOS, the time required for communication between a debugging tool and target can be minimized. This results in increased response speed.

When the RIM and RTOS are supplied with source code, the RIM itself can be reconfigured with respect to the RTOS that is freely reconfigurable as needed for applications. To avoid allocating memory space for unused function in the RIM or unnecessary debugging support within the RTOS, the RTOS can be reconfigured to support only the required abilities. Furthermore, a best suited RIM for debugging the RTOS and that without unnecessary functions, can be generated to minimize useless overhead while debugging.

As an example, examine a situation where a high-speed break must be supported. To excuse a high-speed break, it is necessary to minimize the amount of communication between the target and debugging tool, which is a bottleneck in the current debugging environment. When all functions provided by the ITRON Debugging Interface Specification are implemented with greater importance placed on the RIM side, the modules within a debugging tool determine the conditions that are dependent on the RTOS. Therefore, the target-to-debugging tool communication forms a bottleneck. When a high-speed break must be provided, the objective will not be achieved by normal means because of the above-mentioned problem.
To solve this problem, the break hit decision routine implemented in the RIM should be incorporated in the RTOS. When this method is used, the task ID decision routine is embedded in the RTOS so that a break point is set inside the RTOS (this is not a place where a break is normally positioned). An instruction for calling the routine is positioned at a place where a break point should normally exist. This ensures that the debugging speed increases because the amount of communication between the host and target dramatically decreases. However, the routine placed within the RTOS causes a larger amount of overheads to the target than that in normal situations. The debugging personnel should exercise judgment to select one of the various methods.

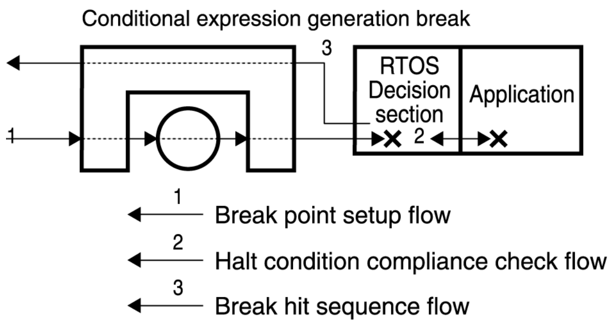

Figure 4: Flow of Operations when a Special Break Routine is Implemented

3. Common Regulations

This chapter explains the common concepts in the ITRON Debugging Interface Specification.

3.1 Interface Function Registration/Unregistration

The ITRON Debugging Interface Specification ensures that all functions are available for a debugging tool or RIM when the pointers to them are registered in the structure ***T_INTERFACE*** that stores interface function pointers.
All functions offered by the ITRON Debugging Interface must be called by acquiring the pointers to the functions from the above-mentioned interface structure, except for ***dbg_ini_inf***. Except when functions are bound statically, the function pointer values registered in the interface structure may change. Therefore, use of a local copy or similar processing operation must not be performed, because the changes will not be reflected.

The structure ***T_INTERFACE*** stores the pointers to all the interface functions. Structure members are arrayed in the order of entries in the specifications. Consequently, the pointers to functions that are outside the scope of the specifications are arrayed in random order. Structure members for all the pointers to nonexisting or unsupported functions must store ***NULL*** (= 0). Therefore, before calling interface initialization related functions (***dbg_ini_inf***, ***dbg_ini_rim***), the debugging tool must put ***NULL*** into all pointers to the unsupported functions included in the interface and functions possessed by the RIM.
The following example shows a debugging interface initialization routine for the debugging tool side:

Program source

```
  /* Interface structure initialization */
ZeroMemory(&interface, sizeof(T_INTERFACE));
  /* TIF function registration */
interface.tif_xxx_yyy = xxx_yyy;
  /* Interface initialization */
dbg_ini_inf(...);
  /* RIM initialization */
if( interface->dbg_ini_rim != (void *)0l)
    (*interface->dbg_ini_rim)(...);
```

Program source

3.2 Consistency

The term consistency means that data is retained throughout a single operation. The term consistency assurance means assuring that data agrees with the information on the target throughout a single operation.
The operation is judged to be inconsistent if the system is unstable (e.g., operation in a critical section of an OS) when, for example, an RTOS-dependent information read process is executed.

When the debugging tool performs a process after stopping the user target, you may conclude that all functions assure consistency. However, if the user target is operated in a critical section of an OS while it is halted, consistency is not assured.

Conditions for function and consistency assurance are listed below:

- **Single memory block read (tif_get_mem)**
  When a procedure is performed to read a specified memory block, the targeted memory block must not be written to.
- **Plural memory block read (tif_get_bls)**
  When a procedure is performed to read a specified memory block, no targeted memory blocks must be written to. (If a single memory block is called more than once, the consistency among the blocks is not assured.)
- **Task status retrieval (rif_ref_obj)**
  The current execution position must not be a critical section of a current OS. In addition, the pointer to the TCB and the TCB itself must be read with consistency assurance. Further, there must be no contradiction in the data constructed with the read information.

3.3 Prohibition on Target Halt

The prohibition on target halt means that the target operation related procedures defined in the ITRON Debugging Interface Specification must not be used. When the target is requested to continue running in all operations, these functions must not be called.

Target operation-related functions that are defined in the ITRON Debugging Interface Specification are listed as follows:

- **Target execution**
  tif_sta_tgt
- **Target stop**
  tif_stp_tgt
- **Target execution break**
  tif_brk_tgt
- **Target execution resumption**
  tif_cnt_tgt

For some functions, ‘consistency assurance’ and ‘permission for target halt’ may be used simultaneously. In such an instance, the RIM must return the ***E_NOSPT*** error when consistency cannot be assured permanently without halting the target, also the RIM must return ***E_FAIL*** when consistency cannot be assured temporarily without halting the target.

3.4 Types

This section describes unique types defined in the ITRON Debugging Interface Specification.

Table 13: Unique Types

Type name	Meaning
BITMASK	Bit mask (detailed later)
ER	Move than 16-bit integer for storing error code
FLAG	32-bit unsigned integer
ER_ID	ID or ER, whichever integer greater A positive value indicates ID and negative value indicates ER.
DT_xxx	Identical type that large enough to store variable defined as xxx in ITRON Kernel Specification
ID	Unsigned integer that large enough to store object number on debugging interface
INT	Signed integer that exists on host with natural length
UINT	Unsigned integer that exists on host with natural length
VP	Void pointer on host
VP_INT	Type that large enough to store VP and INT
LOGTIM	Integer that indicates log time (unit defined at implementation)

Further, a type beginning with the prefix ***DT_*** is defined to store variables of a type defined in the target ITRON kernel specification within the RIM and debugging tool. This type is 'a variable that is large enough to store target data'. It may not always coincide with the target type size. (When the target INT is 16 bits, ***DT_INT*** can be 32 bits.)

Even if a defined type name is the same as the type of the ITRON kernel specification, the debugging interface basically concludes that it does not comply with the ITRON kernel specification. More specifically, ER and ID are defined in the ITRON Kernel Specification. However, they are uniquely defined in the ITRON Debugging Interface Specification as well.

3.5 Bit Mask

The ITRON Debugging Interface uses a bit mask in order to set enabled or disabled. A bit mask is a set of 1-bit flags.

The first item of the bit mask corresponds to the LSB. Therefore, when expressed in C, the status of the n-th flag must be stored so that it can be got as indicated below:

Program source
```
(( bitmask >> n) & 1)
```
Program source

Bit masks are classified into the following types according to length:

BITMASK	With natural length that exceeds maximum count used within specification
BITMASK_8	1 byte (8 bits)
BITMASK_16	2 bytes (16 bits)
BITMASK_32	4 bytes (32 bits)
BITMASK_64	8 bytes (64 bits)

When a bit mask with a length not exceeding 64 bits is to be created, the minimum specified type meeting the requirements must be used.

When the length of a bit mask exceeds 64 bits or cannot be fixed, the bit mask must be defined as a 1-byte bit mask array (***BITMASK_8***[]). Therefore, when expressed in C, the n-th item must be stored so that it can be read with the following syntax:

Program source
```
(( bitmask [n>>3] >> (n & 7)) & 1)
```
Program source

3.6 Structure and Keys of Getting Information

To get information, the ITRON Debugging Interface Specification uses a special structure and a key for specifying the information to be got.

The following functions are used to get the information that applies this rule.

- **rif_ref_cfg : Get of kernel configuration**
- **dbg_ref_dbg : Get of tool information related debugging**
- **dbg_ref_rim : Get of information related RIM**

To specify the information to be got, the ITRON Debugging Interface Specification uses key code consisting of four 8-bit integers. In this document, the key codes are described as follows. Within a program, etc., the prefix ***INF_*** may be attached to a key to indicate that the key is a key for getting information.

Key code := ***first key*** ["." ***second key*** ["." ***third key*** ["."***fourth key***]]]

(Example: ***BREAK.CONDITION.MAX***, ***INF_HOST.INF_NAME***)

The second and subsequent keys of a key code can be omitted. Omitted keys are handled as ***DEFAULT*** = (0).

The structure of getting information ***T_INFO*** is detailed below:

```
typedef struct          t_info_result_buf
{
    UINT sz                     : Buffer size
    VP ptr                      : Pointer to region where character string or special type be
                                  stored
}   T_INFO_RESULT_BUF;

typedef union           t_info_result
{
        INT value               : 32-bit signed integer
        T_INFO_RESULT_BUF buf
                                : Value of special type

typedef struct          t_info
{
    char key[4]                 : Key for specifying information
    T_INFO_RESULT result:Corresponding value for key
}   T_INFO;
```

Two types of information can be got: 32-bit integer, and character string or special type. The information type can be presumed from the most significant bit of the last key. In the above example where "***BREAK.CONDITION.MAX***" is used, the information type can be derived from the third key (***MAX***). The table below shows the relationship between the type and the most significant bit of the last key.

Table 14: The most Significant Bit of the Last Key and Got Information Type

Most significant bit	Got type
0	32-bit integer
1	Character string or special type

T_INFO::result.buf.sz is a variable that retains the length of the buffer for getting the character string. However, when a character string or special type is read, its length is stored in ***T_INFO::result.buf.sz***. When an integer value is read, the value is undefined.
A storage region must be furnished separately by the caller for getting of a character string or special type. The caller gets an adequately large storage region. It stores the pointer to the acquired region in ***T_INFO::result.buf.ptr*** and the acquired size in ***T_INFO::result.buf.sz***. The callee stores the information about character strings and special types in a specified region in such a manner that the transfer length does not exceed the got size. Since a terminal symbol is always attached to a character string, in case that ***T_INFO::result.buf.sz*** is set to 1, no read data is obtained even if the function ends normally. If ***T_INFO::result.buf.sz*** is set to 0, the ***E_PAR*** error occurs.
If the buffer size is smaller than the transfer data length in situations where a special type is to be read, the behavior of the function is stipulated by an 'implement definition'. However, if ***T_INFO::result.buf.sz*** is 0, the ***E_PAR*** error occurs.

If an invalid key code* is contained in one of the ***T_INFO***s, which is specified as an argument, in situations where two or more items of information are read simultaneously, the function turns out to an error. In such an instance, the function does not give a report or assurance about whether information other than the invalid key code is read correctly.

Key code insertion occurs so that the first key is the first item for the array (***T_INFO::key***). To clarify operations, the example below shows the implementation of the key code generation function (in C++).

Program source

```
static char StringBuffer[MAX_STRBUF_LENGTH];

static inilne void MAKE_KEYCODE
    ( T_INFO * info, char key1, char key2 = 0, char key3 = 0, char key4 = 0)
{
    info->key[0] = key1;
    info->key[1] = key2;
    info->key[2] = key3;
    info->key[3] = key4;
    info->result.buf.sz = MAX_STRBUF_LENGTH;
    info->result.buf.ptr = StringBuffer;
}
```

Program source

The key code "***0.0.0.0***" has a special meaning. When "***0.0.0.0***" is passed as a key code, the function of getting information returns a succeeding key code of previously got key code. Further, if the first element of a key code array passed as an argument is "***0.0.0.0***", that element denotes the first key code, and the information acquisition function gets a key code with the smallest value.
In other words, when there are five ***T_INFO*** arrays with a key code "***0.0.0.0***", the information corresponding to each arraied information from first to fifth is got. However, operations performed at execution of a subsequent function remain unchanged even after continuous acquisition by "***0.0.0.0***". Therefore, note that the same information is got even if a function is executed two or more times using the ***T_INFO*** arrays, all of which consist of "***0.0.0.0***".

*. The 'invalid key code' refers to a key code that is neither defined by the ITRON Debugging Interface Specification nor contained in a unique specification. All key codes defined by the ITRON Debugging Interface Specification must have a certain value.

Program source

```
    //Checks whether each functional unit of RIF is supported
T_INFO support[6];

MAKE_KEYCODE (&support[0], INF_RIF, INF_UNIT, INF_OBJ, 0);
for(i = 1;i<6;i++)
    MAKE_KEYCODE (&support[i], 0, 0, 0, 0);

    //Now, everything from RIF.UNIT.OBJ to RIF.UNIT.CTX will be got.
dbg_ref_rim (support, 6, 0);
```

Program source

Since this structure of getting information structure is supported by more than two function, it is conceivable that different functions may use different information key codes (e.g., the ***CFG*** key may be used for ***dbg_ref_dbg***). Whether the function returns an error, associated value, or invalid value in such an instance is determined by an 'implement definition'. The caller must not assume that information can be got even if the function does not match a key code, or must not expect that an error will be reported when such a procedure is performed.

Each vendor can freely create the key for getting information instead of the use of the key defined in the ITRON Debugging Interface Specification. In such a situation, it is strongly recommended that the second and third high-order bits of the key[*] are both 1. The ITRON Debugging Interface Specification assures that no key definitions formulated in the future will overlap this range.

*. 64 keys in total (0x60-0x7f and 0xe0-0xff).

3.7 Error Codes

3.7.1 E_xxx error and ET_xxx error

Error codes defined as ***E***_xxx in the ITRON Kernel Specification are expressed as ***ET***_xxx in the ITRON Debugging Interface Specification. Error ***ET***_xxx represents an error that may be caused by target operations. On the other hand, error ***E***_xxx represents an error that may occur at the host. For example, ***E_NOMEM*** means that an insufficient memory error has occurred at the host, and ***ET_NOMEM*** means that an insufficient memory error has occurred at the target. Error ***ET***_xxx, which denotes an error at the target, has the same value as the error in the ITRON Kernel Specification. On the other hand, error ***E***_xxx, which denotes an error that may occur at the host, is defined at a position 128 units away from the position in kernel specification. For example, when ***ET_ID*** is -18, ***E_ID*** is -146.

3.7.2 Common errors

Common errors are errors that may occur to all functions defined in the ITRON Debugging Interface Specification.

E_OK

Processing ended normally.

E_NOMEM

Memory was not allocated to host due to memory insufficiency.

E_NOSPT

Function is not supported. This error occurs when the function specified by a flag is not implemented or is inoperative.

E_FAIL

Function could not answer the request due to some factor. However, this error is not serious enough to affect target program execution. If the request is issued again, it may be executed properly. This is a general error that is not serious.
When a function returns this error, the status internally changed by the function must be restored to such a level* that the meaning is the same as that prevailing at the function start. Further, it must not be assumed that a debugging tool calls a function with the same parameters (retry) immediately after it caused an error.
(Example of ***E.FAIL*** error: As the current execution position was in the kernel's critical section at ***rif_ref_obj*** issuance, queue processing was not properly achieved. When ***tif_set_reg*** was issued, all the registers were not written into.)

*. If an argument is invalidated when a function ends with an error, the argument itself is also invalidated. Therefore, the values of the argument need not to be restored because the both meanings are equivalent in the end. When implementation is performed in such a manner that allocated memory is freed at a certain time, absolutely unused memory need not to be freed on the spot.

E_SYS

Function could not answer the request due to some factor. And target computer suspends its execution with an inconsistent state. Even if the request is issued again, normal processing is not performed. This is a general error that is serious. If a function for the target access interface used within an RTOS access interface function terminates unexpectedly with ***E_SYS***, it must return ***E_SYS***.
(Example: When ***rif_ref_obj*** issued, memory read mechanism of debugging tool did not normally operate and failed to get information. While writing ***tif_set_reg*** to a register, it failed to complete the write process so some register values were not updated.)
When the ***E_SYS*** error occurs, a debugging tool should notify the user of the fatal error and state clearly that the operations of the debugging environment (target and debugging tool) are unstable.

3.7.3 Similar errors

This section explains the differences between similar errors defined in the ITRON Debugging Interface Specification.

E_ID and E_NOID

- **E_ID**
  The specified ID range was outside the valid range. The error recurs as long as an ID number within the specified ID range is used.
- **E_NOID**
  ID numbers were not sufficient to assign ID automatically. This error recurs until at least one ID number for automatic assignment is available by means of object destruction and so on

ET_OBJ, ET_NOEXS, and ET_OACV

- **ET_OBJ**
  Although the object assigned to the specified ID existed at the target, the operation was not performed successfully. The error recurs until the cause is removed. (For example, Function reports ***ET_OBJ*** during exclusive kernel’s operation (critical section) of the object.)
- **ET_NOEXS**
  No object with the specified ID exists at the target. The error recurs until the object assigned to the specified ID is generated.
- **ET_OACV**
  Although the object assigned to the specified ID exists at the target, the operation was denied because of an object access violation (e.g. privilege fault). The error recurs until the privilege level of the caller or callee, etc., is changed.

3.8 Variable-Length Storage Region

The ITRON Debugging Interface Specification uses the following two methods to obtain a task ID list and other variable-length information.

- **Separate-space variable-length region (suffix *-IST*)**
  The region for variable-length information storage will be allocated separately from the structure of getting information. This method is used when, for example, the same structure contains plural item of variable-length information.
- **Same-space variable-length region (suffix *-ary*)**
  The region contiguous to the get information structure will be used as the region for getting variable-length information.

Details are given in the following subsections.

3.8.1 Separate-space variable-length region

The separate-space variable-length region consists of variables with two unique suffixes and a region for variable-length data storage.

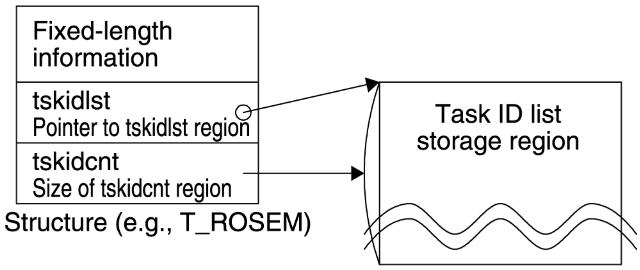

Figure 5: Separate-space Variable-length Region (Task ID)

- **Suffix *lst***
  This variable stores the pointer at the beginning of the region that stores variable-length data.
- **Suffix *cnt***
  This variable stores the size of the variable-length data storage region (in item units).

In the specification, the above variables are described as a pointer variable with the suffix ***lst*** and are contiguous to a variable with the suffix ***cnt***.

3.8.2 Same-space variable-length region

The same-space variable-length region consists of a variable that indicates the size of the storage region and a region contiguous to the structure that is used as the variable-length data stor age region.

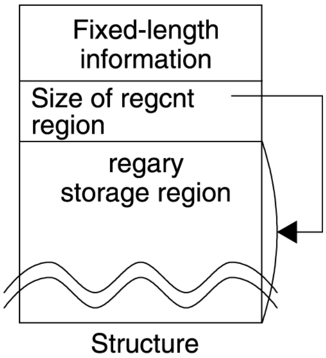

Figure 6: Same-space variable-length Region

- **Suffix ary**
  Array that stores variable-length data
- **Suffix cnt**
  Variable that stores size of array that stores variable-length data (in item units)

In the specification, the above is described as an array or pointer variable that has the suffix ***ary*** and is contiguous to a variable with the suffix ***cnt***.

3.9 Identification Number (ID)

The ITRON Debugging Interface Specification assigns an identification number (ID) to a setting for identification purposes when break point or memory polling (watch point), etc., setting is performed. However, note that this identification number (ID) differs from the ID defined by the ITRON Kernel Specification.

IDs can be assigned to the following functions:

- **RIF break point**
- **TIF break point**
- **Polling (watch point)**
- **RIF log**
- **TIF log**

The characteristics of the IDs are summarized below:

- **The value is 1 or greater positive quantity.**
  0 or less-numbered cannot basically be handled. When a normal method is used, setup items with an 0 or less-numbered value cannot be operated.
- **ID values are not always consecutive.**
  Even when IDs are assigned continuously with the automatic number assignment function, etc., such assigned values are not always consecutive.
- **The value may be reused.**
  Once an ID is freed, it may be reused. However, two or more setup items cannot exist with the same ID and function.
- **The values are independent of each other as far as they have different function.**
  An ID is assigned to each function. Therefore, setup items for the same functions (e.g., TIF break point and RIF break point) may have the same ID. However, the entities of the setup items differ as far as they have different functions even if they have the same ID.

The IDs assigned to the above five functions are declared as an ***ID*** type. On the other hand, the IDs defined by the ITRON Kernel Specification are declared as a ***DT_ID*** type. For the handling of ID type (***DT_ID*** type) variables defined in the ITRON Specification, refer to the ***ITRON Kernel Specification*** and other relevant documents.

3.10 Register Name

The ITRON Debugging Interface Specification uses a character string to identify the registers of the target computer. The following rules apply to the register identification character strings:

- **Characters**
  The character string for a register name must consist of uppercase alphabetical letters (A to Z) and numbers (0 to 9).
- **Character count limitation**
  No register name may exceed 8 characters (termination included) in length.
- **Unique name**
  Each register name must be a name (abbreviation) in a target chip hardware manual or used by an assembler created by a target chip manufacturer. If different names are used to indicate the same register between target and debugger, alias should be given to the register name in debugger side. However, the name must clearly indicate the characteristics of the register.

The following functions use register names:

- **rif_get_rdt : Get of description table**

3.11 Flag

The ITRON Debugging Interface Specification provides all functions with flags for function selection (except for callback function and some supported functions). These flags are used as part of parameters to use functions defined in the ITRON Debugging Interface Specification, including the consistency assurance and automatic number assignment.

The bits of these flags have the following meanings:

Table 15: Functions of Flags

Bit mask	Meaning
0xFF000000	Flag with prefix ***FLG_*** , defined in this specification
0x00FF0000	Reserved
0x0000FF00	Flag region that can be defined freely by the RIM and debugging tool
0x000000FF	Option for each mechanism (Begins with ***OPT_***)

FLG_DEFAULT (= 0) indicates a state with no flag.

For these flags, new items can be added by each vendor. In such a case, use of low-order bits 8 to 15 is strongly recommended. The ITRON Debugging Interface Specification assures that no new flag will be defined in that region.
Every function defined in the ITRON Debugging Interface Specification returns the ***E_NOSPT*** error when an incoming flag cannot be processed by it.

3.12 Register Set Description Table

The register set description table consists of information of register value storage location and register name. The RIM and debugging tools operate the registers and context in accordance with the information written in this register set description table.

The following functions handle the register set description table:

rif_get_rdt	Get of description table
rif_get_ctx	Get of task context
rif_set_ctx	Set of task context
tif_get_reg	Read of register value
tif_set_reg	Write of register value

The register set description table structure (***T_GRDT***) is shown below:

```
typedef struct    t_grdt_regary
{
        char * strname      : Pointer indicating register name
        UINT length         : Length (in bytes)
        UINT offset         : Storage offset position
}         T_GRDT_REGARY;

typedef struct    t_grdt
{
  UINT regcnt               : Count of registers
  UNIT ctxcnt               : Count of registers that can be contained in context
  T_GRDT_REGARY regary[]
                            : Register information
}       T_GRDT;
```

The register set description table has the following features:

- **Stores register name, size, and storage location**
  The register information (***T_GRDT::regary***) in the register set description table stores the register name, register size (in bytes), and offset which is needed for the kernel to load and store the register value. The register length and offset position are required for getting, setup, and similar operations. These values define the offset position and data length concerning the target register value storage within a region that retains the register value.
- **Retains context and registers operated by RIM**
  The register set description table stores two types of information: context information, and register information. The first half of the register set description table lists the registers that can serve as the context for the target OS, and the latter half lists all registers that the RIM may operate.
  T_GRDT::ctxcnt retains the context count of the target OS. The first ***ctxcnt*** registers of ***T_GRDT::regary*** are OS task contexts. On the other hand, ***T_GRDT::regcnt***, retains the count of all registers that are written in the register set description table.

- **Applicable to all register operations**
  Register operation functions provided by the target access interface refer to the register set description table implicitly. Therefore, in principle, register operation functions never handle registers which is not in the register set description table.
- **Remains invariable throughout program execution period**
  The register set description table offered by the RIM does not change throughout a program execution* period. The RIM must not rewrite the contents of the table during execution. Further, the debugging tool must not rewrite the contents of the table that is obtained from the RIM through ***rif_get_rdt*** use.

The four functions using the register set description table, except for ***rif_get_rdt***, retain the enable/disable identification information (***BITMASK_8 * valid***) as an argument. ***valid*** correlates to each element of ***regary***, and the elements are valid when the associated bits are nonzero. Furthermore, the enable/disable identification information obtained as a result of operation is stored again in ***valid***.
When ***valid*** is NULL, all the registers are targeted for operation, and the operation results are not stored.

An example is shown in ***Table 16***. In this example, the task context merely has a program counter (PC) and stack pointer (SP). Further, the RIM may operate a status register (SR) and general-purpose register (R14) in addition to the PC and SP.:

Table 16: Typical Register Set Description Table

Field	Description
regcnt	4
ctxcnt	2
regary [0]	{“PC”, 4, 0}
regary [1]	{“SP”, 4, 4}
regary [2]	{“SR”, 4, 8}
regary [3]	{“R14”, 4, 12}

In accordance with the information contained in the register set description table, the functions for getting ***rif_get_ctx*** and ***tif_get_reg*** store the task contexts and register values in a specified region. The example below shows a typical program execution that is performed using the register set description table indicated in the preceding example:

Program source

```
char buffer[16];
BITMASK_8 valid = 0xa:
tif_get_reg (buffer, &valid, FLG_DEFAULT);
```

Program source

*. The term program execution indicates the range of dbg_ini_rim to dbg_fin_rim.

When the above program is executed and all the operations are ended normally, the function ***tif_get_reg"*** stores a register value in the variable ***buffer*** as follows:

Table 17: Register Storage

Offset	Contents
0 to 3	Nothing stored
4 to 7	Stuck pointer (SP)
8 to 11	Nothing stored
12 to 15	General register (R14)

3.13 Special Blocking Mode

Although, in principle, all of functions execute by the non-blocking mode, ITRON Debugging Interface permits them to execute by the special blocking mode. It is assumed that the special blocking mode is used for processes that do not take a considerable execution time. Use of this special blocking mode facilitates program implementation.

When a function is executed in the special blocking mode, the program is blocked until execution ends. However, to prevent the program being stopped within the function permanently, the special blocking mode times out automatically after a certain time specified by the implementor. If the process is discontinued by a timeout, the function returns ***E_FAIL***.

The special blocking mode prevents certain operations (e.g. update of user interface) being stopped due to the other related operation blocked. Therefore, the time for blocking should be reasonable time for which users can keep waiting[*]. The actual timeout time is implementation-dependent.

The special blocking mode can be used by specifying the ***OPT_BLOCKING*** option flag. The special block mode is supported by the following functions:

- **rif_cal_svc : Issue of service call**
- **tif_set_pol : Set of memory data change report**
- **tif_cal_fnc : Issue of function**

*. It is usually said that the user can only wait several seconds for processing without being notified. However, if the user is notified before or during processing that processing will take a considerable time, the timeout time can be increased as needed. However, note that the user must not be forced to wait for an unlimited period.

4. RTOS Support Function Guideline

4.1 Standardization of Implemented Functionalities

This guideline is to standardize RTOS support functionalities that are implemented by "ITRON Debugging Interface Specification complying debugging tools." Functionality elements of RTOS support provided by ITRON Debugging Interface Specification are listed below:

- **Get of ITRON object status**
- **Handle of task context**
- **Issue of service call**
- **OS-dependent break and trace**
- **OS-dependent execution history (service call, task transition, debugging log, etc.)**

The individual functions and their implementation methods are summarized below:

Get of ITRON object status

This functionality retrieves internal RTOS object information that are normally difficult to inspect, and display them to user.

Examples of information to be got

- **Task**
  Priority, stack, wait factor, waiting object, etc.
- **Synchronous object**
  Control block data, waiting task, etc.
- **Ready queue**
  Running task ID and executable task list
- **System-related information**
  Current context mode and kernel internal status

Handle of task context

This functionality provides the way to handle context information such as register contents, including stack pointer, and program counter. In case of acquisition, the context information can be retrieved from an appropriate region regardless of the task status.

Issue of service call

The issue of service call provides a means of issuing an RTOS service call with appropriate parameter. For example, this can be used to invoke semaphore release operation form debug tool. This functionality is not limited to RTOS service calls. Service calls of other software components can also be invoked, resulting in enhanced debugging capability.

OS-dependent break and trace

The ITRON Debugging Interface Specification supports the following RTOS-dependent break functions:

- **Task-related break**
  Operates break with specifying task ID.
  Operates break for specified task without halting execution of other tasks.
  Operates break when service call is invoked by specified task.
- **Object-related break**
  Operates break when specified object is operated.
- **System-related break**
  Operates break upon context switching.
  Operates break upon dispatch to specified task.

A method for halting the target at each break can be selected.

- **Halts entire system.**
- **Halts targeted task only and continues execution of system (RTOS support required).**

OS-dependent execution history

This function gets the execution history to monitor the system behavior. The ITRON Debugging Interface Specification supports the following functions.

- **Dispatch history**
  Gets task execution transition history.
- **Issue of service call**
  Gets parameters, error codes, and other historical information about issued service calls.
- **User event history**
  Gets comments and other historical information which are described by user.

4.2 Level Indications

Service calls are classified into different levels. Debugging tools clearly compliant with the ITRON Debugging Interface Specification must clearly indicate the levels they support. The user is then allowed to determine the available capabilities.
The ITRON Debugging Interface Specification uses dependent level descripion for RIF and TIF. The subsequent subsections describe the RIF and TIF level indications.

4.2.1 RIF level indication

The RIM (RIF) level is indicated for each functional unit, which is an aggregation of functions that provide RTOS support functions.

The functional units are listed below (abbreviations in brackets):

- **Get of object status [OBJ]**
- **Get of context manipulation [CTX]**
- **Issue of service call [SVC]**
- **Set of break [BRK]**
- **Get of break condition [CND]**
- **Execution history [LOG]**

The RIM must describe which fuctional units are supported.
The debugging tool must describe the implementation of a connection mechanism (user interface, etc.) for each functional unit as the RIF level.

A level description example is shown below. When the combination shown in ***Table 18*** is used, the end user can use the minimum functions for get of object status, context manipulation, issue of service call, set of break, and execution history*.

Table 18: Level Indication Example

Functional unit	RIM	Debugging tool
OBJ	○	○
CTX	○	○
SVC	○ (***tif_alc_mbt*** required)	○(***tif_cal_svc*** unsupported)
BRK	○	
CND	○	× (Conditional break nonsupported)
LOG	○(RIM applicable independently)	× (UI offered)
Other	Partly expanded	TIF level [R]

*: The debugging tool does not support a function for getting execution history. However, it is available because it can be executed by the RIM independently. Strictly speaking, it means that the TIF log mechanism cannot be used.

