

MTASK2015

April 2015

**A Multi-Task kernel (MT) for
the SHARP PC-1500/A and TRS80 PC-2**

Copyright 1992 - 1993 - 2015 - Christophe Gottheimer

1/ Presentation

This is my implementation of a small multi-task kernel (MT). I started to develop it in 1992 when I was working on **Motorolla 680x0** targets **vxWorks**. I feel very funny to try to go out of the way in this small computer. This was of course possible because the BASIC ROM was able to divert the interruptions. I read an article in a french computer review speaking about a small example of this diversion. I adapted it and after I developed several MT-kernels, with each time more functionalities.

One of the last I developed support an access to the MT-kernel through a BASIC instruction **TASK**. This was useful to "speak" with a task without calling an assembly routine interface.

I stopped the development to start another, more complex: A small multi-process operating system named MMPS (see: <http://www.pc1500.com/mmeps.html>).

Several years ago, I recovered the tapes where the MT-kernel was saved, but I was not able to reload it or to extract a binary. I still own my written notes. Some weeks ago, finally, I succeed to extract the binary images with some parts corrupted, but my notes helped me to rebuild a working kernel.

I also use this work to change some kernel request and add some new ones like the timer.

Content of the package:

COPYING	- The GNU GPL v2
README	- A small README manual
mk.sh	- A shell script to build the mt2015-* binary images
mtask2015.pdf	- The whole MT-kernel documentation in PDF
mtask2015.abw	- The whole MT-kernel documentation source for AbiWord
mtask2015.asm	- The assembly source code of the MT-kernel
mtfull.asm	- The include method for the full image source code
callregs.asm	- The assembly source code of the CALLREGS routine
taskbas.asm	- The assembly source code of the BASIC instruction TASK
keyboard.asm	- The assembly source code of a small keyboard driver
exall.asm	- The assembly source code of all examples
exsched.asm	- The assembly source code of a small "schedule" task
Images/std/	- Directory with all modules images for standard unit
Images/1500A/	- Directory with all modules images for the PC-1500A
Images/ex/	- Directory with all modules images of the examples

Note that for machine a very small amount of memory, images without the BASIC instruction **TASK** are provided.

To build the Multi-Task Kernel from sources, the **lhTools-0.7.2** or higher are required.

2/ Installation

2.1/ Machine requirements

The MT-kernel alone needs 2Kb, so it is very difficult to use it in PC-1500 without a memory module. A module of at least 4Kb (CE-151) or a PC-1500A is highly recommended. For the images with the **TASK** instruction and the keyboard driver, 2.8Kb is required. Special images for PC-1500A are provided, because of the different memory scheme; with these images, the task structures are stored into the dedicated ML area (**&7F00 . . &7F82**). Note that standard images also work into the PC-1500A.

The keyboard driver needs to be installed into a **ROM A02** or higher. If the function **PEEK &E2B9** answers **56**, the machine has a good ROM, and **full** images may be used. Else, the driver and the **TASK** instruction will be not usable; in this case, use the **reduced** images. **CALLREGS** works on all ROMs. Note that the PC-1500A have always the good ROM.

Note: The images for **CE-151** is also working in a PC-1500A without module.

If a RESET occurs (i.e. a **NEW0? :CHECK**), the last step 2.5/ need to be done to reactivate the keyboard driver.

2.2/ Images naming

MTASK2015 is delivered with 2 kinds of images: **.wav** for audio download with the **CE-150** or **CE-162** cassette interface and **.bin158** for a serial download with the **CE-158** serial interface.

The **reduced** images come with the **MT-kernel** and **CALLREGS**. The **full** images come with the **TASK** command, the **MT-kernel** and **CALLREGS**.

The images naming is **Images/std/mt2015-full-ceMMM.EEE** for full image and **Images/std/mt2015-reduced-ceMMM.EEE** where **MMM** is the module (**151**, **155**, **159** or **161** [also for **CE-163**]), and **EEE** is **.wav** for audio or **.bin158** for serial.

The **.bin158** are binary images with the **CE-158** header included to a download with the **CLOAD M** command.

The dedicated images for PC-1500A have the same naming convention, but they are located in **Images/1500A/** directory.

2.3/ Audio download

Connect the **PC-1500** to a **CE-150** or **CE-162** audio cassette interface and plug the audio jack wire.

After, enter a **CLOAD M** command and start to play the WAV file. After 3.30 minutes, **MTASK2015** is loaded. See 4.5/ to start **MTASK2015**.

- When loading the **full** WAV image, **MTASK"2015:Fccc** is displayed where **ccc** is the module: **151 155 159** or **161**.
- When loading the **reduced** WAV image, **MTASK"2015:rccc** is displayed where **ccc** is the module: **151 155 159** or **161**.

2.4/ Serial download

Connect the **PC-1500** to a **CE-158** interface and plug the serial wire between the host computer and the interface.

On host, configure the serial line parameters with **2400** bauds, **8 bits**, **No parity** and **1-bit stop: 2400/8/N/1**.

Switch the **CE-158** interface ON and after the **PC-1500**. Configure the serial parameters and set the device in input mode:

```
SETCOM 2400,8,N,1  
SETDEV CI  
OUTSTAT 0  
CLOAD M
```

Start the transfer on the host PC. After less than 1 minute, **MTASK2015** is loaded. See 2.5/ to start **MTASK2015**.

With the **lhTools**, the utility **lhcom** may be used as follow for serial download:

```
lhcom -s -Y /dev/tty<...>=2400,8,N,1 Images/<...>/mt2015<...>.bin158
```

2.5/ Initialization

Depending of the images you have loaded, the following steps need to be done to initialize **MTASK2015**, but only **full** images needs these initializations steps.

CE-151:

```
POKE &785B, &47, &FD
POKE &79D1, &24
POKE &79D4, &55
```

CE-155:

```
POKE &785B, &3F, &FD
POKE &79D1, &20
POKE &79D4, &55
```

CE-159:

```
POKE &785B, &2F, &FD
POKE &79D1, &14
POKE &79D4, &55
```

CE-161 and CE-163:

```
POKE &785B, &0F, &FD
POKE &79D1, &04
POKE &79D4, &55
```

Type the **POKE**'s addresses and values very carefully, because a mistake may crash the **PC-1500** and the whole memory may be corrupted or lost !

Be sure to call at least one time the **t_TASKINIT** routine (see 3/) or a **TASK CLEAR** (full images) to properly clear the task structures and some pointers. This need to be done after loading the **MTASK2015** binary, or after a crash or a soft reset (**NEW? CHECK:**)

Welcome to MTASK2015 !

3/ Starting the MT-kernel

Before any use after loading **MTASK2015**, or before working from a clean situation, a call to **t_TASKINIT** has to be performed. This clears all the task structures, the internal pointers and install a relocatable vector for **t?KRNREQ** useful to develop common task code. This is simply done by **CALL t_TASKINIT** where **t_TASKINIT** is the address defined below according of the binary image loaded.

For example, to call **t_TASKINIT** with the CE-155 image: **CALL &3FF1**

On full images, just do **TASK CLEAR**

At each Power-OFF, the MT-kernel is halted, and interruption vector is not restored by the ROM when powering on. So at each Power-ON, a **TASK RUN** need to be done to restart the scheduler. The task structures are not affected by this command except for the caller; its task structure is reinitialized. The relocatable vector for **t?KRNREQ** is also reinstalled by **t_ARUN**. This is simply done by **CALL t_ARUN** where **t_ARUN** is the address defined below according of the binary image loaded.

