T-Monitor Specification

T-Monitor 1.B0.01

Aug. 2002

Contents

History of Revisions

[Version 1.B0.01]

- Added a `Kill` command to the monitor commands, for forcing process termination.
- Added an extended services execution function (`tm_extsvc`) to the monitor service functions for program support.

Chapter 1

Overview

T-Monitor is the basic monitor program in T-Engine, intended to be resident in ROM for provision of the following functions.

1. System Functions
   - Hardware Initialization
   - System Startup
   - Exception/Interrupt/Trap Handling Functions
2. Debugging Functions
   - Memory Operations
   - Register Operations
   - IO Operations
   - Disassembly
   - Program and Data Loading
   - Program Execution
   - Breakpoint Operations
   - Trace Execution
   - Disk Read, Write, and Boot Operations
3. Program Support Functions
   - Monitor Service Functions

T-Monitor is to a large extent dependent on the CPU and T-Engine board hardware. The specifications given here are therefore limited to common aspects. More detailed T-Monitor specifications are defined for each T-Engine board implementation.

Chapter 2

System Functions

2.1 Hardware Initialization

When the system is reset, T-Monitor is started initially and performs the following processing.

1. Hardware Initialization
   Performs the basic hardware initialization as necessary for system startup.

2. Hardware Self-Diagnosis
   Performs memory checking and other necessary self-diagnosis. If a problem is detected in the system, T-Monitor reports the problem and aborts system startup. If a serial port is available for debugging use, T-Monitor outputs error messages to this serial port and waits for monitor command input.

2.2 System Startup

When hardware initialization is completed, starts the system in accord with either of the following startup modes.

1. Automatic Startup Mode
   T-Monitor searches in order the disk drives attached to the system, boots from the first bootable disk discovered and starts up the system (same as `BootDisk` command). If no bootable disk is found, it starts the system in ROM. If there is no system in ROM, it waits for monitor command input.

2. Monitor Startup Mode
   T-Monitor waits for monitor command input, without starting up the system. Depending on the implementation, there may be other startup modes besides the above. Startup mode can be set on the board DIP switches or in data stored in nonvolatile memory. The specific method is implementation-dependent.

2.3 Exception/Interrupt/Trap Handling Functions

T-Monitor keeps a single vector table for all exceptions, interrupts and traps (EIT), executing the handlers registered there. Details of the vector table are specified separately for each implementation. For undefined vectors, a T-Monitor exception handler is run by default as set in initialization processing.

When an undefined EIT is raised, this handler outputs the information to the serial port used for debugging and passes control to T-Monitor. Essentially no interrupts are used in T-Monitor. All interrupts are disabled while monitor operations are in progress.

Chapter 3

Debugging Functions

3.1 Console Connection

A debugging console is connected to the T-Engine debugging serial port (RS232C) and used for debugging. The serial communications specifications while a console is connected are as follows.

Baud rate	: 38,400 bps (or 115,200 bps)
Data length	: 8 bits
Stop bit	: 1 bit
Parity	: none
Flow control	: XON/XOFF
Character code	: ASCII code
End of received line	: CR (0x0d)
End of sent line:	: CRLF (0x0d, 0x0a)

- Baud rate is set on the board by DIP switches, or in nonvolatile memory data, etc.; the actual setting method is implementation-dependent.

3.2 Command Format

1. Commands T-Monitor displays the prompt "`TM>`" on the debugging console and waits for command input.
   Commands have the following format. One line can contain up to 256 characters.

   ```
   <command name> <parameter 1>, <parameter 2>, ... <CR/LF>
   ```

   - A `<Command name>` is not case-sensitive.
   - The `<command name>` and first `<parameter>` are separated by a blank space or tab.
   - Each `<parameter>` is separated by '`,`'. If a parameter is omitted, entering '`,`' only for that parameter maintains the proper correspondence of values to parameter names.

   Multiple commands can be entered on the same line separated by '`;`'. Lines beginning with '`*`' are ignored as comment lines. A blank line with CR/LF only is ignored.

2. Control Codes

   The following control codes from the debugger console are supported.

`Ctrl-X` (0x18), `Ctrl-U` (0x15)	Undo (delete) line entry
`Ctrl-H` (0x08), `DEL` (0x7f)	Undo (delete) 1 character entry
`Ctrl-S` (0x13, XOFF)	Pause display scroll
`Ctrl-Q` (0x11, XON)	Resume display scroll
`Ctrl-C` (0x03)	Kill command
`Ctrl-F` (0x06), `ESC [ C`	Move cursor right (→)
`Ctrl-B` (0x02), `ESC [ D`	Move cursor left (←)
`Ctrl-P` (0x10), `ESC [ A`	Call up previous line (↑)
`Ctrl-N` (0x0e), `ESC [ B`	Call up next line (↓)
`Ctrl-K` (0c0b)	Delete after cursor

3. Numeric Values

   Numeric values are written using the following notation.