4.2.2 TIF level indication

The TIF level description not only notifies the end user of available functions but also provides an index for a RIM implementer.
The TIF level is provided for each function and is roughly divided into the following two types (abbreviations in brackets):

- **Necessary function [R]**
  This type of function must be implemented in the form of a debugging tool that complies with the ITRON Debugging Interface Specification. The RIM implementer need not check whether a necessary function exists. When the extended part of the necessary function is used, the function may return ***E_NOSPT***.
- **Extended function [E]**
  An extended function is mainly defined for convenience (e.g., conditional break). It may not be implemented depending on the debugging tool. Therefore, the RIM creator must not issue a call without checking that a Function offering an extended function exists.

4.2.3 Other interface

For the RIF "***rif_ref_cfg*** (get of kernel configuration)" and some functions with a name beginning with ***dbg_***, the required [R] and extended [E] description are used as with the TIF. Since these functions mainly serve as information for RIM and debugging tool creator, they need not be described even when they are implemented.

4.3 Terms and Definitions

4.3.1 Debugging tool

In the ITRON Debugging Interface Specification, the monitoring tools for checking whether the target program and target hardware are normally operating are collectively called ***debugging tools***. The debugging tool does not contain a target or target program. However, it may contain a debugging agent (described later) needed for debugging and offered by tool manufacturers.

Using “debuggers” as a term is avoided intentionally. Debuggers usually indicates programs. If a tool including peripheral tools are also referred to as debuggers, the difference between the target and debuggers would be vague. The term debugging tools is therefore used to distinguish such a difference.

4.3.2 Debugging agent

In the ITRON Debugging Interface Specification, the software is collectively called a ***debugging agent*** as far as it functions as a program on the target hardware to provide support for debugging when a debugging environment is created. This term is used without regard to the implementation form no matter whether such software is an internal part of an OS or a task.

4.4 Break Mechanism

The ITRON Debugging Interface Specification furnishes mechanism and functions to support a break in consideration of RTOS. This section details such a break mechanism.

The break mechanism consists of the following functions:

rif_set_brk	Request of break point set
rif_del_brk	Delete of break point
rif_rep_brk	Report of break hit
rif_ref_brk	Get of set break information
rif_ref_cnd	Get of break condition
tif_set_brk	Set of break point
tif_del_brk	Delete of break point
tif_rep_brk	Break report

The following subsection details the characteristics of the break mechanism.

4.4.1 Decision of callback

When a debugging tool reaches break point defined by the function ***tif_set_brk*** on the target access interface, a debugging tool calls the callback function ***tif_rep_brk*** to let the RIM determine whether or not to halt the operation.
Meanwhile, when RIM reaches a break point defined by the function ***rif_set_brk*** on the RTOS access interface, the RIM calls the callback function ***tif_rep_brk*** to report a halt. However, ***rif_rep_brk*** does not have a return value and cannot decide on a break suspension. When ***tif_rep_brk*** returns ***E_TRUE***, the debugging tool allows a break operation to continue. However, if ***tif_rep_brk*** returns ***E_FALSE***, the debugging tool aborts a break operation and resumes target execution. However, ***rif_rep_brk*** cannot decide on a break abortion.

The break operation is executed with the above-mentioned sequential operation.

An explanation is given below with an example.

1. The debugging tool uses ***rif_set_brk*** to inform the RIM of the user break point setting request. The RIM uses the ***tif_set_brk*** function to set a break point at a specific address.

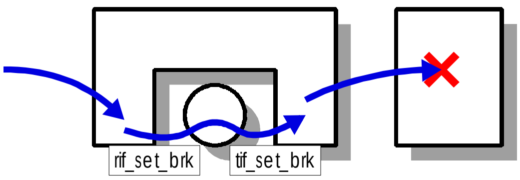

Figure 7: Setting of Break Point

2. When target execution is initiated by the RIM or user and the target program satisfies the break point setting conditions, the debugging tool function is exercised to break the target. From this time on, execution of the target program is broken.

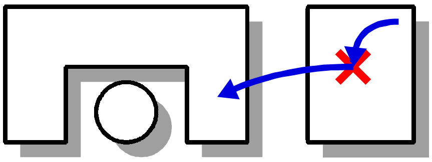

Figure 8: Break Hit

3. When the break point that caused a halt was set by ***tif_set_brk***, the debugging tool calls the callback function ***tif_rep_brk*** to report the occurrence of a break. In this case, the break point ID and break parameters set by ***tif_set_brk*** are passed as arguments.

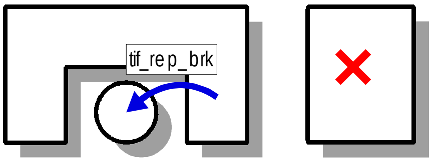

Figure 9: ***tif_rep_brk*** Call

4. The RIM makes full use of TIF functions to collect adequate information for determining whether or not to halt target execution according to the preset break, and then makes a judgment. The break process continues with ***Step 5*** when target execution should be halted or continues with ***Step 5***, when target execution should not be halted.

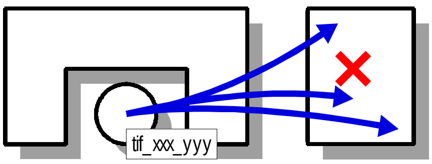

Figure 10: Information Collection

5. If RIM decides to halt target execution as a result of information collection and if the break point has been set by ***rif_set_brk***, RIM calls the ***rif_rep_brk*** callback function. If target should halt, ***E_TRUE*** is returned by the callback.

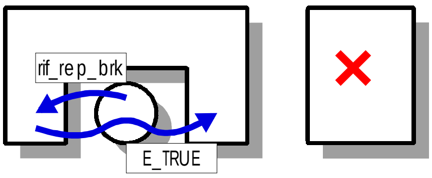

Figure 11: Operation of ***rif_rep_brk*** when Conditions Satisfied

6. The debugging tool allows continuing the break operation, and then notifies the user that target execution halted at the break point.

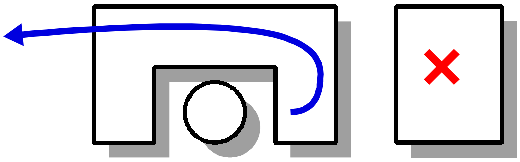

Figure 12: Continuation of Break Operation

5'. If, as a result of information collection, the RIM concludes that target execution should not be halted, RIF function immediately returns ***E_FALSE***.

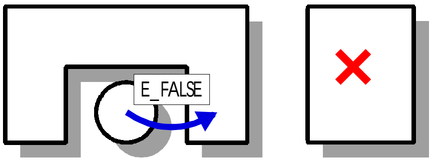

Figure 13: Operation of ***rif_rep_brk*** when Conditions Not Satisfied

6'. The debugging tool resumes the target program execution that was halted in ***step 2***.

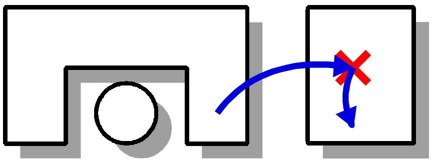

Figure 14: Abortion of Break Operation and Resumption of Target Program Execution

4.4.2 Break of condition-getting type

The ITRON Debugging Interface Specification provides another break support mechanism that acquires break conditions only (see ***Section 2.5.1***). In order to use this feature, the debugging tool must provide a conditional break capability.

The condition-getting type break mechanism consists of the following function:

rif_ref_cnd Get of break condition

The operation flow is shown below:

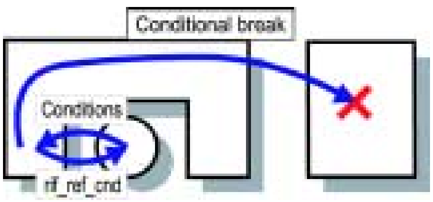

Figure 15: Break of Condition-getting Type

1. Based on user request, the debugging tool in turn calls ***rif_ref_cnd*** to have RIM generate RTOS-dependent conditions.
2. The RIM generates conditions that satisfies the request, and then returns them to the debugging tool.
3. With the generated conditions, the debugging tool sets a break point at a user-specified address.

When the condition-getting type brake mechanism is used, the debugging tool performs break point setup. Therefore, ***rif_rep_brk*** will not be called due to a break point set by this method.

4.5 Trace Log Mechanism

To support acquisition of OS-dependent execution history, the ITRON Debugging Interface Specification furnishes a trace log mechanism, which consists of a series of functions and a group of functions. This section details the trace log mechanism.

The trace log mechanism consists of the following functions:

rif_set_log	Set of trace log
rif_del_log	Delete of trace log set
rif_sta_log	Request of trace log function start
rif_stp_log	Request of trace log acquisition stop
rif_get_log	Get of trace log

Trace log mechanism operations can be roughly divided into six types: set, start, execution, get, end, and delete. During a single use of the trace log mechanism, the trace log mechanism process performs one setup operation, two or more series of 'start, execution, and end' operations, two or more getting, and one deletion.

Each of these operations is detailed in the following subsections.

4.5.1 Set

For trace log setting, the function ***rif_set_log*** is used to set trace logs as required times.

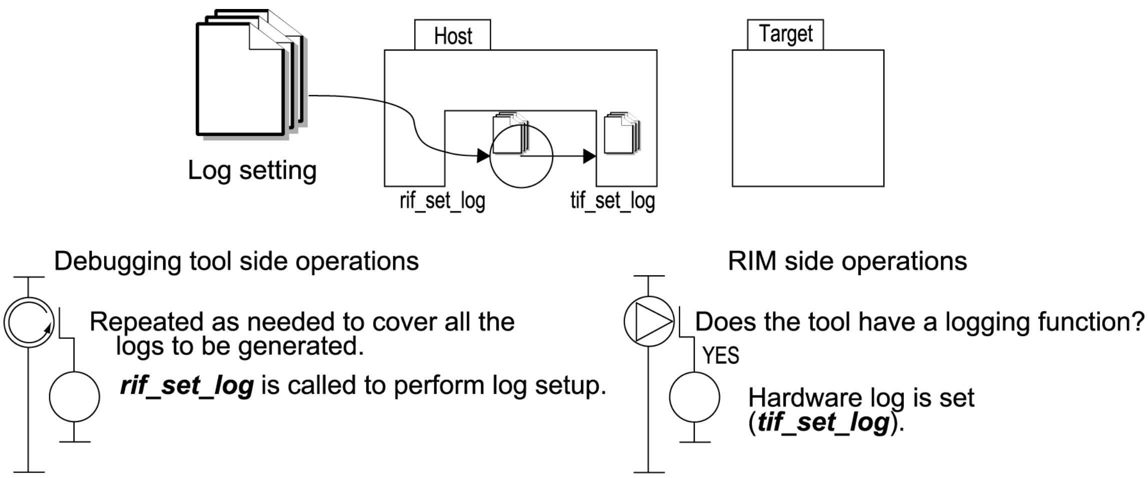

Figure 16: Set of Trace Log

At this stage, no operations are performed to affect on target systems*. The debugging tool gives trace settings to the RIM. The RIM performs necessary operations to prepare for subsequent log triggering (may occasionally optimize setup, for example, by merging the given settings with previously defined settings).
If, for example, the debugging tool can get a memory access log, the RIM uses the ***tif_set_log*** function to perform relevant setting at the same time.

*: This statement is made from the user viewpoint. The target may be more or less manipulated depending on the implementation. However, such a manipulation must not be perceivable by the user.

4.5.2 Start

For trace log starting, a process is performed for getting preselected trace logs.

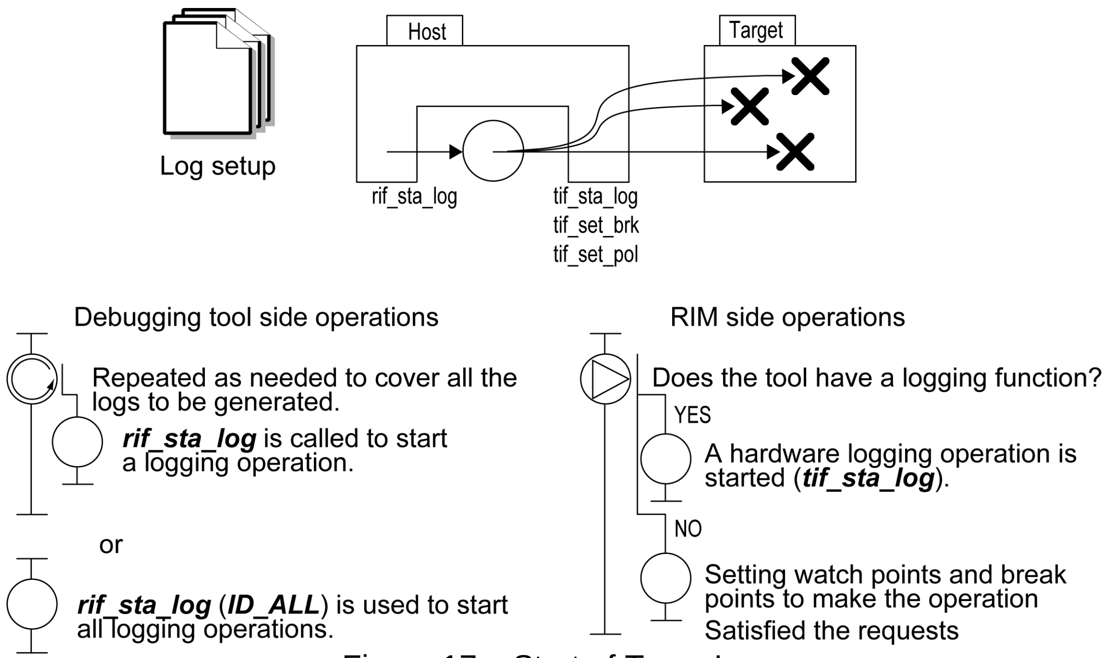

Figure 17: Start of Trace Log

Tracing is prepared according to the preset configuration. If the debugging tool itself has a trace log mechanism, ***tif_sta_log*** is called by RIM to pass settings and to enable the trace. Otherwise, break points and watch points are set to satisfy the settings, and necessary logging information is gathered by using memory read and other TIF APIs through various callbacks invoked upon subsequent target execution.

4.5.3 Execution

For execution, the program is run to get log information.

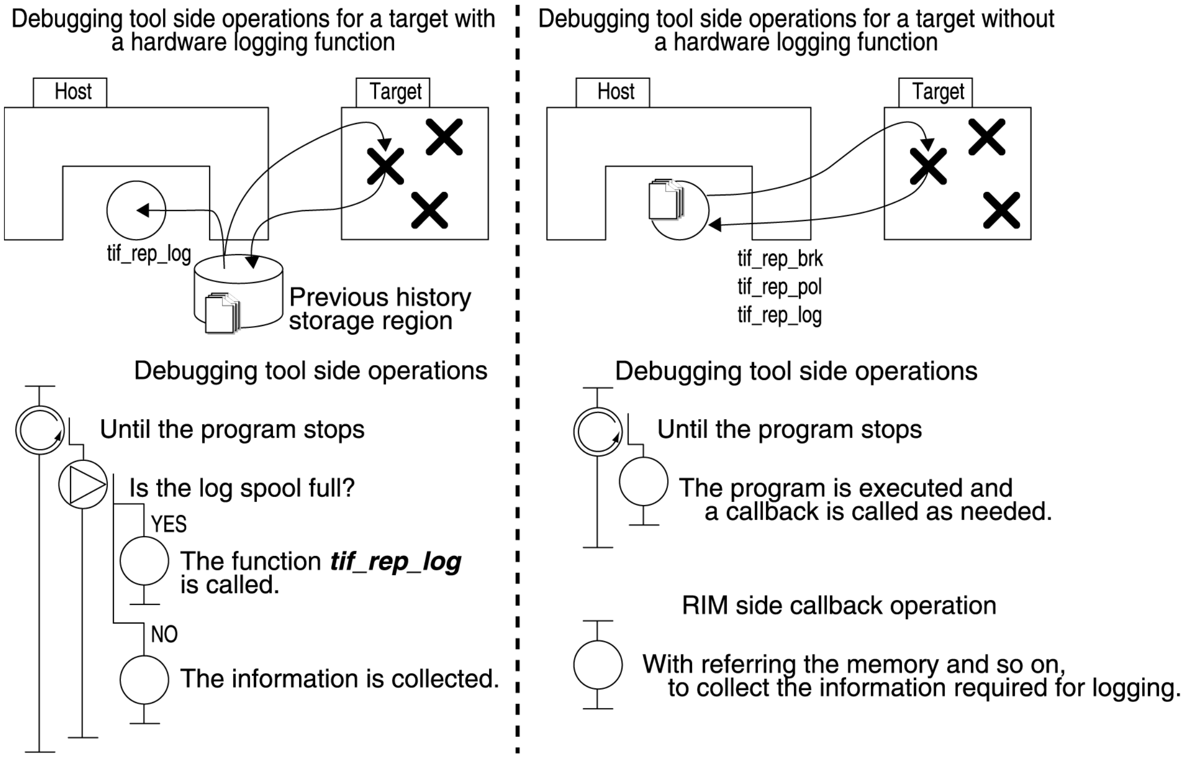

Figure 18: Execution of Trace Log

If the debugging tool has a hardware log mechanism, all mechanisms related to get log depend on hardware. The RIM executes callbacks that are generated only when the hardware log mechanism log buffer (called the ***previous history storage region*** in the ITRON Debugging Interface Specification) becomes full and when trace log retrieval ends.
On the other hand, if there is no apropriate hardware log mechanism, the RIM uses watch point and break point callback functions to collect the necessary information required for logging.

4.5.4 Get

After the end of program execution for getting log, the debugging tool gets the log.

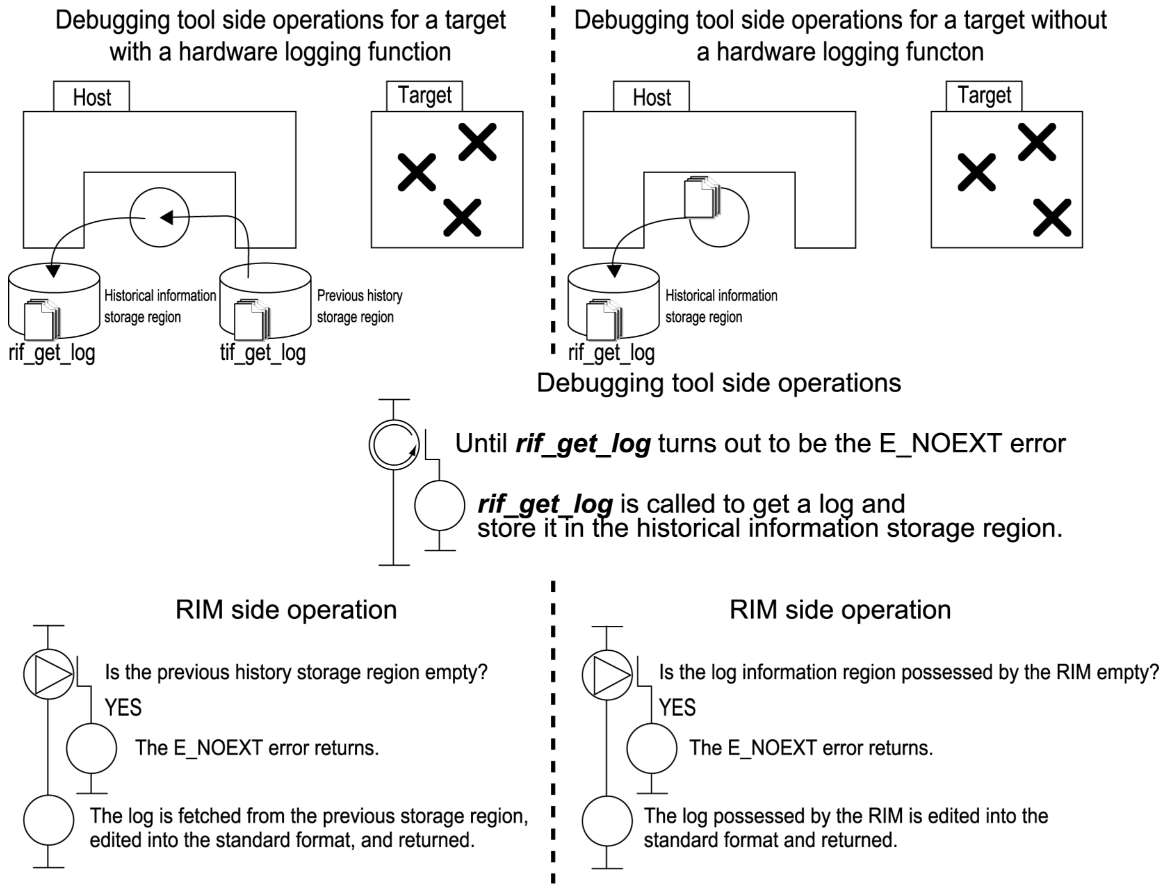

Figure 19: Trace Log Getting

The debugging tool calls ***rif_get_log*** to get one record of log. Each record is obtained in the order they gatherd, and then stored in the historical information storage region.
Since the historical information storage region stores trace log in a standard format defined by the ITRON Debugging Interface Specification, the log may be viewed by some standard viewer, apart from debugging tool used to collect them.

Log acquisition can be performed at any time while the log mechanism is operating.*
Possible log acquisition timings are shown below:

- **When log spool becomes full**
- **When program ends**

*: The getting order denotes the storage order. It does not precisely represent the chronology of getting log.

4.5.5 End

After getting of trace log is completed, the debugging tool terminates the log mechanism.

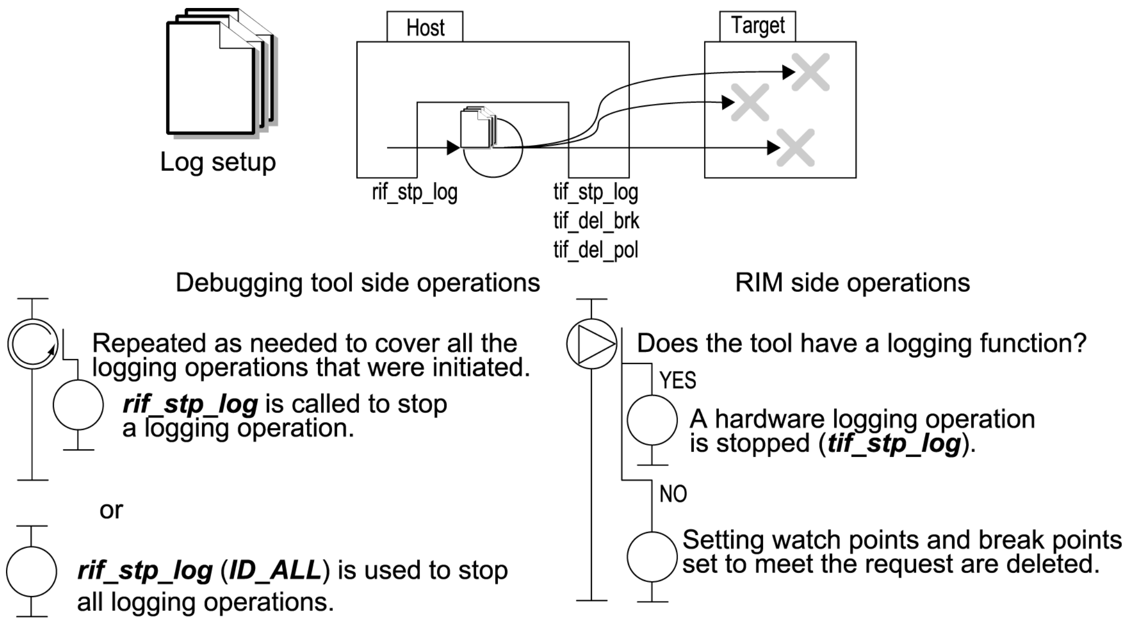

Figure 20: End of Trace Log

The RIM frees prepared resources for trace log retrieval such as a TIF break.
When the debugging tool has a log mechanism, the hardware log mechanism enabled by the ***tif_sta_log*** function is disabled by the ***tif_stp_log*** function. Under other circumstances, the break points and watch points set for getting information are deleted. In both cases the memory acquired by the RIM as the log storage region is also released at the same time.

4.5.6 Delete

When the log setup is no longer needed after completion of the entire get log, the debugging tool deletes the log setting.

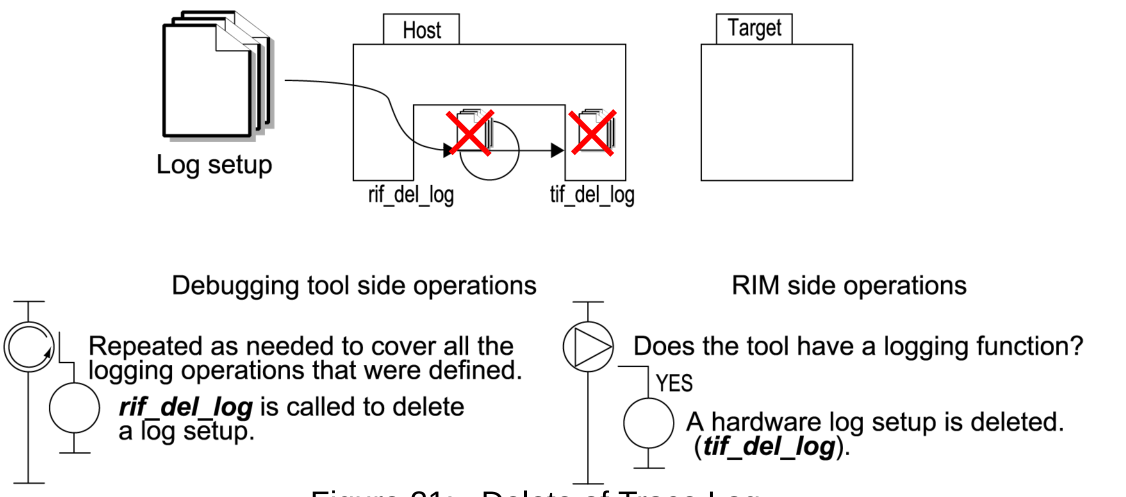

Figure 21: Delete of Trace Log

Upon deletion of settings, the RIM frees the corresponding settings of trace log retrieval, and hardware trace log settings if the RIM uses them.

5. RTOS Access Interface

5.1 Functional Unit

All the functions on the RTOS access interface are grouped into two or more functional units. Function availability is determined on an individual functional unit basis.
When each functional unit is available, it means that all the functions composing the functional unit are implemented and that the key code for identifying that functional unit exists (however, the ***E_NOSPT*** error may be returned if some functions of Functions are not implemented).

The functional units are given below (Abbreviations is in parenthesis):

- **Get of object status [OBJ]**
- **Get of context [CTX]**
- **Issue of service call [SVC]**
- **Set of break [BRK]**
- **Get of break condition [CND]**
- **Execution history [LOG]**

Keys

RIF 4_H

.UNIT 20_H

.OBJ 1_H [1]

Supports the "get of object status" functional unit.

.LOG 2_H [1]

Supports the "get of execution history" functional unit.

.SVC 3_H [1]

Supports the "issue of service call" functional unit.

.BRK 4_H [1]

Supports the "set of break" functional unit.

.CND 5_H [1]

Supports the "get of break condition" functional unit.

.CTX 6_H [1]

Supports the "get of context" functional unit.

5.2 Get of object Status

rif_ref_obj Get of object status [OBJ]◯

ER rif_ref_obj

(VP p_result, UINT objtype, DT_ID objid, FLAG flags)

VP p_result

Result storage location

UINT objtype

Object type

DT_ID objid

The object ID of the target to be got

FLAG flags

Various flags

This function gets the status of an object that currently exists on the RTOS.

For getting object status, a flag for specifying the object type (ObjType) and a result packet for status storage are used. The read upper limit for the "waiting task ID list" and other variable-length data described as "type *identifier-***lst***" is determined when this function is called with the upper-limit value set for the "count parameter" described as "***UINT*** identifier-***cnt***", which is corresond with "type *identifier-AIB". In this case, the smaller data berween the "read upper-limit value" substituted before operation and the "actual variable-length data count" is substituted into the "count parameter" after operation. If the "actual variable-length data count" exceeds the "read upper-limit value", variable-length data transfer does not take place beyond the upper-limit value.

Table 19 shows the relationship between the ***R_ROSEM.wtskcnt*** data and the data stored in the ***wtsklst***-specified region when ***rif_ref_obj*** is issued to a semaphore having 10 tasks in a waiting list.

Table 19: Operation Performed in Relation to a Semaphore Having 10 Tasks in a Waiting List

T_ROSEM.wtskcnt		**Data Stored in wtsklst-specified Region**
Before Execution	***After Execution***	
0	***0***	***Nothing is stored.***
1	***1***	***ID of the task positioned at the beginning of the waiting queue.***
2	***2***	***The first waiting task ID and the second waiting task ID.***
10	***10***	***IDs of the waiting tasks from the first one to the last one.***
11	***10***	***Same as above.***

At the beginning of the packet returned by ***rif_ref_obj***, a bit mask is positioned to indicate whether the subsequent field is valid or not. The first candidate for the bit mask is a structure member that follows "***valid***". If a structure member is a pointer to another structure, the enable/disable identification information about the structure member indicated by the pointer is stored, but the enable/disable identification information about the pointer to another structure is not stored. If the pointer is invalid, all members of the structure indicated by the pointer are invalidated. For detailed bit mask descriptions, see ***Section 3.5***.
As an example, the table below shows the fields of the structure ***T_ROMPF*** that stores the information about a fixed-length memory pool (***OBJ_FMEMPOOL***) and the corresponding bit mask bit positions:

Table 20: Relationship between ***T_ROMPF*** Members and Bit Mask Bit Positions

Bit Position	Structure Member
0	***T_ROMPF::mpfatr***
1	***T_ROMPF::blksz***
2	***T_ROMPF::fblkcnt***
3	***T_ROMPF::blkcnt***
4	***T_ROMPF::ablkcnt***
5	***T_ROMPF_BLKLST::htskid***
6	***T_ROMPF_BLKLST::blkadr***
7	***T_ROMPF::wtskcnt***
8	***T_ROMPF::wtsklst***

Object identification flags (***ObjType***) and result packets are shown below. If NC is attached to the end of an object name, it means that the associated items and arguments are invalid.

- **OBJ_SEMAPHORE (0x80): Semaphore**

```
typedef struct      t_rosem
{
      BITMASK valid       : Valid field flag
      DT_ATR sematr       : Semaphore attribute
      DT_UINT isemcnt     : Initial semaphore count
      DT_UINT maxsem      : Semaphore maximum value
      DT_UINT semcnt      : Semaphore count value
      DT_UINT wtskcnt     : Waiting task count (also used as the wtsklst upper limit)
      DT_ID * wtsklst     : Pointer to the region for storing the waiting task ID list
      T_ROSEM;
}
```

Gets semaphore-related information. Before execution, ***wtsklst*** and ***wtskcnt*** must be initialized.

- **OBJ_EVENTFLAG (0x81): Event flag**

typedef struct t_roflg_wflglst
{
DT_ID wtskid : Waiting task ID
DT_FLGPTN wflgptn : Wait flag pattern for each task
DT_UINT wflgmode : Wait mode for each task

} **T_ROFLG_WFLGLST;**

typedef struct t_roflg
{
BITMASK valid : Valid field flag
DT_ATR flgatr : Flag attribute
DT_FLGPTN iflgptn : Initial flag pattern
DT_FLGPTN flgptn : Flag pattern
DT_UINT wtskcnt : Waiting task count (also used as the upper limit for the wflglst)
T_ROFLG_WFLGLST * wflglst
: Pointer to information about task with this flag
} **T_ROFLG;**

Gets the information about an event flag. Before execution, ***wtskcnt***, ***wtsklst***, ***wflgptn***, and ***wflgmode*** must be initialized.

