Retro

Tiny

Multitasking

kernel

for Z80 based computers

RTM/Z80

v 2.6

Software and documentation by Szilagyi Ladislau

Contents

Introduction – learning about multitasking

The RTM/Z80 project is intended to offer, to the retro-computer hobbyists and to anyone willing to learn about multitasking systems, the necessary knowledge needed to understand and learn the basics of this interesting but difficult area of software engineering.

Nothing compares to learning through practice, and I hope that using RTM/Z80 will help.

I was “exposed” to the “multitasking” topic some 40 years ago.

Back in 1978, as a programmer, I was allowed to use a computer named PDP-11, built by the late Digital Equipment Corporation, using RSX-11M as operating system. This was one of the first good multitasking, real-time operating systems ever designed.

Also, my job included programming an INTEL 8080 based computer.

I managed to write for this INTEL 8080 computer a small kernel (less than 8 Kbytes), containing multitasking support, I/O drivers (on interrupts) for the real-time clock, serial console, digital and analog inputs/outputs, punched paper tape (!) reader, etc.

It was used in Romania until the late 80’s in various industrial projects.

40 years after, I switched to ZILOG Z80, using the occasion provided by the creators of the RC2014 home-brew Z80-based computer, and I tried to re-build my old multitasking kernel on this new Z80 environment.

This is how the RTM/Z80 project started.

Some months later, I finished my first version of RTM/Z80 and I opted to make-it accessible to anyone interested.

In my opinion, it is a very good tool if you want to learn about multitasking.

I consider that, when someone tries to learn something connected to complex software systems, the best way is to be provided with all the possible resources, including the source code, a working hardware-software system, user manuals and implementation examples or demo programs.

Therefore, all the parts of this project, including the source code, are made accessible to anyone.

Summary

In computing, **multitasking** is the concurrent execution of multiple tasks over a certain period of time. New tasks can interrupt already started ones before they finish, instead of waiting for them to end.

As a result, a computer executes segments of multiple tasks in an interleaved manner, while the tasks share common processing resources such as central processing units (CPUs) and main memory. Multitasking automatically interrupts the running program, saving its state (computer register contents) and loading the saved state of another program while transferring control to it.

Multitasking is a common feature of computer operating systems. It allows more efficient use of the computer hardware; where a program is waiting for some external event such as a user input or an input/output transfer with a peripheral to complete, the central processor can still be used with another program.

RTM/Z80 is a multitasking kernel, built for Z80 based computers, written in Z80 assembly language, providing its users with an Application Programming Interface (API) accessible from programs written in the C language and the Z80 assembly language.

It is intended to be a simple and easy to use learning tool, for those who want to understand the tips and tricks of the multitasking software systems.

RTM/Z80 can be used on the following environments:

- Z80SIM Z80 simulator (e.g. on Windows, under CygWin)
- Z80ALL standalone homebrew Z80 computer (25MHz Z80, 4 x 32KB RAM, KIO, VGA, PS/2, DS1302)
- RC2014 homebrew Z80 computer, using the following hardware configuration options:
    - SC108(Z80 + 2x64KB RAM) + SC110(CTC+SIO)
    - any Z80 board + 64/128KB RAM + SC110(CTC+SIO)
    - any Z80 board + 512KB RAM+512KB ROM Memory Module + SC110(CTC+SIO)
- RCBUS based homebrew Z80 computer, using the following hardware configuration options:
    - SC706(Z80) + (SC707 / SC714 RAM) + (SC716(SIO) + SC718(CTC)) / SC725(CTC+SIO)
- any CPU Z80 board supporting IM2 + any 64KB RAM board + any CTC board + any SIO (or KIO board) (in this case, the I/O ports must be set in the source code, see manual, chapter Porting RTM/Z80 to other hardware)

RTM/Z80 mandatory hardware requirements:

- (at least) 64KB RAM
- support of Z80 Interrupt Mode 2
- CTC, interrupt enabled
- SIO or KIO, interrupt enabled

Basic RTM/Z80 characteristics and features:

- RTM/Z80 applications may run directly under CP/M 2.2 on any Z80 based computer as a .COM executable program, or may run on any simulated/emulated Z80 system, or on any Z80 retro/home brew computer (e.g. RC2014).
- It's very easy to build applications that use RTM/Z80. Using the HiTech C compiler, assembler and linker, the application can be quickly compiled, assembled and linked with the RTM/Z80 library, in a minimal number of steps.
- It's easy to configure. It can be quickly tailored according to the user options. Can be built as an object code library or can be ported to a (E)EPROM + RAM configuration.
- It's quite small. In its minimal version, it requires less than 4KB ROM plus 8K RAM. However, RTM/Z80 can be scaled-up easily: if your RC2014 hardware is provided with a 512KB ROM + 512KB RAM memory module, then RTM/Z80 can be configured to run very large multitasking applications (with total code & read-only data size up to 512KB) capable to allocate/deallocate very large amounts of read-write data (up to 460KB)
- It's fast enough. The following measurements are valid for a Z80 CPU running at 7.3728 MHz:
    - Task switching time (switching from a task executing a semaphore "signal" to another task "waiting" for the semaphore) is under 160 microseconds.
    - Allocate/deallocate a block of dynamic memory takes under 130 microseconds
    - Run/stop task operations are executed in less than 220 microseconds
- Its functions are easy to be used (no complicated C or assembler data structures need to be initialized)
- Provides 8KB of RAM dynamic memory, enabling users to allocate/deallocate blocks of memory of variable sizes, from 16 bytes to 4 K bytes.
- Users can define and run up to 32 concurrent tasks. The highest priority task gains CPU access.
- RTM/Z80 is basically a "cooperative" multitasking system; however, round-robin pre-emptive scheduling is possible, and can be switched on/off using the API
- Semaphores can be used to control access to common resources. Semaphores are implemented as "counting" semaphores. There is no limit regarding the number of semaphores that can be used.
- Queues and mailboxes can be used for inter-task communication. Messages can be sent and received between tasks. There is no limit regarding the number of queues or mailboxes that can be used.
- Timers can be used to delay/wait for variable time intervals (multiple of 5 milliseconds). There is no limit regarding the number of timers that can be used.
- Interrupt driven I/O drivers are used to perform I/O requests targeted to serial hardware devices (console, printer, etc.) for baud rates up to 115.2 K

RTM/Z80 is a “low profile” multitasking system; it is written entirely in Z80 assembler language and uses as a building platform the HiTech Z80 software, which may be run on a “real” Z80 computer or on a Z80 “simulator”. However, you can use also the C language and “mix” C code with Z80 assembly code when writing RTM/Z80 applications.

RTM/Z80 does not pretend to be a “real-time” system; for this target, you need much more powerful CPU power; the 7.3728 MHz Z80 is a low-placed processor in this perspective.

Building RTM/Z80 applications does not imply the use of any Unix/Linux development platform. All you need is CP/M, knowledge of Z80 assembly language or C language and being used to operate the HiTech tools (assembler, C compiler, linker).

RTM/Z80 is not a “competitor” of the many Z80 multitasking systems available on the market, it is only a learning tool for those who want to understand the “tips & tricks” of multitasking; because of this, its structure is simple and straightforward. However, the author tried to build also a versatile and efficient system, with performances comparable with other popular Z80 multitasking systems.

Many “hot” multitasking topics are discussed here, including dynamic memory management, priority inversion issue, task synchronization and communication mechanisms, starting from the basics but exposing also more complicated multitasking issues.

All the RTM/Z80 configuration options are grouped in a single text file, named “config.mac”. CP/M SUBMIT files are provided to help building RTM/Z80 and/or the RTM/Z80 applications. Details explaining how to configurate RTM/Z80, how to build applications written for RTM/Z80 and how to execute them on CP/M or RC2014 are included in the last part of this manual. Porting RTM/Z80 to any Z80-based, interrupts enabled (IM2) computer is possible.

Have a nice time studying this manual! Of course, any improvement suggestions related to the manual and/or the software are welcome!

Cooperative vs. Preemptive multitasking

In **preemptive** multitasking, the operating system can initiate a context switching from the running process to another process.

In other words, the operating system allows stopping the execution of the currently running process and allocating the CPU to some other process.

The OS uses some criteria to decide for how long a process should execute before allowing another process to use the operating system.

The mechanism of taking control of the operating system from one process and giving it to another process is called preempting or preemption.

In **cooperative** multitasking, the operating system never initiates context switching from the running process to another process.

A context switch occurs only when the processes voluntarily yield control periodically or when idle or logically blocked to allow multiple applications to execute simultaneously.

Also, in this multitasking, all the processes cooperate for the scheduling scheme to work.

Early versions of both Windows and Mac OS used cooperative multitasking. Later on, preemptive multitasking was introduced in Windows NT 3.1 and in Mac OS X. However, preemptive multitasking has always been a core feature of Unix based systems.

RTM/Z80 is basically a cooperative multitasking system; however, a mechanism called “round-robin scheduling” may be enabled/disabled using the API, providing access to (limited) preemptive multitasking.

See the “**Round robin scheduling**” chapter for details.

RTM/Z80 Application Programming Interface

RTM/Z80 functions may be called from C and/or Z80 assembler source code.

All these functions must be called after RTM/Z80 is started-up (using the StartUp function, which is the only exception to this rule).

At RTM/Z80 start-up, hardware interrupts are enabled. In applications written for RTM/Z80, disabling hardware interrupts should be used with care (if hardware interrupts are disabled for long periods of time, interrupt driven I/O and the real time clock will be critically affected and the system may crash).

For the C language, the calling procedure of RTM/Z80 functions is simple, you need only to insert in your source code the appropriate #include statements, then use the C language procedure calling, for example:

```
#include <dlist.h>
#include <balloc.h>
#include <rtsys.h>
…
struct Semaphore* S;
void (*fp)(void);
…

void AnotherTask(void)
{
…
  S=MakeSem();
  Signal(S);
  StopTask(GetCrtTask());
}

void Task(void)
{
  fp = AnotherTask;
  RunTask(0x60, (void*)fp, 5); /* Run AnotherTask with stack size=60H, prio=5 */
…
  Wait(S);
…
  ShutDown();
}

void main(void)
{
…
  fp = Task;
  StartUp(0x1E0, (void*)fp, 10); /* Start up, run Task with stack size=1E0H, prio=10 */
}
```

The RTM/Z80 functions, when called from C, save/restore all the Z80 registers (AF, BC, DE, HL, IX, IY, AF', BC', DE' and HL'), therefore the tasks written in C can be executed safely, even if they call LIBC functions using also the secondary set of registers (AF', BC', DE' and HL'), with the precaution to assign a large enough stack to these tasks (0x1E0 is a good-enough choice, see chapter Tasks).

When using code written in **Z80 assembler**, there are two possible approaches:

1) **The first approach** (“function parameters on the stack”) uses the stack to pass the parameters values to the RTM/Z80 functions:

```
; Calling Routine, defined as:
;char, short, void* Routine(P1, P2, ..., Pn);
;
       PUSH   Pn           ;last param pushed first!
       ...
       PUSH   P2
       PUSH   P1
       CALL   _Routine     ; !!! NOTE the leading _
;             ; return value in HL, or L
;             ; P1, P2, ..., PN are still on the stack
```

This approach has also the advantage of saving and restoring the values of all the Z80 registers (AF, BC, DE, HL, IX, IY, AF’, BC’, DE’ and HL’). If the function returns values, these are stored in HL (or L) registers, depending on the size of the returned value.