Hexadecimal	`H'`⟨hexadecimal string⟩	`h'`⟨hexadecimal string⟩
	`0x`⟨hexadecimal string⟩	⟨0 to 9 ⟩ ⟨ hexadecimal string ⟩
Decimal	`D'`⟨numeric string⟩	`d'`⟨numeric string⟩
Octal	`Q'`⟨octal numeric string⟩	`q'`⟨octal numeric string⟩
Binary	`B'`⟨binary numeric string⟩	`b'`⟨binary numeric string⟩

⟨Number⟩:	'0' to '9'
⟨Binary digits⟩:	'0' , '1'
⟨Octal digits⟩:	'0' to '7'
⟨Hexadecimal digits⟩:	'0' to '9', 'A' to 'F', 'a' to 'f'

   A numeric string with no prefix is considered to be a hexadecimal string.

4. Character Strings

   A character string is any series of characters enclosed by '`"`'; these are used as parameters with some commands.

5. Register Names

   A register name is a special symbol dependent on the CPU; these are used as parameters with some commands.

6. Expressions

   An expression consists of numeric values or register names joined by operators '`+`', '`-`', '`*`', and '`/`'. Expressions are used as parameters with some commands. Operations including those with '`*`' and '`/`' are always performed from left to right. A register name means the value of the register it designates.

   [Example]

`8000 + d'250`	—	`H'80FA`
`1000 + 100 * 2`	—	(`H'1000` + `H'100`) ∗ 2
`R0 + 100`	—	Register `R0` value + `h'100`

   '`&`' is an operator indicating indirect reference, with the value of the expression up to that point being the memory address contents (word). Additional layers of indirect reference are possible by stringing together multiple '`&`'.

   [Example]

`AC000000 + 4`	—	`H'AC0000004` memory contents
`R0 & + 8`	—	A memory contents of the address which is the register `R0` value memory contents +8

Expressions may be used in value parameters (address, size, etc.) with all commands.

3.3 List of Commands

The following notation is used in the command explanations.

`(foo)`	Short form of command
`[foo]`	Optional designation
`[foo]..`	Optional repeated designation
`foo|bar`	Choice of designations
`Byte`	8 bits
`Half-word`	16 bits
`Word`	32 bits

The T-Monitor commands are listed below.

Command name		Description
Dump	(D)	Dump Memory
DumpByte	(DB)	Dump Memory
DumpHalf	(DH)	Dump Memory
DumpWord	(DW)	Dump Memory
Modify	(M)	Modify Memory
ModifyByte	(MB)	Modify Memory
ModifyHalf	(MH)	Modify Memory
ModifyWord	(MW)	Modify Memory
Fill	(F)	Fill Memory
FillByte	(FB)	Fill Memory
FillHalf	(FH)	Fill Memory
FillWord	(FW)	Fill Memory
SearchChar	(SC)	Search Memory
SearchByte	(SCB)	Search Memory
SearchHalf	(SCH)	Search Memory
SearchWord	(SCW)	Search Memory
Compare	(CMP)	Compare Memory
Move	(MOV)	Move Memory
InputByte	(IB)	Input from IO Port
InputHalf	(IH)	Input from IO Port
InputWord	(IW)	Input from IO Port
OutputByte	(OB)	Output to IO Port
OutputHalf	(OH)	Output to IO Port
OutputWord	(OW)	Output to IO Port
Disassemble	(DA)	Disassemble
Register	(R)	Dump/Modify Register
Go	(G)	Execute Program
BreakPoint	(B)	Set Breakpoint
BreakClear	(BC)	Clear Breakpoint
Step	(S)	Step Trace
Next	(N)	Next Trace
BackTrace	(BTR)	Back Trace
Load	(LO)	Load Program/Data
ReadDisk	(RD)	Read Disk
WriteDisk	(WD)	Write Disk
InfoDisk	(ID)	Display Disk Information
BootDisk	(BD)	Boot from Disk
Kill	(KILL)	Kill Process
Help	(H) (?)	Display Help Message
Exit	(EX)	Exit Monitor

Dump/DumpByte/DumpHalf/DumpWord

Dump Memory

[Format]

```
Dump     (D)  [<start address>][, {<end address>|#<data count>}]
DumpByte (DB) [<start address>][, {<end address>|#<data count>}]
DumpHalf (DH) [<start address>][, {<end address>|#<data count>}]
DumpWord (DW) [<start address>][, {<end address>|#<data count>}]
```

[Description]

Displays the memory contents in the designated address range in the following ⟨units⟩.

Dump, DumpByte	Byte unit	⟨data count⟩ is in bytes
DumpHalf	Half-word unit	⟨data count⟩ is in half-words
DumpWord	Word unit	⟨data count⟩ is in words

The address range for the operation is either of the following.

⟨start address⟩ to ⟨end address⟩ + ⟨unit⟩ − 1
⟨start address⟩ to ⟨start address⟩ + ⟨data count⟩ * ⟨unit⟩ − 1

If ⟨start address⟩ or ⟨end address⟩ is not at a ⟨unit⟩ byte boundary, the address is aligned with a boundary.
If ⟨start address⟩ is omitted, the dump starts from the next address following the range of the previous Dump Memory command.
If ⟨end address⟩ is omitted, 64 bytes of data are displayed regardless of ⟨unit⟩.
Access is made only to memory in the designated range, in the designated unit. No write access is made to memory by this command.