- **OBJ_DATAQUEUE (0x82): Data queue**

typedef struct t_rodtq
{
BITMASK valid : Valid field flag
DT_ATR dtqatr : Data queue attribute
DT_UINT dtqcnt : Data queue capacity
DT_UINT stskcnt : Count of tasks waiting for sending (also used as the upper limit for wstsklst)
DT_UINT * stsklst : Pointer to region storing ID list of tasks waiting for transmission
DT_UINT rtskcnt : Count of tasks waiting for reception (also used as the upper limit for wrtsklst)
DT_ID * rtsklst : Pointer to the region for storing the ID list of tasks waiting for reception
DT_UINT itemcnt : Count of queue data (also used as the upper limit for itemlst)
DT_VP_INT * itemlst: Pointer to the region for storing the list of all items
} **T_RODTQ;**

Gets the information about a data queue. Before execution, ***stskcnt***, ***wstsklst***, ***rtskcnt***, ***wrtsklst***, ***itemcnt***, and ***itemlst*** must be initialized.

- **OBJ_MAILBOX (0x83): Mailbox**

typedef struct t_rombx
{
BITMASK valid : Valid field flag
DT_ATR mbxatr : Mailbox attribute
DT_PRI maxmpri : Maximum priority
DT_UINT wtskcnt : Count of waiting tasks (also used as the upper limit for wtsklst)
DT_ID * wtsklst : Pointer to the region for storing the ID list of waiting tasks
DT_UINT msgcnt : Count of message headers (also used as the upper limit for msglst)
DT_T_MSG ** msglst: Pointer to the region for storing the list of all messages
} **T_ROMBX;**

Gets the information about a mailbox. Before execution, ***wtskcnt***, ***wtsklst***, ***msgcnt***, and ***msglst*** must be initialized.

- **OBJ_MUTEX (0x84): Mutex**

typedef struct t_romtx
```
{
    BITMASK valid       : Valid field flag
    DT_ATR mtxatr       : Mutex attribute
    DT_PRI ceilpri      : Upper-limit priority
    DT_ID htskid        : ID of the task that locks a mutex
    DT_UINT wtskcnt     : Count of waiting tasks (also used as the upper limit for wtsklst)
    DT_ID * wtsklst     : Pointer to the region for storing the ID list of waiting tasks
}   T_ROMTX;
```
Gets the information about a mutex. Before execution, ***wtskcnt*** and ***wtsklst*** must be initialized.

- **OBJ_MESSAGEBUFFER (0x85): Message buffer**

typedef struct t_rombf_msglst
```
{
    DT_VP msgadr        : Message addresses
    DT_UINT msgsz       : Message length
}   T_ROMBF_MSGLST;
```

typedef struct t_rombf
```
{
    BITMASK valid       : Valid field flag
    DT_ATR mbfatr       : Message buffer attribute
    DT_UINT maxmsz      : Message maximum size
    DT_SIZE mbfsz       : Buffer region size
    DT_UINT stskcnt     : Count of tasks waiting for sending (also used as the upper limit
                          for stsklst)
    DT_ID * stsklst     : Pointer to region storing ID list of waiting tasks
    DT_UINT rtskcnt     : Count of tasks waiting for reception (doubles as rtsklst upper limit)
    DT_ID * rtsklst     : Pointer to the region for storing the ID list of waiting tasks
    DT_SIZE fmbfsz      : Free region size
    DT_UINT msgcnt      : Count of messages (also used as the upper limit for msgls)
    T_ROMBF_MSGLST * msglst
                        : Pointer to information about messages
}   T_ROMBF;
```
Gets the information about a message buffer. Before execution, ***stskcnt***, ***wtsklst***, ***msgcnt***, ***msglst***, and ***msgszlst*** must be initialized.

- **OBJ_RENDEZVOUSPORT (0x86): Rendezvous port**

typedef struct t_ropor
```
{
    BITMASK valid       : Valid field flag
    DT_ATR poratr       : Rendezvous port attribute
    DT_UINT maxcmsz     : Call message maximum size
    DT_UINT maxrmsz     : Response message maximum size
    DT_UINT ctskcnt     : Count of tasks waiting for a call (also used as the upper limit for
                          ctsklst)
    DT_ID * ctsklst     : Pointer to the region for storing the IDs of all the tasks waiting
    DT_UINT atskcnt     : Count of tasks waiting for acceptance (doubles as atsklst upper
                          limit)
```

DT_ID * atsklst : Pointer to the region for storing the IDs of all the tasks waiting for acceptance
} **T_ROPOR;**

Gets the information about a rendezvous port. Before execution, ***ctskcnt***, ***atskcnt***, ***ctsklst***, and ***atsklst*** must be initialized.

- **OBJ_RENDEZVOUS (0x87): Rendezvous**

typedef struct t_rordv
{
BITMASK valid : Valid field flag
DT_ID tskid : ID of a task waiting for a rendezvous
} **T_RORDV;**

Gets the information about a rendezvous.

- **OBJ_FMEMPOOL (0x88): Fixed-length memory pool**

typedef struct t_rompf_blklst
{
DT_ID htskid : ID number of task that acquired block
DT_VP blkadr : Block starting address
} **T_ROMPF_BLKLST;**

typedef struct t_rompf
{
BITMASK valid : Valid field flag
DT_ATR mpfatr : Fixed-length memory pool attribute
DT_SIZE blksz : Block size
DT_UINT fblkcnt : Count of remaining fixed-length memory blocks
DT_UINT blkcnt : Count of all memory blocks
DT_UINT ablkcnt : Count of allocated block (blklst upper limit)
T_ROMPF_BLKLST *ablklst : Pointer to detailed information about blocks

DT_UINT wtskcnt : Count of tasks waiting for getting (wtsklst upper limit)
DT_UINT * wtsklst : Pointer to region storing IDs of tasks waiting for get
} **T_ROMPF;**

Gets the information about a fixed-length memory pool. Before execution, ***ablkcnt***, ***ablklst***, ***wtskcnt***, and ***wtsklst*** must be initialized.

- **OBJ_VMEMPOOL (0x89): Variable-length memory pool**

typedef struct t_rompl_blklst
{
DT_SIZE blksz : Block size
DT_ID htskid : ID number of task that got block
DT_VP blkadr : Block starting address
} **T_ROMPL_BLKLST;**

typedef struct t_rompl
{
BITMASK valid : Valid field flag
DT_ATR mplatr : Variable-length memory pool attribute
DT_SIZE mplsz : Variable-length memory pool region size
DT_UINT fblksz : Maximum gettable size

DT_UINT ablkcnt : Count of allocated block (upper limit for blklst)
T_TOMPL_BLKLST * ablklst
: Pointer to detailed information about blocks
DT_UNIT wtskcnt : Count of tasks waiting for getting (wtsklst upper limit)
DT_ID * wtsklst : Pointer to region storing IDs of tasks waiting for getting
} **T_ROMPL;**

Gets the information about a variable-length memory pool. Before execution, ***ablkcnt***, ***ablklst***, ***wtskcnt***, and ***wtsklst*** must be initialized.

- **OBJ_TASK (0x8a): Task**

typedef struct t_rotsk
{
BITMASK valid : Valid field flag
DT_ATR tskatr : Task attribute
DT_VP_INT exinf : Extension information
DT_FP task : Startup address
DT_PRI itskpri : Initial priority
DT_VP stk : Starting address of initial stack
DT_SIZE stksz : Stack size
DT_STAT tskstat : Task status
DT_PRI tskpri : Current task priority
DT_PRI tskbpri : Task base priority
DT_STAT tskwait : Factor of a task's wait
DT_ID wobjid : ID of an object to wait for
DT_TMO lefttmo : The remaining time before timeout
DT_UINT actcnt : Activation requests queuing count
DT_UINT wupcnt : Wake-up requests queuing count
DT_UINT suscnt : Suspension requests count
} **T_ROTSK;**
Gets the information about a task.

- **OBJ_READYQUEUE (0x8b): Ready queue (NC: objid)**

typedef struct t_rordq
{
BITMASK valid : Valid field flag
DT_ID runtskid : ID of the currently executed task
DT_UINT tskcnt : Count of ready (and running) tasks (upper limit for tsklst)
DT_ID * tsklst : Pointer to the region for storing the IDs of all the executable tasks
} **T_RORDQ;**
Gets the information about a ready queue. If no executable task exists, the value 0 returns to ***runtskid*** and ***tskcnt***. In this case, the ***tsklst*** data has no change.
Before execution, ***tskcnt*** and ***tsklst*** must be initialized.

- **OBJ_TIMERQUEUE (0x8c): Timer queue (NC: objid)**

typedef struct t_rotmq_quelst
{
UINT objtype : Pointer to the region for storing the types of waiting objects
DT_ID wobjid : Pointer to the region for storing the IDs of waiting objects
DT_TMO lefttmo : Pointer to the region for storing the remaining wait time
} **T_ROTMQ_QUELST;**

```
typedef struct    t_rotmq
    BITMASK valid        : Valid field flag
    DT_SYSTIM systim     : System time at the time of getting information
    DT_UINT quecnt       : Count of waiting objects in a timer queue (upper limit for quelst)
    T_TORMQ_QUELST * quelst
                         : Pointer to information about objects in timer queue
}   T_ROTMQ;
```

Gets the information about a timer queue.
The timer queue information contains the types of all events (cyclic handler, alarm handler, overrun handler, and task) to be activated by a time event and the scheduled times for the generation of such events. As regards a cyclic handler, however, the information will not be got not from all activating positions but from the next activating position.
The type of a waiting object is stored with a constant described as ***OBJ_xxx*** which use to specify the object with ***rif_ref_obj***.
Before execution, ***quecnt***, ***objtyplst***, ***wobjidlst***, and ***lefttmolst*** must be initialized.

- **OBJ_CYCLICHANDLER (0x8d): Cyclic handler**

```
typedef struct    t_rocyc
{
    BITMASK valid        : Valid field flag
    DT_ATR cycatr        : Attribute
    DT_VP_INT exinf      : Extension information
    DT_FP cychdr         : Start address
    DT_RELTIM cyctim     : Cycle
    DT_RELTIM cycphs     : Initial phase
    DT_STAT cycstat      : Cyclic handler start status
    DT_RELTIM lefttim    : Remaining time
}   T_ROCYC;
```

Gets the information about a cyclic handler.

- **OBJ_ALARMHANDLER (0x8e): Alarm handler**

```
typedef struct    t_roalm
{
    BITMASK valid        : Valid field flag
    DT_ATR almatr        : Attribute
    DT_VP_INT exinf      : Extension information
    DT_FP almhdr         : Startup address
    DT_STAT almstat      : Alarm handler start status
    DT_RELTIM lefttim    : Remaining time
}   T_ROALM;
```

Gets the information about an alarm handler.

- **OBJ_OVERRUNHANDLER (0x8f): Overrun handler**

```
typedef struct    t_roovr
{
    BITMASK valid        : Valid field flag
    DT_ATR ovratr        : Attribute
    DT_FP ovrhdr         : Start address
    DT_STAT ovrstat      : Handler start status
```

DT_OVRTIM lefttmo: Remaining processor time
} **T_ROOVR;**

Gets the information about an overrun handler.

- **OBJ_ISR (0x90): Interrupt service routine**

typedef struct t_roisr
{
BITMASK valid : Valid field flag
DT_ATR isratr : Attribute
DT_VP_INT exinf : Extension information
DT_FP isrfnclst : Registered routine start address
DT_INTNO inhno : Applied interrupt handler number
} **T_ROISR;**

Gets the information about an interrupt service routine.

- **OBJ_KERNELSTATUS (0x91): Kernel information (NC: objid)**

typedef struct t_roker
{
BITMASK valid : Valid field flag
BOOL actker : Kernel start status (TRUE = activated)
BOOL inker : Kernel code execution (TRUE = execution in progress)
BOOL ctxstat : Context status ***(sns_ctx)***
BOOL loccpu : CPU locked status ***(sns_cpu)***
BOOL disdsp : Dispatch disabled status ***(sns_dsp)***
BOOL dsppnd : Dispatch suspended status ***(sns_dpn)***
DT_SYSTIM systim : System time
DT_VP intstk : Stack for non-task context
DT_SIZE intstksz : Stack size for non-task context
} **T_ROKER;**

Gets the information about kernel status.

actker is a variable that indicates the kernel start status. It is ***FALSE*** in the target start sequence. It is ***TRUE*** when a system call become available after completion of kernel initialization.

inker is a variable that indicates whether currently executed code is a kernel code or not*.

- **OBJ_TASKEXCEPTION (0x92): Task exception handler**

typedef struct t_rotex
{
BITMASK valid : Valid field flag
DT_TEXPTN pndptn: Suspended exception factor
DT_FP texrtn : Exception handler start address
} **T_ROTEX;**

*. The statuses that are judged as a kernel operation are as follows; a sequence between exception occurrence and handler stertup, a sequence between handler termination and dispatcher termination, or a sequence between target startup and initial application task starup.

Gets the information about a task exception handler.

- **OBJ_CPUEXCEPTION (0x93): CPU exception handler (objid corresponds to an exception factor)**

typedef struct t_roexc
{
BITMASK valid : Valid field flag
DT_FP excrtn : Exception handler activating address
} **T_ROEXC;**

Gets the information about CPU exception.

Supplementary explanation

Implement-dependent information is defined as a structure member that follows each structure. For definition of a unique object, the employed object identification constant must be outside the range from 0 to 255. When performing a unique object operation, it is best to set up a flag to clarify it.

Flags

OPT_GETMAXCNT (1)

Even when the variable-lengh data count exceeds the upper limit value, this flag throughly tracks and gets the data count.

OPT_VENDORDEPEND (2)

Gets implement-dependent information.

FLG_NOCONSISTENCE (10000000$_H$): Nonconsistency flag

When this flag is specified, the data to get need not be consistent (e.g., the task is not freed from the waiting state although there is no factor of the task wait).

FLG_NOSYSTEMSTOP (20000000$_H$): An explicit system halt is not permitted.

When this flag is specified, ***tif_brk_tgt*** must not be used within a function to halt the system. If this flag is not supported, the ***E_NOSPT*** error occurs.

Keys

RIF 4$_H$

- **.RIF_REF_OBJ** 1$_H$
  - **.FLG_NOCONSISTENCE** 1$_H$ [1]
    The FLG_NOCONSISTENCE flag is available.
  - **.FLG_NOSYSTEMSTOP** 2$_H$ [1]
    The ***FLG_NOSYSTEMSTOP*** flag is available.
  - **.OPT_VENDORDEPEND** 10$_H$ [1]
    The ***OPT_VENDORDEPEND*** option is available.
  - **.OPT_GETMAXCNT** 11$_H$ [1]
    The ***OPT_GETMAXCNT*** option is available.
  - **.STATICPARAMETER** 12$_H$
    - **.OBJ_SEMAPHORE** 80$_H$ [T]
      This structure has semaphore information that is statically determinative.
    - **.OBJ_EVENTFLAG** 81$_H$ [T]
      This structure has event flag information that is statically determinative.
    - **.OBJ_DATAQUEUE** 82$_H$ [T]
      This structure has data queue information that is statically determinative.
    - **.OBJ_MAILBOX** 83$_H$ [T]
      This structure has mailbox information that is statically determinative.
    - **.OBJ_MUTEX** 84$_H$ [T]
      This structure has mutex information that is statically determinative.
    - **.OBJ_MESSAGEBUFFER** 85$_H$ [T]
      This structure has message box information that is statically determinative.
    - **.OBJ_RENDEZVOUSPORT** 86$_H$ [T]
      This structure has rendezvous port information that is statically determinative.
    - **.OBJ_RENDEZVOUS** 87$_H$ [T]
      This structure has rendezvous information that is statically determinative.

.OBJ_FMEMPOOL 88_H [T]

This structure has fixed-length memory pool information that is statically determinative.

.OBJ_VMEMPOOL 89_H [T]

This structure has variable-length memory pool information that is statically determinative.

.OBJ_TASK $8A_H$ [T]

This structure has task information that is statically determinative.

.OBJ_READYQUEUE $8B_H$ [T]

This structure has ready queue information that is statically determinative.

.OBJ_TIMERQUEUE $8C_H$ [T]

This structure has timer queue information that is statically determinative.

.OBJ_CYCLICHANDLER $8D_H$ [T]

This structure has cyclic handler information that is statically determinative.

.OBJ_ALARMHANDLER $8E_H$ [T]

This structure has alarm handler information that is statically determinative.

.OBJ_OVERRUNHANDLER $8F_H$ [T]

This structure has overrun handler information that is statically determinative.

.OBJ_ISR 90_H [T]

This structure has interrupt service routine information that is statically determinative.

.OBJ_KERNELSTATUS 91_H [T]

This structure has kernel information that is statically determinative.

.OBJ_TASKEXCEPTION 92_H [T]

This structure has task exception information that is statically determinative.

.OBJ_CPUEXCEPTION 93_H [T]

This structure has CPU exception information that is statically determinative.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_CONSIST (-225)

Consistency was not assured (however, it is not handled as an error if ***FLG_NOCONSISTENCE*** is set).

E_NOEXS (-42)

The targeted object was not found on the target.

E_PAR (-145)

A parameter value was invalid.

ET_OBJ (-41)

The targeted object on the target was inoperative.

ET_ID (-18)

The specified kernel object ID was invalid.

ET_OACV (-27)

An invalid object on an target was accessed (tskid < 0).

5.3 Get of Task Context

5.3.1 Get of register set description table

rif_get_rdt Get of description table [CTX]◯

ER rif_get_rdt (const T_GRDT ** ppk_pgrdt, FLAG flags)

const T_GRDT ** ppk_prgrdt

Pointer to the region that stores the pointer to the register set description table structure

FLAG flags

Flags

This instruction gets the pointer to the register table that contains the context information about a targeted task. The body of this table is located in the RIM and its contents are constant. If the debugging tool is to be used to modify the contents, make a copy of the contents with the debugging tool and then modify the contents of the copy.

The register table has the details of registers that need to be saved in case of task switching. The structure "***T_GRDT***" is detailed below. For the register table, see ***Section 3.12***.

```
typedef struct    t_grdt_regary
{
    char * strname       : Pointer to register name
    UINT length          : Length (in bytes)
    UINT offset          : Storage offset position
}   T_GRDT_REGARY;

typedef struct    t_grdt
{
    UINT regcnt          : Count of registers
    UINT ctxcnt          : Count of registers that can be contained in context
    T_GRDT_REGARY regary[]
                         : Register information
}   T_GRDT;
```

Supplementary explanation

The register set description table contains all the registers that compose the context and all the registers to be operated by the RIM. The targeted task context consists of ***T_GRDT::ctxcnt*** specified number of elements beginning with the start of ***T_GRDT::regary***. ***T_GRDT::regcnt*** indicates the count of registers to be operated by the RIM.

Keys

RIF 04_H

.RIF_GET_RDT 02_H

.REGISTER 2_H

.SIZE 04_H [W]

Size (in bytes) of adequate region for register storage.

.CONTEXT 12_H

.SIZE 04_H [W]

Size (in bytes) of adequate region for context storage.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused for some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_PAR (-145)

A parameter value was invalid.

5.3.2 Get of task context

rif_get_ctx Get of task context [CTX]◯

ER rif_get_ctx

(VP p_ctxblk, BITMASK_8 * p_valid, DT_ID tskid, FLAG flags)

VP p_ctxblk
Leading pointer that indicates the region for storing got context

BITMASK_8 * p_valid
Pointer to validation flag about register table items
(NULL: Targets entire context)

DT_ID tskid
ID of a targeted task

FLAG flags
Flags

This function gets and stores task context in accordance with the register table that is got by ***rif_get_rdt***. This function permits the debugging tool to get context from an appropriate region at all times irrespective of the current task status.

The variable "***p_ctxblk***" is the pointer to the buffer that stores context obtained upon execution of this function. Before executing this function, the debugging tool must create a region that is large in size enough to store the context. The size of this buffer can be got by using the information acquisition key code ***RIF.RIF_GET_RDT.CONTEXT.SIZE***. It can also be calculated from the register table got by the function ***rif_get_rdt***. When the region size is determined by calculation, it is necessary to furnish a region that is large enough to store only the context portion of the register table.

Storage is performed in accordance with the storage offset position and register length written in the register table got by ***rif_get_rdt***. For the register table, see ***Section 3.12***.

p_valid specifies whether the registers should be enabled or disabled. When given as an argument for the function, ***p_valid*** does not store disabled registers. This function also stores the result of getting targeted register in ***p_valid***. However, if ungot registers are essential to the targeted task context, an error such as ***ET_MACV*** is returned depending on the situation*. The information stored in regions related to ungot registers is implement-dependent. Even if the enable/disable information is given in excess of the number of registers (***T_GRDT::ctxcnt***) composing the context, excess registers will not be got.
If NULL is specified for ***p_valid***, the whole context is targeted for getting so that the details of the result will not be stored.

When the flag ***OPT_APPCONTEXT*** is specified, the context is got on the application level. If the task is stopped inside the kernel, the RIM uses the current stack frame, etc., to generate and return the context that prevailed before kernel code entry.

*. For example, the floating-point register is not required for tasks that do not perform floating-point calculations. Even if the floating-point register is contained in the register table in such a situation, the function may return E_OK without getting the floating-point register.

Flags

OPT_APPCONTEXT (1)

Handles context in application level as a target.

FLG_NOCONSISTENCE (10000000_H): Nonconsistency flag

When this flag is specified, the got data need not be consistent (e.g., the task is not cleared from the waiting state although there is no factor of the task's wait).

FLG_NOSYSTEMSTOP (20000000_H): An explicit system halt is not permitted.

When this flag is specified, ***tif_brk_tgt*** must not be used within a function to halt the system. If this flag is not supported, the ***E_NOSPT*** error occurs.

Keys

RIF	04_H
.RIF_GET_CTX	03_H
.FLG_NOCONSISTENCE	01_H [1]

The ***FLG_NOCONSISTENCE*** flag is available.

.FLG_NOSYSTEMSTOP	02_H [1]

The ***FLG_NOSYSTEMSTOP*** flag is available.

.OPT_APPCONTEXT	10_H [1]

The ***OPT_APPCONTEXT*** option is available.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation faiure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_CONSIST (-225)

Consistency was not assured (however, it is not handled as an error if ***FLG_NOCONSISTENCE*** is set).

ET_OBJ (-41)

The targeted object on the target was inoperative.

ET_OACV (-27)

An invalid object on an target was accessed (tskid < 0).

ET_ID (-18)

The specified kernel object ID was invalid.

E_PAR (-145)

A parameter value was invalid.

ET_NOEXS (-42)

The target object was not found on the target.

5.3.3 Set of task context

rif_set_ctx Set of task context [CTX]◯

ER rif_set_ctx

(VP p_ctxblk, BITMASK_8 * valid, FLAG flags)

VP p_ctxblk

Pointer to the region that stores the context to be set

BITMASK_ * p_valid

Pointer to validation flag about register table items
(NULL: Targets entire context)

FLAG flags

Flags

This function sets task context in accordance with the register table that is obtained by ***rif_get_rdt***. The use of this function permits the debugging tool to set appropriate context at all times irrespective of the current task status.

Setup is performed in accordance with the information in the register table obtained by ***rif_get_rdt***. The variable "***p_ctxblk***" is the pointer to the buffer that stores the context to be set upon execution of this function. Before executing this function, the debugging tool must store the context data to be set in a specified region in accordance with the register table obtained by ***rif_get_rdt***. For the register table, see ***Section 3.12***.

p_valid specifies whether the registers should be enabled or disabled. When given as an argument for the function, ***p_valid*** does not set disabled registers. This function also stores the result of targeted register acquisition in ***p_valid***. However, if registers that cannot be set are essential to the targeted task context, an error such as ***ET_MACV*** is returned depending on the situation*. Even if enable/disable information is given in excess of the number of registers (***T_GRDT::ctxcnt***) composing the context, excess registers will not be set.
If NULL is specified for ***p_valid***, the whole context is targeted for setup so that the result details will not be stored.

When the flag "***OPT_APPCONTEXT***" is specified, the context in application level will be set.

Flags

OPT_APPCONTEXT (1)

Handles context in application level as a target.

FLG_NOSYSTEMSTOP (20000000$_H$): An explicit system halt is not permitted.

When this flag is specified, ***tif_brk_tgt*** must not be used within a function to halt the system. If this flag is not supported, the ***E_NOSPT*** error occurs.

*. For example, the floating-point register is not required for tasks that do not perform floating-point calculations even if it is contained in the register table. In such a situation, the function may return E_OK without setting the floating-point register even when it is targeted for setup.

Keys

RIF 01_H

.RIF_SET_CTX 13_H

.FLG_NOSYSTEMSTOP 02_H [1]

The ***FLG_NOSYSTEMSTOP*** flag is available.

.OPT_APPCONTEXT 10_H [1]

The ***OPT_APPCONTEXT*** option is available.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_CONSIST (-225)

Consistency was not assured (however, it is not handled as an error if ***FLG_NOCONSISTENCE*** is set).

ET_OBJ (-41)

The targeted object on the target was inoperative.

ET_OACV (-27)

An invalid object on a target was accessed (tskid < 0).

ET_ID (-18)

The specified kernel object ID was invalid.

E_PAR (-145)

A parameter value was invalid.

ET_NOEXS (-42)

The target object was not found on the target.

5.4 Issue of Service Call

5.4.1 Issue of service call

rif_cal_svc Issue of service call [SVC]◯

ER rif_cal_svc (T_RCSVC * pk_psvc , FLAG flags)

T_RCSVC * pk_psvc
Information of call issuance

FLAG flags
Flags

This function issues a service call. Since issuance is executed in non-blocking mode, the end of this function does not mean the end of a service call. However, note that issuance is executed in the special blocking mode only when ***OPT_BLOCKING*** is set explicitly. The execution process performed in the special blocking mode times out at the pre-selected timeout time.

Contents of ***T_RCSVC***

```
typedef struct    t_rcsvc
{
    DT_FN svcfn          : Function code to be issued
    BOOL tskctx          : Execution with task context (= TRUE)
    DT_ID tskid          : ID of a targeted task (when tskctx = TRUE)
    UINT prmcnt          : Parameter count
    VP_INT paramry[]     : Array that stores list of all parameters
}   T_RCSVC;
```

Supplementary explanation

Since this function is executed in non-blocking mode, the end of this function is not identical with the end of the issued service call. However, if the use of non-blocking mode is prohibited due to the employed RIM implementation method and blocking mode is implemented, the end of this function can be regarded as the end of service call. So, the termination of this function can be regarded as the service call end. ***RIF.RIF_CAL_SVC.NON-BLOCKING*** should be implement as **FALSE** to let the ***dbg_ref_rim*** function inform the debugging tool that the end of this function is regarded as the end of the service call.

When ***T_RCSVC::tskctx*** is set to **FALSE**, the service call for which this function is set will be executed with nontask context.

Even when ***OPT_BLOCKING*** is specified, the callback function "***rif_rep_svc***" is called unless ***FLG_NOREPORT*** is specified.

When ***rif_cal_svc*** is executed in the special blocking mode, the function may not return control until the service call terminates in the strict sense. In the strict sense, the service call terminates when the stack frame prevailing at function termination is equivalent to the stack frame prevailing when a function call is made by ***rif_cal_svc***. More specifically, if the service call is executed in such a manner as to invoke dispatching, such as a wait within the function, the dispatch to the same task recurs and this function does not return control until the target service call is completed. Furthermore, if the same function is executed recursively within the target function, this function does not return control until termination occurs for the same number of times as the calls. However, when execution is performed in the special blocking mode, a predefined timeout occurs even if the termination does not occur in the strict sense. For details, see ***Section 3.13***, Special Blocking Mode.

T_RCSVC::prmary stores the value to be delivered as a parameter. The method of parameter delivery conforms to the method for the ***μITRON*** 4.0-compliant service call ***cal_svc*** (For structures, etc., the pointers to structures are stored).

Flags

FLG_NOREPORT (80000000$_H$): Report function invalidation

The paired callback function will not be called.

OPT_BLOCKING (1)

Executed in a blocking mode.

Keys

RIF	04$_H$
.RIF_CAL_SVC	04$_H$
.FLG_NOREPORT	03$_H$ [1]
The ***FLG_NOREPORT*** flag is available.	
.OPT_BLOCKING	10$_H$ [1]
The ***OPT_BLOCKING*** flag is available.	
.OPT_APPCONTEXT	11$_H$ [1]
The ***OPT_APPCONTEXT*** option is available.	
.NON-BLOCKING	12$_H$ [1]
A non-blocking SVC issue is supported.	

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_PAR (-145)

A parameter value was invalid.

E_EXCLUSIVE (-226)

Another request was already issued. The function could not receive a new request until execution of the previous request ends.

ET_OBJ (-41)

The targeted object on the target was inoperative.

ET_OACV (-27)

An invalid object on an target was accessed (tskid < 0).

ET_ID (-18)

The specified kernel object ID was invalid.

ET_NOEXS (-42)

The target object was not found on the target.

5.4.2 Cancel of an issued service call

rif_can_svc Cancel of an issued service call [SVC]○

ER rif_can_svc (FLAG flags)

FLAG flags
Flags

This function cancels the service call that is issued immediately before the operation. However, this function aims at getting focus that was lost by issuance. It cannot completely eliminate the influence of the service call.

Flags

OPT_CANCEL (0)
Does not consider the influence of the issued service call **(default)**.

OPT_UNDO (1)
Completely restore the state to the status before the issuance.

Keys

RIF 04_H

.RIF_CAN_SVC 05_H [1]
rif_can_svc is implemented.

.OPT_CANCEL 10_H [1]
The ***OPT_CANCEL*** option is available.

.OPT_UNDO 11_H [1]
The ***OPT_UNDO*** option is available.

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_OBJ (-169)
The targeted object on the target was inoperative.

5.4.3 Report of service call end

rif_rep_svc Report of service call end [SVC:callback]□

void rif_rep_svc (DT_ER result)

DT_ER result
Error code for the last-issued service call

When a service call invocated by ***rif_cal_svc*** ends, the debugging tool calls the callback function ***rif_rep_svc*** to report the service call end to the RIM. ***rif_rep_svc*** is a callback function to receive the error code for the last-issued service call. However, if the ***FLG_NOREPORT*** flag is specified when ***rif_cal_svc*** is used to issue a service call, this function does not report the end.

Supplementary explanation

The argument "***result***" stores an error code (***ET_xxx***) that complies with the kernel specification.

This function is called at the same time as the end of a service call. Therefore, an end report might be made before escape from ***rif_cal_svc***. To avoid such a problem, you should not write the following code:

Program source

```
volatile int flag;
rif_rep_svc(err)
{  flag = 1; }

foo()
{
    rif_cal_svc(....);
            //Clears the flag (reporting may be completed at this time).
    flag = 0;
            //Blocking continues until the service call ends.
    while(flag == 0);
}
```

Program source

Keys

RIF	04_H
.RIF_CAL_SVC	06_H

Error

This function does not return a value.