This way, the user can call any RTM/Z80 function without affecting the values of the main set of Z80 registers (except for the case of functions returning values in HL or L registers, when HL or L is affected).

Remember to push the parameters of a function in the reverse order and prefix the function name with a _ (underline), for example:

```
       LD           HL,5               ;task priority
       PUSH         HL
       LD           HL,TStart          ;task start address
       PUSH         HL
       LD           HL,0E0H            ;stack size
       PUSH         HL
       CALL         _RunTask           ;HL returned as new task TCB addr
       POP          BC                 ;drop parameters
       POP          BC
       POP          BC
```

2) **The second approach** (“function parameters values in registers”) is faster, loading the function parameters values directly into the Z80 registers, but <u>does not save/restore the Z80 registers</u>.

```
; Calling Routine, defined as:
;char, short, void* Routine(P1, P2, ..., Pn);
;
       LD     HL,…         ;Load parameters into registers
       LD     C,…
       CALL   __Routine    ; !!! NOTE the leading __ (double _)
                           ; return value in HL, or L
```

Remember to prefix the function name with a double _ (underline), for example:

```
LD          E,5                 ;task priority
LD          HL,TStart           ;task start address
LD          BC,0E0H             ;stack size
CALL        __RunTask           ;HL returned as new task TCB addr
```

The full set of API functions is described in the following chapters.

The “include” file and the list of parameters, in the case of C language (which is relevant also for the first Z80 assembler language approach), and the Z80 registers to be used for the parameters in case of the second Z80 assembler language approach (the returned value is always stored into HL or L register), are shown for each function.

RTM/Z80 can be configured (using the DEBUG option) to check the parameters of all the API functions. However, if DEBUG option is not selected, no such check will be performed.

The advantage of using the DEBUG option is that wrong API parameters will be detected; the disadvantage is the loss of performance (API calls are slowed down by this parameter checking).

The advantage of not using the DEBUG option is of course the speed gained; the disadvantage is that any wrong-value parameter used at any API call may cause (big) trouble, possible leading to system crash.

It is advisable to start testing an RTM/Z80 application always in DEBUG mode, to find all the possible programming errors, and only then to switch to the faster non-DEBUG mode.

All the “objects” used by RTM/Z80 (semaphores, task control blocks, queues, mailboxes, timers) are allocated in / deallocated from / the “common” dynamic memory, allowing for an efficient use of the available RAM memory.

Starting and shutting down RTM/Z80

#include <rtsys.h>

RTM/Z80 will execute a ”call **_main**” instruction, after being booted. The **“_main**” function must issue an RTM/Z80 **StartUp** call, in order to initialize all the system data structures and to start the first user task. Until **StartUp** is executed, no other RTM/Z80 functions must be called.

Examples:

(as the C language main() function)

```
void   main(void)
{
       …
       StartUp(TaskStackSize, MyFirstTask, TaskPriority);
}
```

(or as a label in assembler)

```
       psect  text
       GLOBAL _main
_main:
       ld     bc,0E0H      ;stack size
       ld     hl,Task      ;task start address
       ld     e,5          ;task priority
       call   __StartUp
       ret
```

(A **psect** is a named section of the program, in which code or data may be defined at assembly time. Usually **text** = code section, **data** = read-only data section and **bss** = read/write data section).

If RTM/Z80 is resident in ROM, then **_main** is set equal to 2800H (at boot, the system code will be copied from (E)EPROM to RAM and executed there, and the area starting from 2800H is reserved for the user) (see the Chapter Using RTM/Z80 in a ROM/RAM configuration).

struct TaskCB* StartUp(short stack_size, void* StartAddr, short Prio);
BC= Stack_Size, HL= StartAddr, E=Prio, returns Task Control Block (TCB) address of the task (or zero) in HL.

Data structures are initialized, hardware is setup, interrupts enabled, and the provided parameters are used to start the first task (See Chapter Tasks).

From this first task, other user tasks may be started. The address of the TCB for this task is returned (or NULL (zero) if no more dynamic memory was available).

void ShutDown(void);
All active, waiting and suspended tasks are stopped, pending I/O operations are aborted, the real-time clock is reset and the RTM/Z80 is shut down. For Z80SIM, control returns to CP/M, for RC2014, control returns to SCM, for Z80ALL, a hardware restart will be needed.

short GetHost(void);
Returns HL=0 if RTM/Z80 is executed on Z80SIM’s CPM, =1 if executed on RC2014, =2 for Z80ALL.

Lists handling

RTM/Z80 uses double-linked lists to manage all the “system objects” (allocated blocks of memory, tasks control blocks, semaphores, queues, mailboxes, timers, I/O requests, etc.).

All these system objects are resident in the dynamic memory (see the next chapter).

Each list has a “list header”, 4 bytes, containing the pointers to first and last list elements.

Each list element has a “link area”, 4 bytes, containing the pointers to the previous and next list elements.

The lists handling routines must be used only for lists with headers and elements allocated in the dynamic memory (using Balloc, see next chapter).

This is because the routines are optimised for speed, and the list and header pointers are incremented / decremented using only the low byte “part” of the 2-byte pointer (e.g. INC L, instead of INC HL).

Given the fact that all the addresses of the memory blocks allocated in the dynamic memory have values with their last 4 bits zero, this works well, but, again, it will not work for list headers and/or list elements placed randomly in the memory (consider the case when the header is placed at 0FFFH, incrementing only the low part of this address will produce the “next” address as 0F00H, which is, of course, wrong).

These functions are accessible only from Z80 assembler, passing the arguments via registers (second approach) and must be called only under interrupts disabled.

__***InitList*** – Initialize a list header

```
;       must be called under interrupts DISABLED
;       HL=ListHeader, returned
;       affected regs: HL,DE
;       A,BC,IX,IY not affected
```

__***AddToL*** – Adds a new list element to a list

```
;       must be called under interrupts DISABLED
;       HL=list header, DE=new
;       return HL=new
;       affected regs: A,BC,DE,HL
;       IX,IY not affected
```

__***FirstFromL*** – Returns a pointer to the first list element, or zero if list is empty

```
;       must be called under interrupts DISABLED
;       HL=list header
;       returns (HL=first and Z=0) or (HL=0 and Z=1 and CARRY=0)
;       affected regs: DE,HL
;       A,BC,IX,IY not affected
```

__***LastFromL*** – Returns a pointer to the last list element, or zero if list is empty

```
;       must be called under interrupts DISABLED
;       HL=list header
;       returns (HL=last and Z=0) or (HL=0 and Z=1)
;       affected regs: DE,HL
;       A,BC,IX,IY not affected
```

__***NextFromL*** – Returns a pointer to the next list element, or zero if this was the last one

```
;       must be called under interrupts DISABLED
;       DE=list header, HL=current element
;       returns (HL=next after current and Z=0) or (HL=0 and Z=1 if end-of-list)
;       affected regs: A,DE,HL
;       BC,IX,IY not affected
```

__***RemoveFromL*** – Removes a list element from the specified list

```
;       must be called under interrupts DISABLED
;       HL=element to be removed
;       Returns HL=Element
;       affected regs: A,BC,DE,HL
;       IX,IY not affected
```

__***InsertInL*** – Inserts a new list element before the specified current element of a list

```
;       must be called under interrupts DISABLED
;       HL=current element, BC=new element
;       returns HL=new element
;       affected regs: A,BC,DE,HL
;       IX,IY not affected
```

__***GetFromL*** – Removes the first element from the specified list (if any), or returns zero

```
;       must be called under interrupts DISABLED
;       HL=list header
;       returns (HL=element and Z=0) or (HL=0 and Z=1 if list empty)
;       affected regs: A,BC,DE,HL
;       IX,IY not affected
```

__***RotateL*** – Rotates the list elements (last becomes first, first becomes second, etc)

```
;       must be called under interrupts DISABLED
;       HL=list header
;       returns (HL=0 and Z=1 if list empty or has 1 element), else (HL=1 and Z=0)
;       affected regs: A,BC,DE,HL
;       IX,IY not affected
```

Memory allocation / deallocation

#include <balloc.h>

Dynamic memory allocation is when an executing task requests that the operating system give it a block of main memory. The task then uses this memory for some purpose.

Tasks may request memory and may also “return” previously dynamically allocated memory. Memory may be returned whenever it is no longer needed. Memory can be returned in any order without any relation to the order in which it was allocated.

8 Kbytes of RAM dynamic memory is available in RTM/Z80. All the “objects” (TCB’s, semaphores, queues, mailboxes, timers, etc.) created and managed by the RTM/Z80 system are allocated / deallocated from this area of memory.

The allocation algorithm used is the so-called “buddy-system” method (see Donald Knuth, The art of computer programming, vol 1-fundamental algorithms).

The buddy memory allocation technique is a memory allocation algorithm that divides memory into partitions to try to satisfy a memory request as suitably as possible. This system makes use of splitting memory into halves to try to give a best fit.

The size of a block is 2^n. Power-of-two block sizes make address computation simple, because all buddies are aligned on memory address boundaries that are powers of two.

When a larger block is split, it is divided into two smaller blocks, and each smaller block becomes a unique buddy to the other. A split block can only be merged with its unique buddy block, which then reforms the larger block they were split from.

In RTM/Z80, the smallest memory block that can be allocated has 16 (2^4) bytes, while the largest has 4096 (2^{12}) bytes.

void* Balloc(short Size);
C=”Size” (0 to 8), returns the address of the allocated memory block in HL, registers A,BC,DE are affected. NULL (zero) is returned if no more free memory is available.

Allocates a block of size 2^{Size+4} bytes and returns its address. Therefore, users may request to allocate blocks of memory starting from the minimal size of 10H(2^4) bytes to the maximal size of 1000H(2^{12}) bytes.

Each allocated memory block contains a “prefix” area (6 bytes), located at the beginning of the block. This area contains a “link” space (4 bytes, see previous chapter), then a byte containing the ID of the owner task, and then a byte containing the block size (0 to 8).
The last two bytes of the “prefix” of an allocated memory block must not be changed by the user, e.g. data must be stored outside (after) this “protected area”.
However, the “link” part of an allocated memory block (first 4 bytes) can be used by the list handling routines.

Therefore, you may use the starting address of an allocated block to add-it to a double-linked list, but when storing data to this block you must avoid altering the “prefix” area.

For example, in Z80 assembler code, after obtaining the address of a block of 10H bytes:

```
            DI
            LD    C,0
            CALL  __Balloc     ;HL=allocated block address
            EI
```

, the caller will use for storage only the area starting from HL+6:

```
            PUSH  HL           ;save block address on stack
            LD    DE,6         ;skip protected area
            ADD   HL,DE        ;HL=address of the storage area (size=0AH)
            EX    DE,HL        ;DE=address of the storage area (size=0AH)
            LD    HL,source    ;move 10 bytes from “source” to the allocated block
            LD    BC,10
            LDIR
            DI
            LD    HL,ListHeader
            CALL  __InitList   ;init list
            POP   DE           ;HL=header,DE=block address
            CALL  __AddToL     ;add block to the list
            EI
            …
ListHeader: defs  4
```

, but the block may be added to a list without any problem (however, this part must be executed under interrupts disabled).