For example, to call **t_ARUN** with the CE-159 image: **CALL &27F4**

With **CALLREGS**, do a:

```
A$="000000000000"  
CALL &callregs-address,A$
```

Note that the *callregs-address* depends of the image installed (see below). For example, to call **t_ARUN** using **CALLREGS** with the CE-161 image:

```
A$="000000000000"  
CALL &07FA,A$
```

On full images, just do **TASK ARUN**

On full images, the keyboard driver comes with the key auto repeat, a special **OFF** to avoid to replay the step 2.5/ at each Power-ON, and the **TASK** instruction mapped to the key **DEF+O**. This is because the ROM does not fetch the keywords outside some tables. Note that **DEF+O** replaces the **MERGE** command in shortcut, but **MERGE** is still accessible by typing **MERGE**.

Depending of the images loaded and the modules inserted, the following routines are located to different addresses (full and reduced have the same):

	t_TASKINIT	t_ARUN	t?KRNREQ	CALLREGS
CE-151	&47F1	&47F4	&47F7	&47FA
CE-155	&3FF1	&3FF4	&3FF7	&3FFA
CE-159	&27F1	&27F4	&27F7	&27FA
CE-161/163	&07F1	&07F4	&07F7	&07FA

In the way to develop and provide common programs with all images of **MTASK2015**, a relocatable vector is installed by **t_ARUN** for **t?KRNREQ**. The vector is at the address **&79FA**. Note that if you use some peripherals, it may safe to call **t_ARUN** to reinstall the relocatable vector. Note that **CALLREGS** or **TASK** are always safe because they use an internal call address instead of the relocatable vector.

4/ MT-Kernel requests

When a task requires a MT-kernel services, it should call a *MT-kernel request*. This is done by calling the **t?KRNREQ** entry point with some parameters in the CPU registers, depending of the request. The register **D** is loaded with the request id to performed. The others parameters may be or not optional. At the return from a request, the **CARRY** is set if it was successful, or it is cleared if the request failed and the register **H** contains the error code.

Be careful to always call the **t?KRNREQ** entry point and never bypass it, else a severe crash may occur and all data and/or program may be corrupted or lost.

For example, to pause the caller during a number of scheduler ticks (**&1FF**):

```
;; BC contains the number of ticks as delay  
LD    B,&01  
LD    C,&FF  
;; D is loaded with the TASK PAUSE identifier, i.e,  
LD    D,&0A  
CALL  t?KRNREQ  
;; If CARRY cleared, an error occurred and code is returned in H  
JR    NC,process-error
```



```

;; L = <prio>

&07 :: t?LOCK:
;; LOCK - Try to take a lock
;; L = <lockid>

&08 :: t?UNLOCK:
;; UNLOCK - Give a lock
;; L = <lockid>

&09 :: t?LOCKWAIT:
;; LOCK WAIT - Take a lock and wait if not free
;; L = <lockid>

&0A :: t?PAUSE:
;; PAUSE - Sleep task for a delay
;; BC = <delay>

&0B :: t?INPUT:
;; INPUT - Wait for a message
;; BC = <address>
;; L = <length>
;; -> H = <receive-length>
;; -> A = <sender-tid>

&0C :: t?PRINT:
;; PRINT - Send a message to a task
;; BC = <address>
;; L = <length>
;; A = <tid>

&0D :: t?PRINTWAIT:
;; PRINT WAIT - Send a message and wait if receiver is
;; not ready
;; BC = <address>
;; L = <length>
;; A = <tid>

&0E :: t?PRINTINPUT:
;; PRINT INPUT - Send a message and wait for message
;; BC = <address>
;; L = <length>
;; A = <tid>
;; -> H = <receive-length>
;; -> A = <sender-tid>

&0F :: t?GOTO:
;; GOTO - Install event handler
;; BC = <handler>

```

```

    ;; When entering into the handler:
    ;; >>    L = <eventid-receive>
    ;; >>    A = <sender-tid>

&10 :: t?UNGOTO:
    ;; GOTO    - Uninstall event handler

&11 :: t?ON:
    ;; ON      - Set event active
    ;;    L = <eventid>

&12 :: t?OFF:
    ;; OFF     - Reset event active
    ;;    L = <eventid>

&13 :: t?CALL:
    ;; CALL    - Send an event to a task
    ;;    L = <eventid>
    ;;    A = <tid>
    ;; Note that a task can not call itself

&14 :: t?TIME:
    ;; TIME    - Arm a timer and wake-up the calling task
    ;; when the delay elapses
    ;;    BC = <delay>

&15 :: t?TIMEEND:
    ;; TIMEEND - Delete running timer

&16 :: t?WAIT:
    ;; WAIT    - Wait for lock, semaphore, message, event
    ;; or timeout

&17 :: t?PEEK:
    ;; PEEK    - Wait on a counter semaphore with a lock
    ;;    L = <cntid><locktid>

&18 :: t?POKE:
    ;; POKE    - Signal on a counter semaphore with a lock
    ;;    L = <cntid><locktid>
    ;; If b7=1 in L, the counter is cleared

&19 :: t?DIM:
    ;; DIM     - Declare a queue
    ;;    BC = <queue>
    ;;    L = <size>

&1A :: t?READ:
    ;; READ    - Read a message from the queue

```

```
;; BC = <msg>
;; -> L = <length>
&1B :: t?READWAIT:
;; READWAIT - Read a message from the queue and wait
;; if empty
;; BC = <msg>
;; -> L = <length>

&1C :: t?DATA:
;; DATA - Send a message into a queue
;; BC = <msg>
;; L = <length>
;; A = <tid> | 0xFF for itself
```

4.2/ MT-kernel error codes

t#ERR_NoSuchKrnReq	#240 (&F0)
t#ERR_NoTaskFree	#241 (&F1)
t#ERR_TaskBusy	#242 (&F2)
t#ERR_NoSuchTask	#243 (&F3)
t#ERR_TaskNotInput	#244 (&F4)
t#ERR_LockNotOwner	#245 (&F5)
t#ERR_UnlockNotOwner	#246 (&F6)
t#ERR_HandlerBusy	#247 (&F7)
t#ERR_TaskCalling	#248 (&F8)
t#ERR_NoSuchQueue	#249 (&F9)
t#ERR_QueueBusy	#250 (&FA)
t#ERR_QueueEmpty	#251 (&FB)
t#ERR_QueueFull	#252 (&FC)
t#ERR_TimerRunning	#253 (&FD)
t#ERR_TimedOut	#254 (&FE)

4.3/ TASK instruction syntax

If the **TASK** instruction is available, its syntax is the following:

```
TASK CLEAR
TASK ARUN
TASK ?<var>
TASK RUN <entry>,<stack>[,<prio>[,<tid>]];<var>
TASK END <tid>
TASK STOP <tid>
TASK CONT <tid>
TASK NEW <prio>[,<tid>]
TASK PAUSE <delay>
TASK LOCK <lockid>
TASK LOCK WAIT <lockid>
TASK UNLOCK <lockid>
TASK INPUT <var$>
TASK PRINT <tid>,<var$>
TASK PRINT WAIT <tid>,<var$>
TASK PRINT INPUT <tid>,<var$>
TASK GOTO [<handler>]
TASK ON <eventid>
TASK OFF <eventid>
TASK CALL <eventid>,<tid>
TASK TIME [<delay>]
TASK WAIT
TASK POKE CLS <lockid>,<cntid>
TASK POKE <lockid>,<cntid>
TASK PEEK <lockid>,<cntid>
TASK DIM <queue>,<size>
TASK READ WAIT <var$>
TASK READ <var$>
TASK DATA <tid>,<var$>
```

If the priority *<prio>* is not fixed when calling **TASK RUN**, the default priority **3** is assumed by the MT-kernel.