5.4.4 Get of function code

rif_ref_svc Get of function code [SVC]◯

ER rif_ref_svc (DT_FN * p_svcfn, char * strsvc, FLAG flags)

DT_FN * p_svcfn

Pointer to the region for storing a function code that corresponds to the name of a service call

char * strsvc

Name of a targeted service call

rif_ref_svc gets a function code from a service call function name. Function codes got by this function can be used for functions that have ***rif_cal_svc***, ***rif_set_brk***, ***rif_set_log***, and other function codes as parameters.

Supplementary explanation

The "***str_svc***" argument (name of the targeted service call) for this function corresponds to an API name that defined by the μITRON Standard. When the prefix "_" for C or a suffix (parameter type, byte count, etc.) for C++ is added to the service call name, the normal operations of the function are not guaranteed. Normal operations will not be guaranteed either if a parameter section is specified in the parenthesis following an API name.

Keys

RIF	04_H
.RIF_REF_SVC	07_H

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_OBJ (-169)

The targeted object on the target was inoperative.

E_PAR (-145)

A parameter value was invalid.

5.4.5 Get of service call name

rif_rrf_svc Get of service call name [SVC]◯

ER rif_rrf_svc

(char * pstr_svc, UINT bufsz, DT_FN svcfn, FLAG flags)

char * pstr_svc
Pointer to the beginning of the region that stores the name of a service call

UINT bufsz
Size of the buffer that stores the name (termination symbol included)

DT_FN svcfn
Function code of a targeted service call

FLAG flags
Flags

rif_rrf_svc gets a service call name in accordance with a function code.

Supplementary explanation

pstr_svc (service call name) is a return value of this function. This return value is an API name defined by the μITRON Standard. The prefix "_" for C language or a suffix (parameter type, byte count, etc.) for C++ language is not added to the function name. Similarly, the parameter section in a parenthesis following an API name is not added.
For the argument "***bufsz***" the size of the buffer region specified by ***p_strsvc*** must be set in bytes. In this instance, ***buflsz*** contains a terminal symbol. To thoroughly get a service call name, therefore, it is necessary that the specified size be not smaller than "service call name length + 1". If this condition is not satisfied, the service call name, including terminal symbol, will be stored without exceeding the above-mentioned length limit. When ***bufsz*** is 1, a normal end occurs with only the terminal symbol stored. However, if ***bufsz*** is 0, the ***E_PAR*** error occurs.

Keys

RIF 04_H

.RIF_RRF_SVC 08_H

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_OBJ (-169)

The targeted object on the target was inoperative.

E_PAR (-145)

A parameter value was invalid.

5.5 Set of Break Point

5.5.1 Set of break point

rif_set_brk Set of break point [BRK]◯

ER_ID rif_set_brk (ID brkid, T_RSBRK * pk_rsbrk , FLAG flags)

ID brkid
Break point ID

T_RSBRK * pk_rsbrk
Pointer to the structure that has the information about the break to be set

FLAG flags
Flags

(Return value) ID brkid
Assigned break point ID

This function offers function for setting an RTOS-dependent break. A break point ID is assigned to a break point. A positive number other than 0 is used to indicate a break point ID. It is used for cancellation and hit notification.

The structure of ***T_RSBRK*** is as shown below:

```
typedef struct    t_rsbrk
{
    UINT brktype        : Break type
    UINT brkcnt         : Count before break
    DT_ID tskid         : Task ID
    DT_ID objid         : Object ID
    UINT objtype        : Object type
    VP_INT brkprm       : Parameter for callback function
    DT_VP brkadr        : Address for break setting
    DT_FN svcfn         : Function code
}   T_RSBRK;
```

brktype consists of one "stop condition", a desired number of "additional conditions", and one "stop procedure" detailed below. The parameters to be used are parenthesized. Note, however, that ***brkcnt*** is valid for all combinations.

Stop conditions

- **BRK_EXECUTE (1)**
  Sets an execution break (***brkadr***, ***tskid***)
- **BRK_ACCESS (2)**
  Sets an access break (***brkadr***, ***tskid***)
- **BRK_DISPATCH (3)**
  Sets a break for a task dispatcher (after execution) (***tskid***)
- **BRK_SVC (4)**
  Performs a break upon an SVC (***tskid, objid***, ***svcfn***)

Additional conditions

- **BRK_ENTER (00_H)**
  Places a break at the start position (***BRK_DISPATCH***, ***BRK_SVC***)
- **BRK_LEAVE (80_H)**
  Places a break at the escape position (***BRK_DISPATCH***, ***BRK_SVC***)

Stop procedures

- **BRK_SYSTEM (0_H)**
  Stops all system when a break occurs.
- **BRK_TASK (40_H)**
  Stops task unit when a break occurs.
- **BRK_REPORT (20_H)**
  Makes a report only (but does not break).

Special values are set to the paramenters, as detailed below:

Table 21: Special Parameter Values Available for Break Setup

Parameter	Value	Meaning
tskid	***ID_ALL (-1)***	***Targets all tasks for a break.***
objid	***ID_ALL (-1)***	***Targets all objects for a break.***
svcfn	***ID_ALL (-1)***	***Breaks upon each SVC.***
brkcnt	***BRK_NOCNT (1)***	***Does not use a count.***

The above values are variously combined for break setup purposes.
Example: Breaks upon the tenth switch to task 2.

Program source
```
T_RSBRK {
    brktype     : BRK_DISPATCH
    brkcnt:     : 10
    tskid       : 2
}
```
Program source

Example: Breaks when task 5 attempts to get semaphore 2.

Program source
```
T_RSBRK {
    brktype     : BRK_SVC
    brkcnt      : BRK_NOCNT
    tskid       : 5
    objtype     : OBJ_SEMAPHORE
    objid       : 2
    ext.svcfn   : -0x25 (wai_sem)
}
```
Program source

The parameters to be ignored depending on the option selection will be basically excluded from consideration. However, if a vendor furnishes a special break setting function, the use of an argument section and the addition of parameters are permitted. However, the following flag must be set for "***flags***" to indicate above mentioned states.

OPT_EXTPARAM (2)

Specifies an extended parameter.

When a task dispatcher is used for setup, the RIM sets breaks at all locations where task dispatch may occur in the kernel.

Supplementary explanation

This function is called by the debugging tool. However, the debugging tool must not set a break that it does not support. (For example, a debugging tool that does not support an access break must not use this function to request access break setup.)

When the function is executed successfully in situations where the automatic number assignment flag "***FLG_AUTONUMBERING***" is specified, the function returns the value of 1 or greater (ID value), which is assigned to a break point. This is also true even when the automatic assignment flag is not specified.

Flags

OPT_NOCNDBREAK (1)

A conditional break can not be used for break setting.

OPT_EXTPARAM (2)

Specifies an extended parameter.

FLG_NOREPORT (80000000$_H$): Report function invalidation

The corresponding callback function will not be called.

FLG_AUTONUMBERING (40000000$_H$): ID automatic assignment

Automatically assigns an ID. The function ignores an argument which is specified with ID. When successful, the function returns the automatically assigned ID.

Keys

RIF 04$_H$

.RIF_SET_BRK 09$_H$

.FLG_NOREPORT 03$_H$ [1]

The ***FLG_NOREPORT*** flag is available.

.FLG_AUTONUMBERING 04$_H$ [1]

The ***FLG_AUTONUMBERING*** flag is available.

.OPT_NOCNDBREAK 10$_H$ [1]

The ***OPT_NOCNDBREAK*** option is available.

.OPT_EXTPARAM 11$_H$ [1]

The ***OPT_EXTPARAM*** option is available.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the opration could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_PAR (-145)

A parameter value was invalid.

E_ID (-146)

The specified object ID was invalid.

E_NOID (-162)

Count of IDs for automatic assignment was insufficient.

ET_OBJ (-41)

The targeted object on the target was inoperative.

ET_OACV (-27)

An invalid object on an target was accessed (tskid < 0).

ET_ID (-18)

The specified kernel object ID was invalid.

ET_NOEXS (-42)

The targeted object was not found on the target.

5.5.2 Delete of break point

rif_del_brk Delete of break point [BRK]◯

ER rif_del_brk (ID brkid, FLAG flags)

ID brkid

ID of the break point to be deleted

FLAG flags

Flags

This function requests the RIM to delete an RTOS-dependent break.
When ***brkid*** is set to ***ID_ALL*** (= 0), the function deletes all break points.

Keys

RIF	04_H
.RIF_DEL_BRK	$0A_H$

Flags

None in particular

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_OBJ (-169)

The targeted object on the target was inoperative.

E_ID (-146)

The specified object ID was invalid.

5.5.3 Report of break hit

rif_rep_brk Report of break hit [BRK:callback]☐

void rif_rep_brk (ID brkid, VP_INT exinf)

ID brkid
ID of the break hit

VP_INT exinf
Extended parameter

When a break set by ***rif_set_brk*** is reached and broken, the RIM uses this callback to report the break. Normally, the Debugging tool requires the "***tif_rep_brk***" callback function to the RIM for calling this function.
An extended parameter can be passed to the function. This parameter uses the value of ***T_RSBRK::brkprm*** when break point is set with ***rif_set_brk***.

Keys

RIF	04_H
.RIF_REP_BRK	$0B_H$

Errors

This function does not have any return value.

5.5.4 Get of break information

rif_ref_brk Get of break information [BRK]◯

ER rif_ref_brk (ID brkid, T_RSBRK * ppk_rsbrk, FLAG flags)

ID brkid
Break point ID

T_RSBRK * ppk_rsbrk
Pointer to the region that stores break information

FLAG flags
Flags

This function gets the break point information that corresponds to the specified break point ID. When the function turns out to be successful, it stores the information about the specified break point ID in the region specified by ***ppk_sbrk***.

Keys

RIF 04_H

.RIF_REF_BRK $0C_H$

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_OBJ (-169)
The targeted object on the target was inoperative.

E_ID (-146)
The specified object ID was invalid.

E_PAR (-145)
A parameter value was invalid.

5.5.5 Get of break condition

rif_ref_cnd Get of break condition [CND]◯

ER rif_ref_cnd

(T_RRCND_DBG * ppk_dbg, T_RRCND_RTOS * pk_rtos,

FLAG flags)

T_RRCND_DBG * ppk_dbg

Pointer to the region that stores the information to check conditions that was set

T_RRCND_RTOS * pk_rtos

Pointer to the region that stores the conditions to be got

FLAG flags

Flags

This function is used to view the RTOS-dependent conditions that should be examined when a debugging tool merely uses its own functions to perform an RTOS-dependent break.

The following RTOS-aware conditions are entered for ***T_RRCND_RTOS***:

typedef struct t_rrcnd_rtos

{

FLAG flags : Contents to be examined

DT_ID objid : ID as a condition

} **T_RRCND_RTOS;**

The following value can be set for "***T_RRCND_RTOS::objid***":

- **CND_CURTSKID (0)**
  Conditions under which the ID of the currently executed task is equal to ***id***

This function returns the method of checking the conditions that is set to ***T_RRCND_DBG***. The following items of information to be checked is returned:

typedef struct t_rrcnd_dbg

{

DT_VP execadr : Execution address (NULL: NC)

DT_VP valadr : Address for comparison (NULL: NC)

UINT vallen : Data length (1, 2, or 4 bytes)

VP_INT value : Data or pointer value

} **T_RRCND_DBG;**

The conditions generated by ***T_RRCND_DBG*** is stated as “when program counter reachs ***execadr*** and ***vallen*** bytes data from the memory address ***valadr*** is ***value***”. When ***NULL*** is stored at ***execadr***, this expression becomes a conditional expression that is independent of the program counter. If ***valadr*** is omitted, this expression turns out to be a conditional expression that is independent of memory data. However, if this function generates conditions under which ***execadr*** and ***valadr*** are both ***NULL***, the debugging tool that has executed this function concludes that all the conditions are invalid.

T_RRCND_DBG::value stores the value that is compared. If the value is greater than ***VP_INT***, ***value*** must also store the pointer to the region that stores this value.

Supplementary explanation

This function checks whether the range of specified IDs is valid. However, it does not check whether tasks exist.

Keys

RIF	04_H
.RIF_REF_CND	$0D_H$

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_PAR (-145)
A parameter value was invalid.

E_CND (-228)
The conditions can not be set.

ET_ID (-18)
The specified kernel object ID was invalid.

5.6 Execution History (Trace Log)

5.6.1 Set of trace log

rif_set_log Set of trace log [LOG]◯

ER_ID rif_set_log

(ID logid, UINT logtype, VP pk_rslog, FLAG flags)

ID logid
: ID number to be assigned to the log to be set

UINT logtype
: Type of the log to be set

VP pk_rslog
: Pointer to the region that stores the trace log setup information

FLAG flags
: Flags

(Return value) ID logid
: Unique value for identifying the log that is set

This function passes the setup information for get trace log to the RIM and make a request to get it.

The following values can be used as ***logtype***:

- **LOG_TYP_INTERRUPT (1)**
  Interrupt
- **LOG_TYP_ISR (2)**
  Interrupt service routine
- **LOG_TYP_TIMERHDR (3)**
  Timer handler
- **LOG_TYP_CPUEXC (4)**
  CPU exception
- **LOG_TYP_TSKEXC (5)**
  Task exception
- **LOG_TYP_TSKSTAT (6)**
  Task state
- **LOG_TYP_DISPATCH (7)**
  Task dispatch
- **LOG_TYP_SVC (8)**
  Service call
- **LOG_TYP_COMMENT (9)**
  Comment (It is a log which consists of a character string only; mainly written by the user)

LOG_ENTER (0$_H$) and ***LOG_LEAVE (80$_H$)*** exist. The former is used as an additional flag to activator or start. The latter is used to terminate an operation. If these desired position specifers are omitted, it is concluded that ***LOG_ENTER*** is specified (e.g., ***LOG_TYP_TSK*** | ***LOG_ENTER***: gets a log in relation to a task startup).

The following structures are assigned to the above-mentioned various types. These structures are used for ***pk_rslog***. When "***ID_ALL (= -1)***" is specufued, parameters marked "ID_ALL available", all IDs will be targeted. Substitution must be conducted by casting into the respective type as necessary.

LOG_TYP_INTERRUPT (1): Interrupt (start, end)

```
typedef struct    t_rslog_interrupt
{
    DT_INTNO intno      : Interrupt number (ID_ALL available)
}   T_RSLOG_INTERRUPT;
```

LOG_TYP_ISR (2): Interrupt service routine (start, end)

```
typedef struct    t_rslog_isr
{
    DT_ID isrid         : Interrupt service routine ID (ID_ALL available)
    DT_INTNO intno      : Interrupt number (ID_ALL available)
}   T_RSLOG_ISR;
```

Note: If ***intno*** is***ID_ALL***, ***isrid*** is automatically set to***ID_ALL***.

LOG_TYP_TIMERHDR (3): Timer event handler (start, end)

```
typedef struct    t_rslog_timerhdr
{
    UINT type           : Handler type (OBJ_ALL available)
          (stores the "OBJ_xxx" constant that is used for rif_ref_obj::objtype.)
          (all types will be targeted when OBJ_ALL(= ID_ALL) is specified.)
    DT_ID hdrid         : Handler ID (ID_ALL available)
}   T_RSLOG_TIMERHDR;
```

LOG_TYP_CPUEXC (4): CPU exception (start, end)

```
typedef struct    t_rslog_cpuexc
{
    DT_EXCNO excno      : CPU exception code (ID_ALL available)
}   T_RSLOG_CPUEXC;
```

LOG_TYP_TSKEXC (5): Task exception (start, end)

```
typedef struct    t_rslog_tskexc
{
    DT_ID tskid         : Task ID (ID_ALL available)
}   T_RSLOG_TSKEXC;
```

LOG_TYP_TSKSTAT (6): Task state

```
typedef struct    t_rslog_tskstat
{
    DT_ID tskid         : Task ID (ID_ALL available)
}   T_RSLOG_TSKSTAT;
```

Note: The tasks state is regarded as the execution-ready state without distinction between executing state and execution-ready state.

LOG_TYP_DISPATCH (7): Task dispatch start

typedef struct t_rslog_dispatch

{

DT_ID tskid : Task ID (***ID_ALL*** available)

} **T_RSLOG_DISPATCH;**

LOG_TYP_SVC (8): System call (start, end)

typedef struct t_rslog_svc

{

DT_FN svcfn : Function code

DT_ID objid : Targeted object ID (ignored when the SVC does not have a target; ***ID_ALL*** available)

DT_ID tskid : Task ID (***ID_ALL*** available)

BITMASK param : Parameter to be targeted (***ID_ALL*** available)

} **T_RSLOG_SVC;**

Note: When ***ID_NONTSKCTX(=0)*** is specified for tskid, the nontask context will be targeted. ***ID_ALL*** means both the task context and nontask context. ***param*** specifies the parameters to be logged and logs the parameters that correspond to the bit positions at which the value is 1. When ***LOG_ENTER*** is specified, the leftmost argument corresponds to the first parameter. When ***LOG_LEAVE*** is specified, the return value is the first parameter, and the second and subsequent parameters are the arguments.

LOG_TYP_COMMENT (9): Comment

typedef struct t_rslog_comment

{

UINT length : Comment character string length

} **T_RSLOG_COMMENT;**

Supplementary explanation

Some logs are output in a specified order. The following logs are output in a predetermined order. The logs on the left-hand side are displayed first.

- ***LOG_TYP_DISPATCH|LOG_LEAVE, LOG_TYP_TSKEXC***
- ***LOG_TYP_DISPATCH|LOG_ENTER, LOG_TYP_TSKSTAT***

LOG_TYP_SVC|LOG_LEAVE does not detect the end of the following service calls:

- ***ext_tsk***
- ***exd_tsk***

LOG_TYP_TSKEXC|LOG_LEAVE will not be detected in the following situation:

- Non-local jump (longjmp) from task exception handler*

LOG_TYP_TSKSTAT does not distinguish between the executable state (READY) and executing state (RUNNING). It recognizes both states as a READY state. The READY state and RUNNING state are acquired by ***LOG_TYP_DISPATCH***.

*. Refers to process that uses longjmp, setjmp, etc., to forcibly pass process to specific function irrespective of function execution order

When the function is successfully executed in situations where the automatic number assignment flag ***FLG_AUTONUMBERING*** is specified, the function returns a value of 1 or greater (ID value), which is assigned to a log item. This is also true even when the automatic assignment flag is not specified.

Flag

FLG_AUTONUMBERING (40000000$_H$): ID automatic assignment

Automatically assigns an ID. If an argument is specified as the ID, it is ignored by the function. When the function is successfully executed, it returns the automatically assigned ID.

Keys

RIF 04$_H$

.RIF_SET_LOG 0E$_H$

.FLG_AUTONUMBERING 04$_H$ [1]

The ***FLG_AUTONUMBERING*** flag is available.

.OPT_BUFFUL_STOP 10$_H$ [1]

The ***OPT_BUFFUL_STOP*** option is available.

.OPT_BUFFUL_FORCEEXEC 11$_H$ [1]

The ***OPT_BUFFUL_FORCEEXEC*** option is available.

Errors

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_ID (-146)

The specified object ID was invalid.

E_NOID (-162)

Count of IDs for automatic assignment was insufficient.

E_OBJ (-169)

The targeted object on the target was inoperative.

ET_ID (-18)

The specified kernel object ID was invalid.

E_PAR (-145)

A parameter value was invalid.

5.6.2 Delete of trace log

rif_del_log Delete of trace log [LOG]◯

ER rif_del_log (ID logid, FLAG flags)

ID logid
ID of the trace log to be deleted

FLAG flags
Flags

This function deletes the trace log setting specified by ***rif_set_log***. It deletes all the log setting when ***logid*** is set to ***ID_ALL (=-1)***.

Supplementary explanation

Trace logs validated by ***rif_sta_log*** cannot be deleted.

Keys

RIF	04_H
.RIF_DEL_LOG	$0F_H$

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_ID (-146)
The specified object ID was invalid.

E_OBJ (-169)
The targeted object on the target was inoperative.

E_EXCLUSIVE (-226)
Another request has already been issued. The function could not receive a new request until execution of the previous request ends.

5.6.3 Request of trace log function start

rif_sta_log Request of trace log function start [LOG]◯

ER rif_sta_log (ID logid, FLAG flags)

ID logid
ID number assigned to the trace log function to be started

FLAG flags
Flags

This function starts executing the trace log function in accordance with the setting defined by ***rif_set_log***. When ***ID_ALL (=-1)*** is specified, all the specified logs are validated.

Supplementary explanation

Getting trace log takes place in non-blocking mode. You should therefore note that the end of this function does not mean the end of getting trace log. In reality, getting log operation is performed during a program run resumption after the call of this function.

Supplementary explanation

Even when the trace log function is exercised two or more times for the log setting for the sigle ID, the function returns ***E_OK***. The all specified log settings are stopped by a single stop procedure even if the trace log function is exercised two or more times.

Keys

RIF	04_H
.RIF_STA_LOG	10_H

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_ID (-146)
The specified object ID was invalid.

E_OBJ (-169)
The targeted object on the target was inoperative.

5.6.4 Request of trace log stop

rif_stp_log Request of trace log stop [LOG]◯

ER rif_stp_log (ID logid, FLAG flags)

ID logid
ID of the trace log to be stopped

FLAG flags
Flags

This function stops the specified trace logging operation. All logs are targeted when ***logid*** is set to ***ID_ALL (=-1)***.

Supplementary explanation

This function aims at clearing the break points or other settings for get trace log. It does not cancel the trace log settings.
Storage of the data specified by ***rif_set_log*** must be assured before and after this function.

Supplementary explanation

Even when this function is executed for an already terminated log setting, it returns ***E_OK***.

Keys

RIF	04_H
.RIF_STP_LOG	11_H

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_ID (-146)
The specified object ID was invalid.

E_OBJ (-169)
The targeted object on the target was inoperative.

5.6.5 Get of trace log

rif_get_log Get of trace log [LOG]◯

ER rif_get_log (T_RGLOG * ppk_rglog, FLAG flags)

T_RGLOG * ppk_rglog
Pointer to the region that stores the standard trace log information

FLAG flags
Flags

rif_get_log requires to get logs stored in the RIM. The RIM issues ***tif_get_log*** as needed to get primitive log information and remakes this information into a return value. When a highly functional debugging tool is used, the RIM may use the data got by ***tif_get_log*** as the return value without remaking.
When ***rif_get_log*** gets one log, it moves the read position to the next log. To get all logs, the debugging tool calls this function two or more times. When no log remains, ***rif_get_log*** returns the ***E_OBJ*** error.

The contents of ***T_RGLOG*** are indicated below:

```
typedef struct t_rglog
{
    UINT logtype        : Log type
    LOGTIM logtim       : Time stamp
    BITMASK valid       : Validation flag
    UINT bufsz          : Buffer region (buf) size (in bytes)
    char buf[]          : Buffer region to store information (detailed later)
} T_RGLOG;
```

T_RGLOG is required to have a sufficient region for storing "the type and the data to be stored in a buffer" (mentioned later) in addition to essential items, "***logtype***", "***logtim***", and "***valid***".

The generated log type enters the ***T_RGLOG::type*** position. ***T_RSLOG::buf*** stores the information that corresponds to the specified type. For a log type that permits the designation of startup and end, the specifiers "***LOG_ENTER***" and "***LOG_LEAVE***" are set to "***T_RGLOG::type***". The log types and the information to be stored are detailed below. Note that the information logged at startup is different from information logged at termination only when ***LOG_TYP_DISPATCH*** is used.

LOG_TYP_INTERRUPT (1): Interrupt handler

```
typedef struct t_rglog_interrupt
{
    DT_INHNO inhno      : Interrupt handler number
} T_RGLOG_INTERRUPT;
```

LOG_TYP_ISR (2): Interrupt service routine

```
typedef struct  t_rglog_isr
{
    DT_ID isrid         : Interrupt service routine ID
    DT_INTNO inhno      : Interrupt handler number
}   T_RGLOG_ISR;
```

LOG_TYP_TIMERHDR (3): Timer event handler

```
typedef struct  t_rglog_timerhdr
{
    UINT type           : Timer type
            (stores the constant "OBJ_xxx" that is used for rif_ref_obj::objtype).
    DT_ID hdrid         : Timer event handler ID
    DT_VP_INT exinf     : Extension information
}   T_RGLOG_TIMERHDR;
```

LOG_TYP_CPUEXC (4): CPU exception

```
typedef struct  t_rglog_cpuexc
{
    DT_ID tskid         : ID of a targeted task
}   T_RGLOG_CPUEXC;
```

If the cause of an CPU exception is outside the task, ***tskid*** is 0.

LOG_TYP_TSKEXC (5): Task exception

```
typedef struct  t_rglog_tskexc
{
    DT_ID tskid         : ID of a targeted task
}   T_RGLOG_TSKEXC;
```

LOG_TYP_TSKSTAT (6): Task state

```
typedef struct  t_rglog_tskstat
{
    DT_ID tskid         : Task ID
    DT_STAT tskstat     : Status of task at transition destination
    DT_STAT tskwait     : Wait state
    DT_ID wobjid        : ID of waiting object
}   T_RGLOG_TSKSTAT;
```

LOG_TYP_DISPATCH|LOG_ENTER (7): Task dispatch start

```
typedef struct  t_rglog_dispatch_enter
{
    DT_ID tskid         : ID of executed task
    UINT disptype       : Dispatch type
}   T_RGLOG_DISPATCH_ENTER;
```

The dispatch types are as follows:

DSP_NORMAL (0)
Dispatch from task context

DSP_NONTSKCTX (1)
Dispatch from interrupt process or CPU exception

LOG_TYP_DISPATCH|LOG_LEAVE (135): Task dispatch end

typedef struct t_rglog_dispatch_leane

{

DT_ID tskid : ID of task about to be executed

} **T_RGLOG_DISPATCH_LEAVE;**

LOG_TYP_SVC (8): Service call

typedef struct t_rglog_svc

{

DT_FN fncno : Function code

UINT prmcnt : Parameter count

DT_VP_INT prmary []:Parameters

} **T_RGLOG_SVC;**

LOG_TYP_COMMENT (9): Comment (log consisting of a character string only)

typedef structt_rglog_comment B

{

UINT length : Character string length

char strtext [] : Character string (NULL-terminated string) - May be broken

} **T_RGLOG_COMMENT;**

Before the call of this function, the debugging tool must store the size (in bytes) of the buffer region specified by ***T_RGLOG::buf*** in the ***T_RGLOG*** structure member ***bufsz***.

Supplementary explanation

As regards a log (***LOG_TYP_COMMENT::strtext***) that is marked "May be broken", a transfer is made to the extent possible even if the buffer region is insufficient. However, the minimum required meaningful unit must be assured even if the transfer has to be broken before completion due to buffer region insufficiency.* The enable/disable bit map (explained later) for such a broken parameter remains enabled and the return value is ***E_NOMEM*** error.

T_RSLOG::valid indicates a valid field of items to be stored in ***T_RSLOG::buf***. The items are sequentially mapped into bit map in order. As regards ***LOG_TYP_SVC_ENT***, for instance, ***fncno***, ***prmcnt***, and ***Prmary [n]*** are assigned to the least significant bit, the second least bit, and the third+n least bit, respectively. | is stored in the enabled item, while 0 is stored in the disabled item. However, ***T_RGLOG_COMMENT::strtext*** is handled in the unit of the entire character string and not in the character unit. Bits irrelevant to items are all 0.

T_RGLOG_SVC::prmcnt, got by a log type-service call start (***LOG_TYP_SVC|LOG_ENTER***) stores the maximum number of obtained parameters. As regards the normally got portion of ***T_RGLOG_SVC::prmary***, the leftmost argument is handled as the first one and the bit corresponding to ***T_RGLOG::valid*** is 1. If, for example, parameter is got partially, note that the number of function arguments does not match ***T_RGLOG_ SVC::prmcnt***.
T_RGLOG_SVC::prmcnt, got by a log type-service call end (***LOG_TYP_SVC|LOG_LEAVE***) stores the maximum number of got parameters, including the return value. For the normally got portion of ***T_RGLOG_SVC::prmary***, the return value and function leftmost argument are handled as the first and second ones, respectively, and the bit corresponding to ***T_RGLOG::valid*** is 1.

*. Strtext is a NULL-terminated character string. To assure that a NULL-terminated character string is meaningful, it is necessary to add a terminal symbol to break when the remaining buffer size is 1 byte.

Some logs are output in a specified order. The following logs are output in a predetermined order. The logs on the left side are displayed first.

- ***LOG_TYP_DISPATCH|LOG_LEAVE, LOG_TYP_TSKEXC***
- ***LOG_TYP_DISPATCH|LOG_ENTER, LOG_TYP_TSKSTAT***

LOG_TYP_SVC|LOG_LEAVE does not detect termination of the following functions:

- ***ext_tsk***
- ***exd_tsk***

LOG_TYP_TSKEXC|LOG_LEAVE will not be detected in the following situation:

- Non-local jump (longjmp) from task exception handler

LOG_TYP_TSKSTAT does not distinguish between the execution-ready state (READY) and executing state (RUNNING). It recognizes both states as a READY state. The READY state and RUNNING state are got by ***LOG_TYP_DISPATCH***.

Option

OPT_PEEK (1)

Gets a trace log without deleting it from the spool.

Keys

RIF	04_H
.RIF_GET_LOG	12_H
.OPT_PEEK	10_H [1]
The ***OPT_PEEK*** option is available.	
.STRUCT_SVC	11_H [1]
Uses a dedicated structure for the start/end of ***LOG_TYP_SVC***.	

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_OBJ (-169)

The targeted object on the target was inoperative.

E_PAR (-145)

A parameter value was invalid.

5.6.6 Reconfigur of trace log mechanism

rif_cfg_log Reconfigur of trace log mechanism [LOG]◯

ER rif_cfg_log (T_RCLOG * pk_rclog, FLAG flags)

T_RCLOG * pk_rclog
Pointer to the packet that stores trace log configuration information

FLAG flags
Flags

This function changes the trace log mechanism configuration.

The structure "***T_RCLOG***" which stores trace log configuration information is detailed below:

```
typedef struct    t_rclog
{
    UINT type          : Trace log configuration type
    DT_VP bufptr       : Pointer to the trace log buffer
    DT_SIZE bufsz      : Trace log buffer size
}   T_RCLOG;
```

T_RCLOG::type stores the trace log mechanism setup information. The buffer getting method and log buffer full state operation can be specified as the setup information. The following values can be used as setup information (The ***E_NOSPT*** error occurs if an unsupported method is selected).

Buffer getting method

- ***LOG_HARDWARE (0)***
  Gets buffer with TIF-based hardware log mechanism
- ***LOG_SOFTWARE (1)***
  Gets buffer with software-based log mechanism executed by RIM alone

Operation when buffer full

- ***LOG_BUFFUL_STOP (0)***
  Stops getting trace when buffer full
- ***LOG_BUFFUL_FORCEEXEC (4)***
  Continues getting buffer by discarding oldest information when buffer full

T_RCLOG::bufptr and ***T_RCLOG::bufsz*** set the guide for RTOS history storage region creation by the RIM and debugging tool. When getting log is intended, the specified region is used as the log buffer.

Supplementary explanation

If a log mechanism is used without these setting mentioned above, the operation follows implement definition.

If ***LOG_HARDWARE*** is specified and the RIM checks the key code ***DEBUGGER.LOG.NUM*** and concludes that it has no hardware log mechanism, the function must return ***E_NOSPT***.
If the log buffer region overlaps with a program region (data or code region) or a nonexistent memory space is specified, the RIM returns the ***ET_MACV*** error.