In C code, this “protected” area is evident examining the structure of the allocated memory block:

```
struct bElement {
       void* next;
       void* prev;
       char Status; /* 0 = available, else ID of owner task */
       char Size; /* 0=10H to 8=1000H */
       /* DATA – must be stored here */
};
```

Note: calling __Balloc from assembly language (parameter values in registers) must be done under interrupts disabled.

short BallocS(short MemSize);
BC=MemSize (size in bytes, should be <= 1000H), returns BC=block size (0 to 8). If the parameter provided is > 1000H, it is “cut” to 1000H.

Returns the block size (0 to 8) corresponding to the provided memory buffer size; may be used to compute the parameter to be used at Balloc call, given the size of the buffer to be allocated, for example:

```
Balloc(BallocS(0x100)); /* allocate 0x100 = 256 bytes */
```

short Bdealloc(void* addr_block, short Size);
HL=block, C=Size (0 to 8) (Size is used only if compiled with DEBUG option), returns HL=NULL(0) if deallocation failed, else not NULL, registers A,BC,DE are affected.

Deallocates (frees) the specified memory block of address addr_block and size = 2^{Size+4}.

If DEBUG option was used at compiling the system, size is provided as a check parameter in order to avoid deallocating wrong blocks of memory. In this case, NULL (zero) is returned if wrong address or wrong size is provided or if the memory block is not allocated.

Note: calling __Bdealloc from assembly language (parameter values in registers) must be done under interrupts disabled.

void* Extend(void* addr_block);
HL=block, returns HL=extended block or NULL if extending fails, registers AF,BC,DE are affected.

Allocates a new memory block, double the size of the specified block, copies all the data from the old block to the new block, deallocates the old block, returns the new block address.

Note: calling __Extend from assembly language (parameter values in registers) must be done under interrupts disabled.

short GetMaxFree(void);
Returns in HL the “size” of the largest free memory block, as a value in the range 0 to 8.
Registers A,BC,DE,HL are affected.

If no more memory is available, returns -1 (0FFFFH).

short GetTotalFree(void);
Returns in HL the total amount (in bytes) of free memory available to be allocated.
Registers A,BC,DE,HL,IX are affected.

struct TaskCB* GetOwnerTask(void* adrblock)
HL= block. Returns HL=TCB address of task who owns the block of memory (the task who made the allocation of the block).

Using the upper 64KB RAM memory (if available)

Usually, the RC2014 hardware has 64K RAM plus 32K EPROM. Certain RC2014 hardware configurations (e.g. Steve Cousins SC108, Phillip Stevens MemoryModule) are provided with two banks of 64KB RAM (the “lower” and the “upper” 64KB RAM bank).

For these configurations, RTM/Z80 provides a practical way to store and retrieve large amounts of data (up to 48KB) in the “upper” RAM bank:

void LowToUp100H(void* From, void* To)
IX=address of the source data (in the lower RAM bank), IY=address of the destination buffer (in the upper RAM bank).
A total of 256 bytes (100H) are moved from the lower RAM to the upper RAM, and the IX and IY registers are incremented with 256 (in case you are using a call from Z80 assembly language, this means the pointers are updated, in position to handle a next (possible) LowToUp100H call).

void UpToLow100H(void* From, void* To)
IY=address of the source data (in the lower RAM bank), IX=address of the destination buffer (in the upper RAM bank).
A total of 256 bytes (100H) are moved from the upper RAM to the lower RAM, and the IX and IY registers are incremented with 256 (in case you are using a call from Z80 assembly language, this means the pointers are updated, in position to handle a next (possible) UpToLow100H call).

Be warned, these functions alter the contents of all Z80 registers!

These functions are provided only when, in the configuration file **config.mac**, the following two settings are chosen when building the RTM/Z80 system:

```
WATSON      equ 0  ;1=Watson is used (not for CP/M)
RAM128K     equ 1  ;0=only 64K RAM, 1= 2 x 64K RAM available
```

This means that we indicated that we have 2 x 64KB RAM, but we do not want that WATSON utility uses the upper 64KB RAM bank, making possible to use this upper 64KB RAM as a temporary storage-area for our RTM/Z80 application data.

The highest address, from this upper 64KB RAM bank, that is allowed to be accessed by these functions must not exceed 0DF00H; this is critical to be respected as a mandatory rule, because after the mentioned address there are stored the “service” routines that move bytes from the “lower” to the “upper” RAM banks (these routines, loaded by the **hexboo**t program, are stored at the same addresses in both the “lower” and “upper” RAM banks, in order to make possible the execution of these “service” routines).

If overwritten, these routines will fail to perform their duty, causing even a system crash.

So, as a basic rule, less than 56K bytes must be “stored” using the LowToUp100H function calls.

After the execution of the RTM/Z80 application that stored data (suppose less than 56KB) to the “upper” RAM bank using LowToUp100H, the data may be retrieved using UpToLow100H, to be displayed at the console (see **testxr.c** in folder TEST).

Using the extra 2 x 32KB RAM memory for Z80ALL

The Z80ALL standalone Z80 computer has 4 x 32KB RAM. Besides the ‘default’ 64KB RAM, it is possible to access the two extra 32KB RAM banks (mapped to 0 – 8000H):

- Move bytes from a 32KB bank to the dynamic memory
- Move bytes from the dynamic memory to a 32KB bank

, using the following functions:

void MoveFrom32K(int bank, void* source, void* dest, int count)
Moves ‘count’ bytes from ‘source’ address of the 32KB bank number ‘bank’ (0 or 1) to the ‘dest’ address in the dynamic memory; the source address must be <= 8000H, and the dest address must belong to the dynamic memory buffer.

void MoveTo32K(int bank, void* source, void* dest, int count)
Moves ‘count’ bytes from ‘source’ address in the dynamic memory to the ‘dest’ address of the 32KB bank number ‘bank’ (0 or 1); the dest address must be <= 8000H, and the source address must belong to the dynamic memory buffer

When the DEBUG condition is selected (in config.mac), these constraints are verified, and if not fulfilled, the Move function will NOT be executed; however, if DEBUG is not selected, it is the responsibility of the programmer to provide correct parameter values.

Tasks

#include <rtsys.h>

In RTM/Z80, up to 32 tasks can be executed concurrently. Each task has a “TCB” (Task Control Block), allocated by RTM/Z80 at task start, which contains also its “private stack” placed in the high (last) part of the TCB. Each task is uniquely defined by the address of its “TCB” (Task Control Block), created when the task is ‘started’.

A task may have one of the following states: *active, waiting* or *suspended*.

User tasks priorities range from 1 to 127, while the RTM/Z80 system tasks (I/O drivers, CMD task) are assigned with the range of priorities from 128 to 255 (with the special case of the garbage collector task, having the lowest possible priority = 0).

Active tasks are given access to the CPU time according to their priority (highest priority first), but in the situation when there are more than one active concurrent tasks, an optional round robin mechanism is provided to facilitate a fair sharing of the CPU time (CPU time slices are given according to the tasks priorities).

Tasks have also a unique ID (from 1 to 255), used by RTM/Z80 to “stamp” every block of memory allocated by the task. The “garbage cleaner” task will search for blocks of memory “stamped” with the task ID when trying to deallocate blocks of memory owned by terminated (stopped) tasks.

Active tasks may be suspended or stopped when necessary. After stopping a task, it can be started again, but a limit of 255 total “task executions” is imposed because the need for a unique ID for each task.

For example, it is possible to start then stop a task repeatedly, in a loop, for a limited number of times, but no more than 255 times.

Struct TaskCB* RunTask(short Stack_Size, void* StartAddr, short Prio);
BC= Stack_Size, HL= StartAddr, E=Prio (1 to 127), returns TCB address of the task in HL.

A task with the provided start address and priority is created (a TCB is allocated in the dynamic memory) and started, becoming active (its TCB address is inserted into the active tasks queue, according to its priority, then control is passed to the highest priority task from the active tasks queue). If the priority provided does not fit in the range 1 to 127, it will be set to 1.

If the TCB could not be allocated (there is no more free memory), or the maximum number of active tasks was reached, or more than 255 tasks were started, NULL(0) is returned.

The first parameter (Stack_Size) sets the size of stack for the new task.

The first 32 (20H) bytes in the TCB are used as a “private” area by the RTM/Z80 to store important data related to the task (priority, stack pointer, ID etc.), but the rest of the TCB area is used as the task stack area.

Because of this structure of the TCB area, and considering that all allocated memory blocks are sized as powers of 2, it is important to choose carefully the value of stack_size.

For example, it make sense to use the value 0E0H, because added to 20H (size of “private” area), it gives 100H, a power of 2 (the system will allocate for the TCB exactly 100H).

It would be unwise to use the value 0E8H, because it will make the system to allocate 200H (0E8H + 20H = 108H, and the “next” power of 2 is 200H), and it will generate a “loss” of 0F8H bytes (the system allocates 200H, but since the stack_size requested is 0E8H, it will set SP=TCB+20H+0E8H, wasting the last 0F8H bytes of the TCB – these bytes cannot be accessed nor used by the current task or other tasks).

Special care must also be given to correctly select an appropriate magnitude of the stack size, depending on the stack usage of the task.

Normally, 60H should be enough for a task written in Z80 assembly language (60H=96 bytes, equivalent to 48 CALL or PUSH instructions), but in the case when the task is written in C and heavily uses C library functions (e.g. sprintf), a larger size must be chosen (e.g. 1E0H).

If the system was built using the DEBUG setting, RTM/Z80 checks (at task switching time) the current active task stack pointer value versus the “remaining” stack space and issues a warning if the remaining stack space drops below 20H, by printing on the terminal (console) the current task TCB address (in hexadecimal) followed by an ‘!’.

Also, the ***StackLeft*** function (see below) may be used to obtain the amount of available stack space, and in case the available stack space dropped too low, the ***IncTaskStack*** function (see below) may be used to increase the task stack size.

It is important to keep low the number of concurrent active tasks; too many active tasks will affect the performance of the system.

short StopTask(struct TaskCB*);

HL=taskTCB, returns HL=NULL(0) if the parameter provided is not the address of a task TCB, else not NULL. Only valid parameters are accepted, a NULL (zero) value is returned when the function is called using a non-existent TCB address.

The specified task is terminated (task is removed from the active tasks queue), task’s TCB is deallocated, all timers started by the task are stopped, all I/O operations issued by the task are stopped. Control is passed to the highest priority task from the remaining active tasks queue.

The garbage cleaner task will automatically deallocate all the blocks of memory allocated but not yet deallocated by the terminated task (memory leaks).

After the last active user task executes StopTask, the RTM/Z80 system hosting the user applications is automatically shutdown.

struct TaskCB* GetCrtTask(void);
Returns HL=address of current task TCB. Returns the TCB address of the current task.

void Suspend(void);
Suspends the execution of the current task (task is removed from the active tasks queue). Control is passed to the highest priority task from the active tasks queue.

short Resume(struct TaskCB*);
HL=taskTCB, returns HL=NULL(0) if the parameter provided is not the address of a task TCB, or if the task is not suspended, else returns not NULL.

Resumes the execution of the specified suspended task (it is inserted into the active tasks queue, according to its priority, then control is passed to the highest priority task from the active tasks queue), and not NULL is returned.

The parameter passed to the function must be a valid TCB address of a suspended task, otherwise NULL (zero) is returned.

short StackLeft(struct TaskCB*);
DE=taskTCB, Returns in HL the size of the stack still available for the specified task. Registers A,BC,DE,HL are affected. The size of the stack still available is calculated subtracting from the stack size the size of the area already “used” by the PUSH operations executed before this call.