[Typical Usage]

```
TM> Dump AC100000
AC100000: 00 09 80 04 45 03 E0 05 E0 09 00 0A 00 0B 56 0C ....E.........V.
AC100010: 04 0D 00 0E 03 01 E0 03 E1 05 E8 FF 8E 00 00 00 ................
AC100020: 1B D6 1B D6 1B D6 1B D6 9E 00 00 00 8E 00 01 C0 ................
AC100030: C6 16 D0 0C 00 FF 80 46 80 10 00 00 88 12 22 4C .......F......''L

TM> DumpHalf AC100000, AC100010
AC100000: 0900 0480 0345 05E0 09E0 0A00 0B00 0C56  ....E.........V.
80100010: 040D                                     ..

TM> DumpWord AC100000, #9
AC100000: 04800900 05E00345 0A0009E0 0C560B00  ....E.........V.
AC100010: 0E000D04 03E00103 FFE805E1 0000008E  ................
AC100020: D61BD61B                             ....
```

Modify/ModifyByte/ModifyHalf/ModifyWord

Modify Memory

[Format]

```
Modify     (M)  [<start address>][, <set value>]..
ModifyByte (MB) [<start address>][, <set value>]..
ModifyHalf (MH) [<start address>][, <set value>]..
ModifyWord (MW) [<start address>][, <set value>]..
```

[Description]

Modifies the memory at ⟨start address⟩ in the following ⟨unit⟩.

Modify, ModifyByte	Byte unit
ModifyHalf	Half-word unit
ModifyWord	Word-unit

If ⟨start address⟩ is not a ⟨unit⟩ byte boundary, the address is aligned with a boundary.
If ⟨start address⟩ is omitted, modification starts from the next address following the previous Modify Memory.
An ⟨expression⟩ or ⟨character string⟩ can be designated in ⟨set value⟩.
An ⟨expression⟩ is set as a ⟨unit⟩ byte value. A ⟨character string⟩ is set as a byte data array packed with zeros at the end to align it with a ⟨unit⟩ byte boundary. Consecutive ⟨set value⟩ designations may be made for up to a maximum of 128 bytes. If ⟨set value⟩ is omitted, the memory contents are modified interactively. In interactive mode, the following input has special meaning.

'.'	Stop command execution
'^'	Go back one address
(CR/LF only)	Go to next address without setting

Access is made only to memory in the designated range, in the designated unit. Memory is not read by this command, except in interactive mode.

[Typical Usage]

```
TM> ModifyByte AC100000
AC100000: 00 -> 12
AC100001: 09 -> 34
AC100002: 80 -> ^
AC100001: 34 -> .

TM> ModifyHalf AC100000, "ABCD", 56, 78

TM> ModifyWord AC100000
AC100000: 44434241 ->
AC100004: 00780056 -> .
```

Fill/FillByte/FillHalf/FillWord

Fill Memory

[Format]

```
Fill     (F)  <start address>,{<end address>|#<data count>},<set value>[,<set value>]..
FillByte (FB) <start address>,{<end address>|#<data count>},<set value>[,<set value>]..
FillHalf (FH) <start address>,{<end address>|#<data count>},<set value>[,<set value>]..
FillWord (FW) <start address>,{<end address>|#<data count>},<set value>[,<set value>]..
```

[Description]

Fills the designated memory range with a ⟨set value⟩ array repeated in the following ⟨unit⟩.

Fill, FillByte	Byte unit	⟨data count⟩ is in bytes
FillHalf	Half-word unit	⟨data count⟩ is in half-words
FillWord	Word unit	⟨data count⟩ is in words

The address range for the operation is either of the following.

⟨start address⟩ to ⟨end address⟩ + ⟨unit⟩ − 1
⟨start address⟩ to ⟨start address⟩ + ⟨data count⟩ * ⟨unit⟩ − 1

If ⟨start address⟩ or ⟨end address⟩ is not at a ⟨unit⟩ byte boundary, the address is aligned with a boundary.
An ⟨expression⟩ or ⟨character string⟩ can be designated in ⟨set value⟩. An ⟨expression⟩ is set as a ⟨unit⟩ byte value. A ⟨character string⟩ is set as a byte data array packed with zeros at the end to align it with a ⟨unit⟩ byte boundary. Consecutive ⟨set value⟩ designations may be made for up to a maximum of 128 bytes.
Access is made only to memory in the designated range, in the designated unit. No memory is read by this command.

[Typical Usage]

```
TM> Fill AC101000, #10, 57

TM> Dump AC101000, #12
AC101000: 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 57 WWWWWWWWWWWWWWWW
AC101010: 00 00                                           ..

TM> FillWord AC101000, AC10101f, 12, 34

TM> Dump AC101000, #22
AC101000: 12 00 00 00 34 00 00 00 12 00 00 00 34 00 00 .......4.......4
AC101010: 12 00 00 00 34 00 00 00 12 00 00 00 34 00 00 .......4.......4
AC101020: 00 00                                        ..
```


SearchChar/SearchByte/SearchHalf/SearchWord

Search Memory

[Format]

```
SearchChar (SC)  <start address>,{<end address>|#<data count>},
                                                <search value>[,<search value>]..
SearchByte (SCB) <start address>,{<end address>|#<data count>},
                                                <search value>[,<search value>]..
SearchHalf (SCH) <start address>,{<end address>|#<data count>},
                                                <search value>[,<search value>]..
SearchWord (SCW) <start address>,{<end address>|#<data count>},
                                                <search value>[,<search value>]..
```

[Description]

Searches the memory contents in the designated address range for the ⟨search value⟩ array in the following ⟨unit⟩, and if the ⟨search value⟩ array is found, displays its initial address. The search ends when up to 64 results have been displayed.