Keys

RIF	04_H
.RIF_CFG_LOG	13_H

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_PAR (-145)
A parameter value was invalid.

ET_MACV (-26)
An invalid memory region on the target was accessed.

5.7 Other RTOS-related Information

5.7.1 Get of kernel configuration

rif_ref_cfg Get of kernel configuration [R]◯

ER rif_ref_cfg

(T_INFO * p_information, UINT packets, FLAG flags)

T_INFO * p_information

Pointer to the beginning of a get information structure array

UINT packets

Length of the get information structure array indicated by p_information

FLAG flags

Flags

This function gets a kernel configuration.*

To get information, this function uses the function for getting information ***T_INFO*** and key code. For details, see ***Section 3.6***. ***rif_ref_cfg*** can get key codes under the ***INF_CFG*** key.

Keys

CFG 7_H

.CPUEXCEPTION 17_H

.MIN 1_H [W]

Minimum value of the internal exception causes that the kernel uses

.MAX 2_H [W]

Maximum value of the internal exception causes that the kernel uses

.NUM 3_H [W]

Count of internal exception causes that the kernel uses

.SYSTIM 20_H

.TICK_D 1_H [W]

Denominator when the timer resolution is expressed in milliseconds (ms)

.TICK_N 2_H [W]

Numerator when the timer resolution is expressed in milliseconds (ms)

.UNIT_D 3_H [W]

Denominator when the timer unit is expressed in milliseconds (ms)

.UNIT_N 4_H [W]

Numerator when the timer unit is expressed in milliseconds (ms)

*. In the ITRON Debugging Interface Specification, the information changed by kernel reconfiguration is defined as the kernel configuration. You should remember this definition if you have difficulty selecting ***dbg_ref_rim*** (explained later) or ***rif_ref_cfg*** function as a new information item to add in.

.LOGTIM 21_H

.TICK_D 1_H [W]

Denominator when the log time resolution is expressed in milliseconds (ms)

.TICK_N 2_H [W]

Numerator when the log time resolution is expressed in milliseconds (ms)

.UNIT_D 3_H [W]

Denominator when the log time unit is expressed in milliseconds (ms)

.UNIT_N 4_H [W]

Numerator when the log time unit is expressed in milliseconds (ms)

.INTERRUPT 22_H

.MIN 1_H [W]

Minimum value of the external interrupt factors that the kernel uses

.MAX 2_H [W]

Maximum value of the external interrupt factors that the kernel uses

.NUM 3_H [W]

Count of external interrupt factors that the kernel uses

.ISR 25_H

.MIN 1_H [W]

Minimum ISR number offered by kernel

.MAX 2_H [W]

Maximum ISR number offered by kernel

.NUM 3_H [W]

Number of ISRs offered by kernel

.MAKER 23_H [W]

Manufacturer code

.PRIORITY 24_H

.MIN 1_H [W]

Minimum value of the priority levels available to the kernel

.MAX 2_H [W]

Maximum value of the priority levels available to the kernel

.OBJ_SEMAPHORE 80_H

.MIN 1_H [W]

Minimum value of assignable IDs

.MAX 2_H [W]

Maximum value of assignable IDs

.OBJ_EVENTFLAG 81_H

.MIN 1_H [W]

Minimum value of assignable IDs

.MAX 2_H [W]

Maximum value of assignable IDs

.OBJ_DATAQUEUE 82_H

.MIN 1_H [W]

Minimum value of assignable IDs

.MAX	2_H [W]
Maximum value of assignable IDs	
.OBJ_MAILBOX	83_H
.MIN	1_H [W]
Minimum value of assignable IDs	
.MAX	2_H [W]
Maximum value of assignable IDs	
.OBJ_MUTEX	84_H
.MIN	1_H [W]
Minimum value of assignable IDs	
.MAX	2_H [W]
Maximum value of assignable IDs	
.OBJ_MESSAGEBUFFER	85_H
.MIN	1_H [W]
Minimum value of assignable IDs	
.MAX	2_H [W]
Maximum value of assignable IDs	
.OBJ_RENDEZVOUSPORT	86_H
.MIN	1_H [W]
Minimum value of assignable IDs	
.MAX	2_H [W]
Maximum value of assignable IDs	
.OBJ_RENDEZVOUS	87_H
.MIN	1_H [W]
Minimum value of assignable IDs	
.MAX	2_H [W]
Maximum value of assignable IDs	
.OBJ_FMEMPOOL	88_H
.MIN	1_H [W]
Minimum value of assignable IDs	
.MAX	2_H [W]
Maximum value of assignable IDs	
.OBJ_VMEMPOOL	89_H
.MIN	1_H [W]
Minimum value of assignable IDs	
.MAX	2_H [W]
Maximum value of assignable IDs	
.OBJ_TASK	$8A_H$
.MIN	1_H [W]
Minimum value of assignable IDs	
.MAX	2_H [W]
Maximum value of assignable IDs	
.OBJ_CYCLICHANDLER	$8D_H$
.MIN	1_H [W]
Minimum value of assignable IDs	

.MAX 2_H [W]
Maximum value of assignable IDs
.OBJ_ALARMHANDLER $8E_H$
.MIN 1_H [W]
Minimum value of assignable IDs
.MAX 2_H [W]
Maximum value of assignable IDs
.PRVER $A0_H$ [S]
Version number of the kernel
.SPVER $A1_H$ [S]
ITRON Specification version number

If the above ***.MAX*** key code is 0 and ***.MIN*** key code is 0, it means that the associated function is not supported.
.MIN is a key code of getting information to indicate the lower limit for an object ID or other item used by the system. If the employed debugging tool does not display such system objects, their values can be replaced by the object ID minimum value (1) available to the user.*

Supplementary explanation

If a nonexistent key code of getting information is specified or if this function is called together with a buffer having a size of "0", the function returns ***E_PAR*** (parameter error).

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_PAR (-145)
A parameter value was invalid.

E_OBJ (-169)
The targeted objuect on the target was inoperative.

*. According to the ITRON Specification, system objects customarily have a negative object ID. Meanwhile, user tasks can only use a positive object ID. Therefore, if system objects are not displayed, the ***INF_MIN*** value is not so important.

This page is intentional blank.

6.Target Access Interface

6.1 Memory Operations

6.1.1 Allocate memory (on host)

tif_alc_mbh Allocate memory (on host) [R] □

ER tif_alc_mbh (VP * p_blk, UINT blksz, FLAG flags)

VP * p_blk
Pointer to the region that stores the pointer to the beginning of an allocated block

UINT blksz
Block size

FLAG flags
Flags

To create a work region for a memory read, the debugging tool provides the RIM with a means of memory allocation. When the C library is available to the host, the debugging tool only call the ***malloc*** function. However, the RIM must not assume that the C library is implemented in the host on which the debugging tool runs. Therefore, the RIM must not internally call the ***malloc*** function.

Keys

TIF	05_H
.TIF_ALC_MBH	01_H

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_PAR (-145)
A parameter value was invalid.

6.1.2 Allocate memory (on target)

tif_alc_mbt Allocate memory (on target) [E] □

ER tif_alc_mbt (DT_VP * p_blk, DT_SIZE blksz, FLAG flags)

DT_VP * p_blk
Region for storing the pointer to the beginning of an allocated memory region

DT_SIZE blksz
Size (in bytes) of the memory region to be allocated

FLAG flags
Flags

When the debugging tool can manage the memory on the target,* this function is executed to allocate the memory on the target for the purpose of performing an operation, for instance, "to let the RIM write a glue routine** on the target".

If dynamic memory allocation is unable, there is no need to support this function. In such an instance, the RIM must allocate a region itself.

Keys

TIF 05_H

.TIF_ALC_MBT 02_H [1]

Supports this function.

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_PAR (-145)
A parameter value was invalid.

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

ET_NOMEM (-33)
The request could not be executed due to insufficient memory on the target.

---

*: The assumed situation is such that a function for emulating a memory within a space where no physical memory exists, which some general-purpose debuggers have, is implemented.

**: For an SVC issue, the RIM may generate a temporary program for calling a targeted SVC. Such a program is called a glue routine.

6.1.3 Free memory (on host)

tif_fre_mbh Free memory (on host) [R] □

ER tif_fre_mbh (VP blk, FLAG flags)

VP blk

Pointer to the beginning of the memory block to be freed

FLAG flags

Flags

This function frees a memory that is allocated on a host. On most of the hosts, it is assumed that this function corresponds to the C library's "free" function.

Supplementary explanation

When "***blk***" is contained in a closed section between the block start position and the "block length - 1" position, this function normally frees memory.

Keys

TIF 05_H

.TIF_FRE_MBH 03_H

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_PAR (-145)

A parameter value was invalid.

E_OBJ (-169)

The targeted object on the target was inoperative.

6.1.4 Free memory (on target)

tif_fre_mbt Free memory (on target) [E] □

ER tif_fre_mbt (DT_VP blk, FLAG flags)

DT_VP blk
Pointer to the beginning of the memory block to be freed

FLAG flags
Flags

This function frees a memory that is allocated to the target.

Supplementary explanation

When ***blk*** is contained in a closed section between the block start position and the "block length - 1" position, this function normally frees memory.

Keys

TIF 05_H

.TIF_FRE_MBT 04_H [1]

Supports this function.

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_PAR (-145)
A parameter value was invalid.

ET_NOMEM (-33)
The request could not be executed due to insufficient memory on the target.

ET_OBJ (-41)
The targeted object on the target was inoperative.

6.1.5 Read memory (memory block)

tif_get_mem Read memory [R] □

ER tif_get_mem

(VP p_result, DT_VP memadr, DT_SIZE memsz, FLAG flags)

VP p_result
: Pointer to the beginning of the storage region

DT_VP memadr
: Read starting address

DT_SIZE memsz
: Length of the data to be read (in bytes)

FLAG flags
: Flags

tif_get_mem reads the data in the target memory that has a length of memsz and begins with memadr. Before a function call, the RIM creates a buffer with a length greater than memsz, and sets it in ***p_result***. The debugging tool stores the read memory data in ***p_result***-specified region as a byte string.

Extension

The following extended functionalities are defined:

Flags

FLG_NOCONSISTENCE (10000000_H): Nonconsistency flag

When this flag is specified, the data that is got need not be consistent (e.g., the task is still in the waiting state although there is no factor of the task wait).

FLG_NOSYSTEMSTOP (20000000_H): An explicit system halt is not permitted.

When this flag is specified, ***tif_brk_tgt*** must not be used within the function to halt the system. If this flag is not supported, the ***E_NOSPT*** error occurs.

Extension

Supplementary explanation

The read access size is determined by the debugging tool.

Keys

TIF	05_H
.TIF_GET_MEM	05_H
.FLG_NOCONSISTENCE	01_H [1]
Supports the ***FLG_NOCONSISTENCE*** flag.	
.FLG_NOSYSTEMSTOP	02_H [1]
Supports the ***FLG_NOSYSTEMSTOP*** flag.	

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

ET_MACV (-26)

An invalid memory region on the target was accessed.

E_PAR (-145)

A parameter value was invalid.

E_CONSIST (-225)

Consistency was not assured (however, it is not handled as an error if ***FLG_NOCONSISTENCE*** is set).

6.1.6 Read memory (block set)

tif_get_bls Read memory by block set 0[R] □

ER tif_get_bls

(VP p_result , T_BLKSET * blkset, FLAG flags)

VP p_result
Pointer to the region that stores the results of a read

T_BLKSET * blkset
Structure specifying the read location

FLAG flags
Flags

This function reads the contents of the target memory by a block set. The block set retains positions consisting of a memory address and byte length within a target memory space. ***tif_get_bls*** can read the target memory space indicated by the block set in batch processing.

The ***T_BLKSET*** structure is an aggregate that stores memory blocks, which are read units.

```
typedef struct t_blkset
{
    UINT blkcnt             : Count of blocks
    T_MEMBLK blkary []:  Block array
} T_BLKSET;

typedef struct t_memblk
{
    DT_VP blkptr            : Pointer to store the memory block data
    DT_SIZE blksz           : Byte count of memory block data
} T_MEMBLK;
```

The read contents of the target memory are stored sequentially in the ***p_result***-defined memory space in the order specified by the block set. If the memory is read with the following block set, the read data is stored as indicated in ***Table 22***.

T_BLKSET pk_blkset = { 3, { { 0x1000, 128} , { 0x2000, 1} , { 0x3000, 64} } }

Table 22: Relation Between Block Set and Data Arrangement

Starting offset	0	128	129
Data length	128 bytes	1 byte	64 bytes
Data address	0x1000 to 0x1080	0x2000	0x3000 to 0x3040

Supplementary explanation

When this function returns ***E_OK***, it assures that the required block set is normally read in accordance a with required conditions. If any one of requested blocks is unsuccessfully read, the ***E_MACV*** error occurs. Furthermore, if ***FLG_NONCONSISTENCE*** (described later) is not specified and consistency cannot be assured for all regions instead of on an individual memory block basis, the ***E_CONSIST*** error occurs, unlike when ***tif_get_mem*** is executed continuously.

The read access size is determined by the debugging tool.

Before a function call, the RIM must create a buffer that is large enough to store the result, and store it in ***p_result***.

Extension

The following operation can be executed with extended functions:

Flags

FLG_NOCONSISTENCE (10000000_H): Nonconsistency flag

When this flag is specified, the data that is got need not be consistent (e.g., the task is still in the wait state although there is no cause of the task's wait).

FLG_NOSYSTEMSTOP (20000000_H): An explicit system halt is not permitted.

When this flag is specified, ***tif_brk_tgt*** must not be used within the function to halt the system. If this flag is not supported, the ***E_NOSPT*** error occurs.

Extension

Keys

TIF	05_H
.TIF_GET_BLS	06_H
.FLG_NOCONSISTENCE	01_H [1]
Supports the ***FLG_NOCONSISTENCE*** flag.	
.FLG_NOSYSTEMSTOP	02_H [1]
Supports the ***FLG_NOSYSTEMSTOP*** flag.	

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

ET_MACV (-26)

An invalid memory region on the target was accessed.

E_PAR (-145)

A parameter value was invalid.

E_CONSIST (-225)

Consistency was not assured (however, it is not handled as an error if ***FLG_NOCONSISTENCE*** is set).

6.1.7 Write memory (memory block)

tif_set_mem Write memory by memory block [R] □

ER tif_set_mem

(VP storage , DT_VP memadr, DT_SIZE memsz, FLAG flags)

VP storage
Pointer to the beginning of the region that retains the data to be written

DT_VP memadr
Address on the target where data is written

DT_SIZE memsz
Length of data to be written (in bytes)

FLAG flags
Flags

This function writes to the target memory by memory block in accordance with the stored contents in ***storage***. For details, see ***Section 6.1.5***.

Extension

The following operation can be executed with extended function:

Flags

FLG_NOCONSISTENCE (10000000_H): Nonconsistency flag

When this flag is specified, the data that is got need not be consistent (e.g., the task is still in the wait state although there is no factor of the task wait).

FLG_NOSYSTEMSTOP (20000000_H): An explicit system halt is not permitted.

When this flag is specified, ***tif_brk_tgt*** must not be used within the function to halt the system. If this flag is not supported, the ***E_NOSPT*** error occurs.

Extension

Supplementary explanation

The write access size is determined by the debugging tool.

Keys

TIF 05_H

.TIF_SET_MEM 07_H

.FLG_NOCONSISTENCE 01_H [1]

Supports the ***FLG_NOCONSISTENCE*** flag.

.FLG_NOSYSTEMSTOP 02_H [1]

Supports the ***FLG_NOSYSTEMSTOP*** flag.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

ET_MACV (-26)

An invalid memory region on the target was accessed.

E_PAR (-145)

A parameter value was invalid.

E_CONSIST (-225)

Consistency was not assured (however, it is not handled as an error if ***FLG_NOCONSISTENCE*** is set).

6.1.8 Write memory (block set)

tif_set_bls Write memory by block set [R] □

ER tif_set_bls (VP storage, T_BLKSET * blkset, FLAG flags)

VP storage
: Pointer to the region that stores the data to be written

T_BLKSET * blkset
: Pointer to the structure that indicates the write destination

FLAG flags
: Flags

This function writes data into the memory on the target by block set. For details, see ***Section 6.1.6***.
This function and the ***tif_get_bls*** function are opposite. If the following operation is performed, it must be assured that the memory data remains unchanged (except for spaces with a real-time capability or dynamically changing contents).

Program source

```
{
        //Writing the read data as it is
if(get_bls(buffer,blkset,0) == E_OK)
        set_bls(buffer,blkset,0);
}
```

Program source

Supplementary explanation

If any of the specified block sets fails, the function ends with ***E_MACV***. In this instance, ***tif_set_bls*** does not assure or report the extent to which ***blkset*** is written.

The write access size is determined by the debugging tool.

Extension

The following operation can be executed as extended functions:

Flags

FLG_NOCONSISTENCE (10000000_H): Non consistency flag

When this flag is specified, the data that is got need not be consistent (e.g., the task is still in the wait state although there is no factor of the task's wait).

FLG_NOSYSTEMSTOP (20000000_H): An explicit system halt is not permitted.

When this flag is specified, ***tif_brk_tgt*** must not be used within the function to halt the system. If this flag is not supported, the ***E_NOSPT*** error occurs.

Extension

Keys

TIF 05_H

.TIF_SET_BLS 08_H

.FLG_NOCONSISTENCE 01_H [1]

Supports the ***FLG_NOCONSISTENCE*** flag.

.FLG_NOSYSTEMSTOP 02_H [1]

Supports the ***FLG_NOSYSTEMSTOP*** flag.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

ET_MACV (-26)

An invalid memory region on the target was accessed.

E_PAR (-145)

A parameter value was invalid.

E_CONSIST (-225)

Consistency was not assured (however, it is not handled as an error if ***FLG_NOCONSISTENCE*** is set).

6.1.9 Set of change report

tif_set_pol Set of memory data change report [E] □

ER_ID tif_set_pol

(ID polid, DT_VP adr, DT_INT value, UINT length, FLAG flags)

ID polid
Polling ID

DT_VP adr
Memory address where a change is detected

DT_INT value
Value to be compared

UINT length
Byte length of a targeted memory block (1, 2, 4, or 8)

FLAG flags
Flags

(Return value) ID polid
Any value identifying this polling setting

This function sets a polling to be performed by a debugging tool. The debugging tool performs a polling to monitor data at a specific memory address. If there is any change in the data, the debugging tool uses a callback function to report it. However, this operation may not keep up with rapid data changes.
If ***OPT_CMPVALUE*** is specified, the debugging tool compares ***value*** with the memory data. If they differ, the debugging tool calls the ***tif_rep_pol***. If ***OPT_CMPVALUE*** is not specified, the debugging tool saves the memory data at the time of ***tif_set_pol*** setting, and compares it with the current data. If they differ, the debugging tool calls the ***tif_rep_pol*** function.

Supplementary explanation

Unlike an access break, ***tif_set_pol*** does not report unless the contents change.

A memory data update and a ***tif_rep_pol*** function call are not concurrent.

When the function is executed successfully in situations where the automatic number assignment flag ***FLG_AUTONUMBERING*** is specified, the function returns a value of 1 or greater (ID value) that is assigned to a setup item. This is also true even when the automatic assignment flag is not specified.

Flags

OPT_CMPVALUE (2)

Sets a value to be compared.

FLG_AUTONUMBERING (40000000_H): ID automatic assignment

Automatically assigns an ID. If a specified argument is same as the ID value, it is ignored by the function. When the function is successfully executed, it returns the automatically assigned ID.

Keys

TIF 05_H

.TIF_SET_POL 09_H [1]

Supports this function.

.FLG_AUTONUMBERING 04_H [1]

Supports the ***FLG_AUTONUMBERING*** flag.

.OPT_CMPVALUE 10_H [1]

Supports the ***OPT_CMPVALUE*** option.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

ET_MACV (-26)

An invalid memory region on the target was accessed.

E_PAR (-145)

A parameter value was invalid.

E_ID (-146)

The specified object ID was invalid.

E_NOID (-162)

Count of IDs for automatic assignment was insufficient.

E_OBJ (-169)

The targeted obuject on the target was inoperative

6.1.10 Delete of change report setting

tif_del_pol Delete of change report setting [E] □

ER tif_del_pol (ID polid, FLAG flags)

ID polid
ID to be deleted

FLAG flags
Flags

This function deletes a change report (polling) that is set by ***tif_set_pol***. When ***ID_ALL*** (=-1) is specified, all the change reports are deleted.

Supplementary explanation

This function can also be called from the report function ***tif_rep_pol***.

Keys

TIF 05_H

.TIF_DEL_POL $0A_H$ [1]

Supports this function.

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_ID (-146)
The specified object ID was invalid.

E_OBJ (-169)
The targeted object on the target was inoperative

6.1.11 Change report

tif_rep_pol Report of memory data change [E:callback]◯

void tif_rep_pol (ID polid, DT_INT value, FLAG flags)

ID polid
: Polling ID

DT_INT value
: Memory data value after a change

FLAG flags
: Flags

When a debugging tool detects a memory data change with a polling process that is performed by ***tif_set_pol***, this function reports the change.

Keys

TIF	05_H
.TIF_REP_POL	$0B_H$

Errors

This function does not have a return value.

6.2 Register Operations

6.2.1 Read of register value

tif_get_reg Read of register value [R] □

ER tif_get_reg (VP r_result, BITMASK_8 * p_valid, FLAG flags)

VP r_result
: Pointer to the beginning of the region that stores a register value

BITMASK_8 * p_valid
: Pointer to validation flag about register table items
(NULL: Targets entire context)

FLAG flags
: Flags

This function gets the register value of the current target in accordance with the contents of the register set description table.

The variable ***p_result*** is the pointer to the buffer for storing the register value that will be got by execution of this function. Before execution of this function, the debugging tool must create a region that is large enough to store the register value. The key code of getting information ***RIF.RIF_GET_RDT.REGISTER.SIZE*** should be used for the size of the buffer. The buffer size can also be calculated from the register table got by the function ***rif_get_rdt***. In such a case, a region large enough to store all the registers indicated by the register table must be furnished.

p_valid specifies whether the registers should be enabled or disabled. When it is given as a function argument, disabled registers will not be got. Furthermore, this function stores the got results of targeted registers in ***p_valid***. When all the targeted registers are got normally, this function returns ***ET_SYS*** or other errors depending on the situation. The information stored in regions related to the registers which could not be got is implement-dependent. Even if the enabled/disabled information is given in excess of the number of registers (***T_GRDT::regcnt***), excessive registers will not be got.
If ***NULL*** is specified for ***p_valid***, all registers are targeted for getting so the result details will not be stored.

Extension

The following operation can be executed as extended function:

Flags

FLG_NOCONSISTENCE (10000000_H): Nonconsistency flag

When this flag is specified, the data that is get need not be consistent (e.g., the task is still in the wait state although there is no factor of the task's wait).

FLG_NOSYSTEMSTOP (20000000$_H$): An explicit system halt is not permitted.

When this flag is specified, the ***tif_brk_tgt*** must not be used in the function to the function to halt the system. If this flag is not supported, the ***E_NOSPT*** error occurs.

Extension

Supplementary explanation

The read register value is stored in accordance with the endian of the target.

If a non-existent register is selected as the read operation target, the function returns the ***E_PAR*** error.

Keys

TIF	05$_H$
.TIF_GET_REG	0C$_H$
.FLG_NOCONSISTENCE	01$_H$ [1]
Supports the ***FLG_NOCONSISTENCE*** flag.	
.FLG_NOSYSTEMSTOP	02$_H$ [1]
Supports the ***FLG_NOSYSTEMSTOP*** flag.	

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_CONSIST (-225)

Consistency was not assured. (however, it is not handled as error if ***FLG_NOCONSISTENCE*** is set).

E_PAR (-145)

A parameter value was invalid.

ET_MACV (-26)

An invalid memory region on the target was accessed.

6.2.2 Wite register

tif_set_reg Write of register value [R] □

ER tif_set_reg (VP storage, BITMASK_8 * p_valid, FLAG flags)

VP storage
Pointer retaining the value to be written

BITMASK_8 * p_valid
Pointer to validation flag about register table items
(NULL: Targets entire context)

FLAG flags
Flags

This function changes the value of a register on the target.

Supplementary explanation

The value to be written in a register must be stored in the endian of the target.

If a nonexisting register is specified as the write destination, the function returns the ***E_PAR*** error.

Keys

TIF	05_H
.TIF_SET_REG	$0D_H$

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_CONSIST (-225)
Consistency was not assured (however, it is not handled as an error if ***FLG_NOCONSISTENCE*** is set).

E_PAR (-145)
A parameter value was invalid.

ET_MACV (-26)
An invalid memory region on the target was accessed.

6.3 Target Operations

6.3.1 Start of target execution

tif_sta_tgt Start of target execution [R] □

ER tif_sta_tgt (DT_VP staaddr, FLAG flags)

DT_VP staaddr
Starting address

FLAG flags
Flags

This function executes the target from a specified address. It starts to execute target from a specified address while retaining the current register values and target system status.

The write access size is determined by the debugging tool.

Extension

Flag

OPT_RESTART (1)
Restarts target (ignores argument ***staadr***).

Extension

Supplementary explanation

This function can be executed only when the target is stopped or temporarily broken. If the function cannot be executed in such a state, it returns the ***E_EXCLUSIVE*** error.

Keys

TIF	05_H
.TIF_STA_TGT	$0E_H$
.OPT_RESTART	10_H [B]
OPT_RESTART is available.	

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_PAR (-145)

A parameter value was invalid.

E_EXCLUSIVE (-226)

Another request has already been issued. The function could not receive a new request until execution of the previous request ends.

6.3.2 Stop of target execution

tif_stp_tgt Stop of target execution [E] □

ER tif_stp_tgt (FLAG flags)

FLAG flags
Flags

This function stops the target when it is issued.

Supplementary explanation

When this function executes target switches to a stop state even when it has been stopped or broken. Target execution resumption from the stop state depends on an implement definition.

Keys

TIF 05_H

.TIF_STP_TGT $0F_H$ [1]

Supports this function.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

6.3.3 Break of target execution

tif_brk_tgt Break of target execution [E] □

ER tif_brk_tgt (FLAG flags)

FLAG flags
Flags

This function stops the target in such a manner that its execution can be resumed later.

Keys

TIF 05_H

.TIF_BRK_TGT 10_H [1]

Supports this function.

Supplementary explanation

If this function is executed while the target is stopped, the ***E_EXCLUSIVE*** error occurs. (***E_OK*** occurs in a break state.)

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_EXCLUSIVE (-226)

Another request has already been issued. The function could not receive a request until the execution of previous request ends.

6.3.4 Resumption of target execution

tif_cnt_tgt Resumption of target execution [R] □

ER tif_cnt_tgt (FLAG flags)

FLAG flags
Flags

This function resumes a target execution in break state.

Supplementary explanation

If this function is executed when the target is not in a break state, the ***E_EXCLUSIVE*** error occurs.

Keys

TIF	05_H
.TIF_CNT_TGT	11_H

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_EXCLUSIVE (-226)

Another request has already been issued. The function could not receive a new request until execution of the previous request ends.

6.4 Hardware Break Operations

6.4.1 Set of break point

tif_set_brk Set of break point [R] □

ER_ID tif_set_brk (ID brkid, T_TSBRK * pk_tsbrk, FLAG flags)

ID brkid
Break point ID

T_TSBRK * pk_tsbrk
Pointer to the structure having break point information

FLAG flags
Flags

(Return value) ID brkid
Assigned break point ID

This function not only sets a break point on the target but also sets a callback routine for such a break.

The contents of ***T_TSBRK*** are given below:

```
typedef struct      t_tsbrk
{
     UINT brktype        : Break type
     DT_VP brkadr        : Address at which a break is set
     VP_INT brkprm       : Callback routine report flag
}    T_TSBRK;
```

The meaning and the value that the ***brktype*** parameter can be set are shown below:

- **BRK_EXECUTE (1)**
  Execution break

Supplementary explanation

When the function is executed successfully in situations where the automatic number assignment flag ***FLG_AUTONUMBERING*** is specified, the function returns the value of 1 or greater (ID value), which is assigned to a setup item. This is also true even when the automatic assignment flag is not specified.

Flag

FLG_NOREPORT (80000000$_H$): Report function invalidation

The paired callback function will not be called.

Extension

The following operation can be executed as extended functions:

For the ***brktype*** parameter, the following value can also be set:

- **BRK_ACCESS (2)**
  Access break

When an access break is specified, at least one of the following access specifiers must be set. However, two or more can be specified simultaneously.

- **ACS_READ (0x100)**
  Invokes break when read performed at target address
- **ACS_WRITE (0x200)**
  Invokes break when write performed at target address
- **AS_MODIFY (0x400)**
  Invokes break when modification made at target address

When the employed debugging tool supports a conditional break function recommended by the ITRON Debugging Interface Specification, setting ***OPT_CNDBRK*** to the ***flags*** parameter enables to use the following ***T_TSBRK_CND*** instead of ***T_TSBRK***. For use of ***T_TSBRK_CND***, the RIM must cast a ***T_TSBRK_CND*** type variable into the ***T_TSBRK*** type and pass it to ***tif_set_brk***.

```
typedef struct    t_tsbrk_cnd
{
    UINT brktype       : Break type
    DT_VP brkadr       : Address at which a break is set
    VP_INT brkprm      : Callback routine report flag
    DT_VP cndadr       : Address to be set for a conditional break
    VP_INT cndval      : Value to be set for a conditional break
    UINT cndvallen     : Byte lenght (1, 2, or 4) of the value to be set for a conditional
                         break
}   T_TSBRK_CND;
```

When this structure and ***OPT_CNDBRK*** are used, a conditional expression (*cndadr == cndval) is added to regular break conditions. A break is regarded as a provisional break hit only when these two conditions are satisfied, and ***tif_rep_brk*** is called as needed.

Flags

OPT_CNDBREAK (4)

Uses a conditional break mechanism of the debugging tool.

FLG_AUTONUMBERING (40000000$_H$): ID automatic assignment

Automatically assigns an ID. If the ID value is specified as an argument, it is ignored by the function. When the function is successfully executed, it returns the automatically assigned ID.

Extension

Keys

TIF 05_H

.TIF_SET_BRK 13_H

.FLG_AUTONUMBERING 04_H [1]

Supports the ***FLG_AUTONUMBERING*** flag.

.OPT_CNDBREAK 10_H [1]

Supports the ***OPT_CNDBREAK*** option.

.BRK_ACCESS 11_H [1]

An access break is available.

Errors

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_NOID (-162)

Count of ID for automatic assignment was insufficient .

E_OBJ (-169)

The targeted object on the target was inoperative.

ET_ID (-18)

The specified kernel object IDs was invalid.

E_PAR (-145)

A parameter value was invalid.

ET_MACV (-26)

An invalid memory region on the target was accessed.

6.4.2 Delete of break point

tif_del_brk Delete of break point [R] □

ER tif_del_brk (ID brkid, FLAG flags)

ID brkid
Break point ID

FLAG flags
Flags

This function deletes a break point that corresponds to a specified ID.

The following special parameter can be set to specify the ID for deletion.

- **ID_ALL (-1)**
  Deletes all break points.

Keys

TIF	05_H
.TIF_DEL_BRK	14_H

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_ID (-146)
The specified object ID was invalid.

E_OBJ (-169)
The targeted object in the target was inoperative.