For example, if the task was created with a stack size of 60H (96 bytes), and the previous 2 CALL and 5 PUSH instructions “consumed” 7*2=14 bytes before the call of the function, the StackLeft function returns the value 82=96-14.

If the “stack left” drops below 20H (32 bytes), future CALL or PUSH operations risks “altering” the task TCB “private area”, leading to unpredictable system behaviour or even system crash.

In the case of “stack-hungry” code or “re-entrant” algorithms or tasks written in the C language using extensive C library functions, this function should be used frequently in order to provide an early warning.

As a generic rule, tasks written in C language using C library functions must have larger stack sizes (1E0H is a safe choice), while for the tasks written in Z80 assembler, an acceptable TCB size is 60H.

short IncTaskStack(short stackSize);
BC = stackSize. Returns not NULL if stack size was increased to the new value, else 0 if no more memory available or the provided size is less than the actual size of the stack.

This function sets the size of the current active (executing) task stack to the value provided as a parameter. A new (larger) TCB is built for the task, old stack is moved to new TCB, then the old TCB is deallocated. This way, the old contents of the stack is entirely preserved.

short GetTaskSts(struct TaskCB*);
BC=task TCB. Returns HL=1 if the task is active, 2 if is waiting a semaphore or 3 if is suspended. Returns HL=NULL(0) if the parameter provided is not the address of a task TCB.
This function returns the status of the specified task.

struct TaskCB* GetTaskByID(short ID);
C=ID. Returns the task TCB with the provided ID, or NULL if no such task exists.

This function searches the TCB of a task by its ID. Might be used to find out the owner of an allocated block of memory (see also ***GetOwnerTask*** from memory allocation/deallocation).

void GetTaskPrio(struct TaskCB*);
BC=TaskTCB. Returns HL=priority, or -1 if the parameter provided is not the address of a real task TCB.

This function returns the specified task's priority.

void SetTaskPrio(struct TaskCB*, short Prio);
BC=TaskTCB, E=Prio. Returns HL=0 if the parameter provided is not the address of a real task TCB, or if the priority specified is > 127, else returns 1.

The specified priority is set to the specified task, then control is passed to the highest priority task from the active tasks queue. That means, the current task may loose/gain access to the CPU, depending on the chosen new priority.

Using the 512KB ROM + 512KB RAM memory module (if available)

Usually, the RC2014 hardware has 64K (or 128KB) RAM plus 32K EPROM.

If your system has also a 512KB ROM + 512KB RAM memory module, then RTM/Z80 can be configured to run very large multitasking applications (with total code & read-only data size up to 512KB) capable to store and retrieve very large amounts of read-write data (up to 460KB).

In CONFIG.MAC, if your system has a 512KB ROM + 512KB RAM memory module, the system setting **M512** must be set to 1 and there is another setting, **EXTM512** that must be used also in order to configurate RTM/Z80.

There are two available configurations:

- 1- Only basic extended memory access is provided
- 2- Advanced code overlays and extended dynamic memory is provided

OPTION 1 – basic extended memory

If EXTM512 is not selected (EXTM512=0), then only basic memory access functions are provided:

short Save100H(void* From, void* To)
HL=source address (from 0 to BF00H), BC=high part of the destination (e.g. if the destination address is 1A000H then BC must be set to 1A0H - destination address may range from 00000H to 6FF00H); HL=1 is returned if OK, else HL=0 is returned if parameters are off-range.

A total of 256 bytes are saved to the 512KB RAM.

short Load100H(void* To, void* From)
DE=destination address (from 0 to BF00H), BC=high part of the source (e.g. if the source address is 1A000H then BC must be set to 1A0H - source address may range from 00000H to 6FF00H); HL=1 is returned if OK, else HL=0 is returned if parameters are off-range.

A total of 256 bytes are loaded from the 512KB RAM.

OPTION 2 - advanced code overlays & extended dynamic memory

If EXTM512 is selected (EXTM512=1), then code overlays support and more advanced memory access functions are provided.

Code overlays support

The memory is partitioned as follows:

P0=0000H – 3FFFH : partition reserved for RTM/Z80 kernel + main app task & read-only data
P1=4000H – 7FFFH : partition reserved for overlays (app tasks & read-only data)
P2=8000H – BFFFH : partition reserved for RTM/Z80 read-write data & main dynamic memory
P3=C000H – FFFFH : partition reserved for RTM/Z80 extended dynamic memory

An RTM/Z80 application will be configured as follows:

The main task (called at **StartUp**), built as a .COM executable, will be loaded in P0 ; its size must not exceed 16KB (RTM/Z80 kernel + app code + read-only data).

All the other tasks (with individual code + read-only size up to 16KB) will be stored on the 512KB EPROM, and will be executed using the function **RunTask512** :

void* RunTask512(short Stack_Size, void* StartAddr, short Prio, short ROMbank);

The first 3 parameters are identical with the RunTask parameters; the 4'th parameter ROMbank (with values from 1 to 31) is the index of the 16KB partition from the 512KB EPROM that contains the code of the task. The task will be loaded in the P1 partition.

The TESTS/512 folder contains an example of such an RTM/Z80 application where T.COM is the 'main' task(calling T1) and T1(calling T2) & T2 must be stored on the 512KB EPROM (T.C, T1.C, T2.C, MAKET.SUB, T1ROM.HEX, T2ROM.HEX, TROM.HEX, TBASE.SYM).

Extended dynamic memory support

If EXTM512 is selected (EXTM512=1), buffers of memory (with sizes up to 16KB bytes) may be allocated/deallocated, like in the case of the main dynamic memory. To access these buffers, a separate function will "select" the 16KB memory bank that contains the buffer. This extended dynamic memory can be used in parallel with the main dynamic memory.

char* alloc512(short size, char* bank)
BC=size of buffer to be allocated (in bytes), DE=pointer of memory bank.
Returns a pointer to the allocated buffer and the index of the 16KB memory bank that will contain this buffer, or NULL if no more memory is available. Buffers up to (16KB – 6) bytes may be allocated.

void free512(char* buf, char bank)
DE=buffer, A=bank
Deallocates (frees) the buffer.

void set512bank(char bank)
A=bank
Selects the bank of 16KB memory to be used. All read/write operations using a buffer, allocated with this method, must be prefixed with a call of set512bank.

short Get512Free(void)
Returns (in KB) the size of extended dynamic memory available.

In TESTS/512, tallo512.c is an example about how to use these extended dynamic memory support functions.

Semaphores

#include <rtsys.h>

A **semaphore** is a mechanism used to control the tasks access to a common resource.

Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores and are used to implement locks. RTM/Z80 implements **counting semaphores**; they contain a counter (2 bytes) and a priority-ordered queue of tasks waiting for this semaphore.

The counter indicates how many resources are available, while the queue is used to manage the tasks willing to access those resources. The queue is managed as a priority-ordered queue.

Two basic operations are defined: Signal and Wait.

Basically, Signal may be described as “if the waiting queue is empty, increment the counter, else take the first task from the queue and insert-it into the active tasks list (highest priority task will be given access to CPU)”, and Wait as “if the counter is zero, insert the current task into the queue (the remaining highest priority task will be given access to CPU), else decrement the counter and resume execution”.

For example, suppose we have a queue with limited storage capacity, let’s say 16 bytes.

In order to allow the access to this queue, two counting semaphores are needed:

- AvailableSpace, initialized with the counter=16
- AvailableData, initialized with the counter=0

Write-to-queue can be described as:

Wait(AvailableSpace)
Move a byte from the caller to the queue
Signal(AvailableData)

Read-from-queue can be described as:

Wait(AvailableData)
Move a byte from the queue to the caller
Signal(AvailableSpace)

struct Semaphore* MakeSem(void);
returns HL=SemAddr (or NULL if no dynamic memory available).

Allocates memory for the semaphore, set counter=0 and sets-up an empty queue.

void* ResetSem(struct Semaphore* SemAddr);
HL=SemAddr. If the parameter provided is not the address of a real semaphore or if a task is waiting for the semaphore, NULL(0) is returned, else a not NULL value is returned.

Resets the already allocate semaphore (set counter=0 and sets-up an empty queue).

short Signal(struct Semaphore* SemAddr);
HL=SemAddr. If the parameter provided is not the address of a real semaphore, NULL(0) is returned, else a not NULL value is returned. The __Signal assembler function returns only CARRY=1 if wrong semaphore address provided, else CARRY=0.

If the semaphore queue is not empty, the first task is extracted from the queue and inserted into the active tasks queue, according to its priority, then control is passed to the highest priority task.

If the queue was empty, the semaphore counter is incremented. The queue of tasks waiting for the semaphore is not a FIFO or LIFO queue, but a priority-ordered queue (according to the priority of the tasks waiting for the semaphore). Therefore, always the highest-priority task waiting for the semaphore will obtain the access to CPU time.

short Wait(struct Semaphore* SemAddr);
HL=SemAddr. If the parameter provided is not the address of a real semaphore, NULL(0) is returned, else a not NULL value is returned. The __Wait assembler function returns only CARRY=1 if wrong semaphore address provided, else CARRY=0.

If the semaphore counter is > 0, it is decremented and execution of the current task continues. Else, the current task is extracted from the active tasks queue and inserted into the semaphore queue, according to its priority, then control is passed to the highest priority task from the active tasks queue.

short DropSem(struct Semaphore* SemAddr);
HL=SemAddr. Returns a NULL when the parameter provided is not a semaphore or the semaphore queue is not empty, else returns a not NULL value. Frees the memory allocated for the semaphore.

short GetSemStatus(struct Semaphore* SemAddr);
HL=semaphore address. Returns in HL the semaphore counter. Returns a NULL when the parameter provided is not a semaphore.

Data Queues

#include <queue.h>

A data queue can store a number of data batches in a buffer allocated when making the queue.

Each data batch is a vector of 2-byte (word) elements.

An obvious use of the data queue is to send/receive pointers between tasks.

A queue may accept data batches with sizes up to 255 * 2 bytes.

The number of data batches allowed for a queue is limited to 255.

However, there is also a limitation related to the total size of such a buffer containing data batches (buffer size must be smaller than 1000H, the maximum size of a memory block possible to be allocated).

Examples of queues:
- A queue of 255 words (total size = 200H)
- A queue of 32 sets of 16 words (total size = 400H)
- A queue of 16 sets of 128 words (total size = 1000H)

A data queue is “guarded” by two private semaphores, used to manage the full/empty queue events.

Practically, the Write to queue and Read from queue operations can be described as follows:

Write:
Wait for available space in the queue
Move the data batch from the caller to the queue
Signal data available

Read:
Wait for data available
Move the data batch from the queue to the caller
Signal space available in the queue

struct Queue* MakeQ(short batch_size, short batch_count);
B=batch size (number of 2-bytes to be moved), C=batches count, returns HL=queue header (or NULL if no dynamic memory available). Returns NULL (zero) if the batch_size>255 or batch_count > 255 or no memory was available for the queue or buffer.

Allocates memory for the queue and initializes the data queue according to the provided batch_size and count of batches, allocates the necessary buffer to store the batches.

short DropQ(struct Queue* queue);
BC=queue. Returns NULL if the parameter provided is not the address of a queue or if the queue contains data not being read yet, else returns a not NULL value.

Deallocates the buffer allocated to the queue and to the data batches.

short WriteQ(struct Queue* queue, void* info);
BC=queue, DE=pointer to data. Returns NULL if the parameter provided is not the address of a queue, else returns a not NULL value.