SearchChar, SearchByte	Byte unit	⟨data count⟩ is in bytes
SearchHalf	Half-word unit	⟨data count⟩ is in half-words
SearchWord	Word unit	⟨data count⟩ is in words

The address range for the operation is either of the following.

⟨start address⟩ to ⟨end address⟩ + ⟨unit⟩ − 1
⟨start address⟩ to ⟨start address⟩ + ⟨data count⟩ * ⟨unit⟩ − 1

If ⟨start address⟩ or ⟨end address⟩ is not at a ⟨unit⟩ byte boundary, the address is aligned with a boundary.
An ⟨expression⟩ or ⟨character string⟩ can be designated in ⟨search value⟩. An ⟨expression⟩ is set as a ⟨unit⟩ byte value. A ⟨character string⟩ is set as a byte data array packed with zeros at the end to align it with a ⟨unit⟩ byte boundary. Consecutive ⟨search value⟩ designations may be made for up to a maximum of 128 bytes.
Access is made only to memory in the designated range, in the designated unit. No write access is made to memory by this command.

[Typical Usage]

```
TM> SearchChar AC101000, AC10101f, 12
AC101003:
AC10100B:
AC101013:
AC10101B:

TM> SearchWord AC101000, #20, 12, 34
AC101000:
AC101008:
AC101010:
AC101018:
```

Compare

Compare Memory

[Format]

```
Compare (CMP) <start address>, {<end address>|#<byte count>}, <compare address>
```

[Description]

Compares the memory contents in the designated address range with the memory contents starting from ⟨compare address⟩, displaying addresses having different contents along with the memory contents at those addresses, in byte units. The comparison is stopped when up to 64 results have been displayed. The address range for the operation is either of the following.

⟨start address⟩ to ⟨end address⟩
⟨start address⟩ to ⟨start address⟩ + ⟨byte count⟩ − 1

Access is made only to memory in the designated range, in byte units. No write access is made to memory by this command.

[Typical Usage]

```
TM> Compare AC100000, AC100fff, AC110000

TM> Compare AC100000, AC100fff, AC120000
AC100020: 34 -> AC120000: 00
AC100021: 56 -> AC120000: 00
:       :
```

Move

Move Memory

[Format]

```
Move (MOV) <start address>, {<end address>|#<byte count>}, <destination address>
```

[Description]

Moves the memory contents in the designated address range to ⟨destination address⟩. There may be overlap between the source and destination address ranges.
The address range for the operation is either of the following.

⟨start address⟩ to ⟨end address⟩
⟨start address⟩ to ⟨start address⟩ + ⟨byte count⟩ − 1

This operation does not write to the source memory, and does not read from the destination memory. Access is made only to memory in the designated range, in byte units. No write access is made to the source memory by this command, and no read access is made to the destination memory.

[Typical Usage]

```
TM> Move AC100000, #1000, AC110000
```


InputByte/InputHalf/InputWord

Input from IO Port

[Format]

```
InputByte (IB) <IO address>
InputHalf (IH) <IO address>
InputWord (IW) <IO address>
```

[Description]

Reads and displays data from the designated ⟨IO address⟩ in the following ⟨unit⟩.

InputByte	Byte unit
InputHalf	Half-word unit
InputWord	Word unit

Error results if ⟨IO address⟩ is not a ⟨unit⟩ byte boundary.
Access is made only to the designated IO address, in the designated unit. No write access is made to the IO port by this command.
In a memory-mapped IO system, the designated memory address is accessed as an IO address.

[Typical Usage]

```
TM> InputByte 310
310: 5F
```

OutputByte/OutputHalf/OutputWord

Output to IO Port

[Format]

```
OutputByte (OB) <IO address>,<byte data>
OutputHalf (OH) <IO address>,<half-word data>
OutputWord (OW) <IO address>,<word data>
```

[Description]

Writes data in the following ⟨unit⟩ at the designated ⟨IO address⟩.

OutputByte	Byte unit
OutputHalf	Half-word unit
OutputWord	Word unit

Error results if ⟨IO address⟩ is not a ⟨unit⟩ byte boundary.
Access is made only to the designated IO address, in the designated unit. No read access is made from the IO port by this command.
In a memory-mapped IO system, this command writes to the memory address designated as IO address.