6.4.3 Break report

tif_rep_brk Break report [R:callback] ◯

ER tif_rep_brk (ID brkid, VP_INT param)

ID brkid
Break point ID

VP_INT param
Report parameter (see ***Section 6.4.1***)

This function reports that the target is stopped at a break point specified by ***tif_rep_brk***. In this callback function, the RIM checks whether the conditions for this break are satisfied and determines whether or not to break the system. When this function concludes that the conditions are satisfied, the debugging tool performs a specified operation and escapes the function. And then, it continues a target stop process. If the function does not conclude that the conditions are satisfied, it cancels a target stop process and resumes target execution.

A series of break operations is show below:

1. A break setting request is delivered by ***tif_set_brk*** to the RIM.
2. The RIM uses ***tif_set_brk*** to set a break point at a location that satisfies the request.
3. When the debugging tool reaches the break point, it checks whether it has been set by ***tif_set_brk***.
4. If so, the debugging tool executes ***tif_rep_brk*** using the break ID and report flag as arguments.
5. The callback function check whether the currently stopped conditions satisfiey the requested break setting on the basis of the report parameter, break ID, and ***tif_set_brk*** argument (when the request is satisfied, proceed to the step 6. If not, proceed to the step 6').
6. When the request is satisfied, ***tif_rep_brk*** calls ***rif_rep_brk***.
7. After a necessary process is performed by ***rif_rep_brk***, ***tif_rep_brk*** returns ***E_TRUE***.
8. The debugging tool reports the user that the target is broken (the target is in a break state in the steps 3 or later operation).

6'. If the request is not satisfied, ***E_FALSE*** is returned.
7'. The debugging tool resumes the target operation.

Supplementary explanation

When this function returns ***E_TRUE***, the debugging tool continues a break operation. On the other hand, when this function returns ***E_FALSE***, the debugging tool suspends a break operation to stop the election of target. However, if ***BRK_REPORT*** is specified as a stop state operation for the target break point, the break operation does not continue even if this function returns ***E_TRUE***.
While this function is making a decision, target execution is in a break state. However, this does not hold true when ***BRK_REPORT*** is specified as the stop state operation for the target break point.

Keys

TIF 05_H

.TIF_REP_BRK 12_H

Supports this function.

.FLG_AUTONUMBERING 04_H [1]

Supports the ***FLG_AUTONUMBERING*** flag.

Errors

E_TRUE (0)

Decision routine return parameter **(TRUE)**
Concludes that a break hit has occurred, and continues a break process.

E_FALSE (-229)

Decision routine return parameter **(FALSE)**
Concludes that the conditions are false, and continues target execution.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

6.5 Symbol Table Operations

6.5.1 Reference of symbol table value

tif_ref_sym Reference of symbol table value [R] □

ER tif_ref_sym (INT * p_value , char * strsym , FLAG flags)

INT * p_value
: Pointer to the region that stores a value indicated by a symbol

char * strsym
: Symbol name (NULL-terminated string)

FLAG flags
: Flags

This function gets a value of the symbol table that is specified by ***strsym***.

Supplementary explanation

Only a symbol value (address) can be got by ***tif_ref_sym***. An equation cannot be evaluated in principle. More specifically, arithmetic operation, logic operation, array (***dummy[n]***), indirect operator (****dummy***), address operator (***&dummy***), and member selection equation (***a.b, c->d***) cannot be used.

Keys

TIF	05_H
.TIF_REF_SYM	15_H

Errors

E_OK (0)
: Normally ended.

E_NOSPT (-137)
: An unsupported operation was executed.

E_NOMEM (-161)
: The request could not be executed due to insufficient host memory.

E_FAIL (-227)
: The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
: An irrecoverable (fatal) error occurred for some reason.

E_OBJ (-169)
: The targeted object on the target was inoperative.

E_PAR (-145)
: A parameter value was invalid.

6.5.2 Reference of symbol in symbol table

tif_rrf_sym Reference of symbol in symbol table [E] ☐

ER tif_rrf_sym

(char * p_sym , UINT maxlen , INT value , FLAG flags)

char * p_sym
: Stores the corresponding symbol

UINT maxlen
: Maximum size (termination code excluded) of a symbol storage region

INT value
: The key value for reverse search

FLAG flags
: Flags

This function searches for a symbol that is closest to the key.

For a symbol search, the following flags can be exclusively used:

OPT_SEARCH_COMPLETELY (0)
Searches for only a symbol that perfectly matches the search key **(default)**.

OPT_SEARCH_FORWARD (1)
Search forward (in increasing address direction) for symbol closest to specified value.

OPT_SEARCH_BACKWARD (2)
Search forward (in decreasing address direction) for symbol closest to specified value.

Supplementary explanation

When ***OPT_SEARCH_FORWARD*** or ***OPT_SEARCH_BACKWARD*** is specified, the search ends when the start or end of the address space is reached. ***OPT_SEARCH_FORWARD*** and ***OPT_SEARCH_BACKWARD*** are provided to get the name of the service call that is currently being executed by the RIM. The operation to be performed when more than one symbol is assigned to the searched value is implementation-dependent. However, for the above reason, a function name, etc., should be prefered in a code region, and a global variable name, etc., should be prefered in a data region.

maxlen indicates the size of a symbol name storage buffer. ***maxlen*** indicates the prevailing length when a terminating character is included. Therefore, when ***maxlen*** is 1, the character string is void so that ***E_OK*** is returned. When ***maxlen*** is 0, the ***E_PAR*** error occurs.

Keys

TIF 05_H

.TIF_RRF_SYM 16_H [1]

Supports this function.

.OPT_SEARCH_FORWARD 10_H [1]

The ***OPT_SEARCH_FORWARD*** option is available.

.OPT_SEARCH_BACKWARD 11_H [1]

The ***OPT_SEARCH_BACKWARD*** option is available.

.OPT_SEARCH_COMPLETELY 12_H [1]

The ***OPT_SEARCH_COMPLETEL***Y option is available.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_OBJ (-169)

The targeted object on the target was inoperative.

E_PAR (-145)

A parameter value was invalid.

6.6 Function Execution

6.6.1 Function call

tif_cal_fnc Function call [E] □

ER tif_cal_fnc (T_TCFNC * pk_tcfnc, FLAG flags)

T_TCFNC * pk_tcfnc
Pointer to the structure that stores the service call information to be issued

FLAG flags
Flags

This function uses a debugging tool's function to call a function. Function execution basically takes place in a non-blocking mode. Upon completion of function execution, the callback function "***tif_rep_fnc***" is called.

The contents of the "***T_TCFNC***" structure are show below:

```
typedef struct    t_tcfnc_prmary
{
    UINT prmsz          : Parameter size (in bytes)
    VP prmptr           : Pointer to the parameter storage region
}   T_TCFNC_PRMARY;

typedef struct    t_tcfnc
{
    DT_VP fncadr        : Function address
    DT_VP stkadr        : Stack pointer for a function issue
    UINT retsz          : Size (in bytes) of the result storage region
    VP retptr           : Pointer to the region that stores execution results
    UINT resultsz       : Count of parameter
    T_TCFNC_PRMARY prmary[]
                        : Parameter
}   T_TCFNC;
```

To store the function return value, the RIM creates a buffer and stores the pointer to the buffer region in ***T_TCFNC::resultptr*** and the buffer region size in ***T_TCFNC::resultsz***. After function execution, ***tif_cal_fnc*** stores the function return value in the buffer. If the debugging tool concludes that the size is inadequate for return value storage, an error occurs before issuing. Whether the debugging tool conducts a return value type check or not is implementation-dependent.

When the debugging tool passes parameters, it expands the parameters so that ***T_TCFNC::param[0]*** is the leftmost parameter of the function to be executed. The debugging tool may sometimes place the parameters in the target stack area as they are. Therefore, if the size setting is smaller than the size required by the function to be executed, two parameters may be combined.

If the ITRON Debugging Interface Specification cannot be implemented in non-blocking mode, the get information key code item "***TIF.TIF_CAL_FNC.NON-BLOCKING***" must be set to ***FALSE (=0)***. If, in this instance, this function is executed without specifying ***OPT_BLOCKING***, it returns ***E_NOSPT***. Even when this function is executed in a non-blocking mode, the callback function ***tif_rep_fnc*** is called.

Supplementary explanation

When ***tif_cal_fnc*** is executed in blocking mode, this function does not return control until the called function terminates in the strict sense. In the strict sense, the called function terminates when the stack frame at function termination is equivalent to the stack frame when a function is called by ***tif_cal_fnc***. More specifically, if dispatching occurs within the called function and control is passed to another task, this function does not conclude that the function is terminated. In some cases, this function does not return control until the associated function is exited, irrespective of the context status. This also holds true for the end report ***tif_rep_fnc*** for ***tif_cal_fnc***.

Flags

FLG_NOREPORT (80000000_H): Report function invalidation

The paired callback function will not be called.

OPT_BLOCKING (1)

Performs execution in blocking mode.

Keys

TIF	05_H
.TIF_CAL_FNC	17_H [1]
Supports this function.	
.FLG_NOREPORT	03_H [1]
Supports the ***FLG_AUTONUMBERING*** flag.	
.OPT_BLOCKING	11_H [1]
Supports the ***OPT_NON-BLOCKING*** option.	
.NON-BLOCKING	12_H [1]
Supports a non-blocking function call.	

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_EXCLUSIVE (-226)

Another request has already been issued. The function could not receive a function execution request until the execution of the previous request ends.

E_PAR (-145)

A parameter value was invalid.

ET_MACV (-26)

An invalid memory region on the target was accessed.

ET_NOMEM (-33)

The request could not be executed due to insufficient memory on the target.

6.6.2 Report of function execution end

tif_rep_fnc Report of function execution end [E:callback]○

void tif_rep_fnc (FLAG flags)

FLAG flags
Flags

This function reports the end of a function that was issued by ***tif_cal_fnc*** in a non-blocking mode. The return value is to be stored in the region specified by ***tif_cal_fnc***.

Keys

TIF 05_H

.TIF_REP_FNC 18_H [1]

Supports this function.

Error

This function does not have a return value.

6.7 Trace Log Operations

6.7.1 Set of trace log

tif_set_log Set of trace log [E] □

ER_ID tif_set_log (ID logid, T_TSLOG * pk_tslog, FLAG flags)

ID logid
ID assigned to selected log information

T_TSLOG * pk_tslog
Pointer to the structure that stores trace log setting information

FLAG flags
Flags

(Return value) ID logid
Assigned log ID (independent of ***rif_set_log***)

This function performs trace log setting.

The contents of the structure ***T_TSLOG*** are indicated below:

```
typedef struct    t_tslog
{
     UINT logtype        : Log type flag
     DT_VP staadr        : Starting address
     DT_VP endadr        : Ending address (NULL if the range is not specified)
     DT_VP valptr        : Read start position (NULL: event occurrence position)
     DT_SIZE valsz       : Data length (in bytes)
}    T_TSLOG;
```

The following values can be set for ***T_TSLOG::logtype***:

The following values can be used exclusively:

- ***LOG_INSTRUCTION (0)***
  Instruction **(default)**
- ***LOG_DATA (1)***
  Data

When ***LOG_DATA*** is specified for ***logtype***, at least one of the following operation options must be specified. However, two or more can be specified simultaneously.

- **ACS_READ (0x100)**
  Read
- **ACS_WRITE (0x200)**
  Write
- **ACS_MODIFY (0x400)**
  Modification (Read Modify Write)

When the buffer for getting log is full, the following options can be selected exclusively as the performed operation.

LOG_BUFFUL_STOP (0)
Stops getting a trace when the buffer becomes full **(default)**.

LOG_BUFFUL_CALLBACK (2)
Executes callback function when the buffer becomes full.

LOG_BUFFUL_ FORCEEXEC (1)
Continues with getting log by discarding oldest data when the buffer becomes full.

The above options are valid for a log that is set by the execution of this function.
Let us assume that three different logs are activated. The first log (ID: 1) is the one for which no option is set. For the second log (ID: 2), ***OPT_BUFFUL_CALLBACK*** is set. For the third log (ID: 3), ***FLG_NOREPORT*** is set. When the buffer later becomes full due to target program execution and the debugging tool concludes that the currently got log event cannot be stored, a forced termination is issued to the logs having ID 1 and ID 3 for which ***OPT_BUFFUL_STOP*** is set by default, and ***tif_rep_log*** receives an ID1 end event (***EV_STOP***). The debugging tool does not report to the ID 3 because ***FLG_NOREPORT*** is set for it. Since ***OPT_BUFFUL_CALLBACK*** is set for the ID 2, ***tif_rep_log*** is called by ***EV_REPORT***. If, in this instance, a buffer read or other appropriate process is not performed in ***tif_rep_log*** and the buffer becomes full again, ***EV_BUFFER_FULL*** calls ***tif_rep_log*** for all existing logs.

Supplementary explanation

The ***T_TSLOG::staadr*** and ***T_TSLOG::endadr*** variables define the memory region to be targeted for log event generation. This region is a closed section [***staadr, endadr***], and the address ***endadr*** is targeted. If ***staadr > endadr***, the ***E_PAR*** error occurs.

The variable ***T_TSLOG::endadr*** defines the memory region to be targeted for event generation. The variable ***T_TSLOG::valsz*** defines the length of memory to be read at the time of event generation. If ***T_TSLOG::valsz*** is set to 0, only events will be stored.
The variable ***T_TSLOG::valptr*** specifies the address where a read operation begins when an event occurs. When a log event occurs in a closed section ***[staadr, endadr]*** in situations where a specific address is set, ***T_TSLOG::valsz*** bytes are read beginning with ***T_TSLOG::valptr*** and recorded. On the other hand, if ***T_TSLOG::valptr*** is set to ***NULL***, the address where an event is generated becomes the start point. If, in this situation, an event is generated, to access a certain address (***evtadr***) in a closed section [***staadr, endadr***], ***length*** bytes data is read from ***evtadr*** and stored.

When the function is executed successfully in situations where the automatic number assignment flag ***FLG_AUTONUMBERING*** is specified, the function returns the value of 1 or more (ID value), which is assigned to a setup item. This is also true even when the automatic assignment flag is not specified.

Flags

FLG_NOREPORT (80000000$_H$): Report function invalidation

The paired callback function will not be called.

FLG_AUTONUMBERING (40000000$_H$): ID automatic assignment

Automatically assigns an ID. If an argument is used to specify the ID, it is ignored by the function. When the function is successfully executed, it returns the automatically assigned ID.

OPT_BUFFUL_STOP (0)

When the buffer becomes full, this flag stops getting trace operation **(default)**

OPT_BUFFUL_FORCEEXEC (1)

When the buffer becomes full, this flag discards the oldest data and continues to get logs.

OPT_BUFFUL_CALLBACK (2)

When the buffer becomes full, this flag executes tif_rep_log.

Keys

TIF 05$_H$

.TIF_SET_LOG 19$_H$ [1]

Supports this function.

.FLG_NOREPORT 03$_H$ [1]

The "***FLG_NOREPORT***" flag is available.

.FLG_AUTONUMBERING 04$_H$ [1]

Supports the ***FLG_AUTONUMBERING*** flag.

.OPT_BUFFUL_FORCEEXEC 11$_H$ [1]

The ***OPT_BUFFUL_FORCEEXEC*** option is available.

.OPT_BUFFUL_CALLBACK 12$_H$ [1]

The ***OPT_BUFFUL_CALLBACK*** option is available.

.LOG_INSTRUCTION 13$_H$ [1]

The log type ***LOG_INSTRUCTION*** is available.

.LOG_DATA 14$_H$ [1]

The log type ***LOG_DATA*** is available.

.LOG_READ 15$_H$ [1]

LOG_READ is available.

.LOG_WRITE 16$_H$ [1]

LOG_WRITE is available.

.LOG_MODIFY 17$_H$ [1]

LOG_MODIFY is available.

Errors

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)
The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)
An irrecoverable (fatal) error occurred for some reason.

E_ID (-146)
The specified object ID was invalid.

E_NOID (-162)
Count of IDs for automatic assignment was insufficient.

E_OBJ (-169)
The targeted object on the target was inoperative.

ET_MACV (-26)
An invalid memory region on the target was accessed.

E_PAR (-145)
A parameter value was invalid.

6.7.2 Delete of trace log setting

tif_del_log Delete of trace log setting [E] □

ER tif_del_log (ID logid, FLAG flags)

ID logid
ID of the log to be deleted

FLAG flags
Flags

This function deletes logs that are set by ***tif_set_log*** completely or partially. ***tif_set_log*** is explained earlier.

Supplementary explanation

When ***logid*** is set to ***ID_ALL(=1)***, all the logs will be targeted. Note that this ***logid*** is given by ***tif_set_log***. It is independent of the ID of lag that is used for ***rif_set_log***.

Keys

TIF 05_H

.TIF_DEL_LOG $1A_H$ [1]

Supports this function.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_ID (-146)

The specified object ID was invalid.

E_OBJ (-169)

The targeted object on the target was inoperative.

E_EXCLUSIVE (-226)

Another request has already been issued. The function could not receive a new request until the execution of the previous request ends.

6.7.3 Start of trace log

tif_sta_log Start of trace log [E] □

ER tif_sta_log (ID logid, FLAG flags)

ID logid

ID of the log to be activated

FLAG flags

Flags

This function starts to get a trace log in accordance with the data set by ***tif_set_log***. When ***logid*** is set to ***ID_ALL(=1)***, this function validates all the log settings defined by ***tif_set_log***.

Supplementary explanation

Even when this function is executed for a second time with respect to a log setting that has already been started, the function ends normally. However, the specified log setting is stopped by a single stop procedure even if it has plurally been activated.

Keys

TIF 05_H

.TIF_STA_LOG $1B_H$ [1]

Supports this function.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_ID (-146)

The specified object ID was invalid.

E_OBJ (-169)

The targeted object on the target was inoperative.

6.7.4 Stop of trace log

tif_stp_log Stop of trace log [E] □

ER tif_stp_log (ID logid, FLAG flags)

FLAG flags
Flags

This function stops a specified trace log which is currently got.

Supplementary explanation

This function does not concern the target execution status.
Even when this function is executed for a second time with respect to an already stopped log setting, the function ends normally. However, the specified log setting is started by a single start procedure even if it has plurally been stopped.

Keys

TIF 05_H

.TIF_STP_LOG $1C_H$ [1]

Supports this function.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_ID (-146)

The specified object ID was invalid.

E_OBJ (-169)

The operation targeted was not found or operative.

6.7.5 Trace logs callback

tif_rep_log Trace logs callback [E:callback] ○

void tif_rep_log (ID logid, UINT event, FLAG flags)

ID logid
: ID of the log that is the factor of generation

UINT event
: Factor of the call of this function

FLAG flags
: Flags

This function is called to perform an appropriate process when a factor is generated by a trace log operation or when a callback is set by the function for getting trace log ***tif_set_log***. The function also performs a process when, for instance, a log is deleted due to a buffer-full condition.

The probable factors of generation are enumerated below:

EV_BUFFER_FULL (1)
The trace buffer is full.

EV_STOP (2)
The trace log function is stopped.

EV_REPORT (4)
The report conditions specified by ***tif_set_log*** are satisfied.

Supplementary explanation

When a log is brought to a forced termination due, for instance, to a buffer-full condition, the RIM needs not to call ***tif_stp_log*** for the targeted ID.
If it is necessary to get trace log on the target while this callback is being called, this function does not assure to get trace log data.
As regards a log for which ***OPT_BUFFUL_CALLBACK*** is specified by ***tif_set_log***, the first buffer-full condition is reported as ***EV_REPORT***. If there are two or more logs for which ***OPT_BUFFUL_CALLBACK*** is specified, ***EV_REPORT*** is issued for all such logs. If no appropriate process is performed later and the buffer-full condition, which was the factor for the issue of ***EV_REPORT***, is not cleared, ***EV_BUFFER_FULL*** is called for all remaining logs as an unrecoverable error. If no appropriate process is performed for this buffer-full condition, the debugging tool forcibly terminates all the logs and reports an ***EV_STOP*** to terminate the process.

Keys

TIF 05_H

.TIF_REP_LOG $1D_H$ [1]
Supports this function.

Errors

This function does not have a return value.

6.7.6 Get of trace log

tif_get_log Get of trace log [E] □

ER tif_get_log (VP p_result, FLAG flags)

VP p_result
Pointer to the region that stores a trace log

FLAG flags
Flags

This function gets a trace log source that is retained by a debugging tool. The trace log source is memory data on the target that the debugging tool has got as log information. When a log is directly written into memory or onto a disk not with debugging tool, but with a debugging task and so on, this function cannot got a log.
After ***tif_set_log*** gets one log, it moves the read position to the next log. To get all the logs, the RIM has to call this function two or more times. When the remaining log count is 0, ***tif_get_log*** returns the ***E_OBJ*** error.

The data of structure for getting log ***T_TGLOG*** are shown below:

typedef struct t_tglog
{
ID logid : Corresponding log ID
DT_VP staadr : Preselected starting address
DT_VP endadr : Preselected ending address
UINT logtype : Log type information
LOGTIM logtim : Time stamp
DT_SIZE bufsz : Buffer size
char buff[] : The region that stores a value which was got
} **T_TGLOG;**

Notes:

1. When the ***tlogid*** does not exist ***(tlogid=0)***, it is necessary to be determined from an address and so on.

2. The value specified for ***bufsz*** indicates the maximum length that can be get by ***buf***. When the function is executed, ***bufsz*** stores the size of the stored data. For details, see ***Section 5.2***.

Option

OPT_PEEK (1)
Gets a trace log without deleting it from the spool.

Keys

TIF 05H
.TIF_GET_LOG 1EH [1]
Supports this function.
.OPT_PEEK 10H [1]
Supports the ***OPT_PEEK*** option.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_OBJ (-169)

The targeted object on the target was inoperatve.

E_PAR (-145)

A parameter value was invalid.

7. Other Interfaces

7.1 Debugging Tool Operations

7.1.1 Get of debugging tool information

dbg_ref_dbg Get of debugging tool information [R] □

ER dgb_ref_dbg

(T_INFO * pk_rdbg, UINT packets, FLAG flags)

T_INFO * pk_rdbg

Pointer to beginning of array of structure that stores information about debugging tool

UINT packets

T_INFO structure array length

FLAG flags

Flags

The RIM uses this function to examine the type of a debugging tool, the operations the debugging tool performs, and other information.
The function for getting information ***T_INFO*** and key codes are used for getting information about this function. For details, see ***Section 3.6***.

The contents of ***T_INFO*** are shown below:

```
typedef struct    t_info_result_buf
{
    UINT sz                 :  Buffer size
    VP ptr                  :  Pointer to region storing character string or special type
}   T_INFO_RESULT_BUF;

typedef union     t_info_result
{
    INT value               :  32-bit signed integer
    T_INFO_RESULT_BUF buf
                            :  Value of special type
}   T_INFO_RESUT;

typedef struct    t_info
{
    char key [4]            :  Key for indentifying information
    T_INFO_RESULT result
                            :  Value corresponding to key
}   T_INFO;
```

Keys

DEBUGGER		1_H
.CNDBREAK		1_H
.NUM		3_H [W]
	Count of conditional breaks that can be set (0: not supported)	
.LOG		2_H
.NUM		3_H [W]
	Count of hardware logs that can be set (0: not supported)	
.NAME		80_H [S]
	Any character(s) for debugging tool identification	
HOST		2_H
.ENDIAN		1_H [W]
	Host computer endian (0: little; 1: big)	
.NAME		80_H [S]
	Any character(s) for host computer identification	
TARGET		3_H
.ENDIAN		1_H [W]
	Target computer endian (0: little; 1: big)	
.REGISTER		2_H
.NUM		3_H [W]
	Count of target computer registers	
.NAME		80_H [S]
	Any character(s) for target device identification	

Supplementary explanation

The information that can be got with this function includes all the key codes with ***INF_TIF*** as the first key as described in ***Chapter 6***.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error for some reason

E_OBJ (-169)

The targeted object on the target was inoperative.

E_PAR (-145)

A parameter value was invalid.

7.2 RIM Operations

7.2.1 RIM initialization

dbg_ini_rim RIM initialization [R] ◯

ER dbg_ini_rim (VP param)

VP param
Parameter sent from debugging tool

This function initializes the RIM at a debugging tool activation. Callback functions are registered at this stage. This function is executed after the ***dbg_ini_inf*** function described in ***Section 7.3***. Therefore, it is assured that all the functions offered by the debugging tool side are available on the interface.
The parameter value is not especially stipulated. However, it is possible that parameters will be standardized in compliance with the guidelines (e.g., Windows-DLL guidelines) within the debugging interface.

Supplementary explanation

When this function returns an error other than ***E_OK***, debugging tool judges that the function failed in RIM initialization. In such a case, the debugging tool must not read the other interface functions that belong to the RIM side.

Errors

E_OK (0)

Normally ended.

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason

(No implicit error exists.)

7.2.2 RIM finalization process

dbg_fin_rim RIM finalization process [R] ◯

ER dbg_fin_rim (VP param)

VP param

Parameter sent from debugging tool

This function performs the RIM finalization process. The debugging tool must call this function before the end of the program, and the RIM must free all got sources within this function. The parameter value is not especially stipulated. However, it is possible that parameters will be standardized in compliance with the guidelines (e.g., Windows-DLL guidelines) within the debugging interface.

Supplementary explanation

When this function ends with other than ***E_OK***, the debugging tool must not call any functions that are offered subsequently by the RIM.

Errors

E_OK (0)

Normally ended.

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

(No implicit error exists.)

7.2.3 Get RIM-related information

dbg_ref_rim Get RIM-related information [R] ◯

ER dbg_ref_rim

(T_INFO * ppk_rrim, UINT packets, FLAG flags)

T_INFO * ppk_rrim

Pointer to beginning of array of information storage structure

UINT packets

Length of array indicated by ***ppk_rrim***

This function gets the RIM function and other RIM-related information. The information obtained in this manner enables the debugging tool to acquire information including that of function that are available on the RTOS access interface.

The function for getting information ***T_INFO*** and key codes are used to get information with this function. For details, see ***Section 3.6***.

Keys

OS 8_H

.NAME 80_H [S]

Any character(s) for target OS identification ("ITRON")

Supplementary explanation

The information that can be got with this function includes all the key codes with ***INF_RIF*** (described in ***Chapter 5***) as the first key.

Errors

E_OK (0)

Normally ended.

E_NOSPT (-137)

An unsupported operation was executed.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_OBJ (-169)

The targeted object on the target was inoperative.

E_PAR (-145)

A parameter value was invalid.

7.3 Interface Operations

7.3.1 Interface initialization

dbg_ini_inf Interface initialization [E] ◯

ER dbg_ini_inf (T_INTERFACE * ppk_interface, VP param)

T_INTERFACE * ppk_interface
Pointer to the region that stores entry point for each function

VP param
Parameter offered by debugging tool side

This function reports the location of the function pointer table to access interface functions and initializes the function pointer table. It is executed by the debugging tool side. In this function, the RIM registers the pointer to a function to be offered by RIM itself on the interface in ***ppk_interface***.

Before execution of this function, the debugging tool must offer pointers to the following functions:

- **dbg_ref_dbg**
- **Functions on TIF**
  (Note: No callback on RIF need to be registered at this stage.)

In this function, the RIM must offer pointers to the following functions:

- **dbg_ini_rim**
- **dbg_ref_rim**
- **dbg_fin_rim**
- **Functions on RIM**
  (Note: No callback on the TIF need be registered at this stage.)

T_INTERFACE is a structure that has the pointers to all functions offered in compliance with the ITRON Debugging Interface Specification.

This function need not to be executed in an environment where all the functions are bound statically.

Errors

E_OK (0)
Normally ended.

E_NOSPT (-137)
An unsupported operation was executed.

E_NOMEM (-161)
The request could not be executed due to insufficient host memory.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued).

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_PAR (-145)

A parameter value was invalid.

This page is intentional blank.

8. Recommended Guidelines

This chapter explains the recommended guidelines for the ITRON Debugging Interface Specification. The recommended guidelines need not to be complied with. However, they contain items concerning compatibility. It is therefore best if debugging tool or RIM implementation is in compliance with the guidelines to provide support for a large number of debugging tools and RIMs.

8.1 RIM Guideline

8.1.1 RIM operation guideline

- **Access in undefined state before target initialization**
  In a situation where the target is not initialized, the debugging tool might not be able to gain accessing. If any operation is performed in such a state, function returns a system error "***E_SYS***". Also, the resulting information is invalid.

8.1.2 RIM data format for supplying

The RIM is implemented in the manufacturer’s debugging tool. Therefore, specific guidelines apply to its data format for supply.

The following data formats are supported in the current specifications:

- **Supplies C source program**
- **Supplied with library**

With the use of any other method of supply is intended, the RIM creation side must introduce a thunk layer to establish a link between the module main body and C language interface.

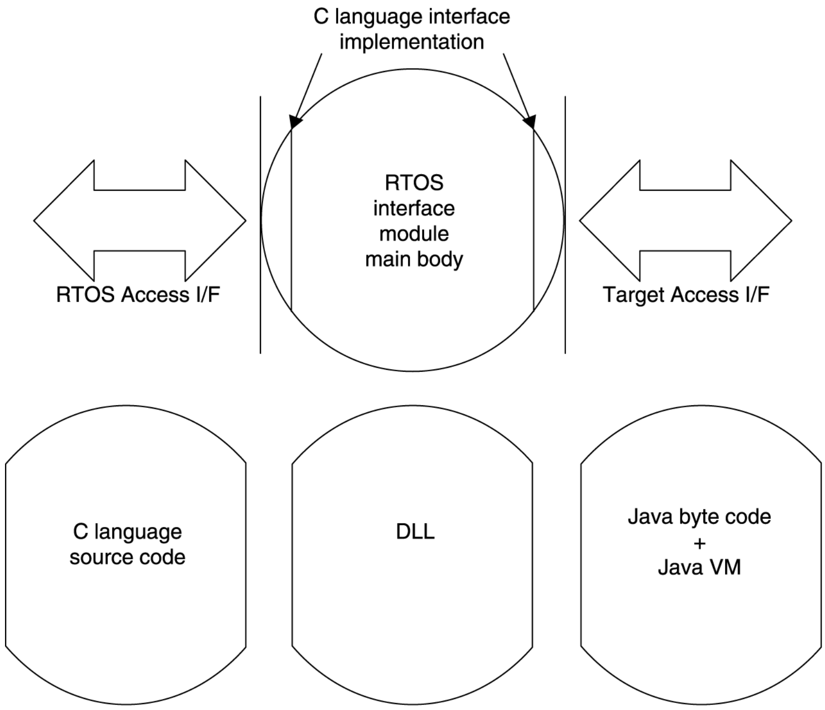

Figure 22: Special RIM Supply Method

8.1.3 Speed enhancement and debugging agent

The current debugging interface uses a callback to check for a break point hit. However, when actual devices are used instead of simulation, the information transfer between host and target is mostly via a serial interface. Therefore, frequent callbacks lowers the debugging tool speed. Under such circumstances, RTOS manufacturers should introduce ‘debugging tasks geared to increase speed’ to operate debugging tools at high speed (this function is effective for breaks and trace logs whose speed should be increased).

Function examples of debugging tasks used for such purposes are listed below:

- **Break-related function**
- **Function for satisfying some break conditions in debugging task**
- **Trace log function**
- **Function for getting trace log closed only in target without resort to debugging tool**

In addition to the above, we think it is possible to offer more effective functions and higher-speed operations depending on the RTOS characteristics.