Adds the data batch pointed by the provided address “info” to the queue. Practically, a number of 2*batch_size bytes will be copied from “info” to the queue. If no more space is available in the queue buffer, the caller will wait for the next ReadQ to free the necessary space.

short ReadQ(struct Queue* queue, void* buf);
BC=queue, DE=pointer to buffer. Returns NULL if the parameter provided is not the address of a queue, else returns a not NULL value.

Read and extract from the queue buffer the next data batch and stores it to the provided buffer. Practically, a number of 2*batch_size bytes will be copied from the queue to the provided buffer. If no data batch is available in the queue, the caller will wait for the first WriteQ to put a data batch in the queue.

short GetQSts(struct Queue* queue);
HL=queue. Returns in HL the number of data batches not yet read. Returns -1 if the parameter provided is not the address of a queue.

Mailboxes

#include <mailbox.h>

Mailboxes are provided to facilitate sending and receiving messages of fixed length between tasks. The length of a message is limited to 249 bytes.

For each message sent, a block of memory is allocated and the data is copied to it from the sender memory area.

When receiving a message, the data is copied from this block of memory to the receiver buffer, then the block of memory used to transport the data is deallocated.

To synchronize between senders and receivers of mails, an internal semaphore belonging to the mailbox is used.

Practically, the Write to mailbox and Read from mailbox operations can be described as follows:

Write:

Allocate a block of memory to store the message
Move the message from the caller to the block of memory
Add the block of memory to the mailbox list of messages
Signal message available

Read:

Wait for message available
Get the first block of memory from the mailbox list
Move the message from the block of memory to the caller
Deallocate the block of memory

struct Mailbox* MakeMB(short MesageSize);
C=MessageSize, returns HL=mailbox or NULL if no dynamic memory is available.

Allocates memory for the mailbox and initializes the mailbox message queue, sets the given size as the length (in bytes) of the messages and resets the internal semaphore. Returns NULL (zero) only if the given message size was > 249.

short DropMB(struct Mailbox* mb);
HL=mailbox

Deallocates the buffer allocated to the mailbox. Returns NULL if the parameter provided is not the address of a mailbox or if the mailbox queue is not empty (there are messages not yet read), else returns a not NULL value.

short SendMail(struct MailBox* MBox, void* Msg);
HL=MBox, DE=Msg

Allocates a memory buffer, copies the message to this buffer, adds-it to the mailbox list. Returns NULL (zero) only if could not allocate or the provided parameter is not a real mailbox.

short GetMail(struct MailBox* MBox, void* DestBuffer);
HL=Mbox, DE=buffer. Returns NULL (zero) only if the provided parameter is not a real mailbox.

Waits for the first available mail, then extracts the first mail from the mailbox list, copies the content to the destination buffer and deallocates the space needed for the message.

short GetMBSts(struct MailBox* Mbox);
HL=Mbox. Returns in HL the number of mails not yet read. Returns -1 only if the provided parameter is not a real mailbox.

Timers

When certain code sequences need to be executed repeatedly at certain time intervals, RTM/Z80 timers must be used. Timers use the computer CTC's interrupts (they occur each 5 ms for RC2014, or 2.5 ms for Z80ALL).

#include <rtclk.h>

struct RTClkCB* MakeTimer(void);
Returns HL=Timer or NULL if no dynamic memory available.
Allocates dynamic memory for the timer.
.
short DropTimer(struct RTClkCB* Timer);
HL=timer
Deallocates the buffer allocated to the timer. Returns NULL if the parameter provided is not the address of a timer or the timer is started (in this case, it must be stopped first).

short StartTimer(struct RTClkCB* Timer, struct Semaphore* Sem, short Ticks, char Repeat);
HL = Timer, DE = Sem, BC = Ticks, A=Repeat
Initializes and starts the Timer, using the provided Ticks parameter. After 5 * Ticks (2.5 * Ticks for Z80ALL) milliseconds, a **Signal**(Sem) will be executed by the real-time clock driver. If the Repeat parameter is zero, the timer is then stopped, else it continues to issue a **Signal**(Sem) each 5*Ticks milliseconds. A NULL(0) is returned if the Timer is already started.

Too many started Timers may affect the performance of the system!

short StopTimer(struct RTClkCB* Timer);
HL = Timer. Stops the specified timer. A NULL(0) is returned if the parameter provided is not a Timer.

long GetTicks(void);
Returns current counter of ticks (4 bytes: DE=low(counter), HL=high(counter)).

The system maintains a 4-bytes counter of 5ms ticks, which is made accessible using this function.

This counter is incremented each time a real time interrupt occurs (at each 5ms), in a round-robin manner (when reaching the maximum 4-bytes storage capacity, it is reset to zero).

Therefore, time intervals shorter than 2^{32} x 5 ms can be measured with 5ms accuracy.

short GetTimerSts(struct RTClkCB* Timer);
HL=Timer. Returns in HL the current (remaining) ticks for the timer. HL= -1 is returned if the parameter provided is not a real Timer.

void RoundRobinON(void);
Enables (starts) the use of "round-robin" scheduling.

void RoundRobinOFF(void);
Disables (stops) the use of "round-robin" scheduling.

Input/Output - Console (terminal) I/O functions

#include <io.h>

Read and write functions are provided, enabling reading or writing strings of characters from/to the terminal (console). The usual way to issue an I/O request should be:

```
I/O function(buffer, count, &Sem);
Wait(&Sem);
```

The I/O requests are sent from the user task to the CON_Driver, a high priority task, using a queue to handle these requests. The CON_Driver reads from the queue the I/O request and initiates the I/O operation; at the completion of the I/O operation, a Signal will be issued for the semaphore specified in the I/O request.

In the case of the read function:

- the ENTER key will be interpreted as an end of the read request, in this case the read buffer will contain only the characters read before ENTER, followed by the 'string terminator' character (0).
- the BACKSPACE key is processed as "erase last entered character", enabling line editing.

void CON_Read(void* buf, char len, void* SemAddr);
HL=buf, DE=Sem, C=len

Issues a read request to the terminal console, providing a buffer address, a counter (<=255 chars) and a semaphore address. The characters read will be stored into the buffer, followed by an extra byte with 0 (zero) value, acting as a 'string terminator' character (space must be provided in the buffer to accommodate this extra byte).

The console device driver will "process" the command and after finishing it will issue a Signal for the given semaphore. The Semaphore must be initialized before issuing any I/O command.

void CON_Write(void* buf, short len, void* SemAddr);
HL=buf, DE=Sem, BC=len

Issues a write request to the terminal console, providing a buffer address, a counter (<=255 chars) and a semaphore address. The requested number of characters will be written, without any other extra characters (no CR, LF or zero will be appended).

The console device driver will "process" the command and after finishing it will issue a Signal for the given semaphore. The Semaphore must be initialized before issuing any I/O command.

short CTRL_C(void);
Returns HL=1 if CTRL C was pressed, else returns HL=0. The control for CTRL C is made only during write operations or if no input/output operations are active.

char CON_Status(void);
Returns A=0 if no key was pressed during an 'idle' period of time (no active CON_Read or CON_Write), else returns the key pressed.

Asynchronous serial communications I/O functions

void Reset_RWB(void);
Initializes the communications support routines.

void WriteB(void* buf, short len, void* SemAddr);
HL=buf, DE=Sem, C=len (up to 255 bytes)

Issues a write request, providing a buffer address, a counter and a semaphore address. The device driver will ”process” the command and after finishing it will issue a Signal for the given semaphore. The Semaphore must be initialized before issuing any I/O command.

void ReadB(void* buf, char len, Semaphore* S, void* Timer, short TimeOut);
HL=buf, DE=Sem, C=len (up to 255 bytes), IY=Timer, IX=TimeOut

Issues a read request, providing a buffer address, a byte counter, a semaphore address, a timer address and a timeout interval (in 5ms units). If the requested number of bytes is read before the timeout expires, or if the timeout expires before receiving all the requested bytes, a Signal(S) is issued.

For example, if 255 bytes are expected to be read at a baud rate of 115.2K, a timeout equal to 10 is recommended (10*5=50 milliseconds).

short GetCountB(void);
Returns A (or low part of returned value) = the number of NOT read characters, and H (or high part of returned value) = I/O error code (10H=parity err, 20H=rx overrun err, 40H=crc/framing err), if any (0=no error).

Called after a ReadB, returns the number of NOT read bytes and the error/success code of the read request.

An example of how to use the “communications” read function:

```
            call  __Reset_RWB
            call  __MakeTimer
            ld    (Timer),hl
            call  __MakeSem
            ld    (Sem),hl
            ex    de,hl         ;DE=sem addr
            ld    hl,Buf
            ld    c,255
            ld    ix,10         ;timeout 50 ms
            ld    iy,(Timer)
            call  __ReadB
            ld    hl,(Sem)
            call  __Wait
            call  __GetCountB
            or    a             ;check # NOT read
            jr    z,AllBytesRead
            ld    a,h
            cp    …             ;see err code
            …
AllBytesRead:                   ;all CHAR_COUNT were read
```

These functions are intended to read and write bytes via the Z80 SIO serial asynchronous interface, offering support to implement communication protocols. The ability to handle such I/O functions is based on some technical characteristics of RTM/Z80 and Z80 SIO.

The RTM/Z80 has no “disabled” instruction sequences that execute longer than 175 microseconds (for a Z80 running at 7.3728 MHz). The ZILOG SIO has an internal 3-byte buffer to handle “not-read” bytes. These circumstances enable the RTM/Z80 serial communications driver to handle baud rates up to 115.2 K. Let’s explain why.

A baud rate of 115.2Kbits is equivalent to an average of 80 microseconds/byte.

Because RTM/Z80 has no “disabled” instruction sequences that execute longer than 175 microseconds, this leads to up to 3 bytes “waiting to be read” inside the disabled 175 microseconds interval, compatible with the size of the ZILOG SIO 3-byte internal buffer mentioned above.

Furthermore, a special 256 bytes “ring” buffer is used to accumulate incoming unsolicited inputs. When no I/O operation is active or during a WriteB request, the “unsolicited” inputs are stored in this buffer. These bytes will be included in the response to the next ReadB request.

This way, the “bytes receiver” task may stay “busy” (without effectively reading bytes) several milliseconds, before risking to lose inputs.

This makes possible safe serial communications at baud rates up to 115.2K even under heavy data processing, with the condition that “data communication” tasks will keep “I/O-less” computing sequences shorter than 20 milliseconds.

The CMD console handler system task provides a command (HEX) able to read a .HEX file, and optionally execute-it. Large (~16 K bytes) .HEX files may be loaded, even while the system is busy executing other user application tasks.

Of course, a too heavy use of some “critical” RTM/Z80 functions (e.g. RunTask, StopTask, IncTaskStack, Balloc, Bdealloc) while reading bytes on the serial interface will raise the likelihood of I/O errors (reported by SIO), therefore it is recommended to keep such function’s calls at a low frequency.

Also, because the system checks frequently for incoming unsolicited inputs, to store them in the “ring” buffer, the overall performance of the system is affected. If your application does not need to support I/O communications protocols, it’s best to deselect the IO_COMM configuration option, enhancing the system performance (see Chapter Configuring RTM/Z80).

X-MODEM protocol support functions

XMODEM is a simple file transfer protocol developed by Ward Christensen for use in his 1977 MODEM.ASM terminal program.

It allowed users to transmit files between their computers when both sides used MODEM.