[Typical Usage]

```
TM> OutputHalf 310, 513F
```


Disassemble

Disassemble

[Format]

```
Disassemble (DA) [<start address>][, <instruction steps>]
```

[Description]

Disassembles from the designated ⟨start address⟩ for the designated number of ⟨instruction steps⟩ and displays the result.
If ⟨start address⟩ is omitted, the operation starts from the next address after the previous Disassemble command. In case return was made to the monitor after program execution due to a break, exception or the like, the PC register value at that point becomes the `Disassemble` command ⟨start address⟩.
If ⟨instruction steps⟩ is omitted, disassembly proceeds for 16 steps.

- Disassembly support is implementation-dependent.

[Typical Usage]

```
TM> Disassemble AC1000d8
<Disassembly result>
```

Register

Dump/Modify Register

[Format]

```
Register (R) [<register name>[, <set value>]]
```

[Description]

Modifies the contents of ⟨register name⟩. If ⟨set value⟩ is omitted, displays the contents of ⟨register name⟩. The names that can be designated in ⟨register name⟩ are CPU-dependent. The names are not case-sensitive.
A group of registers can be designated in ⟨register name⟩ by the following single-character designations.

G	:	General registers
C	:	Control/system registers
D	:	DSP registers
F	:	Floating point registers
A	:	All registers

If ⟨register name⟩ is omitted, all general registers are displayed.

- Specific register names are implementation-dependent.

[Typical Usage]

```
TM> Register
<Display of all general registers>
TM> Register C
<Display of all control/system registers>
TM> Register R0, 1234567
TM> Register R0
R0: 01234567
```

Go

Execute Program

[Format]

```
Go (G) [<execution start address>][, <execution end address>]
```

[Description]

Executes a program from the designated ⟨execution start address⟩. The ⟨execution end address⟩ is set as a temporary software breakpoint, with control returning to the monitor when the ⟨execution end address⟩ is reached.
If ⟨execution start address⟩ is omitted, execution starts from the address designated by the current PC register value.
Control is returned from the executed program to the monitor in any of the following cases.

- When the set breakpoint is reached.
- When an exception not supported in the program is raised.
- When a monitor service function causes processing to go from the program to the monitor.

[Typical Usage]

```
TM> Go AC1000d8, AC10434
Break (S) at AC10434
```

- "`At XXXX`" is the program counter of the next instruction to be executed.

BreakPoint

Set Breakpoint

[Format]

```
BreakPoint (B) [<break address>[,<break attribute>][, <executed command>]]
```

[Description]

Sets a breakpoint with the designated ⟨break attribute⟩ at the designated ⟨break address⟩. If parameters are not designated, displays all set breakpoints.
The following can be set as ⟨break attribute⟩. S is the default if none is designated.

S	:	When the instruction at ⟨break address⟩ is reached, stops just before execution of that instruction.
E	:	Stops when the instruction at ⟨break address⟩ is reached.
R	:	Stops after a read operation in the memory at ⟨break address⟩.
W	:	Stops after a write operation in the memory at ⟨break address⟩.
RW	:	Stops after a read or write operation in the memory at ⟨break address⟩.

S is a software breakpoint, used as a software method of stopping execution such as by embedding a trap instruction at the break address. Since this requires writing to memory, it cannot be used to set breakpoints for instructions in ROM or other read-only memory. As many as 8 software breakpoints may be set.
E, R, W, and RW are hardware breakpoints, making use of hardware functionality to stop program execution. These breakpoints can be set even for instructions in ROM or other read-only memory. Note that whether execution stops before or after the break condition is met depends on the hardware function. If the hardware supports stopping both immediately before and after the condition is met, stopping immediately before is the standard choice. In this case it is sometimes possible to choose stopping after the condition is met by designating "break attribute +" . In some cases the operand size can be designated with R, W, or RW. This is done by appending :B (byte), :H (half-word), or :W (word) after R, W, or RW. If no operand size is designated, :B (byte) is assumed.
Break attribute examples:

E+	Stop right after instruction execution.
R:W	Stop right before word data is read.
RW+	Stop right after byte data is read or written.

Since hardware breakpoints depend on hardware functions, the details vary with the hardware. In some cases there may be no hardware breakpoint function at all; or, functions in addition to those above may be supported.
A monitor command string (up to 80 characters) to be executed when a break occurs is designated in ⟨executed command⟩. Designating the `Go` command as ⟨executed command⟩ causes execution to be continued automatically after a break.

- The actually supported break attributes are implementation-dependent.

[Typical Usage]

```
TM> BreakPoint AC100100, "Register R0; Go"
```


BreakClear

Clear Breakpoint

[Format]

```
BreakClear (BC) [<break address>][, <break address>]..
```

[Description]

Clears the breakpoint setting at the designated ⟨break address⟩. If ⟨break address⟩ is omitted, all set breakpoints are cleared.

[Typical Usage]

```
TM> BreakClear AC100100
```


Step

Step Trace

[Format]

```
Step (S) [<execution start address>][, <instruction steps>]
```

[Description]

Executes a program trace from the designated ⟨execution start address⟩ for the designated number of ⟨instruction steps⟩, while showing a disassembly display of the executed instructions. The instruction shown in the disassembly display is the next instruction to be executed. That is, after executing one step, the instruction to be executed next is displayed. If the hardware lacks a disassemble function, only the address and memory contents are displayed. If ⟨instruction steps⟩ is omitted, 1 step is the default. If ⟨execution start address⟩ is omitted, trace is executed from the address designated by the current PC register value. During trace execution, all breakpoints are invalid.