The values for *<tid>*, *<prio>*, *<eventid>*, *<lockid>*, *<cntid>* are from **0** to **7**. For *<tid>*, the value **&FF** relates the calling task id.

The queue size is a number from **6** to **255**. The kernel “eats” **5** bytes for its internal operations. So in a queue of **20** bytes, the biggest message size will be **15** bytes.

If an error occurs will calling the *MT-kernel-request* with **TASK**, the normal error handling is assumed. The BASIC will show **ERROR** *eee* where *eee* is the decimal error code listed in step 4.2/.

4.4/ CALLREGS

The routine **CALLREGS** provides an “*enhanced*” call extension with register values passing and retrieving. The routine **CALLREGS** is designed to call only the *MT-kernel-request*.

To call a *MT-kernel-request* with **CALLREGS**, fill a string variable as follow:

```
var$="aabbccddhhll"
```

where:

aa is a 8-bits hexdigits to be loaded in **A**

bb is a 8-bits hexdigits to be loaded in **B**

cc is a 8-bits hexdigits to be loaded in **C**

dd is a 8-bits hexdigits to be loaded in **D**

hh is a 8-bits hexdigits to be loaded in **H**

ll is a 8-bits hexdigits to be loaded in **L**

The value *dd* is the *MT-kernel request id*. If *aa* contains the *task id (TID)* concerned by the *MT-kernel request*, **&FF** always relates the calling **TID**.

Finally calls the **CALLREGS** routine with a:

```
CALL callregs-addr,var$
```

At the return, the **var\$** contains "*a'b'c'h'l'*" where *a' b' c' h' l'* are the returned values from the registers **A B C H** and **L**. If an error occurred during the *MT-kernel request* (**CARRY** cleared), **var\$** is loaded with "#*ee*" where *ee* is the hexadecimal error code listed in 4.2/.

For example, to pause (**&0A**) for a ticks value of **&123**, **BC** has be loaded with the delay, do

```
D$="0001230A0000"
```

```
CALL callregs,D$
```

```
D$
```

and you see:

```
"0001230000"
```

Depending of the images loaded and the modules inserted, the routine **CALLREGS** is located to different addresses (full and reduced have the same):

	CALLREGS
CE-151	&47FA
CE-155	&3FFA
CE-159	&27FA
CE-161/163	&07FA

CALLREGS:

```
A$="*****01****"  
CALL CALLREGS;A$  
A$ shows "0t*****" with t the TID.
```

5.3/ t?RUN

Create a new task. The task entry point should set in **HL**, the stack is **BC**, and the priority in **L[b3..b0]**. It is possible to request a specific **TID** in **L[b7..b4]**, or to let the kernel assigns the **TID** with **&F** in **L[b7..b4]**. If the created task has a higher priority than the caller, it takes the hand immediately.

Syntax:

```
LD    BC,<stack>  
LD    HL,<entry>  
LDA   <tid><prio> | &F<prio>  
LD    D,&02  
CALL  t?KRNREQ  
JR    NC,error_handler
```

Output: If the **CARRY** is set, **A** contains the created **TID**, else **H** contains the error code.

Error:

```
t#ERR_NoTaskFree    No more room in task array to create the new task  
t#ERR_TaskBusy      The TID given in L[b7..b4] is already used
```

BASIC:

```
TASK RUN <entry>,<stack>[,<prio>[,<tid>]];<var>
```

The **TID** is returned to **<var>**. If **<prio>** is not given, **TASK** set it to **3**.

CALLREGS:

```
A$="tpsspp02ppcc"  
CALL CALLREGS;A$  
A$ shows "0t*****" with t the TID.
```

5.4/ t?END

Terminates the task specified by **TID**. If **TID** is **&FF**, the calling task is terminated. If a task terminates itself, **t?END** never returns.

Syntax:

```
LDA   <tid> | &FF  
LD    D,&03  
CALL  t?KRNREQ  
JR    NC,error_handler
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_NoSuchTask No task with the given **TID** exists

BASIC:

```
TASK END <tid>
```

CALLREGS:

```
A$="0t****03****"  
CALL CALLREGS;A$
```

5.5/ t?STOP

Stops the task specified by **TID**. If **TID** is **&FF**, the calling task is stopped. The task still exists, but the scheduler will never start it, until a **t?CONT** is done for this task.

Syntax:

```
LDA <tid> | &FF  
LD D,&04  
CALL t?KRNREQ  
JR NC,error_handler
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_NoSuchTask No task with the given **TID** exists

BASIC:

```
TASK STOP <tid>
```

CALLREGS:

```
A$="0t****04****"  
CALL CALLREGS;A$
```

5.6/ t?CONT

Continues (restart) the task specified by **TID**. A task can not perform a **t?CONT** to itself.

Syntax:

```
LDA <tid>  
LD D,&05  
CALL t?KRNREQ  
JR NC,error_handler
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_NoSuchTask No task with the given **TID** exists

BASIC:

```
TASK CONT <tid>
```

CALLREGS:

```
A$="0t****05****"  
CALL CALLREGS;A$
```

5.7/ t?NEW

Change the priority of a task specified by **TID**. The new priority is in **L** in the range **0..7**. The priority **0** is the highest priority and **7** is the the lowest. The scheduler starts the task with the highest priority until they suspend. After the tasks of less priority, and so on, until the tasks of the lowest priority. If the **TID** is **&FF**, the priority of the calling task is changed.

Syntax:

```
LDA <tid> | &FF  
LD L,<prio>  
LD D,&06  
CALL t?KRNREQ  
JR NC,error_handler
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_NoSuchTask No task with the given **TID** exists

BASIC:

```
TASK NEW <prio>[,<tid>]
```

CALLREGS:

```
A$="0t****06**0p"  
CALL CALLREGS;A$
```

5.8/ t?LOCK

Tries to takes the ownership of a lock. The *lockid* is in range **0..7**. If the lock is free, the calling task becomes the owner, else the request returns an error. When a task terminates, the locks are “freed” by the kernel, if the task owns some locks.