When a debugging agent incorporating the above functions is offered, the user can conduct debugging operations in an appropriate environment by selecting one of three environments (or two out of three environments in some situations).

- **Debugging environment in which this functions operates with large debugging agent and small RIM to eliminate bugs that can be detected with relative ease**

- **Debugging environment with small debugging agent and large RIM can minimize relative load on target with a view to simulating real environment though the function is limited. The environment is suitable for eliminating bugs that cannot easily be detected, for example, a bug with time limitations**
- **Debugging environment in which only RIM is used to impose no load on target**

We expect that the user debugging situation will be improved when two or more sets of RIM and debugging agent are offered to permit selective use depending on the debugging situation (trade-off between overhead and function).

8.2 Windows-DLL Creation Guideline (32-bit RIM)

8.2.1 Type

Host-side types offered by a Windows-DLL are fixed as shown below:

Table 23: 32-bit RIM DLL Host Types

Type Name	Meaning	Bit Length
BOOL	Boolean value	***32 bits***
ER_ID	Greater integer between ***ID*** and ***ER***. ***ID*** represents a positive value. ***ER*** represents a negative value.	***32 bits***
ID	Unsigned integer with sufficiently large size to store object number on debugging interface	***32 bits***
INT	Signed integer that exists on host and has natural length	***32 bits***
UINT	Unsigned integer that exists on host and has natural length	***32 bits***
VP	Void pointer on host	***32 bits***
VP_INT	Type with sufficiently large size to store ***VP*** and ***INT***	***32 bits***
LOGTIM	Log time (accuracy stipulated by ‘implement definition’)	***64 bits***

Table 24: 32-bit RIM DLL Target Types

Type Name	Meaning	Bit Length
DT_B, DT_UB, DT_VB	***8-bit data type***	***8 bits***
DT_H, DT_UH, DT_VH	***16-bit data type***	***16 bits***
DT_W, DT_UW, DT_VW	***32-bit data type***	***32 bits***
DT_D, DT_UD, DT_VD	***64-bit data type***	***64 bits***
DT_SYSTIM, DT_RELTIM, DT_OVRTIM, DT_TMO	***Time-related type (type of the absolute time, the relative time, or the period of relative time)***	***64 bits***
Other	***All other types***	***32 bits****

*. For a 64-bit RIM DLL, this is handled as 64-bit data.

In some cases, these types might be duplicates of those that are stipulated by Windows. Such duplication can be avoided by the following method:

Program source
```
#define TYPE WINDOWS_TYPE
#include <windows.h>
#undef TYPE
    //Subsequently, TYPE can be used as WINDOWS_TYPE.
```
Program source

8.2.2 Structure bits alignment

As with Windows, a RIM created as a Windows-DLL and a debugging tool to accept such a RIM-DLL must comply with the following alignment rules when they declare their respective structures defined by the debugging interface.

Table 25: Windows DLL Creation Guideline Bits Alignment

Data Type	Alignment
DT_B, DT_UB	Aligned at byte boundary
DT_H, DT_UH	Aligned at even-numbered byte boundary
32-bit data type	Aligned at 32-bit boundary
LOGTIM, DT_SYSTIM	Aligned at 64-bit boundary
Structure	Adjusting to alignment requirements of the member which has maximum size in the same structure
Union	Adjusting to alignment requirements of first member

8.2.3 Function export

A RIM-DLL must export the symbol of the following function:

- **dbg_ini_inf: Interface initialization**

8.3 File Format of Standard Execution History

The ITRON Debugging Interface Specification stipulates a standard format for storing an got execution history in a file.
The file is stored in ASCII format with tokens separated by one or more blank characters.*
Also, note that the symbols '.', '|', ':', and ';' are treated as delimiters.**
The syntax is shown below:

Syntax format

Non-termination symbol	Italic
Termination symbol	Bold Gothic
Comment	Character string following symbol '#'
Character string	Expressed by character string (xxx) and comment

Standard history file
Configuration data group   Execution history data group

Configuration data group
Configuration data   Configuration data group
Configuration data

Configuration data
Key code: Value list**;**

Key code
Key   Subsequent key   Subsequent key   Subsequent key
Key   Subsequent key   Subsequent key
Key   Subsequent key
Key

Subsequent key
. Key

Key
xxx   #Key name

Value list
Value Value list
Value

Value
-   #When value setting skipped, hyphen must be used.
Integer value   #Value notation conforms to C language (decimal and hexadecimal only).
Character string   # Value notation conforms to C.

---

*. Space, carriage return, line feed, and tab
**. Blank characters before and after a delimiter can be omitted.

Execution history data group
Execution history data Execution history data group
Execution history data

Execution history data
Execution history header Type-dependent history data**;**

Execution history header
History type**:** History time

History type
xxx #Name of all log types indicated by LOG_TYP_xxx
xxx **| ENTER** #LOG_TYP_xxx|LOG_ENTER
xxx **| LEAVE** #LOG_TYP_xxx|LOG_LEAVE

History time
- #When value setup skipped, hyphen must be used.
Integer value

Type-dependent history data
Value list #As many as parameter members of each log needed type.

Language examples generated from above syntax

Program source

```
CFG.LOGTIM.TICK_N: 1;
CFG.LOGTIM.TICK_D: 1000;
INTERRUPT|ENTER: 0 4;
TASK|ENTER: 180 1;
COMMENT: 200 25 "The program is started.";
```

Program source

This page is intentionuly blank.

9. Reference

9.1 Structures

- **T_MEMBLK [tif_get_bls, tif_set_bls]**

typedef struct t_memblk
{
DT_VP blkptr : Pointer to store memory block data
DT_SIZE blksz : Byte count of memory block data
} **T_MEMBLK;**

- **T_BLKSET [tif_get_bls, tif_set_bls]**

typedef struct t_blkset
{
UINT blkcnt : Count of blocks
T_MEMBLK blkary []:Block array
} **T_BLKSET;**

- **T_RCSVC [rif_cal_svc]**

typedef struct t_rcsvc
{
DT_FN svcfn : Functional code to be issued
BOOL tskctx : Execution with task context (= TRUE)
DT_ID tskid : ID of targeted task (when tskctx = TRUE)
UINT prmcnt : Parameter count
VP_INT prmary[] : Array that stores list of all parameters
} **T_RCSVC;**

- **T_GRDT [rif_get_rdt, tif_get_reg, tif_set_reg]**

typedef struct t_grdt_regary
{
char * strname : Pointer to register name
UINT length : Length (in bytes)
UINT offset : Storage offset position
} **T_GRDT_REGARY;**

typedef struct t_grdt
{
UINT regcnt : Count of registers
UNIT ctxcnt : Count of registers that can be contained in context
T_GRDT_REGARY regary[]
: Register information
} **T_GRDT;**

- **T_INFO [rif_ref_cfg, dbg_ref_dbg, dbg_ref_rim]**

typedef struct t_info_result_buf
{
    UINT sz : Buffer size
    VP ptr : Pointer to region storing caracter string or special type
} **T_INFO_RESULT_BUF;**

typedef struct t_info_result
{
    INT value : 32-bit signed integer
    T_INFO_RESULT_BUF buf
        : Value of special type
} **T_INFO_RESULT;**

typedef struct t_info
{
    char key[4] : Key for information identification
    T_INFO_RESULT result
        : Value corresponding to key
} **T_INFO;**

- **T_RCLOG [rif_cfg_log]**

typedef struct t_rclog
{
    UINT type : Trace log configuration information
    DT_BP bufptr : Pointer to trace log buffer
    DT_SIZE bufsz : Size of trace log buffer
} **T_RCLOG;**

- **T_RGLOG_COMMENT [rif_get_log]**

typedef struct t_rglog_commnet
{
    UINT length : Character string length
    char strtext [] : Character string (NULL-terminated) - May be broken
} **T_RGLOG_COMMENT;**

- **T_RGLOG_CPUEXC [rif_get_log]**

typedef struct t_rglog_cpuexc
{
    DT_ID tskid : Targeted task ID
} **T_RGLOG_CPUEXC;**

- **T_RGLOG_DISPATCH_ENTER [rif_get_log]**

typedef struct t_rglog_dispatch_enter
{
    DT_ID tskid : ID of task in executing state
    UINT disptype : Dispatch type
} **T_RGLOG_DISPATCH_ENTER;**

- **T_RGLOG_DISPATCH_LEAVE [rif_get_log]**

```
typedef struct    t_rglog_dispatch_leave
{
    DT_ID tskid             :  ID of task going to be in executing state
}   T_RGLOG_DISPATCH_LEAVE;
```

- **T_RGLOG_INTERRUPT [rif_get_log]**

```
typedef struct    t_rglog_interrupt
{
    DT_INHNO inhno          :  Interrupt handler number
}   T_RGLOG_INTERRUPT;
```

- **T_RGLOG_ISR [rif_get_log]**

```
typedef struct    t_rglog_isr
{
    DT_ID isrid             :  Interrupt service routine ID
    DT_INHNO inhno          :  Interrupt handler number
}   T_RGLOG_ISR;
```

- **T_RGLOG_SVC [rif_get_log]**

```
typedef struct    t_rglog_svc
{
    DT_FN fncno             :  Functional code
    UINT prmcnt             :  Parameter count
    DT_VP_INT prmary[]:  Parameter
}   T_RGLOG_SVC;
```

- **T_RGLOG_TIMERHDR [rif_get_log]**

```
typedef struct    t_rglog_timerhdr
{
    UINT type               :  Timer type
          (stores constant OBJ_xxx used for rif_ref_obj::objtype)
    DT_ID hdrid             :  Time event handler ID
    DT_VP_INT exinf         :  Extension information
}   T_RGLOG_TIMERHDR;
```

- **T_RGLOG_TSKEXC [rif_get_log]**

```
typedef struct    t_rglog_tskexc
{
    DT_ID tskid             :  Targeted task ID
}   T_RGLOG_TSKEXC;
```

- **T_RGLOG_TSKSTAT [rif_get_log]**

```
typedef struct    t_rglog_tskstat
{
    DT_ID tskid             :  Task ID
    DT_STAT tskstat         :  Status of task at transition destination
    DT_STAT tskwait         :  Wait state
    DT_ID wobjid            :  ID of waiting object
}   T_RGLOG_TSKSTAT;
```

- **T_ROALM [rif_ref_obj]**

typedef struct t_roalm

{

BITMASK valid : Valid field flag
DT_ART almatr : Attribute
DT_VP_INT exinf : Extension information
DT_FP almhdr : Startup address
DT_STAT almstat : Alarm handler start status
DT_RELTIM lefttim : Remaining time

} **T_ROALM;**

- **T_ROCYC [rif_ref_obj]**

typedef struct t_rocyc

{

BITMASK valid : Valid field flag
DT_ART cycatr : Attribute
DT_VP_INT exinf : Extension information
DT_FP cychdr : Startup address
DT_RELTIM cyctim : Cycle
DT_RELTIM cycphs : Initial phase
DT_STAT cycstat : Cyclic handler start status
DT_RELTIM lefttim : Remaining time

} **T_ROCYC;**

- **T_RODTQ [rif_ref_obj]**

typedef struct t_rodtq

{

BITMASK valid : Valid field flag
DT_ATR dtqatr : Data queue attribute
DT_UINT dtqcnt : Data queue capacity
DT_UINT stskcnt : Count of tasks waiting for sending (also used as upper limit for wstsklst)
DT_ID * stsklst : Pointer to region storing ID list of tasks waiting for transmission
DT_UINT rtskcnt : Count of tasks waiting for reception (also used as upper limit for wrtsklst)
DT_ID * rtsklst : Pointer to region storing ID list of tasks waiting for reception
DT_UINT itemcnt : Count of queued data (also used as upper limit for dtqlst)
DT_VP_INT * itemlst : Pointer to region storing list of all items

} **T_RODTQ;**

- **T_ROEXC [rif_ref_obj]**

typedef struct t_roexc

{

BITMASK valid : Valid field flag
DT_FP excrtn : Exception handler start address

} **T_ROEXC;**

- **T_ROFLG[rif_ref_obj]**

typedef struct t_roflg_wflglst
{
DT_ID wtskid : ID of waiting task
DT_FLGPTN wflgptn: Task wait flag pattern
DT_UINT wflgmode : Task wait mode
} **T_ROFLG_WFLGLST;**

typedef struct t_roflg
{
BITMASK valid : Valid field flag
DT_ATR <u>flgatr</u> : Flag attribute
DT_FLGPTN <u>iflgptn</u> : Initial flag pattern
DT_FLGPTN <u>flgptn</u> : Flag pattern
DT_UINT <u>wflagcnt</u> : Waiting task count (also used as upper limit for wflglst)
T_ROFLG_WFLGLST * wflglst
: Pointer to information about task with this flag
} **T_ROFLG;**

- **T_ROISR [rif_ref_obj]**

typedef struct t_roisr
{
BITMASK valid : Valid field flag
DT_ATR isratr : Attribute
DT_VP_INT exinf : Extension information
DT_FP <u>isrfnclst</u> : Starting address of registered routine
DT_INHNO <u>inhno</u> : Corresponded interrupt handler number
} **T_ROISR;**

- **T_ROKER [rif_ref_obj]**

typedef struct t_roker
{
BITMASK valid : Valid field flag
BOOL actker : Kernel start status (TRUE = activated)
BOOL inker : Kernel code execution (TRUE = executing)
BOOL <u>ctxstat</u> : Context status ***(sns_ctx)***
BOOL <u>loccpu</u> : CPU locked ***(sns_cpu)***
BOOL <u>disdsp</u> : Dispatch disabled ***(sns_dsp)***
BOOL <u>dsppnd</u> : Dispatch suspended ***(sns_dpn)***
DT_SYSTIM systim : System time
DT_VP intstk : Stack for nontask context
DT_SIZE intstksz : Stack size for nontask context
} **T_ROKER;**

- **T_ROMBX [rif_ref_obj]**

typedef struct t_rombx
{
BITMASK valid : Valid field flag
DT_ATR mbxatr : Mailbox attribute
DT_PRI maxmpri : Maximum priority
DT_UINT wtskcnt : Count of waiting tasks (also used as upper limit for wtsklst)
DT_ID * wtsklst : Pointer to region storing ID list of waiting tasks
DT_UINT msgcnt : Count of message headers (also used as upper limit for msglst)
DT_T_MSG ** msglst: Pointer to region storing list of all messages
} **T_ROMBX;**

- **T_ROMBF [rif_ref_obj]**

typedef struct t_rombf_msglst
{
DT_VP msgadr : Message address
DT_UINT msgsz : Message length
} **T_ROMBF_MSGLST;**

typedef struct t_rombf
{
BITMASK valid : Valid field flag
DT_ATR mbfatr : Message buffer attribute
DT_UINT maxmsz : Message maximum size
DT_SIZE mbfsz : Buffer region size
DT_UINT stskcnt : Count of tasks waiting for sending (also used as upper limit for wtsklst)
DT_ID * stsklst : Pointer to region storing ID list of waiting tasks
DT_UINT rtskcnt : Count of tasks waiting for reception (also used as upper limit for rtsklst)
DT_ID * rtsklst : Pointer to region storing ID list of waiting tasks
DT_SIZE fmbfsz : Free region size
DT_UINT msgcnt : Count of messages (also used as upper limit for msglst)
T_ROMBF_MSGLST * msglst
: Pointer to information about messages
} **T_ROMBF;**

- **T_ROMPF [rif_ref_obj]**

typedef struct t_rompf_blklst
{
DT_ID htskid : ID number of task that got block
DT_VP blkadr : Block start address
} **T_ROMPF_BLKLST;**

typedef struct t_rompf
{
BITMASK valid : Valid field flag
DT_ATR mpfatr : Fixed-length memory pool attribute
DT_SIZE blksz : Block size

```
    DT_UINT fblkcnt     :  Count of remaining fixed-length memory blocks
    DT_UINT blkcnt      :  Count of all memory blocks
    DT_UINT ablkcnt     :  Count of allocated block (upper limit for blklst)
    T_ROMPF_BLKLST * ablklst
                        :  Pointer to detailed information about each block
    DT_UINT wtskcnt     :  Count of tasks waiting for acquisition (wtsklst upper limit)
    DT_ID * wtsklst     :  Pointer to region storing IDs of tasks waiting for getting
}   T_ROMPF;
```

- **T_ROMPL [rif_ref_obj]**

```
typedef struct    t_rompl_blklst
{
    DT_SIZE blkszi      :  Block size
    DT_ID htskid        :  ID number of task that got block
    DT_VP blkadr        :  Block start address
}   T_ROMPL_BLKLST;

typedef struct    t_rompl
{
    BITMASK valid       :  Valid field flag
    DT_ATR mplatr       :  Variable-length memory pool attribute
    DT_SIZE mplsz       :  Variable-length memory pool region size
    DT_UINT fblksz      :  Maximum size that can be got
    DT_UINT ablkcnt     :  Count of blocks that have got (upper limit for blklst)
    T_ROMPL_BLKLST * ablklst
                        :  Pointer to detailed information about each block
    DT_UINT wtskcnt     :  Count of tasks waiting for get (wtsklst upper limit)
    DT_ID * wtsklst     :  Pointer to region storing IDs of tasks waiting for getting
}   T_ROMPL;
```

- **T_ROMTX [rif_ref_obj]**

```
typedef struct    t_romtx
{
    BITMASK valid       :  Valid field flag
    DT_ATR mtxatr       :  Mutex attribute
    DT_PRI ceilpri      :  Upper-limit priority
    DT_ID htskid        :  ID of task that locks mutex
    DT_UINT wtskcnt     :  Count of waiting tasks (also used as upper limit for wtsklst)
    DT_ID * wtsklst     :  Pointer to region storing ID list of waiting tasks
}   T_ROMTX;
```

- **T_ROOVR [rif_ref_obj]**

```
typedef struct    t_roovr
{
    BITMASK valid       :  Valid field flag
    DT_ATR ovratr       :  Attribute
    DT_FP ovrhdr        :  Startup address
    DT_STAT ovrstat     :  Handler start status
    DT_OVRTIM lefttmo   :  Remaining processor time
```

} **T_ROOVR;**

- **T_ROPOR [rif_ref_obj]**

typedef struct t_ropor
{
BITMASK valid : Valid field flag
DT_ATR poratr : Rendezvous port attribute
DT_UINT maxcmsz : Call message maximum size
DT_UINT maxrmsz : Response message maximum size
DT_UINT ctskcnt : Count of tasks waiting for a call (also used as upper limit for ctsklst)
DT_ID * ctsklst : Pointer to region storing IDs of all tasks waiting for call
DT_UINT atskcnt : Count of waiting tasks (also used as upper limit for atsklst)
DT_ID * atsklst : Pointer to region storing IDs of all waiting tasks
} **T_ROPOR;**

- **T_RORDV [rif_ref_obj]**

typedef struct t_rordv
{
BITMASK valid : Valid field flag
DT_ID tskid : ID of task waiting for rendezvous
} **T_RORDV;**

- **T_RORDQ [rif_ref_obj]**

typedef struct t_rordq
{
BITMASK valid : Valid field flag
DT_ID runtskid : ID of currently executing task
DT_UINT tskcnt : Count of ready tasks (running ones included) (upper limit for tsklst)
DT_ID * tsklst : Pointer to region storing IDs of all executable tasks
} **T_RORDQ;**

- **T_ROSEM [rif_ref_obj]**

typedef struct t_rosem
{
BITMASK valid : Valid field flag
DT_ATR sematr : Semaphore attribute
DT_UINT isemcnt : Initial semaphore count
DT_UINT maxsem : Semaphore maximum value
DT_UINT semcnt : Semaphore count value
DT_UINT wtskcnt : Waiting task count (also used as upper limit for wtsklst)
DT_ID * wtsklst : Pointer to region to storing ID list of waiting tasks
} **T_ROSEM;**

- **T_ROTEX [rif_ref_obj]**

```
typedef struct    t_totex
{
    BITMASK valid      : Valid field flag
    DT_TEXPTN pndptn   : Suspended exception cause
    DT_FP texrtn       : Exception handler startup address
}   T_ROTEX;
```

- **T_ROTMQ [rif_ref_obj]**

```
typedef struct    t_rotmq_quelst
{
    UINT objtyp        : Pointer to region storing types of waiting objects
    DT_ID wobjid       : Pointer to region storing IDs of waiting objects
    DT_TMO lefttmo     : Pointer to region storing remaining wait time
}   T_ROTMQ_QUELST;

typedef struct    t_rotmq
{
    BITMASK valid      : Valid field flag
    SYSTIM systim      : System time prevailing at getting information
    DT_UINT quecnt     : Count of waiting objects in timer queue (upper limit for quelst)
    T_ROTMQ_QUELST * quelst
                       : Pointer to information about waiting objects in timer queue
}   T_ROTMQ;
```

- **T_ROTSK [rif_ref_obj]**

```
typedef struct    t_rotsk
{
    BITMASK valid      : Valid field flag
    DT_ATR tskatr      : Task attribute
    DT_VP_INT exinf    : Extension information
    DT_FP task         : Startup address
    DT_PRI itskpri     : Initial priority
    DT_VP stk          : Initial stack start address
    DT_SIZE stksz      : Stack size
    DT_STAT tskstat    : Task status
    DT_PRI tskpri      : Task current priority
    DT_PRI tskbpri     : Task base priority
    DT_STAT tskwait    : Factor of task wait
    DT_ID wobjid       : ID of object to be waited for
    DT_TMO lefttmo     : Time remaining before timeout
    DT_UINT actcnt     : Count of queued start requests
    DT_UINT wupcnt     : Count of queued wake-up request
    DT_UINT suscnt     : Count of nested forced wait requests
}   T_ROTSK;
```

- **T_RRCND_DBG [rif_ref_cnd]**

```
typedef struct    t_rrcnd_dbg
{
    DT_VP execadr      :  Execution address (NULL: NC)
    DT_VP valadr       :  Address (NULL: NC)
    UINT vallen        :  Data length (1, 2, or 4 bytes)
    VP_INT value       :  Data or pointer value
}   T_RRCND_DBG;
```

- **T_RRCND_RTOS [rif_ref_cnd]**

```
typedef struct    t_rrcnd_rtos
{
    FLAG type          :  Contents to be examined
    DT_ID objid        :  ID as condition
}   T_RRCND_RTOS;
```

- **T_RSBRK [rif_set_brk]**

```
typedef struct    t_rsbrk
{
    UINT brktype       : Break type
    UINT brkcnt        : Count before break
    DT_ID tskid        : Task ID
    DT_ID objid        : Object ID
    UINT objtype       : Object type
    VP_INT brkprm      : Parameter for callback function
    DT_VP brkadr       : Address for break setting
    DT_FN svcfn        : Functional code
}   T_RSBRK;
```

- **T_RGLOG [rif_get_log]**

```
typedef struct    t_rglog
{
    UINT logtype       :  Log type
    LOGTIM logtim      :  Occurrence time
    BITMASK valid      :  Valid field bit map
    UINT bufsz         :  Size of buffer region ‘buf’ (in bytes)
    char buf[]         :  Buffer region for information storage (detailed later)
}   T_RGLOG;
```

- **T_RSLOG_CPUEXC [rif_set_log]**

```
typedef struct    t_rslog_cpuexc
{
    DT_EXCNO excno     :  CPU exception code (ID_ALL available)
}   T_RSLOG_CPUEXC;
```

- **T_RSLOG_DISPATCH [rif_set_log]**

typedef struct t_rslog_dispatch
{
DT_ID tskid : Task ID (***ID_ALL*** available)
} **T_RSLOG_DISPATCH;**

- **T_RSLOG_INTERRUPT [rif_set_log]**

typedef struct t_rslog_interrupt
{
DT_INTNO intno : Interrupt number (***ID_ALL*** available)
} **T_RSLOG_INTERRUPT;**

- **T_RSLOG_ISR [rif_set_log]**

typedef struct t_rslog_isr
{
DT_ID isrid : Interrupt service routine ID (***ID_ALL*** available)
DT_INTNO intno : Interrupt number (***ID_ALL*** available)
} **T_RSLOG_ISR;**

- **T_RSLOG_SVC [rif_set_log]**

typedef struct t_rslog_svc
{
DT_FN svcfn : Functional code (***ID_ALL*** available)
DT_ID objid : Targeted object ID (ignored when SVC does not have target, ***ID_ALL*** available)
DT_ID tskid : Task ID (***ID_ALL*** available)
BITMASK param : Parameter to be got (***ID_ALL*** available)
} **T_RSLOG_SVC;**

typedef struct t_rslog_svc
{
DT_FN svcfn : Functional code (***ID_ALL*** available)
DT_ID objid : Targeted object ID (ignored when SVC does not have target, ***ID_ALL*** available)
DT_ID tskid : Task ID (***ID_ALL*** available)
BITMASK param : Parameter to be got (***ID_ALL*** available)
} **T_RSLOG_SVC;**

typedef struct t_T_RSLOG_TIMERHDR [rif_set_log]
typedef struct t_rslog_timerhdr
{
UINT type : Handler type (***OBJ_ALL*** available)
(Stores constant ***OBJ_xxx*** used for ***rif_ref_obj::objtype***)
(All types are targeted when ***OBJ_ALL***(***= ID_ALL***) is specified.)
DT_ID hdrid : Handler ID (***ID_ALL*** available)
} **T_RSLOG_TIMERHDR;**

- **T_RSLOG_TSKEXC [rif_set_log]**

```
typedef struct    t_rslog_tskexc
{
    DT_ID tskid          : Task ID (ID_ALL available)
}   T_RSLOG_TSKEXC;
```

- **T_RSLOG_TSKSTAT [rif_set_log]**

```
typedef struct    t_rslog_tskstat
{
    DT_ID tskid          : Task ID (ID_ALL available)
}   T_RSLOG_TSKSTAT;
```

- **T_RSLOG_USEREVT [rif_set_log]**

```
typedef struct    t_rslog_comment
{
    UINT length          : Comment character string length
}   T_RSLOG_COMMENT;
```

- **T_TCFNC[tif_cal_fnc]**

```
typedef struct    t_tcfnc_prmary
{
    UINT prmsz           : Parameter size (in bytes)
    VP prmptr            : Pointer to region storing parameter
}   T_TCFNC_PRMARY;

typedef struct    t_tcfnc
{
    DT_VP fncadr         : Function address
    DT_VP stkadr         : Stack pointer for function issue
    UINT retsz           : Size (in bytes) of region storing parameter
    VP retptr            : Pointer to region storing execution results
    UINT prmcnt          : Parameter count
    T_TCFNC_PRMARY prmary[]
                         : Parameter
}   T_TCFNC;
```

- **T_TGLOG [tif_get_log]**

```
typedef struct    t_tglog
{
    ID logid             : Corresponding log ID
    DT_VP staaddr        : Set starting address
    DT_VP endaddr        : Set ending address
    UINT logtype         : Log type information
    LOGTIM logtim        : Time stamp
    DT_SIZE bufsz        : Buffer size
    char buf[]           : The region that stores a value which was got
}   T_TGLOG;
```

- **T_TSBRK [tif_set_brk]**

```
typedef struct    t_tsbrk
{
    UINT brktype        : Break type
    DT_VP brkadr        : Address to set a break
    VP_INT brkprm       : Callback routine report flag
}   T_TSBRK;
```

- **T_TSBRK_CND [tif_set_brk]**

```
typedef struct    t_tsbrk_cnd
{
    UINT brktype        : Break type
    DT_VP brkadr        : Address to set a break
    VP_INT brkprm       : Callback routine report flag
    DT_VP cndadr        : Address to be set for conditional break
    VP_INT cndval       : Value to be set for conditional break
    UINT cndlen         : Byte length (1, 2, or 4) of value to be set for conditional break
}   T_TSBRK_CND;
```

- **T_TSLOG [tif_set_log]**

```
typedef struct    t_slog
{
    UINT logtype        : Log type flag
    DT_VP staadr        : Starting address
    DT_VP endadr        : Ending address (NULL if range not to be specified)
    DT_VP valptr        : Read start position (NULL:  event occurrence position)
    DT_SIZE valsz       : Data length (in bytes)
}   T_TSLOG;
```

9.2 Function List

Get of object status [OBJ] ○
ER rif_ref_obj
(VP p_result, UINT objtype, DT_ID objid, FLAG flags)

Get of description table [CTX] ○
ER rif_get_rdt (const T_GRDT ** ppk_pgrdt, FLAG flags)

Get of task context [CTX] ○
ER rif_get_ctx
(VP p_ctxblk, BITMASK_8 * p_valid, DT_ID tskid, FLAG flags)

Set of task context [CTX] ○
ER rif_set_ctx
(VP p_ctxblk, BITMASK_8 * valid, FLAG flags)

Issue of service call [SVC] ○
ER rif_cal_svc (T_RCSVC * pk_psvc, FLAG flags)

Cancel of an issued service call [SVC] ○
ER rif_can_svc (FLAG flags)

Report of service call end [SVC:callback] □
void rif_rep_svc (DT_ER result)

Get of function code [SVC] ○
ER rif_ref_svc (DT_FN * p_svcfn, char * strsvc, FLAG flags)

Get of service call name [SVC] ○
ER rif_rrf_svc (char * p_strsvc, UINT buf, DT_FN svcfn, FLAG flags)

Set of break point [BRK] ○
ER_ID rif_set_brk (ID brkid, T_RSBRK * pk_rsbrk, FLAG flags)

Delate of break point [BRK] ○
ER rif_del_brk (ID brkid, FLAG flags)

Report of break hit [BRK:callback] □
void rif_rep_brk (ID brkid, VP_INT exinf)

Get of break informationt [BRK] ○
ER rif_ref_brk (ID brkid, T_RSBRK * ppk_rsbrk, FLAG flags)

Get of break condition [CND] ○
ER rif_ref_cnd
(T_RRCND_DBG * ppk_dbg, T_RRCND_RTOS * pk_rtos, FLAG flags)

Set trace log [LOG] ○
ER_ID rif_set_log
(ID logid, UINT logtype, VP pk_rslog , FLAG flags)

Delete of trace log [LOG] ○
ER rif_del_log (ID logid, FLAG flags)

Request of trace log function start [LOG] ○
ER rif_sta_log (ID logid, FLAG flags)

Request of trace log stop [LOG] ○
ER rif_stp_log (ID logid, FLAG flags)

Get of trace log [LOG] ○
ER rif_get_log (T_RGLOG * ppk_rglog, FLAG flags)

Reconfigure of Trace log mechanism [LOG] ○
ER rif_get_log (T_RGLOG * ppk_rglog, FLAG flags)

Get of kernel configuration [R] ○
ER rif_ref_cfg
(T_INFO * p_information, UINT packets, FLAG flags)

Allocate memory (on host) [R] □
ER tif_alc_mbh (VP * p_blk, UINT blksz, FLAG flags)

Allocate Memory (on target) [E] □
ER tif_alc_mbt (DT_VP * p_blk, DT_SIZE blksz, FLAG flags)

Free Memory (on host) [R] □
ER tif_fre_mbh (VP blk, FLAG flags)

Free Memory (on target) [E] □
ER tif_fre_mbt (DT_VP blk, FLAG flags)

Read memory [R] □
ER tif_get_mem
(VP p_result, DT_VP memadr, DT_SIZE memsz, FLAG flags)

Read memory by block set [R] □
ER tif_get_bls
(VP p_result, T_BLKSET * blkset, FLAG flags)

Write memory [R] □
ER tif_set_mem
(VP storage, DT_VP memadr, DT_SIZE memsz, FLAG flags)

Write memory by block set [R] □
ER tif_set_bls (VP storage, T_BLKSET * blkset, FLAG flags)

Set of memory data change report [E] □
ER_ID tif_set_pol
(ID polid, DT_VP adr, DT_INT value, UINT length, FLAG flags)

Delete of change report setting [E] □
ER tif_del_pol (ID polid, FLAG flags)

Report of memory data change [E:callback] ○
void tif_rep_pol (ID polid, DT_INT value, FLAG flags)

Read of register value [R] □
ER tif_get_reg (VP r_result, BITMASK_8 * p_valid, FLAG flags)

Write of register value [R] □
ER tif_set_reg (VP storage, BITMASK_8 * p_valid, FLAG flags)

Start of target execution [R] □
ER tif_sta_tgt (DT_VP staaddr, FLAG flags)

Stop of target execution [E] □
ER tif_stp_tgt (FLAG flags)

Break of target execution [E] □
ER tif_brk_tgt (FLAG flags)

Resumption of target execution [R] □
ER tif_cnt_tgt (FLAG flags)

Set of break point [R] □
ER_ID tif_set_brk (ID brkid, T_TSBRK * pk_tsbrk, FLAG flags)

Delete of break point [R] □
ER tif_del_brk (ID brkid, FLAG flags)

Report break [R:callback] ○
ER tif_rep_brk (ID brkid, VP_INT param)

Reference of in symbol table value [R] □
ER tif_ref_sym (INT * p_value, char * strsym, FLAG flags)

Reference of symbol in symbol table [E] □
ER tif_rrf_sym
(char * p_sym, UINT maxlen, INT value, FLAG flags)

Function call [E] □
ER tif_cal_fnc (T_TCFNC * pk_tcfnc, FLAG flags)

Report of function execution end [E:callback] ○
void tif_rep_fnc (FLAG flags)

Set of trace log [E] □
ER_ID tif_set_log (ID logid, T_TSLOG * pk_tslog, FLAG flags)

Delete of trace log setting [E] □
ER tif_del_log (ID logid, FLAG flags)

Start of trace log [E] □
ER tif_sta_log (ID logid, FLAG flags)

Stop of trace log [E] □
ER tif_stp_log (ID logid, FLAG flags)

Trace logs callback [E:callback] ○
void tif_rep_log (ID logid, UINT event, FLAG flags)

Get of trace log [E] □
ER tif_get_log (VP p_result, FLAG flags)

Get of debugging tool information [R] □
ER dgb_ref_dbg
(T_INFO * pk_rdbg, UINT packets, FLAG flags)

RIM initialization [R] ○
ER dbg_ini_rim (VP param)

RIM finalization process [R] ○
ER dbg_fin_rim (VP param)

Get of RIM information [R] ○
ER dbg_ref_rim
(T_INFO * ppk_rrim, UINT packets, FLAG flags)

Interface initialization [E] ○
ER dbg_ini_inf (T_INTERFACE * ppk_interface, VP param)

9.3 Option Flags

9.3.1 Common flags

FLG_AUTONUMBERING (40000000_H): ID automatic assignment

Automatically assigns ID. If an argument is used to specify the ID, it is ignored by the function. When the function is successfully executed, it returns the automatically assigned ID.