[Typical Usage]

```
TM> Step , 4
<disassembly display-1>
<disassembly display-2>
<disassembly display-3>
<disassembly display-4>
```

Next

Next Trace

[Format]

```
Next (N) [<execution start address>][, <instruction steps>]
```

[Description]

Executes a program trace from the designated ⟨execution start address⟩ for the designated number of ⟨instruction steps⟩, while showing a disassembly display of the executed instructions. In the case of subroutine call instructions, an entire subroutine is traced as one instruction.
The instruction shown in the disassembly display is the next instruction to be executed. That is, after executing one step, the instruction to be executed next is displayed.
If the hardware lacks a disassemble function, only the address and memory contents are displayed.
If ⟨instruction steps⟩ is omitted, 1 step is the default. If ⟨execution start address⟩ is omitted, trace is executed from the address designated by the current PC register value. During trace execution, all breakpoints are invalid.

- Support for next trace execution is implementation-dependent.

[Typical Usage]

```
TM> Next, 4
<disassembly display-1>
<disassembly display-2>
<disassembly display-3>
<disassembly display-4>
```


BackTrace

Back Trace

[Format]

```
BackTrace (BTR) [<frame pointer>][, <display count>]
```

[Description]

Displays a history of function calls saved in the stack, starting from the current frame pointer or from the frame pointer value designated in the parameter.
When display is made from the current frame pointer, the current PC register value is shown first. When the frame pointer is designated in the parameter, the PC register is not displayed. The rest of the display consists of function call return addresses, tracing back in the history.
If ⟨display count⟩ is not designated, up to 16 function call return addresses are shown; if a count is designated, as many items as possible are displayed up to that count as maximum.

- Support for back trace is implementation-dependent. If the function call history is not saved in the stack, this command will not work properly.

[Typical Usage]

```
TM> BackTrace, 2
PC = 80101758
 <-- 80100420
 <-- 80100016
```


Load

Load Program/Data

[Format]

```
Load (LO) <protocol and data format>[, <load start address>]
```

[Description]

Loads program code or data to memory from the debugging console via a serial port.
One of the following is designated in ⟨protocol and data format⟨. The memory address to which the load is to be made is designated in ⟨load start address⟩. Whether or not this is designated depends on the data format, as summarized below.

	⟨Protocol⟩	⟨Data Format⟩	⟨Load Start Address⟩
S	ASCII transfer	S-Format (S3)	Not required
XS	XMODEM	S-Format (S3)	Not required
XM	XMODEM	Memory image data (unconverted)	Required

[Typical Usage]

```
TM> Load XS
Loaded: AC100000 -> AC1023f8

TM> Load XM, AC120000
Loaded: AC120000 -> AC12FFFF
```

ReadDisk

Read Disk

[Format]

```
ReadDisk (RD) <device name>, <start block number>, <block count>, <memory address>
```

[Description]

Reads data from the disk designated in ⟨device name⟩ starting from ⟨start block number⟩ for the number of blocks in ⟨block count⟩, to the designated ⟨memory address⟩.
Block size is specified separately for different devices and media.

Examples:	`pca`	PC card (ATA/CF) #1
	`pcb`	PC card (ATA/CF) #2

- The actual device names are implementation-dependent.

[Typical Usage]

```
TM> ReadDisk pca, 1, 20, AC140000
```


WriteDisk

Write Disk

[Format]

```
WriteDisk (WD) <device name>, <start block number>, <block count>, <memory address>
```

[Description]

Writes data from the designated ⟨memory address⟩ to the disk designated in ⟨device name⟩, starting from ⟨start block number⟩ for the number of blocks in ⟨block count⟩.
The block size depends on the device or media.

Examples:	`pca`	PC card (ATA/CF) #1
	`pcb`	PC card (ATA/CF) #2

- The actual device names are implementation-dependent.

[Typical Usage]

```
TM> WriteDisk hda, 100, 20, AC140000
```

InfoDisk

Display Disk Information

[Format]

```
InfoDisk (ID) <device name>
```

[Description]

Displays information about the disk designated in ⟨device name⟩. The following information is displayed.

Bytes per block
Total blocks

- The actual device names are implementation-dependent.

[Typical Usage]

```
TM> InfoDisk pca
Format: Bytes/block: 512  Total blocks: 8192
```


BootDisk

Boot from Disk

[Format]

```
BootDisk (BD) [<device name>]
```

[Description]

Boots from the disk designated in ⟨device name⟩. If the designated disk is not bootable, returns to waiting for monitor command input. If ⟨device name⟩ is omitted, T-Monitor searches the disks and boots from the first bootable disk found. The order of disk searching proceeds from removable to nonremovable disks. Details are implementation-dependent. If no bootable disk is found, the system state returns to waiting for monitor command input.

Examples: `pca0` 1st partition on PC Card (ATA/CF) #1
`pcb1` 2nd partition on PC Card (ATA/CF) #2

- The actual device names are implementation-dependent.