Syntax:

```
LD L,<lockid>  
LD D,&07  
CALL t?KRNREQ  
JR NC,error_handler
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_LockNotOwner The lock is owned by another task.

BASIC:

TASK LOCK <lockid>

CALLREGS:

A\$="***07**01"**
CALL CALLREGS;A\$

5.9/ t?UNLOCK

Tries to give up the ownership of a lock. If the lock is owned by the task, it is “freed” and the next waiting task may become owner. If the task is not owner of the lock, an error is returned. When a task terminates, the locks are “freed” by the kernel, if the task owns some locks.

Syntax:

LD L,<lockid>
LD D,&08
CALL t?KRNREQ
JR NC,error_handler

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_UnlockNotOwner The lock is owned by another task.

BASIC:

TASK LOCK <lockid>

CALLREGS:

A\$="***08**01"**
CALL CALLREGS;A\$

5.10/ t?LOCKWAIT

Tries to take the ownership of a lock, and waits if the lock is already owned by another task. If the lock is free, the calling task becomes the owner, else the calling task waits until the owner task frees the lock. When a task terminates, the locks are “freed” by the kernel, if the task owns some locks.

Syntax:

LD L,<lockid>
LD D,&09
CALL t?KRNREQ
JR NC,error_handler

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_LockNotOwner The lock is owned by another task.
t#ERR_TimedOut A timer is elapsed, and the request is interrupted

BASIC:

TASK LOCK WAIT <lockid>

*Note: When a timer elapses, the **ERROR 254** is raised in BASIC.*

CALLREGS:

A\$="***09**01"**
CALL CALLREGS;A\$

5.11/ t?PAUSE

Delays the calling task for the amount of ticks given by **BC**. Until the ticks counter is decremented to **0**, the task is no more scheduled.

Syntax:

LD BC, <delay>
LD D, &0A
CALL t?KRNREQ

Output: None

Error: None

BASIC:

TASK PAUSE <delay>

CALLREGS:

A\$="bbcc0A****"**
CALL CALLREGS;A\$

5.12/ t?INPUT

The task goes in the *INPUT-STATE*, and wait infinitely for a synchronous message. The buffer to receive the message is in **BC** and **L** is maximum length for the message. When a another task sends a synchronous message with **t?PRINT**, the tasks returns with **H** the length of the message received, and **A** the **TID** of the sender. The message is copied by the kernel from the "sender-space" to the "receiver-space".

Syntax:

LD BC, <buffer>
LD L, <length>
LD D, &0B

CALL t?KRNREQ

Output: When returning, **H** contains the length of the message received and **A** the **TID** of the sender.

Error: None

BASIC:

TASK INPUT <var\$>

Note: The TASK instruction never returns the sender TID.

CALLREGS:

A\$="**bbcc0B**11"

CALL CALLREGS;A\$

A\$ shows "0**t**bc**ch**11" with **t** the **TID** and **hh** the received length.

5.13/ t?PRINT

Sends a synchronous message pointed by **BC** of the length **L** to the task specified by **TID** in **A**. This task should be in *INPUT-STATE*, else an error is returned. The message is copied by the kernel from the "sender-space" to the "receiver-space". A task can not send a synchronous message to itself.

Syntax:

```
LD   BC,<buffer>
LD   L,<length>
LD   A,<tid>
LD   D,&0C
CALL t?KRNREQ
JR   NC,error_handler
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_NoSuchTask No task with the given **TID** exists
t#ERR_TaskNotInput The receiver task is not in state *INPUT-STATE*

BASIC:

TASK PRINT <tid>,<var\$>

CALLREGS:

A\$="0**t**bbcc0C**11"

CALL CALLREGS;A\$

5.14/ t?PRINTWAIT

Sends a synchronous message pointed by **BC** of the length **L** to the task specified by **TID** in **A**. This task should be in *INPUT-STATE*, else the calling task waits until the remote enters

into *INPUT-STATE*. The message is copied by the kernel from the “*sender-space*” to the “*receiver-space*”. A task can not send a synchronous message to itself.

Syntax:

```
LD    BC,<buffer>
LD    L,<length>
LD    A,<tid>
LD    D,&OD
CALL  t?KRNREQ
JR    NC,error_handler
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_NoSuchTask	No task with the given TID exists
t#ERR_TimedOut	A timer is elapsed, and the request is interrupted

BASIC:

```
TASK PRINT WAIT <tid>,<var$>
```

*Note: When a timer elapses, the **ERROR 254** is raised in BASIC.*

CALLREGS:

```
A$="0tbbccOD*11"
CALL CALLREGS;A$
```

5.15/ t?PRINTINPUT

Sends a synchronous message pointed by **BC** of the length **L** to the task specified by **TID** in **A**. This task should be in *INPUT-STATE*, else the calling task waits until the remote enters into *INPUT-STATE*. The message is copied by the kernel from the “*sender-space*” to the “*receiver-space*”. A task can not send a synchronous message to itself. The caller task immediately enters into the *INPUT-STATE* ready to receive a synchronous message. The kernel insures that this request is performed as an atomic operation. So, there is no risk to lose some synchronous messages even the receiver task has a higher priority than the sender. The task returns with **H** the length of the message received, and **A** the **TID** of the sender.

Syntax:

```
LD    BC,<buffer>
LD    L,<length>
LD    A,<tid>
LD    D,&OE
CALL  t?KRNREQ
JR    NC,error_handler
```

Output: If the **CARRY** is set, the request was successful and **H** contains the length of the message received and **A** the **TID** of the sender, else **H** contains the error code.

Error:

t#ERR_NoSuchTask	No task with the given TID exists
t#ERR_TimedOut	A timer is elapsed, and the request is interrupted

BASIC:

TASK PRINT INPUT <tid>,<var\$>

Note: The TASK instruction never returns the sender TID.

*Note: When a timer elapses, the **ERROR 254** is raised in BASIC.*

CALLREGS:

A\$="0tbbcc0E**11"

CALL CALLREGS;A\$

A\$ shows "0tbcchh11" with **t** the **TID** and **hh** the received length.

5.16/ t?GOTO

Install an event handler pointed by **BC** for the calling task. The event mask is reset to &00 (no event) with this request. If a handler is already installed, the request returns an error. If a task send a event to this task, and if the event is enabled, the handler is called. When entering into the handler, the event is in **L** and **H** and the **TID** of the sender is in **A**. The task enters into the handler even it is stopped, waiting, paused or in *INPUT-STATE*, but not if it is already handling an event. A task can not send an event to itself.

Syntax:

```
LD    BC,<handler>
LD    D,&0F
CALL  t?KRNREQ
JR    NC,error_handler
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_HandlerBusy	The handler is already sets
--------------------------	-----------------------------

BASIC:

TASK GOTO <handler>

CALLREGS:

A\$="**bbcc0F****"

CALL CALLREGS;A\$

5.17/ t?UNGOTO

Remove the event handler for the calling task. The event mask is reset to &00 (no event) with this request.

Syntax:

```
LD    D,&10
CALL  t?KRNREQ
```

Output: None

Error: None

BASIC:

```
TASK GOTO
```

CALLREGS:

```
A$="*****10****"  
CALL CALLREGS;A$
```

5.18/ t?ON

Enable the event id given by **L**. The event id is in range **0 . . 7**. This request is only effective if a handler is installed.

Syntax:

```
LD    L,<eventid>  
LD    D,&11  
CALL t?KRNREQ
```

Output: None

Error: None

BASIC:

```
TASK ON <eventid>
```

CALLREGS:

```
A$="*****11*0l"  
CALL CALLREGS;A$
```

5.19/ t?OFF

Disable the event id given by **L**. The event id is in range **0 . . 7**. This request is only effective if a handler is installed.

Syntax:

```
LD    L,<eventid>  
LD    D,&12  
CALL t?KRNREQ
```

Output: None

Error: None

BASIC:

TASK OFF <eventid>

CALLREGS:

```
A$="*****12*01"  
CALL CALLREGS;A$
```

5.20/ t?CALL

Sends the event specified by **L** to the task specified by **TID** in **A**. To enter into the handler, the receiver task should have installed a handler, enabled the event **L** and not be currently handling. A task can not send an event to itself.