Keith Petersen made a minor update to always turn on "quiet mode", and called the result XMODEM.

XMODEM, like most file transfer protocols, breaks up the original data into a series of "128 byte packets" that are sent to the receiver, along with additional information allowing the receiver to determine whether that packet was correctly received.

If an error is detected, the receiver requests that the packet be re-sent.

Using its asynchronous serial communications platform, RTM/Z80 provides the following functions to implement the XMODEM protocol:

#include <xmodem.h>

short XmSend(struct MailBox* MB_Data);
HL=pointer to user data mailbox; returns 1=OK, -1=Cancelled, -2=Communications failure

Sends the data stored into the mailbox via the XMODEM protocol to a receiver.

The data mailbox must be created using the call MB_Data = MakeMB(129).

The data to be sent via XMODEM must be stored in the mailbox, using SendMail(MB_Data, user_data) calls; the user_data will contain 128 data bytes plus a byte with the value different from EOT=CTRL D (4).

A last mail (end-of-data) must contain in its last byte (byte nr. 129) the value EOT=CTRL D (4); its first 128 bytes are ignored.

As indicated in the original XMODEM protocol, the last 128-byte packet sent must be padded with special SUB bytes (value=1AH), up to the limit of 128 bytes.

short XmRecv(struct MailBox* MB_Data);
HL=pointer to user data mailbox; returns 1=OK, -1=Cancelled, -2=Communications failure

Receives data via the XMODEM protocol from a sender and stores-it to the user mailbox.

The data mailbox must be created using the call MB_Data = MakeMB(129).

The data must be retrieved using GetMail(MB_Data, user_data) calls; the user_data will contain 128 data bytes plus a byte serving as “end” marker (if this byte is equal to EOT=CTRL D (4), it marks the end of the data received – the first 128 bytes from this mail must be ignored).

How to use XmSend / XmRecv

We must create and run a separate task to handle XmRecv, and use GetMail to obtain the data, in another task.

An example follows:

```
#include <dlist.h>
#include <balloc.h>
#include <rtsys.h>
#include <mailbox.h>
#include <io.h>
#include <xmodem.h>

#define EOT 'D'- 0x40

short Xsts;
struct Semaphore* S;
struct MailBox* MB;
char buf[129];

void Xtask(void)
{
  Xsts = XmRecv(MB);
  StopTask(GetCrtTask());
}

void myTask(void) /* low priority */
{
  S = MakeSem();
  MB = MakeMB(129);
  fp = Xtask;
  RunTask(0xE0, fp, 100); /* priority must be higher */
  do
  {
    GetMail(MB,buf);
    if (buf[128] == EOT)
      break;
    /* process the data… */
    Wait(S);
  }
  while(1==1);
  /* then look at the Xsts, just in case… */
  StopTask(GetCrtTask());
}
```

The /* process the data… */ fragment must be substituted with code to handle the data just received; the data received via GetMail may be stored to the upper bank of 64KB RAM (if available), using LowToUp100H call, or written to a CP/M file, using the _bdos call (see next chapter).

If we want to handle XmSend the same way, we must use SendMail to “feed” the data to be sent via the XMODEM protocol; in this case, after sending the last packet of data, we will “mark” buf[128]=EOT before the SendMail to notify XmSend to stop sending data.

See the DEMO chapter for an example (getxfile & putxfile).

Parallel printer I/O support

#include <io.h>

The parallel printer shall be connected to the PIO board (e.g. SC103), using the following connections (via Dupont wires):

PIO A pins(0-7)	PIO B pins(0-7)	Male cable connector (1-25) pins
0		11
1		10
2		1
	0	2
	1	3
	2	4
	3	5
	4	6
	5	7
	6	8
	7	9
GND		GND (e.g.25)

Also, because the parallel printer driver must handle interrupts (the ACK signal from the printer), the interrupt daisy chain on RC2014 (between SC110's SIO and SC103's PIO) must be created, using a Dupont wire.

short LPT_Print(char* buf, short len);
DE=buf, C=len
Prints len characters from the buffer buf to the line printer.
Returns HL=0 : success, 1 : printer off line, 2 : paper out, 3 : printer error

The access to the line printer is protected using a system semaphore; this way, only one task at a time will use the LPT_Print function.

After each LPT_Print call, the calling task will be put on wait for a number of tics (equal to 200 * len/350), in order to allow the printer to physically type the requested characters on paper.

The line printer driver was tested using an EPSON LX-350 parallel printer, with an average speed of 350 chars/sec.

The TEST folder contains a demo program (tprint.as); it prints a CP/M file.

CP/M disk file I/O support

If the host system is provided with file storage facilities (disk, CF card, …) , RTM/Z80 offers a function to access the CP/M files stored on these disks, using the same interface as the CP/M's BDOS.

__***bdos*** (only Z80 assembly language)

The parameters are identical with those used when calling BDOS from CP/M : register C=BDOS function, registers DE = FCB address; register A contains the code returned as in the case of the BDOS call.

Of course, only BDOS functions related to file I/O can be used.

As the CP/M BDOS was not designed for multitasking use, we must "protect" our __**bdos** calls; this must be done by using a system semaphore, named **BDOS_Sem**.

Practically, when accessing disk files, we will use the following code structure:

```
ld    hl,BDOS_Sem
call  __Wait       ;obtain exclusive access to BDOS

;code fragment using _bdos calls , e.g.

ld    de,fcb
ld    c,open
call  __bdos

ld    c,write
call  __bdos

ld    s,close
call  __bdos

ld    hl,BDOS_Sem
call  __Signal     ;release exclusive access to BDOS
```

This mechanism will guarantee that only one task has access at BDOS, at a given time.

See the **tbdos.as** demo program in the folder TESTS (it executes a file copy operation).

When building the system, in the **config.mac** file, the **BDOS** configuration setting must be selected, in order to include the _**bdos** function. The **bdos.obj** and **bios.obj** files must be added to the link command files list.

VGA screen I/O support (for Z80ALL)

The following functions offer an interface to the Z80ALL's VGA screen (48 rows x 64 columns), for the RTM/Z80 system being executed on the Z80ALL standalone Z80 computer:

#include <vga.h>

void CrtClear(void);
Clears the screen (fills the screen with blanks).

void CrtLocate(int column, int row);
B=column, C=row
Positions the screen cursor at (column, row) position; column=0…63, row=0…47

void OutStringVGA(char* msg);
HL=pointer of zero terminated msg
Displays the zero-terminated message at the current cursor position on the screen; TAB, CR, LF, BS can be used as special characters. This function accepts the following VT52 ESC sequences:

```
CURSOR HOME             ESC H (48H)
CURSOR DOWN             ESC B (42H)
CURSOR UP               ESC A (41H)
CURSOR RIGHT            ESC C (43H)
CURSOR LEFT             ESC D (44H)
ERASE TO END_OF_SCREEN  ESC J (4AH)
ERASE TO END-OF_LINE    ESC K (4BH)
SET CURSOR POSITION     ESC Y (59H) Row Column
REVERSE VIDEO           ESC p (70H)
NORMAL VIDEO            ESC q (71H)
CURSOR OFF              ESC f (66H)
CURSOR ON               ESC e (65H)
```

void OutCharVGA(int column, int row, char* ch);
B=column, C=row, E=ch
Positions the screen cursor at (column, row) position (column=0…63, row=0…47) and displays the character ch.

char InCharVGA(int column, int row);
B=column, C=row, returns A=ch
Returns the character being currently displayed at (column, row) position (column=0…63, row=0…47)

PS/2 keyboard I/O support (for Z80ALL)

The following function offers an interface to the PS/2 keyboard, for the RTM/Z80 system being executed on the Z80ALL standalone Z80 computer:

char PS2_Status(void);
Returns L=A = key hit, else 0FFH if no key was hit. A 256-byte round robin buffer is used to accumulate the keys being hit, and PS2_Status reads from this buffer, using a ‘first in, first-out’ logic.

Note

The Z80ALL PS/2 keyboard, when a key is pressed longer than normal, will send the following series of characters (auto repeat protocol):

0, key, 0, key …0, 4.

Real time clock DS1302 support (for Z80ALL)

The following functions offer an interface to the real time clock DS1302, for the RTM/Z80 system being executed on the Z80ALL standalone Z80 computer:

#include <ds1302.h>

void InitRTC(void);
Initializes the real time clock to 00:00:00 and starts it.

long GetTime(void);
returns E = seconds, D = minutes, L = hours, H = 0
Returns the current time, measured since the last InitRTC call.

long DeltaTime(long start, long stop);
returns E = seconds, D = minutes, L = hours, H = 0
Returns the (stop – start) time.

char* TimeToStr(long time)
E = seconds, D = minutes, L = hours, H = 0

Returns the pointer to a string formatted like: “HH:MM:SS” , converting the ‘time’ to string.

This string is however stored to a “static” buffer (it is not allocated in the dynamic memory), therefore it is advisable to be “used” (printed, copied, …) as soon as possible, otherwise the buffer will be overwritten at the next TimeToString call !

The “system status display” (for Z80ALL)

This system component must be selected using the configuration item SYSSTS = 1.

It shows, on the Z80ALL’s VGA screen, the dynamics of an RTM/Z80 application, in real time.

Two main topics are shown on the VGA screen: the dynamic memory configuration (alloc/dealloc blocks), and the tasks status (active, waiting).

At each alloc/dealloc operation, a visual segment representing the actual position and size of the buffer will be shown (or erased).

The tasks are listed with their assigned priorities; when a task is given access to CPU, or when a task will migrate into the “waiting for semaphore” status, the system functions (RunTask, Wait, …) send appropriate outputs to the VGA and small visual signs are shown (or erased).

All this information is shown in real-time. Also, at each real time clock interrupt (2.5 ms), the "ticks counter" of the current active task is incremented.

The StartSampling and StopSampling functions may be called to “define” the “sampling” interval.

void StartSampling(void);

void StopSampling(void);

If these functions are not used, the “sampling” starts at StartUp and finishes at ShutDown.

At RTM/Z80 shutdown, for each task, a "CPU use" percentage is computed and displayed on the VGA screen. Also, the system displays how many bytes remained available in the task’s stack (a very low count should be considered as an "alarm", indicating that the task size should be increased at RunTask)

Thanks to the Z80ALL's 25MHz Z80 speed, this sampling procedure does not affect the overall system performance.

As a conclusion, for Z80ALL, the "RTM/Z80 system status", displayed in real time during the execution of a multitasking application, is very useful for:

- assessing the dynamic memory load (is it dangerously close to the maximum capacity?)
- viewing the task execution dynamics (the tasks active <---> waiting switching)
- learning about the system load (how long, in %, stays the system idle - is the % dangerously low?)

Therefore, a multitasking application can be fine-tuned, in an efficient way, just by watching the “system activities display”, on the VGA screen, during the application’s execution.

Of course, the application tasks must not use the VGA ouput functions provided by RTM/Z80, else the two types of ouput (app & system activities display) will be “mixed” on the VGA screen.