When a partitioned disk is designated in ⟨device name⟩, the system boots from the designated partition even if it is not set as the start partition.

[Typical Usage]

```
TM> BootDisk
```

Kill

Kill Process

[Format]

`Kill`

[Description]

When an exception is raised in an application process and control is passed to T-Monitor, this command forcibly kills the process where the exception was raised and operation of the system as a whole continues. Executing the `Kill` command does not result in return to the monitor.

- Since the application process concept is realized in the upper OS or middleware, the existence of a function for killing processes is implementation-dependent.

[Typical Usage]

```
TM> Kill
```

Help

Display Help Message

[Format]

```
Help (H) [<command name>]
?        [<command name>]
```

[Description]

Displays help on using the command designated in ⟨command name⟩. If ⟨command name⟩ is omitted or designates a nonexistent command, a list of commands is displayed.

[Typical Usage]

```
TM> ? DumpByte
DumpByte (DB) [<start_addr>][, {<end_addr>|#<data_cnt>}]
```

Exit

Exit

[Format]

```
Exit (EX) [<parameter>]
```

[Description]

Exits the monitor and shuts down the system.
If ⟨parameter⟩ is 0 or omitted, the system is stopped and power is turned off.
If ⟨parameter⟩ is -1, the system is reset and restarted.

[Typical Usage]

```
TM> Exit
```

Chapter 4

Program Support Functions

T-Monitor provides the following monitor service functions for use by programs.

Enter Monitor	Enter monitor
Get Character	Input 1 character from console
Put Character	Output 1 character to console
Get Line	Input 1 line from console
Put String	Output character string to console
Execute Command	Execute a monitor command
Read Disk	Read from disk
Write Disk	Write to disk
Info Disk	Get disk information
System Exit	Exit system
Extended SVC	Extended SVC functions

Functions that return error use the same error codes as T-Kernel.
The method of calling monitor service functions from an assembly routine is CPU-dependent. C library functions are provided for calling these service functions from a C language routine.

tm_monitor

Enter Monitor

[C Library Function and Call Format]

```
void tm_monitor( void )
```

[Return Code]

None

[Description]

Enters the monitor from a program and waits for monitor command input.
The monitor `Go` command can be used to resume program execution. If execution is resumed normally, `tm_monitor()` returns control to the program.

tm_getchar

Get Character

[C Library Function and Call Format]

```
INT tm_getchar( INT wait )
```

[Return Code]

≥ 0 : the input character code
-1 : no input (when `wait == 0`)

[Description]

Inputs one character (1 byte) from the debugging console. The entered character is not echoed back. If there is no input, -1 is returned when `wait == 0`; when `wait != 0`, the monitor waits until there is input.

tm_putchar

Put Character

[C Library Function and Call Format]

```
INT tm_putchar( INT c )
```

`c` : the character code to be output n

[Return Code]

-1 : `Ctrl-C` was entered
0 : `Ctrl-C` was not entered

[Description]

Outputs one character (1 byte) to the debugging console.
If `Ctrl-C` (0x03) is entered during output, character output is aborted and -1 is returned.
If the character for output is LF code (0x0A), both CR code (0x0d) and LF code (0x0A) are output (2 characters).

tm_getline

Get Line

[C Library Function and Call Format]

```
INT tm_getline( UB *buff )
```

`buff` : Start address of memory space for storing input string

[Return Code]

≥ 0 : number of characters input
-1 : `Ctrl-C` was entered

[Description]

Inputs one line from the debugging console up to the carriage return (0x0d) or until `Ctrl-C` (0x03) is entered, and puts the result in the designated memory address.
A NULL code(0) is stored at the end of the string as termination. The carriage return or `Ctrl-C` are not stored.
Sufficient space must be provided for buff. Buffer overflow is not detected.
Entered characters are echoed back and undergo the following special key processing.

`Ctrl-X` (0x18), `Ctrl-U` (0x15)	Undo (delete) line entry
`Ctrl-H` (0x08), `DEL` (0x7f)	Undo (delete) 1 character entry
`Ctrl-F` (0x06), `ESC [ C`	Move cursor right (→)
`Ctrl-B` (0x02), `ESC [ D`	Move cursor left (←)
`Ctrl-P` (0x10), `ESC [ A`	Call up previous line (↑)
`Ctrl-N` (0x0e), `ESC [ B`	Call up next line (↓)
`Ctrl-K` (0c0b)	Delete after cursor

tm_putstring

Put String

[C Library Function and Call Format]

```
INT tm_putstring( UB *buff )
```

`buff` : Start address of memory space holding input string

[Return Code]

-1 : `Ctrl-C` was entered
0 : `Ctrl-C` was not entered

[Description]

Outputs characters from the designated memory address to the debugging console, 1 character (byte) at a time up to the NULL code (0).
If `Ctrl-C` (0x03) is entered during output, character output is aborted and -1 is returned.
If the string contains LF code (0x0A), both CR code (0x0d) and LF code (0x0A) are output (2 characters).

tm_command

Execute Command

[C Library Function and Call Format]

```
INT tm_command( UB *buff )
```

`buff` : Start address of memory space holding monitor command array

[Return Code]