Syntax:

```
LD    L,<eventid>  
LD    A,<tid>  
LD    D,&13  
CALL  t?KRNREQ  
JR    NC,error_handler
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

```
t#ERR_NoSuchTask      No task with the given TID exists  
t#ERR_TaskHandling    The receiver task is already entered into a handler
```

BASIC:

```
TASK CALL <eventid>,<tid>
```

CALLREGS:

```
A$="0t****13*11"  
CALL CALLREGS;A$
```

5.21/ t?TIME

Arms the timer delay of the calling task for the amount of ticks given by **BC**. When the timer elapses, the task is “awaken” if it was waiting (**t?PRINTWAIT**, **t?PRINTINPUT**, **t?LOCKWAIT**, **t?READWAIT**, **t?WAIT**, **t?PEEK**), the **CARRY** is cleared and the error **t#ERR_TimedOut** is filled in **H**. If a timer is already running, an error is returned.

Syntax:

```
LD    BC,<delay>  
LD    D,&14  
CALL  t?KRNREQ  
JR    NC,noerror
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_TimerRunning A timer is already armed for the task

BASIC:

TASK TIME <*delay*>

CALLREGS:

A\$="***bbcc14*****"

CALL CALLREGS;A\$

5.22/ t?TIMEEND

Kill the timer for the current task if one was running.

Syntax:

LD D,&15

CALL t?KRNREQ

Output: None

Error: None

BASIC:

TASK TIME

CALLREGS:

A\$="******15*****"

CALL CALLREGS;A\$

5.23/ t?WAIT

Waits infinitely until the caller task is awakened by a timer or by a **t?UNLOCK**, **t?DATA**, or **t?POKE** request. When the timer elapses, the task is “awaken” if it was waiting (**t?PRINTWAIT**, **t?PRINTINPUT**, **t?LOCKWAIT**, **t?READWAIT**, **t?WAIT**, **t?PEEK**), the **CARRY** is cleared and the error **t#ERR_TimedOut** is filled in **H**. If the task is waken by a request, the **CARRY** is set.

Syntax:

LD D,&16

CALL t?KRNREQ

JR C,handle_request

CP H,t#ERR_TimedOut

JR Z,handle_timeout

Output:If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_TimedOut A timer is elapsed, and the request is interrupted

BASIC:

TASK WAIT

Note: When a timer elapses, the **ERROR 254** is raised in BASIC.

CALLREGS:

```
A$="*****16****"  
CALL CALLREGS;A$
```

5.24/ t?PEEK

Waits on a counter semaphore *countid* associated with a lock *lockid* specified by **L**. The *lockid* and the *countid* are in range **0..7**. The *countid* is specified by the bits **6..4** of **L** and the *lockid* by the bits **2..0**. The task tries to own the lock *lockid* when entering into **t?PEEK**. If the lock may be owned, and the counter semaphore *countid* is not **0**, the task returns from the request with the ownership of the lock, and the counter is decremented by **1**. If the counter is **0**, the lock is freed, and the task waits until the counter become not null. If the lock is owned by another task, the calling task waits for the ownership of the lock. If a timer elapses during **t?PEEK**, the request is interrupted, and it returns with the **CARRY** cleared and the error code **t#ERR_TimedOut** into **H**.

Syntax:

```
LD    L,<countid><lockid>  
LD    D,&17  
CALL  t?KRNREQ  
JR    NC,error_handler
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_TimedOut A timer is elapsed, and the request is interrupted

BASIC:

TASK PEEK <lockid>,<countid>

Note: When a timer elapses, the **ERROR 254** is raised in BASIC.

CALLREGS:

```
A$="*****17**c1"  
CALL CALLREGS;A$
```

5.25/ t?POKE

Signal the counter semaphore *countid* associated with a lock *lockid* specified by **L**. The *lockid* and the *countid* are in range **0..7**. The *countid* is specified by the bits **6..4** of **L** and the *lockid* by the bits **2..0**. The task should own the lock *lockid* when entering into **t?POKE**. The counter is incremented by **1** and the lock is freed. If the task is not owner of lock, an

error is returned. If the bit 7 in **L** is set to 1, the counter is reset to 0 and no signal is sent. This is useful to initialize the counters.

Syntax:

```
LD    L,<countid><lockid>
LD    D,&18
CALL  t?KRNREQ
JR    NC,error_handler
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_LockNotOwner The lock is owned by another task.

BASIC:

```
TASK POKE [CLS] <lockid>,<countid>
Note: CLS clears the counter instead of signal it.
```

CALLREGS:

```
A$="*****18**c1"
CALL CALLREGS;A$
```

5.26/ t?DIM

Declares the queue pointed by **BC** of the length **L** for the calling task. The length should be at least 6, because the kernel uses 4 bytes for its pointers, and each messages use 1 byte for its length. If the queue is already declared, an error is returned.

Syntax:

```
LD    BC,<buffer>
LD    L,<length>
LD    D,&19
CALL  t?KRNREQ
JR    NC,error_handler
```

Output: If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

t#ERR_QueueBusy The queue is already installed
t#ERR_QueueFull The queue length is too small (<6)

BASIC:

```
TASK DIM <queue>,<length>
```

CALLREGS:

```
A$="**bbcc19**11"
CALL CALLREGS;A$
```

5.27/ t?READ

The task tries to read an asynchronous message from its queue. The buffer to receive the message is in **BC**. If a message could be read from the queue, it is copied in the space pointed by **BC** and the length is returned into **L**. If the no message is present in the queue, an error is returned. The queue is anonymous.

Syntax:

```
LD    BC,<buffer>
LD    D,&1A
CALL  t?KRNREQ
```

Output: If the **CARRY** is set, the request was successful and **L** contains the length of the message read, else **H** contains the error code.

Error:

```
t#ERR_QueueEmpty    There are no message in the queue
```

BASIC:

```
TASK READ <var$>
```

CALLREGS:

```
A$="**bbcc1A****"
CALL CALLREGS;A$
A$ shows "**bbcc**11" with 11 the received length.
```

5.28/ t?READWAIT

The task tries to read an asynchronous message from its queue. The buffer to receive the message is in **BC**. If a message could be read from the queue, it is copied in the space pointed by **BC** and the length is returned into **L**. If the no message is present in the queue, the task waits until a message is sent (t?DATA) or until a timer elapses. The queue is anonymous.

Syntax:

```
LD    BC,<buffer>
LD    D,&1B
CALL  t?KRNREQ
```

Output: If the **CARRY** is set, the request was successful and **L** contains the length of the message read, else **H** contains the error code.

Error:

```
t#ERR_TimedOut      A timer is elapsed, and the request is interrupted
```

BASIC:

```
TASK READ WAIT <var$>
```

*Note: When a timer elapses, the **ERROR 254** is raised in BASIC.*

CALLREGS:

```
A$="**bbcc1B****"
```

```
CALL CALLREGS;A$
```

A\$ shows "**bbcc**11" with 11 the received length.

5.29/ t?DATA

Sends the asynchronous message pointed by **BC** of the length **L** to the task specified by **TID** in **A**. If the receiver task has no queue, or if there are not enough space in the queue to store the message, an error is returned. On each message, the kernel adds 1 byte for the length when storing it in the queue.