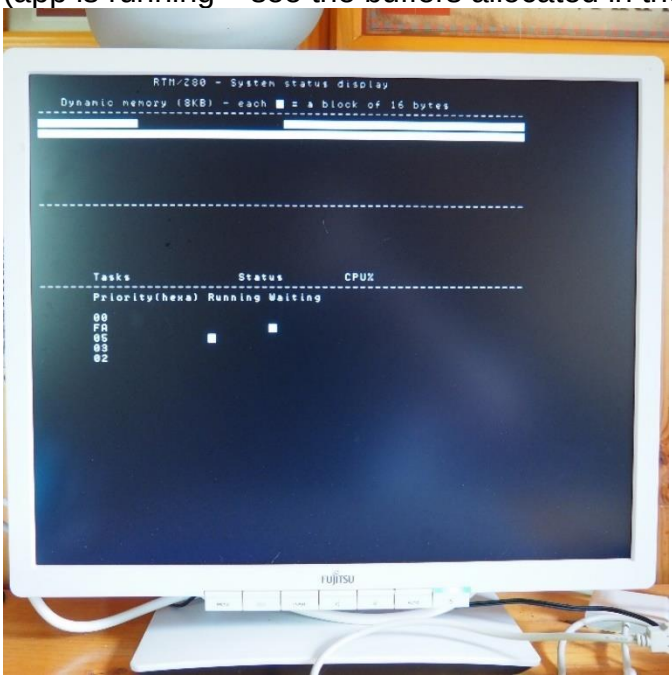
Example: let's use the trrb.c application from the GitHub TEST folder.

```
C>trrb
RTM/Z80 2.6
 T5 counted 50000
 T3 counted 30211
 T2 counted 20084
```

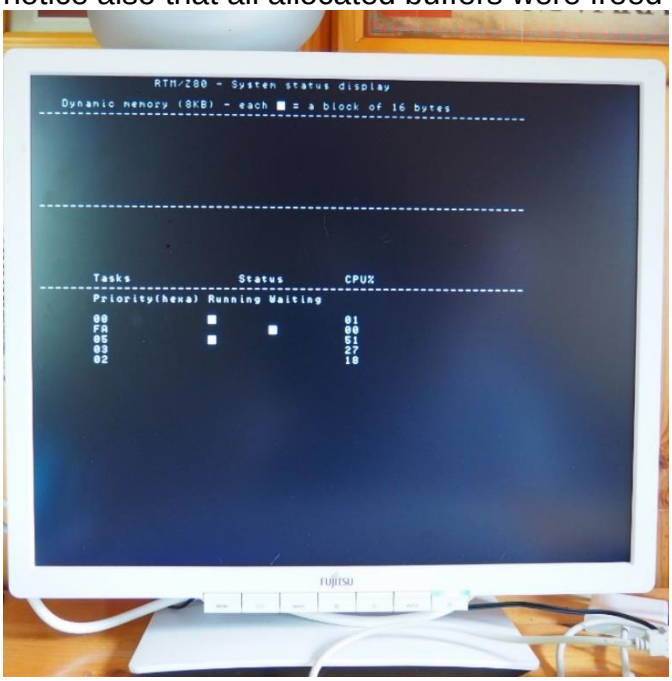
(task T5 priority = 5, task T3 priority = 3, task T2 priority = 2)

Here are two "system activities display" VGA screenshots:

(app is running – see the buffers allocated in the dynamic memory and the task status markings)



(atfter shutdown – see the CPU % allocated to each task; FA = CON driver task, 00 = default task; notice also that all allocated buffers were freed at shutdown – see also next chapter)



The “memory leaks” issue

Sometimes, the tasks allocating blocks of dynamic memory “forget” to deallocate them, before exiting.

These “forgotten” memory blocks are named “memory leaks”; if these accumulate, they rise the fragmentation degree of the dynamic memory (which brings down the performance of the allocation/deallocation mechanisms) and may even lead to “full” dynamic memory.

When memory leaks occur frequently, the applications seem to slow down or even halt.

RTM/Z80 tries to solve this issue using a dedicated “garbage cleaner” task.

A **memory garbage cleaner** (a zero-priority internal task) is provided, automatically deallocating “memory leaks” (memory blocks owned by stopped tasks, but not deallocated by the owner task before stopping).

This special task is started at system start-up and is the last task to exit before system shutdown.

It uses the “ID stamp” contained in each allocated memory block to identify the blocks to be deallocated.

As a task is terminated (using StopTask), the system “stores” the task ID in a queue reserved for the private use of the garbage cleaner, to notify him about a new “job” to clean the possible memory leaks.

The memory garbage cleaner will run only when there are no other active tasks to run (e.g. all other active tasks are terminated or suspended or waiting for semaphores, timers or for the completion of I/O operations).

The Round Robin scheduling

RTM/Z80 is provided with an optional **round robin scheduling** mechanism, enabling a fair sharing of execution time among active tasks.

For example, when the 3 top-priority active tasks with priorities equal to 10, 5, 2 are running continuous sequences of instructions, without executing signal/wait functions, the time slices given to these tasks will be equal to 10*5=50 ms, 5*5=25 ms, 2*5=10 ms, in a round robin sequence.

The round-robin scheduling can be enabled/disabled using dedicated functions (see 2.7 Timers).

The algorithm used to implement this round robin scheduling is simple, not very accurate, but it works without consuming large extra computing power. It is implemented inside the clock interrupt routine.

The steps of the algorithm are:

- When an active task obtains access to the CPU, its priority is loaded into a special counter
- Each real-time clock interrupt (at 5 ms) the special counter is decremented
- When the counter reaches zero, the list of active tasks is “rotated” (first becomes last) and the control is passed to the first active task from the list

Practically, each task is given a “continuous” run time interval equal to its priority multiplied with 5 ms.

Of course, this algorithm interferes with the usual rule of “highest priority task gains the CPU access” and because of this, round robin scheduling must be used with care.

After round-robin scheduling is enabled, the basic rule of “highest priority gains the CPU access” is not anymore valid, therefore this facility must be used only when it is absolutely vital to “share” the CPU time between several tasks executing long sequences of “computation-like” code.

As a basic rule, it is a bad practice to enable round-robin scheduling when the current active tasks will execute frequently Wait, Signal, WriteQ, ReadQ, SendMail, GetMail or I/O operations, because all these API calls imply switching the CPU-control from one task to the other, without the need to apply a “round-robin” switching logic.

On the other hand, it is also a bad practice to disable round-robin scheduling when we have several active tasks busy with lengthy computations or data handling sequences, when a “fair” sharing of the CPU-time is critical.

As a final remark, the RoundRobinON and RoundRobinOFF calls should be used wisely, according to the context and also knowing the fact that the “time-slices” provided are proportional to the task’s priorities.

The “priority inversion” issue

Every multitasking scheduler tries to ensure that of those tasks that are ready to run, the one with the highest priority is always the task that is actually running.

Because tasks share resources, events outside the scheduler's control can prevent the highest priority ready task from running when it should. **Priority inversion** is the term for a scenario in which the highest-priority ready task fails to run when it should.

The real trouble arises at run-time, when a medium-priority task pre-empts a lower-priority task using a shared resource on which the higher-priority task is pending. If the higher-priority task is otherwise ready to run, but a medium-priority task is currently running instead, a priority inversion is said to occur.

Consider the following theoretical case: we have a low-priority task L, a medium-priority task M and high-priority task H. H and L share a resource. Shortly after Task L takes the resource, Task H becomes ready to run. However, Task H must wait for Task L to finish with the resource, so it pends. Before Task L finishes with the resource, Task M becomes ready to run, pre-empting Task L. While Task M (and perhaps additional intermediate-priority tasks) runs, Task H, the highest-priority task in the system, remains in a pending state.

How to deal with the “priority-inversion” in RTM/Z80?

In an application written using RTM/Z80, suppose T has the greatest priority among all tasks.

T may gain access to CPU when:

- RunTask(T) was executed by another task
- StopTask was executed by the current running task
- Suspend was executed by the current running task
- Resume(T) was executed by another task (T executed prior to this a Suspend)
- Another task executes a Signal for a semaphore that contains T in its queue of waiting tasks and T has the greatest priority among all tasks from the queue (T executed prior to this a Wait for that semaphore)
- SetTaskPrio(T) was executed, raising T’s priority above all other tasks’ priorities
- Round-robin mechanism is enabled and T is given access to the CPU (for a number of clock ticks equal to its priority)

Notice that the only case when T obtains the CPU time and (possibly) has not the highest priority among all active tasks is the last, round-robin case.

Let’s consider now the practical case of 3 RTM/Z80 tasks running in the scenario of round-robin being enabled: L with priority 2, M with priority 5 and H with priority 10. Suppose L and H use a semaphore S to obtain access to a shared buffer of data.

H just executed a Wait(S) to gain access to the buffer of data. The scheduler will insert task H into the semaphores’ queue, then will pass the control to the highest priority active task available.

Both M and L are active and available, competing for the use of the CPU time.

But they are made form a quite different stuff:

- L is finishing some short (15 milliseconds) calculus using the data and then will call a Signal(S) to indicate he does not need any more exclusive access to the data
- M has some lengthy calculus to do (during 15 seconds), before stopping.

```
Data:  …
TaskH:
       …
       Wait(S); ← - - - we are here
       …
TaskM:
       …
       Long_computation(); /* takes 15 seconds */
       StopTask(GetCrtTask());
TaskL:
       …
       Quick_calc();/* takes 15 milliseconds */
       Signal(S);
       …
```

After H executes Wait(S), the scheduler gives the control to M, but because the round-robin algorithm, after 5*5=25 milliseconds the control is passed to L, for 2*5=10 milliseconds, then back to M, and so forth.

Now, L is about to finish his part of calculus, but his 10 milliseconds first “time-slice” terminates before finishing its 15 milliseconds calculus to be able to issue the Signal(S). M is given the control for another 25 milliseconds slice, and only then L regains control, spends another 5 milliseconds, finished his calculus and issues the Signal(S). Only 25+10+25+5=65 milliseconds were waisted, from the H’s perspective. But it could be better: the last M's 25 milliseconds were waisted also from the L’s perspective.

We can see that running of M has delayed the running of both L and H. Precisely speaking, H is of higher priority and doesn’t share the semaphore S with M; but H had to wait for M. This is where Priority based scheduling didn’t work as expected because priorities of M and H got inverted. That’s Priority Inversion in action.

In this case, if those 25 milliseconds waisted are important, the solution is to use the SetTaskPrio function. Task H will call SetTaskPrio(TCB_M, 1) before entering Wait(S), to make sure that L will have, for a short period of time, higher priority than M. Then, when resuming from Wait(S), task H will restore the old M priority : SetTaskPrio(TCB_M, 5).

```
TaskH:
…
SetTaskPrio(TCB_M,1);
Wait(S);
SetTaskPrio(TCB_M,5);
…
```

This way, only 10+25+5=40 milliseconds will take from task H Wait to the task L Signal.

Of course, the best solution will be to use also the RoundRobinOFF and RoundRobinON functions:

```
TaskH:
…
RoundRobinOFF();
SetTaskPrio(TCB_M,1);
Wait(S);
SetTaskPrio(TCB_M,5);
RounRobinON();
…
```

As a result, in this later case task H will have to wait only 15 milliseconds.

The “priority-inversion” issue is always difficult to handle.

In the theory of multitasking systems, when using binary semaphores or mutexes, priority inversion can be dealt with some complicated algorithms: priority ceiling, priority inheritance, random boosting, etc. This is mainly because of the binary semaphores and mutexes implementation, prone to (unwanted) automatic task pre-emption.

RTM/Z80 uses counting semaphores, and this avoids a lot of the “priority-inversion” risks, because automatic task pre-emption is not possible when using Wait/Signal for counting semaphores, it can be done only by the round-robin mechanism. And, as shown before, some RTM/Z80 functions (SetTaskPrio, RoundRobinON/OFF), can be used to mitigate this risk.