0	:	Executed a monitor command
No return	:	Entered monitor

[Description]

Executes the string stored at the designated memory address (terminated by NULL code (0)) as a monitor command (array) and returns control to the program.
If the character string is null, the monitor is entered without returning to the program.

tm_readdisk

Read Disk

[C Library Function and Call Format]

```
INT tm_readdisk( UB *dev, INT sblk, INT nblks, VP addr )
```

`dev`	:	Start address of memory space holding device name
`sblk`	:	Start block number
`nblks`	:	Block count
`addr`	:	Memory address

[Return Code]

0	:	Normal completion
≤ 0	:	Error code
`E_NOEXS`	:	Device does not exist
`E_NOMDA`	:	No media
`E_IO`	:	IO error
`E_PAR`	:	Parameter is invalid
`E_MACV`	:	Cannot access memory

[Description]

Reads to the designated memory address the contents of the disk designated by device name, from the designated start block for the designated number of blocks.
Block size is specified separately for different devices and media.

Examples:	`pca`	PC Card (ATA/CF) #1
	`pcb`	PC Card (ATA/CF) #2

- The actual device names are implementation-dependent.

tm_writedisk

Write Disk

[C Library Function and Call Format]

```
INT tm_writedisk( UB *dev, INT sblk, INT nblks, VP addr )
```

`dev`	:	Start address of memory space holding device name
`sblk`	:	Start block number
`nblks`	:	Block count
`addr`	:	Memory address

[Return Code]

0	:	Normal completion
≤ 0	:	Error code
`E_NOEXS`	:	Device does not exist
`E_NOMDA`	:	No media
`E_IO`	:	IO error
`E_PAR`	:	Parameter is invalid
`E_MACV`	:	Cannot access memory
`E_RONLY`	:	Read-only

[Description]

Reads the contents of the designated memory address to the disk designated by device name, from the designated start block for the designated number of blocks.
Block size is specified separately for different devices and media.

Examples:	`pca`	PC Card (ATA/CF) #1
	`pcb`	PC Card (ATA/CF) #2

- The actual device names are implementation-dependent.

tm_infodisk

Info Disk

[C Library Function and Call Format]

```
INT tm_infodisk( UB *dev, INT *blksz, INT *nblks )
```

`dev`	:	Start address of memory space holding device name
`blksz`	:	Start address of memory space holding block size (in bytes)
`nblks`	:	Start address of memory space holding total number of blocks

[Return Code]

0	:	Normal completion
≤ 0	:	Error code
`E_NOEXS`	:	Device does not exist
`E_NOMDA`	:	No media
`E_IO`	:	IO error
`E_MACV`	:	Cannot access memory

[Description]

Gets the block size (in bytes) and total number of blocks in the device designated by device name.

- The actual device names are implementation-dependent.

tm_exit

System Exit

[C Library Function and Call Format]

```
void tm_exit( INT mode )
```

`mode`	: 0	:	Exit system and turn off power
	−1	:	Reset system and restart

[Return Code]

Does not return.

[Description]

Exits the system, either turning off the power or resetting.

tm_extsvc

Extended SVC

[C Library Function and Call Format]

```
INT tm_extsvc( INT fno, INT p1, INT p2, INT p3 )
```

`fno` :	Function number of extended service
`p1,p2,p3` :	Parameters 1 to 3

[Return Code]

0 :	Normal completion
< 0 :	Error code

[Description]

Executes the extended service function designated in fno.
The extended service function numbers, parameters and return codes are all dependent on the monitor implementation.

Chapter 5

Boot Details

5.1 Boot Processing Overview

System boot normally proceeds in the following steps.

1. Search for a bootable device.
2. Load the primary boot program.
3. Load the secondary boot program.
4. Load the operating system.

T-Monitor performs steps 1. and 2. of these.
It also provides monitor service functions enabling disk access by the primary and secondary boot programs.

5.2 Searching for Bootable Device

Searching for a bootable device generally takes place in the following order, but the specific details are implementation-specific.

1. Devices with removable media (floppy disk, CD-ROM, etc.)
2. Devices with removable drive (PC Card, drive connecting by USB interface, etc.)
3. Nonremovable devices (internal hard disk)

In the case of partitioned disks, T-Monitor looks at the partition information, and searches only for a partition marked as start partition. Details of the partition information are as defined in standard PC specifications.

5.3 Loading and Starting Primary Boot Program

The initial block of the disk from which the system is to be booted (the initial block of the start partition if the disk is partitioned) is loaded into memory. This is the primary boot program.

If the block size is smaller than 512 bytes, blocks are loaded consecutively from the initial block until at least 512 bytes are in memory.
Control is then passed to the primary boot program loaded in memory. At this time the monitor passes the following information to the primary boot program.

```
#define L_DEVNM     8       // length of device name
typedef struct BootInfo {
UB  devnm[L_DEVNM];         // physical device name of boot disk
INT part;                   // boot partition number (0 and up,-1: No partition)
INT start;                  // boot partition position (initial block number)
INT blksz;                  // block size (bytes) of boot disk
} BootInfo;
```

The memory address to which the primary boot program is loaded and the method of passing parameters are specified for each implementation.