Syntax:

```
LD BC,<buffer>
```

```
LD D,&1C
```

```
CALL t?KRNREQ
```

```
JR NC,error_handler
```

Output:If the **CARRY** is set, the request was successful, else **H** contains the error code.

Error:

```
t#ERR_NoSuchQueue The task has no queue
```

```
t#ERR_QueueFull There are no space in the queue for the message
```

BASIC:

```
TASK DATA <tid>,<var$>
```

CALLREGS:

```
A$="0tbbcc1C**11"
```

```
CALL CALLREGS;A$
```

6/ Examples

The examples described in this chapter are all given as source in the **mtask2015/** directory. Each memory module has two images built for it address scheme. Use **Images/ex/exall-ceMMM.wav** or **Images/ex/exsched-ceMMM.wav** for an audio download or **Images/ex/exall-ceMMM.bin158** or **Images/ex/exsched-ceMMM.bin158** for a serial download.

6.1/ exall.asm

The 7 examples shown in this source show some inter-tasks interactions. They are designed to be played with the BASIC instruction **TASK**, but **CALLREGS** is fully usable.

In this chapter, we assume that the images are for a **CE-159** module (address from **&2000..&47FF**). We use the memory area from **&4000..&47FF** as stack and queue area. We also assume that the scheduler is reinitialized after each example by a **CALL t_TASKINIT**.

6.1.1/ Flip-flop the II indicator using t?PAUSE

ex1 is at the address **&2C00**

```
ex1:
    ;; Flip-flop the LCD indicator II
    LDA (HIGHLCDFLAG)
    XOR &20
    STA (HIGHLCDFLAG)
    ;; Pause task for a delay of &80 ticks
    LD B,&00
    LD C,&80
    LD D,t?PAUSE
    CALL t?KRNREQ
    JR ex1
```

In this example, the task does a flip-flop with LCD indicator **II**. It performs a **t?PAUSE** for **&0080** ticks, and inverse the LCD indicator.

To start it, do:

```
BASIC:    TASK RUN &2C00,&47FF,2;T
```

When returning, the variable **T** should be **1**. The flip-flop starts.

```
CALLREGS: A$="F247FF022C00"
```

```
CALL &27FA,A$
```

```
A$ -> shows: "0147F37844"
```

6.1.2/ Send HELLO to BASIC task with PRINT

ex3 is at the address **&2C20**

ex3:

```
LD    BC,PC
LDA   *_ex3hello
ADD   BC
;; Send the message pointedb by BC of length L
LD    L,&05
LDA   &00
LD    D,t?PRINTWAIT
CALL  t?KRNREQ
JR    ex3
```

In this example, the task tries continuously to send the message "**HELLO**" to task of **TID 0** (the BASIC task).

To start it, do:

```
BASIC:  TASK RUN &2C20,&47FF,2;T
When returning, the variable T should be 1.
TASK INPUT H$
H$ shows "HELLO"
Doing a t?INPUT will receive the message
```

```
CALLREGS: A$="F247FF022C20"
CALL &27FA,A$
A$ -> shows: "0147F37844"
A$="0076800B0010"
A$ -> shows: "0176800510"
H$ -> shows: "HELLO"
```

6.1.3/ INPUT a message and return in lowercase

ex4 is at the address **&2C40**

ex4:

```
;; Wait for incoming messages
;; BC points to buffer, L is the max length
LD    B,&7B
LD    C,&60
LD    L,&20
LD    D,t?INPUT
CALL  t?KRNREQ
;; Save sender TID and received length
PUSH  A
PUSH  BC
LDA   H
DEC   A
```

```

        STA L
_xor20loop:
        ;; Xor the bit5 (&20). So exchange upper/lower
        LDA (BC)
        XOR &20
        STI (BC)
        DJC _xor20loop
        PUSH HL
        ;; Do a delay before sending back the message
        LD B,&02
        LD C,&00
        LD D,t?PAUSE
        CALL t?KRNREQ
        ;; BC points to the message to send
        ;; L is the length
        ;; A the TID
        POP HL
        POP BC
        LDA H
        STA L
        POP A
        LD D,t?PRINTWAIT
        CALL t?KRNREQ
        JR ex4

```

In this example, the task waits for a message on **t?INPUT**, inverts the bit 5 of all bytes of the message and returns it to sender. A call to **t?PAUSE** is also done to illustrate the *INPUT-STATE* of the caller.

To start it, do:

```

BASIC:  TASK RUN &2C40,&47FF,3;T
        When returning, the variable T should be 1.
        H$="HeLlO-wOrLD"
        TASK PRINT INPUT 1,H$
        H$ shows "hE110-wOrld"

```

```

CALLREGS: A$="F347FF022C40"
        CALL &27FA,A$
        A$ -> shows: "0147F37844"
        H$="HeLlO-wOrLD"
        A$="0176800B0010"
        A$ -> shows: "0176801010"
        H$ -> shows: "hE110-wOrld"

```

6.1.4/ Handle and beep on event 5

ex5 is at the address **&2C80**

ex5:

```
LD BC,PC
LDA *_*_handler
ADD BC
;; Install the event handler pointed by BC
LD D,t?GOTO
CALL t?KRNREQ
;; Enable the event 5
LD L,&05
LD D,t?ON
CALL t?KRNREQ
;; Stop me
LDA &FF ; myself
LD D,t?STOP
CALL t?KRNREQ
RET
__handler:
;; In the handler:
;; >> H and L the event
;; >> A the sendert TID
DI
CALL BEEP1
EI
RET
```

In this example, the task install its event handler with **t?GOTO**, enable the event 5 with **t?ON** and stops itself. When a task send the event 5 to this task (**t?CALL**), a beep is eared. If another event is nothing no beep is eared.

To start it, do:

```
BASIC: TASK RUN &2C80,&47FF,1;T
When returning, the variable T should be 1.
TASK CALL 5,1
A beep is done.
TASK CALL 2,1
No beep.
```

```
CALLREGS: A$="F147FF022C80"
CALL &27FA,A$
A$ -> shows: "0147F37844"
A$="010000130005"
A beep is done.
A$="010000130002"
No beep.
```

6.1.5/ Read time and beep when mm is 0

ex6 is at the address &2CC0

ex6:

```
;; We should insure the re-entrance of
;; the XREG register because this
;; register is used for arguments,
;; computation, save it into our stack
    LD    B,<XREG
    LD    C,>XREG
    LD    L,&07
;; The save/restore loop should be done
;; with interruptions disabled
    DI
pushXREGloop:
    LDI   (BC)
    PUSH A
    DJC  pushXREGloop
;; Call TIME function
    CALL &E5B4    ; TIME
    DEC  BC
    DEC  BC
    DEC  BC
;; Get the minutes
    LDI   (BC)
    INC  BC
    STA  H
    LD   L,&07
;; Restore the XREG register from stack
popXREGloop:
    POP  A
    STD  (BC)
    DJC  popXREGloop
    EI
;; Re-enable the interruptions. From
;; here, could be rescheduled
    LDA  H
;; Check if digit of Minute is 0
    BIT  0F
    JR   NZ,notMM=00
    CALL BEEP1
notMM=00:
;; Pause task for a delay of &40 ticks
    LD   B,&00
    LD   C,&40
    LD   D,t?PAUSE
    CALL t?KRNREQ
    JR   ex6
```

In this example, the task reads the time periodically and beeps while the minute digit is **0**. This programs shows also how to protect against reentry of **XREG** area (**&7A00**).