FLG_NOCONSISTENCE (10000000_H): Nonconsistency flag

When this flag is specified, the got data need not be consistent (e.g., the task is not freed from the waiting state although there is no factor for the task wait).

FLG_NOREPORT (80000000_H): Report function invalidation

The paired callback function is not called.

FLG_NOSYSTEMSTOP (20000000_H): An explicit system stop is not permitted

When this flag is specified, ***tif_brk_tgt*** must not be used within the function to halt the system. If this flag is not supported, the ***E_NOSPT*** error occurs.

9.3.2 Unique flags

OPT_APPCONTEXT (1)

Handles context on application level

OPT_BLOCKING (1)

Performs execution in blocking mode

OPT_CANCEL (0)

Does not consider effect of issued service call **(default)**

OPT_CMPVALUE (2)

Sets value targeted for comparison

OPT_CNDBREAK (4)

Uses conditional break mechanism of debugging tool

OPT_EXTPARAM (2)

Specifies extension parameter

OPT_GETMAXCNT (1)

Even when the upper limit value is smaller than the variable-length data count, this flag tracks to get the data count.

OPT_NOCNDBREAK (1)

Does not use conditional break for break setting

OPT_NORDT (2)

Does not get register set description table

OPT_PEEK (1)

Gets trace log without deleting it from spool

OPT-RESTART (1)

Restarts target (ignores argument staadr)

OPT_SEARCH_BACKWARD (2)

Search backward (in decreasing address direction) to locate symbol closest to specified value

OPT_SEARCH_COMPLETELY (0)

Searches for only symbol that perfectly matches search key **(default)**

OPT_SEARCH_FORWARD (1)

Search backward (in increasing address direction) to locate symbol closest to specified value

OPT_UNDO (1)

Returns to state before issue.

OPT_VENDORDEPEND (2)

Gets implement-dependent information.

9.4 Constants

9.4.1 Object identification constants

OBJ_SEMAPHORE (1)
Semaphore

OBJ_EVENTFLAG (2)
Event flag

OBJ_DATAQUEUE (3)
Data queue

OBJ_MAILBOX (4)
Mailbox

OBJ_MUTEX (5)
Mutex

OBJ_MESSAGEBUFFER (6)
Message buffer

OBJ_RENDEZVOUSPORT (8)
Rendezvous port

OBJ_RENDEZVOUS (9)
Rendezvous

OBJ_FMEMPOOL (10)
Fixed-length memory pool

OBJ_VMEMPOOL (11)
Variable-length memory pool

OBJ_TASK (12)
Task

OBJ_READYQUEUE (14)
Ready queue

OBJ_TIMERQUEUE (15)
Timer queue

OBJ_CYCLICHANDLER (17)
Cyclic handler

OBJ_ALARMHANDLER (18)
Alarm handler

OBJ_OVERRUNHANDLER (19)
Overrun handler

OBJ_ISR (20)
Interrupt service routine

OBJ_KERNELSTATUS (21)
Kernel information

OBJ_TASKEXCEPTION (22)
Task exception handler

OBJ_CPUEXCEPTION (23)
CPU Exception handler

OBJ_ALL (-1u)

Special constant that denotes all objects

9.4.2 Error constants

E_CONSIST (-225)

Consistency was not assured (however, it is not handled as an error if ***FLG_NOCONSISTENCE*** is set).

E_EXCLUSIVE (-226)

Another request is already issued. The function could not receive a new request until execution of the previous request ends.

E_FAIL (-227)

The operation failure was caused by some reason (although the operation could be continued)

E_ID (-146)

The specified object ID was invalid.

E_NOID (-162)

Count of IDs form automatic assignment was insufficient.

E_NOMEM (-161)

The request could not be executed due to insufficient host memory.

E_NOSPT (-137)

An unsupported operation was executed.

E_OBJ (-169)

The targeted object on teh target was inoperative.

E_OK (0)

Normally ended.

E_PAR (-145)

A parameter value was invalid.

E_SYS (-133)

An irrecoverable (fatal) error occurred for some reason.

E_TMOUT (-178)

The process timed out (when ***OPT_BLOCKING*** specified).

E_ID (-18)

The specified kernel object ID was invalid.

ET_MACV (-26)

An invalid memory region on the target was accessed.

ET_NOEXS (-42)

The targeted object was not found on the target.

ET_NOMEM (-33)

The request could not be executed due to insufficient memory on teh target.

ET_OACV (-27)

An illegal target on an target was accessed (tskid < 0).

ET_OBJ (-41)

The targeted object on the target was inoperative.

9.4.3 Break constants

BRK_ACCESS (2)
Sets access break.

BRK_DISPATCH (3)
Sets break for task dispatcher (after execution)

BRK_ENTER (0)
Places break at starting position (***BRK_DISPATCH***, ***BRK_SVC***)

BRK_EXECUTE (1)
Sets execution break.

BRK_LEAVE (128)
Places break at escape position (***BRK_DISPATCH***, ***BRK_SVC***)

BRK_REPORT (32)
Report only (and does not perform break)

BRK_SVC (4)
Breaks with SVC.

BRK_SYSTEM (0)
Stops entire system when break occurs.

BRK_TASK (64)
Stops only task when break occurs.

9.4.4 Log constants

Log type - Object

LOG_TYP_INTERRUPT (1)
Interrupt

LOG_TYP_ISR (2)
Interrupt service routine

LOG_TYP_TIMERHDR (3)
Timer event handler

LOG_TYP_CPUEXC (4)
CPU exception

LOG_TYP_TSKEXC (5)
Task exception

LOG_TYP_TSK STAT (6)
Task status

LOG_TYP_DISPATCH (7)
Task dispatch

LOG_TYP_SVC (8)
Service call

LOG_TYP_COMMENT (9)
Comment (log consisting of character string only; to be written mainly by user)

Log type - Break method

LOG_INSTRUCTION (0)
Instruction

LOG_DATA (4)
Data

Log type - Break conditions

LOG_READ (8)
Read

LOG_WRITE (16)
Write

LOG_MODIFY (32)
Modification (Read Modify Write)

Log mechanism - Configuration setup

LOG_HARDWARE (0)
Uses TIF-based hardware log mechanism for getting

LOG_SOFTWARE (1)
Uses software-based log mechanism executed by RIM alone, for getting

LOG_BUFFUL_STOP (0)
Stops getting log when buffer full

LOG_BUFFUL_CALLBACK (2)
Executes callback function when buffer full

LOG_BUFFUL_FORCEEXEC (4)
Continues getting by discarding oldest data when buffer full

Report events

EV_BUFFER_FULL (1)
The trace buffer is full.

EV_STOP (2)
The trace log function is stopped.

EV_REPORT (4)
The report conditions specified by ***tif_sta_log*** are satisfied.

Dispatch type

DSP_NORMAL (0)
Dispatch from task context

DSP_NONTSKCTX (1)
Dispatch from interrupt process or CPU exception

9.4.5 Other constants

ADR_SYSTEMSTART (0)

Restarts target

CND_CURTSKID (0)

Generates expression in which task ID used as condition

ID_ALL (-1)

Targets all IDs

ID_NONTSKCTX (-127)

Targets nontask context

9.5 Key Code List of Getting Information

First key Value [type]
Explanation of the information that this key can get

.Second key Value [type]
Explanation of the information that this key can get

.Third key Value [type]
Explanation of the information that this key can get

.Fourth key Value [type]
Explanation of the information that this key can get

RIF 4_H

.UNIT 20_H

.OBJ 1_H [1]
Supports the "getting Object status" functional unit.

.LOG 2_H [1]
Supports the "getting execution history" functional unit.

.SVC 3_H [1]
Supports the "service call invocation" functional unit.

.BRK 4_H [1]
Supports the "break setting" functional unit.

.CND 5_H [1]
Supports the "getting break condition" functional unit.

.CTX 6_H [1]
Supports the "getting context" functional unit.

RIF 4_H

.RIF_REF_OBJ 1_H

.FLG_NOCONSISTENCE 1_H [1]
The "***FLG_NOCONSISTENCE***" flag is available.

.FLG_NOSYSTEMSTOP 2_H [1]
The "***FLG_NOSYSTEMSTOP***" flag is available.

.OPT_VENDORDEPEND 10_H [1]
The "***OPT_VENDORDEPEND***" option is available.

.OPT_GETMAXCNT 11_H [1]
The "***OPT_GETMAXCNT***" option is available.

.STATICPARAMETER 12_H

.OBJ_SEMAPHORE 80_H [T]
This structure has semaphore information that is statically determinative.

.OBJ_EVENTFLAG 81_H [T]
This structure has event flag information that is statically determinative.

.OBJ_DATAQUEUE 82_H [T]
This structure has data queue information that is statically determinative.

.OBJ_MAILBOX 83_H [T]
This structure has mailbox information that is statically determinative.

.OBJ_MUTEX 84_H [T]

This structure has mutex information that is statically determinative.

.OBJ_MESSAGEBUFFER 85_H [T]

This structure has message box information that is statically determinative.

.OBJ_RENDEZVOUSPORT 86_H [T]

This structure has rendezvous port information that is statically determinative.

.OBJ_RENDEZVOUS 87_H [T]

This structure has rendezvous information that is statically determinative.

.OBJ_FMEMPOOL 88_H [T]

This structure has fixed-length memory pool information that is statically determinative.

.OBJ_VMEMPOOL 89_H [T]

This structure has variable-length memory pool information that is statically determinative.

.OBJ_TASK $8A_H$ [T]

This structure has task information that is statically determinative.

.OBJ_READYQUEUE $8B_H$ [T]

This structure has ready queue information that is statically determinative.

.OBJ_TIMERQUEUE $8C_H$ [T]

This structure has timer queue information that is statically determinative.

.OBJ_CYCLICHANDLER $8D_H$ [T]

This structure has cyclic handler information that is statically determinative.

.OBJ_ALARMHANDLER $8E_H$ [T]

This structure has alarm handler information that is statically determinative.

.OBJ_OVERRUNHANDLER $8F_H$ [T]

This structure has overrun handler information that is statically determinative.

.OBJ_ISR 90_H [T]

This structure has interrupt service routine information that is statically determinative.

.OBJ_KERNELSTATUS 91_H [T]

This structure has kernel information that is statically determinative.

.OBJ_TASKEXCEPTION 92_H [T]

This structure has task exception information that is statically determinative.

.OBJ_CPUEXCEPTION 93_H [T]

This structure has CPU exception information that is statically determinative.

RIF	04_H
.RIF_GET_RDT	02_H
.REGISTER	2_H
.SIZE	04_H [W]
Size (in bytes) of enough region for register storage	
.CONTEXT	12_H
.SIZE	04_H [W]
Size (in bytes) of enough region for context storage	
RIF	04_H
.RIF_GET_CTX	03_H
.FLG_NOCONSISTENCE	01_H [1]
The "***FLG_NOCONSISTENCE***" flag is available.	
.FLG_NOSYSTEMSTOP	02_H [1]
The "***FLG_NOSYSTEMSTOP***" flag is available.	
.OPT_APPCONTEXT	10_H [1]
The "***OPT_APPCONTEXT***" option is available.	
RIF	04_H
.RIF_SET_CTX	13_H
.FLG_NOSYSTEMSTOP	02_H [1]
The "***FLG_NOSYSTEMSTOP***" flag is available.	
.OPT_APPCONTEXT	10_H [1]
The "***OPT_APPCONTEXT***" option is available.	
RIF	04_H
.RIF_CAL_SVC	04_H
.FLG_NOREPORT	03_H [1]
The "***FLG_NOREPORT***" flag is available.	
.OPT_BLOCKING	10_H [1]
The "***OPT_BLOCKING***" flag is available.	
.NONBLOCKING	12_H [1]
A nonblocking SVC issue is supported.	
RIF	04_H
.RIF_CAN_SVC	05_H [1]
rif_can_svc is implemented.	
.OPT_CANCEL	10_H [1]
The "***OPT_CANCEL***" option is available.	
.OPT_UNDO	11_H [1]
The "***OPT_UNDO***" option is available.	
RIF	04_H
.RIF_CAL_SVC	06_H
RIF	04_H
.RIF_REF_SVC	07_H
RIF	04_H
.RIF_RRF_SVC	08_H

RIF 04_H

.RIF_SET_BRK 09_H

.FLG_NOREPORT 03_H [1]

The "***FLG_NOREPORT***" flag is available.

.FLG_AUTONUMBERING 04_H [1]

The "***FLG_AUTONUMBERING***" flag is available.

.OPT_NOCNDBREAK 10_H [1]

The "***OPT_NOCNDBREAK***" option is available.

.OPT_EXTPARAM 11_H [1]

The "***OPT_EXTPARAM***" option is available.

RIF 04_H

.RIF_DEL_BRK $0A_H$

RIF 04_H

.RIF_REP_BRK $0B_H$

RIF 04_H

.RIF_REF_BRK $0C_H$

RIF 04_H

.RIF_REF_CND $0D_H$

RIF 04_H

.RIF_SET_LOG $0E_H$

.FLG_AUTONUMBERING 04_H [1]

The "***FLG_AUTONUMBERING***" flag is available.

.OPT_BUFFUL_STOP 10_H [1]

The "***OPT_BUFFUL_STOP***" option is available.

.OPT_BUFFUL_FORCEEXEC 11_H [1]

The "***OPT_BUFFUL_FORCEEXEC***" option is available.

RIF 04_H

.RIF_DEL_LOG $0F_H$

RIF 04_H

.RIF_STA_LOG 10_H

RIF 04_H

.RIF_STP_LOG 11_H

RIF 04_H

.RIF_GET_LOG 12_H

.OPT_PEEK 10_H [1]

The "***OPT_PEEK***" option is available.

.STRUCT_SVC 11_H [1]

Uses a dedicated structure for the start/end of ***LOG_TYP_SVC***.

RIF 04_H

.RIF_CFG_LOG 13_H

CFG 7_H

.CPUEXCEPTION 17_H

.MIN 1_H[W]

Minimum value of the internal exception factor that the kernel uses

.MAX 2_H [W]

Maximum value of the internal exception factor that the kernel uses

.NUM 3_H [W]

Count of internal exception factor that the kernel uses

.SYSTIM 20_H

.TICK_D 1_H [W]

Denominator when the timer resolution is expressed in milliseconds (ms)

.TICK_N 2_H [W]

Numerator when the timer resolution is expressed in milliseconds (ms)

.UNIT_D 3_H [W]

Denominator when the timer unit is expressed in milliseconds (ms)

.UNIT_N 4_H [W]

Numerator when the timer unit is expressed in milliseconds (ms)

.LOGTIM 21_H

.TICK_D 1_H [W]

Denominator when the log time resolution is expressed in milliseconds (ms)

.TICK_N 2_H [W]

Numerator when the log time resolution is expressed in milliseconds (ms)

.UNIT_D 3_H [W]

Denominator when the log time unit is expressed in milliseconds (ms)

.UNIT_N 4_H [W]

Numerator when the log time unit is expressed in milliseconds (ms)

.INTERRUPT 22_H

.MIN 1_H [W]

Minimum value of the external interrupt factor that the kernel uses

.MAX 2_H [W]

Maximum value of the external interrupt factor that the kernel uses

.NUM 3_H [W]

Count of external interrupt factor that the kernel uses

.ISR 25_H

.MIN 1_H [W]

Minimum ISR number offered by kernel

.MAX 2_H [W]

Maximum ISR number offered by kernel

.NUM 3_H [W]

Number of ISRs offered by kernel

.MAKER 23_H [W]

Manufacturer code

.PRIORITY		24_H
	.MIN	1_H [W]
	Minimum value of the priority levels available to the kernel	
	.MAX	2_H [W]
	Maximum value of the priority levels available to the kernel	
.OBJ_SEMAPHORE		80_H
	.MIN	1_H [W]
	Minimum value of assignable IDs	
	.MAX	2_H [W]
	Maximum value of assignable IDs	
.OBJ_EVENTFLAG		81_H
	.MIN	1_H [W]
	Minimum value of assignable IDs	
	.MAX	2_H [W]
	Maximum value of assignable IDs	
.OBJ_DATAQUEUE		82_H
	.MIN	1_H [W]
	Minimum value of assignable IDs	
	.MAX	2H [W]
	Maximum value of assignable IDs	
.OBJ_MAILBOX		83H
	.MIN	1_H [W]
	Minimum value of assignable IDs	
	.MAX	2_H [W]
	Maximum value of assignable IDs	
.OBJ_MUTEX		84_H
	.MIN	1_H [W]
	Minimum value of assignable IDs	
	.MAX	2_H [W]
	Maximum value of assignable IDs	
.OBJ_MESSAGEBUFFER		85_H
	.MIN	1_H [W]
	Minimum value of assignable IDs	
	.MAX	2_H [W]
	Maximum value of assignable IDs	
.OBJ_RENDEZVOUSPORT		86_H
	.MIN	1_H [W]
	Minimum value of assignable IDs	
	.MAX	2_H[W]
	Maximum value of assignable IDs	
.OBJ_RENDEZVOUS		87_H
	.MIN	1_H [W]
	Minimum value of assignable IDs	

.MAX		2_H [W]
	Maximum value of assignable IDs	
.OBJ_FMEMPOOL		88_H
.MIN		1_H [W]
	Minimum value of assignable IDs	
.MAX		2_H [W]
	Maximum value of assignable IDs	
.OBJ_VMEMPOOL		89_H
.MIN		1_H [W]
	Minimum value of assignable IDs	
.MAX		2_H [W]
	Maximum value of assignable IDs	
.OBJ_TASK		$8A_H$
.MIN		1_H [W]
	Minimum value of assignable IDs	
.MAX		2_H [W]
	Maximum value of assignable IDs	
.OBJ_CYCLICHANDLER		$8D_H$
.MIN		1_H [W]
	Minimum value of assignable IDs	
.MAX		2_H [W]
	Maximum value of assignable IDs	
.OBJ_ALARMHANDLER		$8E_H$
.MIN		1_H [W]
	Minimum value of assignable IDs	
.MAX		2_H [W]
	Maximum value of assignable IDs	
.PRVER		$A0_H$ [S]
	Version number of the kernel	
.SPVER		$A1_H$ [S]
	ITRON Specification version number	
TIF		05_H
.TIF_ALC_MBH		01_H
TIF		05_H
.TIF_ALC_MBT		02_H [1]
	Supports this function.	
TIF		05_H
.TIF_FRE_MBH		03_H
TIF		05_H
.TIF_FRE_MBT		04_H [1]
	Supports this function.	

TIF 05_H

.TIF_GET_MEM 05_H

.FLG_NOCONSISTENCE 01_H [1]

Supports the "***FLG_NOCONSISTENCE***" flag.

.FLG_NOSYSTEMSTOP 02_H [1]

Supports the "***FLG_NOSYSTEMSTOP***" flag.

TIF 05_H

.TIF_GET_BLS 06_H

.FLG_NOCONSISTENCE 01_H [1]

Supports the "***FLG_NOCONSISTENCE***" flag.

.FLG_NOSYSTEMSTOP 02_H [1]

Supports the "***FLG_NOSYSTEMSTOP***" flag.

TIF 05_H

.TIF_SET_MEM 07_H

.FLG_NOCONSISTENCE 01_H [1]

Supports the "***FLG_NOCONSISTENCE***" flag.

.FLG_NOSYSTEMSTOP 02_H [1]

Supports the "***FLG_NOSYSTEMSTOP***" flag.

TIF 05_H

.TIF_SET_BLS 08_H

.FLG_NOCONSISTENCE 01_H [1]

Supports the "***FLG_NOCONSISTENCE***" flag.

.FLG_NOSYSTEMSTOP 02_H [1]

Supports the "***FLG_NOSYSTEMSTOP***" flag.

TIF 05_H

.TIF_SET_POL 09_H [1]

Supports this function.

.FLG_AUTONUMBERING 04_H [1]

Supports the "***FLG_AUTONUMBERING***" flag.

.OPT_CMPVALUE 10_H [1]

Supports the "***OPT_CMPVALUE***" option.

TIF 05_H

.TIF_DEL_POL $0A_H$ [1]

Supports this function.

TIF 05_H

.TIF_REP_POL $0B_H$

TIF 05_H

.TIF_GET_REG $0C_H$

.FLG_NOCONSISTENCE 01_H [1]

Supports the "***FLG_NOCONSISTENCE***" flag.

.FLG_NOSYSTEMSTOP 02_H [1]

Supports the "***FLG_NOSYSTEMSTOP***" flag.

TIF 05_H

.TIF_SET_REG $0D_H$

TIF 05_H

.TIF_STA_TGT $0E_H$

.OPT_RESTART 10_H [B]

OPT_RESTART is available.

TIF 05_H

.TIF_STP_TGT $0F_H$ [1]

Supports this function.

TIF 05_H

.TIF_BRK_TGT 10_H [1]

Supports this function.

TIF 05_H

.TIF_CNT_TGT 11_H

TIF 05_H

.TIF_SET_BRK 13_H

.FLG_AUTONUMBERING 04_H [1]

Supports the "***FLG_AUTONUMBERING***" flag.

.OPT_CNDBREAK 10_H [1]

Supports the "***OPT_CNDBREAK***" option.

.BRK_ACCESS 11_H [1]

An access break is available.

TIF 05_H

.TIF_DEL_BRK 14_H

TIF 05_H

.TIF_REP_BRK 12_H

Supports this function.

.FLG_AUTONUMBERING 04_H [1]

Supports the "***FLG_AUTONUMBERING***" flag.

TIF 05_H

.TIF_REF_SYM 15_H

TIF 05_H

.TIF_RRF_SYM 16_H [1]

Supports this function.

.OPT_SEARCH_FORWARD 10_H [1]

The "***OPT_SEARCH_FORWARD***" option is available.

.OPT_SEARCH_BACKWARD 11_H [1]

The "***OPT_SEARCH_BACKWARD***" option is available.

.OPT_SEARCH_COMPLETELY 12_H [1]

The "***OPT_SEARCH_COMPLETELY***" option is available.

TIF 05_H

.TIF_CAL_FNC 17_H [1]

Supports this function.

.FLG_NOREPORT 03_H [1]
Supports the "***FLG_AUTONUMBERING***" flag.

.OPT_BLOCKING 11_H [1]
Supports the "***OPT_NONBLOCKING***" option.

.NONBLOCKING 12_H [1]
Supports a nonblocking function call.

TIF 05_H

.TIF_REP_FNC 18_H [1]
Supports this function.

TIF 05_H

.TIF_SET_LOG 19_H [1]
Supports this function.

.FLG_NOREPORT 03_H [1]
The "***FLG_NOREPORT***" flag is available.

.FLG_AUTONUMBERING 04_H [1]
Supports the "***FLG_AUTONUMBERING***" flag.

.OPT_BUFFUL_FORCEEXEC 11_H [1]
The "***OPT_BUFFUL_FORCEEXEC***" option is available.

.OPT_BUFFUL_CALLBACK 12_H [1]
The "***OPT_BUFFUL_CALLBACK***" option is available.

.LOG_INSTRUCTION 13_H [1]
The log type "***LOG_INSTRUCTION***" is available.

.LOG_DATA 14_H [1]
The log type "***LOG_DATA***" is available.

.LOG_READ 15_H [1]
LOG_READ is available.

.LOG_WRITE 16_H [1]
LOG_WRITE is available.

.LOG_MODIFY 17_H [1]
LOG_MODIFY is available.

TIF 05_H

.TIF_DEL_LOG $1A_H$ [1]
Supports this function.

TIF 05_H

.TIF_STA_LOG $1B_H$ [1]
Supports this function.

TIF 05_H

.TIF_STP_LOG $1C_H$ [1]
Supports this function.

TIF 05_H

.TIF_REP_LOG $1D_H$ [1]
Supports this function.

TIF 05_H

.TIF_GET_LOG $1E_H$ [1]

Supports this function.

.OPT_PEEK 10_H [1]

Supports the ***OPT_PEEK*** option.

DEBUGGER 1_H

.CNDBREAK 1_H

.NUM 3_H [W]

Count of conditional breaks that can be set (0: not supported)

.LOG 2_H

.NUM 3_H [W]

Count of hardware logs that can be set (0: not supported)

.NAME 80_H [S]

Unique character(s) for debugging tool identification

HOST 2_H

.ENDIAN 1_H [W]

Host computer's endian (0: little; 1: big)

.NAME 80_H [S]

Unique character(s) for host computer identification

TARGET 3_H

.ENDIAN 1_H [W]

Target computer's endian (0: little; 1: big)

.REGISTER 2_H

.NUM 3_H [W]

Count of target computer registers

.NAME 80_H [S]

Unique character(s) for target device identification

OS 8_H

.NAME 80_H [S]

Unique character(s) for target OS identification ("ITRON")

Appendix A

Member List

In honor of persons who contributed much to the preparation of the specification, the names of the ITRON Debugging Interface Specification Working Group members are listed below (in alphabetical order):

Table 26: Member List

Name	Organization
Kouei Abe	NEC Microcomputer Technology, Ltd.
Kazuyuki Iori	Midoriya Electric Co., Ltd. Design Center
Norihisa Iga	NEC Software Product Engineering Laboratory
Hidehiro Ishii	YDC Corporation
Kazutoyo Inamitsu	Fujitsu Devices Inc.
Shigeto Iwata	eSOL Co., Ltd.
Kazuyuki Uchida	Matsushita Electric Industrial Co., Ltd.
Shinnichiro Eto	Matsushita Information System Reserch Laboratory Hiroshima Co., Ltd.
Yoshinori Kaneko	NEC Microcomputer Technology, Inc.
Takao Kawai	AI Corporation Inc.
Masahiro Kawakami	Oki Electric Industry Co., Ltd.
Motoko Kishitani	MITSUBISHI Electric Semiconductors Systems Corporation
Kenji Kudo	Fujitsu Devices Inc.
Hisaya Kuroda	Sophia Systems Co., Ltd.
Yoshiyuki Koizumi	TOSHIBA Corporation
Masahiko Kohda	Advanced Data Controls Corp.
Shirou Kojima	Fujitsu Devices Inc.
Yasuhiro Kobayashi	Fujitsu Limited
Masaki Gondo	eSOL Co., Ltd.
Masaaki Sakuraba	Fujitsu Devices Inc.
Shigeru Sasaki	Toyota Motor Corporation
Takako Sato	NEC Microcomputer Technology, Ltd.
Shinji Shibata	Firmware Systems Inc.

Table 26: Member List

Name	Organization
Masahiro Shukuguchi	Mitsubishi Electric Micro-Computer Application Software Co., Ltd.
Tetsuo Takagi	DENSO Create Inc.
Hiroaki Takada	Toyohashi Univ. of Technology
Chiharu Takei	YDC Corporation
Tohru Takeuchi	TRON Association
Yuichi Tsukada	Cats Corp.
Shoji Nagata	Matsushita Electric Industrial Co., Ltd.
Satoshi Nagamine	Matsushita Electric Industrial Co., Ltd.
Shigeki Nankaku	Mitsubishi Electric Corporation
Yukio Nomoto	BITRAN Corporation
Shinnichi Hashimoto	Access Co., Ltd.
Yasushi Hasegawa	Fujitsu Devices Inc.
Shinichi Hayakashi	TOSHIBA Corporation
Tadakatsu Masaki	Matsushita Information System Reserch Laboratory Hiroshima Co., Ltd.
Yukihiro Mizukoshi	Oki Electric Industry Co., Ltd.
Satoshi Midorikawa	Midoriya Electric Co., Ltd.
Hiroyuki Muraki	MITSUBISHI Electric Semiconductors Systems Corporation
Kiyoshi Motoki	Fujitsu Devices Inc.
Toshiko Morimoto	YDC Corporation
Shinjiro Yamada	Hitachi Ltd.
Masaru Yamanaka	QNX Software Systems Ltd. Japan
Tatsuo Yamada	Motorola Inc.
Ichiro Yamamoto	LIGHTWELL Co., Ltd.
Akira Yokozawa	TOSHIBA Corporation
Munehiro Yoshida	MITSUBISHI Electric Semiconductors Systems Corporation
Miyoko Yoshimura	eSOL Co., Ltd.
Takayuki Wakabayashi	Toyohashi Univ. of Technology

Appendix B

Numerics

A

B

C

D

E

F

G

H

I

P

R

S

T

U

V

W