To start it, do:

BASIC: TASK RUN &2CC0,&47FF,1;T

When returning, the variable **T** should be **1**.

Look for **TIME** value and when time is **MMDDHH:MOss**, a beep done.

CALLREGS: A\$="F147FF022CC0"

CALL &27FA,A\$

A\$ -> shows: "0147F37844"

When returning, the variable **T** should be **1**.

Look for **TIME** value and when time is **MMDDHH:MOss**, a beep done.

6.1.6/ Wait for counter 7 on lock 6 and beep when signaled

ex7 is at the address **&2D00**

ex7:

```
;; Wait for counter 7 on lock 6
LD    L,&76
LD    D,t?PEEK
CALL  t?KRNREQ
;; Signaled. Do a beep
CALL  BEEP1
JR    ex7
```

In this example, the task waits for the counter semaphore **7** is signaled. The semaphore is associated to the lock **6**.

To start it, do:

BASIC: TASK LOCK 6

TASK POKE CLS 6,7

The BASIC task takes the lock **6**, clears the count **7**. The lock **6** is "free".

TASK RUN &2D00,&47FF,1;T

When returning, the variable **T** should be **1**.

The task is waiting in **t?PEEK**.

TASK LOCK 6

TASK POKE 6,7

A beep is done.

CALLREGS: A\$="00000070006"

CALL &27FA,A\$

A\$="0000001800F6"

CALL &27FA,A\$

A\$="F147FF022D00"

CALL &27FA,A\$

A\$ -> shows: "0147F37844"

```

A$="000000070006"
CALL &27FA,A$
A$="000000180076"
CALL &27FA,A$
A beep is done.

```

6.1.7/ Try to send WORLD with DATA, arm a TIME and WAIT

ex8 is at the address &2D20

ex8:

```

        LD    BC,PC
        LDA   *_ex8world
        ADD   BC
;; Send the message pointed by BC of length L
;; to the TID 0
        LD    L,&05
        LDA   &00
        LD    D,t?DATA
        CALL  t?KRNREQ
;; Arm a timer of &200 ticks
        LD    B,&02
        LD    C,&00
        LD    D,t?TIME
        CALL  t?KRNREQ
;; Do a wait
        LD    D,t?WAIT
        CALL  t?KRNREQ
;; Exit from wait with CARRY set
;; Awaken by a t?UNLOCK, t?DATA, t?POKE
        JR    C,ex8beep
;; CARRY cleared and timer elapsed ?
        CP    H,t#ERR_TimedOut
        JR    NZ,ex8
;; Yes: Flip-flop the LCD indicator G
        LDA   (LOWLCDFLAG)
        XOR   &02
        STA   (LOWLCDFLAG)
        JR    ex8
ex8beep:
        ;; Awaken: do a beep
        CALL  BEEP1
        JR    ex8
        .TEXT
ex8world:
        "WORLD"

```


In this example, the task send "WORLD" with **t?DATA** to the BASIC task, arm a timer (**t?TIME**) and wait (**t?WAIT**). If the timer elapses, the task flip-flop the LCD indicator **G**. If the **t?WAIT** exists with success, a beep is done.

To start it, do:

BASIC: TASK RUN &2D20,&47FF,1;T

When returning, the variable **T** should be 1.

The task is waiting in **t?WAIT**. After a couple of seconds, see the **G** changes.

TASK LOCK 6

TASK UNLOCK 6

A beep is done.

TASK DIM &4700,&20

TASK READ D\$

A\$ -> shows: "WORLD"

CALLREGS: A\$="F147FF022D00"

CALL &27FA,A\$

A\$ -> shows: "0147F37844"

The task is waiting in **t?WAIT**. After a couple of seconds, see the **G** changes.

A\$="000000070006"

CALL &27FA,A\$

A\$="000000080076"

CALL &27FA,A\$

A beep is done.

A\$="000000070006"

CALL &27FA,A\$

A\$="000000180076"

CALL &27FA,A\$

6.2/ exsched.asm

The example described here is a small schedule-reminder. The task waits for a message defining an alarm, and when the time is reached, the message programmed is show to LCD and 5 beeps are done.

The alarm is send as asynchronous message (**t?DATA**) and its format is: "**HhMm CCCCC**" where **Hh** is the hour and **Mm** the minutes of the alarm. When the alarm elapses, the message **CCCCC** is printed to the LCD. If the alarm programmed is already past, the reminder is done immediately.

Imagine that the current time is **17h40m**. A new alarm, ALARM is scheduled for **17h42m**.

To start it, do:

exsched is at the address **&2E00**

BASIC: TASK RUN &2E00,&47FF,0;T

When returning, the variable **T** should be 1.

The task is waiting in **t?READWAIT**.

A\$="1742 ALARM"

TASK DATA 1,A\$

Poll for **TIME**. At *mmdd17.42ss*, the message " **ALARM**" is shown on LCD and 5 beeps are done.

CALLREGS: A\$="F047FF022E00"

CALL &27FA,A\$

A\$ -> shows: "0147F37844"

The task is waiting in **t?READWAIT**.

H\$="1742 ALARM"

A\$="0176801C000A"

CALL &27FA,A\$

A\$="0176801B0A"

Poll for **TIME**. At *mmdd17.42ss*, the message " **ALARM**" is shown on LCD and 5 beeps are done.

The source code shown below is **exsched.asm**

```

        .INCLUDE:      mtask2015.inc

        .CODE
        .COMMENT:     "A schedule-reminder task. Copyright 2015 Christophe Gottheimer
<cgh75015@gmail.com>

schednbttime: .EQU    #10

;; This the schedule-reminder task.
;; The aim of this program is receive some alarms or reminders and to perform
;; some actions when the alarm time is reached.
;; The time checked is hour and minute. A action is a space to print the 5
;; characters of the message to the LCD, or a MT-kernel-request to call, with
;; its arguments: The MT-kernel-request may be one of t?CONT t?PRINT t?CALL or
;; t?DATA.
;; To program an alarm, a message of 10 characters has to be sent to the task
;; with a t?DATA. The format of the message is:
;;   "HHhhMMmm ccccc" where HHhhMMmm is the time of the alarm in ASCII, and
;;   ccccc is the message to be printed at alarm time.
;;   "HHhhMMmmkabchl" where HHhhMMmm is the time of the alarm in ASCII, k is
;;   the MT-kernel-request in binary and a b c h l are the
;;   binary values to fill into the registers D A BC HL
;; To start the task with TASK:
;;   TASK RUN &SCHED,<stack>,1;T
;; To send an alarm at 21:45 to remind to go to sleep:
;;   T$="2145 SLEEP"
;;   TASK DATA T,T$
;; At 21:45, 5 beeps and message SLEEP will be displayed

        .ALIGN:       0100

SCHED:
;; First start. Clear the alarm array: 10 alarms x 8 bytes
LD      B,<schedtab
LD      C,>schedtab
LD      L,[-1][*8]schednbttime
LDA     &FF

schedr0loop:
STI     (BC)
DJC     schedr0loop

;; Declare the queue for incoming alarm messages

```

```

LD    B,<schedqueue
LD    C,>schedqueue
LD    L,&20
LD    D,t?DIM
CALL  t?KRNREQ

```

schedloop:

```

;; Loop on TIME and WAIT. Arm TIME and perform a READ WAIT
LD    B,&00
LD    C,&40
LD    D,t?TIME
CALL  t?KRNREQ

LD    B,<schedmsg
LD    C,>schedmsg
LD    D,t?READWAIT
CALL  t?KRNREQ
;; No error (SCF), an incoming message is present in the queue
JR    C,schedread

;; Error! If Timed-out, its time to check for past-due alarms
;; Other errors, just return to main loop
CP    H,t#ERR_TimedOut
JR    NZ,schedloop

```

schedtime:

```

;; To read the TIME (&E5B4), the task need to insure the XREG
;; re-entrance. To do that, the task save the current XREG
;; into its stack, read the TIME, save the bytes Hh and Mm,
;; restore the XREG. To be sure to work safe, the interruptions
;; are disabled during the save..read..restore steps
LD    B,<XREG
LD    C,>XREG
LD    L,&07
;; Disable the interruptions
DI

```

schedtmloop1:

```

;; Save XREG into stack
LDI   (BC)
PUSH  A
DJC   schedtmloop1
;; Read the current time
CALL  &E5B4
DEC   BC
DEC   BC
DEC   BC
;; Save Mm and Hh
LDD   (BC)
STA   (schedMN)
LDI   (BC)
STA   (schedHR)
;; Restore XREG from stack
INC   BC
INC   BC
LD    L,&07

```

schedtmloop2:

```

POP   A
STD   (BC)
DJC   schedtmloop2
;; Re-enable the interruptions
EI

;; Check for past-due alarms
LD    B,<schedtab
LD    C,>schedtab

```

```

        LD      L,[-1]schednbttime
schedtimeloop:
        LDI     (BC)
        ;; &FF if no alarm entry
        CPA     &FF
        JR      Z,schednexttime
        ;; If (hour < Hh) or ((hour = Hh) and (minute <= Mm))
        ;; the alarm is reached
        CPA     (schedHR)
        JR      NC,schedelapsd
        JR      NZ,schednexttime
        LDA     (BC)
        CPA     (schedMN)
        JR      NC,schedelapsd
        JR      Z,schedelapsd
schednexttime:
        ;; Skip to the next alarm
        LDA     &07
        ADD     BC
        DJC     schedtimeloop
        ;; No more alarm, return to main loop
        JR      schedloop
schedelapsd:
        INC     BC
        LDA     (BC)
        ;; If it is a space, display the 5 characters message after
        CPA     $:space
        JR      NZ,scheddokrnrq
        PUSH    BC
        PUSH    HL
        ;; Display " ccccc" and play 5 beeps
        LD      HL,BC
        LD      C,&06
        CALL    &ED3B
        CALL    BEEP1
        CALL    BEEP1
        CALL    BEEP1
        CALL    BEEP1
        CALL    BEEP1
        POP     HL
        POP     BC
schedackdonexttime:
        DEC     BC
        DEC     BC
        ;; Clear the alarm
        OR      (BC),&FF
        INC     BC
        ;; Check for the next alarm
        JR      schednexttime
scheddokrnrq:
        ;; Check for valid MT-kernel-request
        CPA     t?CONT
        JR      Z,scheddo
        CPA     t?PRINT
        JR      Z,scheddo
        CPA     t?CALL
        JR      Z,scheddo
        CPA     t?DATA
        ;; Not valid, clear the alarm and check next
        JR      NZ,schedackdonexttime
scheddo:
        PUSH    BC
        PUSH    HL
        ;; Retrieve the 6 binary values for the parameters from
        ;; the 5 bytes: DD AA BBCC HLLL

```

```

        LDA    &05
        STA    L
        ADD    BC
scheddoargloop:
        LDD    (BC)
        PUSH  A      ; Push register values: D A BC HL
        DJC    scheddoargloop
        POP    DE
        LDA    E
        POP    BC
        POP    HL
        ;; Call the MT-kernel
        CALL  t?KRNREQ
        POP    HL
        POP    BC
        ;; Clear the alarm and check next
        JR     schedackdonexttime

schedread:
        ;; A incoming message is read. Check for a free alarm
        ;; entry the alarm array
        LD     D,<schedtab
        LD     E,>schedtab
        LD     L,[-1]schednbtime
schedfetchfreeloop:
        LDA    (DE)
        CPA    &FF
        JR     Z,schedfree
        LDA    &08
        ADD    DE
        DJC    schedfetchfreeloop
        ;; Nothing free... Check for alarm
        JR     schedtime

schedfree:
        ;; Alarm free, read the ASCII HhMm from the message
        ;; and convert to binary values
        LD     L,&01
schedconvhrmn:
        LDI    (BC)
        AND    &0F
        AEX
        STA    H
        LDI    (BC)
        AND    &0F
        RCF
        ADC    H
        STI    (DE)
        DJC    schedconvhrmn
        ;; Copy the 6 bytes into the alarm array
        LD     L,&05
schedcploop:
        LDI
        DJC    schedcploop
        ;; Check for alarm
        JR     schedtime

        .BYTE

schedHR:
        &00
schedMN:
        &00

schedtab:
        .FILL: 0050 WITH &FF

```

```
schedqueue:  
  .SKIP: 0018  
schedmsg:  
  .SKIP: 0018  
  
  .END
```

7/ Build programs for MTASK2015 with the lhTools

To build a task program for **MTASK2015**, the source **mtask2015.inc** should be included into the source code. This file defines the **t?KRNREQ** entry point (**&79FA**) and all *MT-kernel-requests* id and all error codes.

Example:

```
.INCLUDE:  mtask2015.inc

.CODE
.PRINT2 " &" ex1 ": Example 1 - Flip-flop the II indicator"
ex1:
    LDA    (HIGHLCDFLAG)
    XOR    &20
    STA    (HIGHLCDFLAG)
; ; Pause task for a delay of 0x80 ticks
    LD     B,&00
    LD     C,&80
    LD     D,t?PAUSE
    CALL  t?KRNREQ
    JR     ex1

.END
```

The source **ex1.asm** should just be assembled by the assembler with **lhasm ex1.asm** to produce a binary file **ex1.bin** located at **&40C5**. Refer to the **lhTools** documentation for further informations (<http://www.pc1500.com/lhTools.html>).

To build a BASIC program using **TASK**, the keyword file **taskbas.kyw** should be imported into the BASIC source. This file defines the **TASK** instruction for BASIC compiler.

Example:

```
.IMPORT:  taskbas.kyw

.BASIC
10 TASK ARUN
20 TASK DIM &78C0,&40
30 TASK READ WAIT M$
40 PRINT "Message from t?READ: ";M$
50 BEEP 5:END

.END
```

The BASIC source **t1.bas** should just be compiled by the assembler with **lhasm t1.bas** to produce a binary file **t1.bin**. When downloading BASIC images built on the PC-1500, to decompile the BASIC, just call **lhdump -K taskbas.kyw image.bin**.

8/ License

Copyright 1992-1993-2015 Christophe Gottheimer <cgh75015@gmail.com>

MTASK2015 is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software Foundation. Note that I am not granting permission to redistribute or modify **MTASK2015** under the terms of any later version of the General Public License.

This program is distributed in the hope that it will be useful (or at least amusing), but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program (in the file "COPYING"); if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.