But, in any particular case, the application context must be carefully studied in order to choose the best solution, which in many cases will be to make a compromise between response times and a fair use of CPU for all tasks.

Queues vs. Mailboxes

The queue mechanism is faster, compared with mailboxes, because no allocation / deallocation of memory blocks is performed during the queue “write” or “read” operations (a single allocation is performed when creating the queue).

However, special care is needed to correctly calibrate the size of the queue buffer (a too low *batch_count* will “block” too frequently in wait the data producer, and a too high *batch_count* will result in an unwisely consume of dynamic memory by using a huge buffer).

Mailboxes assure a safe method of communication between producers and consumers of data, with the disadvantage of being 10% to 20% slower compared with the queues mentioned earlier (the mailbox mechanism of sending/receiving data does allocate/deallocate buffers for each exchanged message).

When the number of messages to be sent is large, and the receiver cannot “read” enough quickly the received messages, the mailbox offers however an advantage, compared to the queues , because the quantity of data sent via a mailbox is limited only by the size of available dynamic memory, while in the case of a queue, the total number of messages that can be “stored” in the queue is limited by the “batch_count” parameter of the queue.

For example, suppose a queue is created using the call MakeQ(5, 7). This will create a queue able to send/receive messages of 10 bytes(=5*2), but with the capability to “store” only 7 of such messages without putting the sender in wait.

If the receiver is too slow to “read” the messages, the sender will be “blocked” in the WriteQ by “Wait for available space in the queue” when the queue will contain 7 messages not yet read by the receiver, being “unblocked” only when the receiver will “read” a message using ReadQ.

The solution will be to use a mailbox (using MakeMB(10)). Now, the only constraint will be the amount of free dynamic memory available to allocate space for the messages being sent, not the number of the messages.

Of course, after some hundreds of messages sent but “non-yet-consumed” by the receiver, the dynamic memory will be entirely “spent”, but this choice (queue speed of handling messages vs. mailbox large number of messages not-yet-processed) is enough flexible to solve most common producer-consumer practical contexts.

Configuring RTM/Z80

RTM/Z80 was written in Z80 assembler, using first the ZAS HiTech's assembler (and later the Z80AS assembler - see https://github.com/Laci1953/Z80AS , compatible with the ZAS assembler) and LINK linker, first with the Udo Munk's Z80SIM and then directly on RC2014's CP/M as a development platform (see next chapter for details).

It was tested, until now, on the following configurations:

- CP/M running under Z80SIM Z80 simulator
- RC2014, with or without CP/M, using the following configurations:
  - SC112 + SC108(Z80 + 32KB SCM EPROM + 2x64KB RAM) + SC110(CTC, SIO) + Digital I/O module
  - SC112 + SC108(Z80 + 32KB RTM/Z80 EPROM + 2x64KB RAM) + SC110(CTC, SIO) + Digital I/O module
  - SC112 + Karl Brokstad's Z80 22c module + Memory Module(32KB RTM/Z80 EPROM + 2x64KB RAM) + SC110(CTC, SIO) + Digital I/O module
  - SC112 + Karl Brokstad's Z80 22c module + 512KB RAM/ROM Memory Module + SC110(CTC, SIO) + Digital I/O module
- Z80ALL (25MHz Z80, 4x32KB RAM, KIO, PS/2 keyboard, VGA screen)

On RC2014 or Z80ALL, RTM/Z80 and its applications can be executed in the following scenarios:

1. RTM/Z80 application loaded from SCM (only for RC2014): Compile the application and link-it to the RTM/Z80 library, then, from the SCM monitor, load the RTM/Z80 and the application in RAM as a single HEX file, and execute-it in RAM
2. RTM/Z80 application run as a .COM file under CP/M: Compile the RTM/Z80 application and link-it to the RTM/Z80 library as a .COM file, and execute the application as a .COM file, starting-it from CP/M
3. RTM/Z80 stored on EPROM (only on RC2014): Compile the application and link-it to the RTM/Z80 library, then, boot RTM/Z80 from EPROM (it will move itself to RAM), then it will load the application as a HEX file and will execute-it in RAM (this is possible when the EPROM from the SC108 or 32KB ROM + 128KB RAM Memory Module or 512KB RAM/ROM Module is changed with an EPROM containing a boot-able image of RTM/Z80 – in fact, it contains several RTM/Z80 versions + Watson + CP/M booter)
4. In the previous situations, after the application terminates, Watson can be loaded to inspect the aftermath of the application execution

The choice of these hardware configurations was imposed because of the need to use Z80 Interrupt Mode 2 (IM2), in order to have separate interrupt servicing routines for the CTC real-time clock interrupts and the read/write SIO serial interface interrupts.

RTM/Z80 can be ported to any other Z80 based computer that enables the use of IM2; if 128 KB of RAM are available, also the Watson utility program may be ported.

A number of configuration options are available (in **config.mac**), including:

- **SIM** – if selected, the RTM/Z80 application will be built to run as a CP/M .COM under Z80SIM;

- **SIM** - if deselected, the system will be built for RC2014
- **RAM128K** - if selected, it means that 2 banks of 64KB RAM are available
- **SC108** - system runs using the Steve Cousins SC108 board
- **Z80ALL** – system runs on Bill Shen's Z80ALL standalone Z80 computer
- **SYSSTS** – (only for Z80ALL) system status will be displayed in real-time on the VGA screen
- **KIO** – the hardware uses a Z80 KIO
- **M512 -** system runs using the 512KB RAM + 512KB ROM memory module
- **EXTM512** – if selected, extended dynamic memory support is provided, enabling 512KB code + allocation of up to 448KB of memory (only if M512 is selected)
- **ROM** – if not selected, RTM/Z80 will be built to be loaded in the RAM at 0000H
- **ROM** – if selected, RTM/Z80 will be built to be stored on EPROM, with thee following parameters as options:

    - **MM** - the RTM/Z80 will be built to be stored in the Memory Module's 32KB EPROM
    - **SC108** - the RTM/Z80 will be built to be stored in the SC108's 32KB EPROM
    - **M512** the RTM/Z80 will be built to be stored in the 512 KB EPROM of the 512 KB ROM/RAM board
    - **BOOT_CODE** – if selected, the RTM/Z80 will be stored at offset 0 in the EPROM, with a supplementary bootstrap code
    - **BOOT_CODE** - if deselected, the RTM/Z80 will be stored at a not null offset in the EPROM, without the bootstrap code

    When using the 32KB EPROM, there is sufficient space in the EPROM's 32KB to store 4 different RTM/Z80 versions, plus the Watson utility program, and the CP/M booter.

    In the case of the 512KB EPROM, 26 different RTM/Z80 versions can be store, plus the Watson utility program, and the CP/M booter.

    These RTM/Z80 versions include a palette of configurations, starting from a "full" version and ending with an "only assembly API" version.

    The following versions of RTM/Z80 are stored in the 32 KB EPROM:

        - Version 1 (FULL: debug, C&assembly API, CMD, async comm I/O) : 10KB

```
DEBUG       equ 1  ;1=debug mode ON: verify task SP, task TCB, dealloc, lists, etc.
SIM         equ 0  ;1=Runs under Z80SIM, 0=Runs on RC2014(SC108+SC110)
DIG_IO      equ 1  ;1=RC2014 Digital I/O module is used
CMD         equ 1  ;1=CON CMD task is included
RSTS        equ 1  ;1=use RST for list routines (not for CP/M)
WATSON      equ 1  ;1=Watson is used
C_LANG      equ 1  ;1=Support for C language API
IO_COMM     equ 1  ;1=Support for async communications I/O
MM          equ 1  ;1=Memory Module is used
RAM128K     equ 1  ;0=only 64K RAM, 1= 2 x 64K RAM available
ROM         equ 1  ;1=sys code on ROM, 0=ROM not used
BOOT_CODE   equ 1  ;1=bootstrap code included in code, 0=no bootstrap code
```

        - Version 2 (no debug, C&assembly API, no CMD, async comm I/O) : 5KB

```
DEBUG       equ 0  ;1=debug mode ON: verify task SP, task TCB, dealloc, lists, etc.
SIM         equ 0  ;1=Runs under Z80SIM, 0=Runs on RC2014(SC108+SC110)
DIG_IO      equ 1  ;1=RC2014 Digital I/O module is used
CMD         equ 0  ;1=CON CMD task is included
RSTS        equ 1  ;1=use RST for list routines (not for CP/M)
WATSON      equ 1  ;1=Watson is used
C_LANG      equ 1  ;1=Support for C language API
IO_COMM     equ 1  ;1=Support for async communications I/O
MM          equ 1  ;1=Memory Module is used
RAM128K     equ 1  ;0=only 64K RAM, 1= 2 x 64K RAM available
ROM         equ 1  ;1=sys code on ROM, 0=ROM not used
BOOT_CODE   equ 0  ;1=bootstrap code included in code, 0=no bootstrap code
```

- Version 3 (no debug, C&assembly API, CMD, no async comm I/O) : < 7KB

```
DEBUG       equ 0  ;1=debug mode ON: verify task SP, task TCB, dealloc, lists, etc.
SIM         equ 0  ;1=Runs under Z80SIM, 0=Runs on RC2014(SC108+SC110)
DIG_IO      equ 1  ;1=RC2014 Digital I/O module is used
CMD         equ 1  ;1=CON CMD task is included
RSTS        equ 1  ;1=use RST for list routines (not for CP/M)
WATSON      equ 1  ;1=Watson is used
C_LANG      equ 1  ;1=Support for C language API
IO_COMM     equ 0  ;1=Support for async communications I/O
MM          equ 1  ;1=Memory Module is used
RAM128K     equ 1  ;0=only 64K RAM, 1= 2 x 64K RAM available
ROM         equ 1  ;1=sys code on ROM, 0=ROM not used
BOOT_CODE   equ 0  ;1=bootstrap code included in code, 0=no bootstrap code
```

- Version 4 (no debug, only assembly API, no CMD, no async comm I/O) : <4KB

```
DEBUG       equ 0  ;1=debug mode ON: verify task SP, task TCB, dealloc, lists, etc.
SIM         equ 0  ;1=Runs under Z80SIM, 0=Runs on RC2014(SC108+SC110)
DIG_IO      equ 0  ;1=RC2014 Digital I/O module is used
CMD         equ 0  ;1=CON CMD task is included
RSTS        equ 1  ;1=use RST for list routines (not for CP/M)
WATSON      equ 1  ;1=Watson is used
C_LANG      equ 0  ;1=Support for C language API
IO_COMM     equ 0  ;1=Support for async communications I/O
MM          equ 1  ;1=Memory Module is used
RAM128K     equ 1  ;0=only 64K RAM, 1= 2 x 64K RAM available
ROM         equ 1  ;1=sys code on ROM, 0=ROM not used
BOOT_CODE   equ 0  ;1=bootstrap code included in code, 0=no bootstrap code
```

- **DEBUG** – if selected, all possible “defensive” checks will be performed for all the system functions parameters. TCB, queue, mailbox, timer, semaphore addresses will be checked for validity, Suspend/Resume functions will be checked for non allowed situations (for example, an active task cannot be “resumed”, and a task waiting for a semaphore cannot be “resumed”), available stack space will be checked (with warnings issued when stack space drops below 20H). In the case when RTM/Z80 is compiled in DEBUG mode, many of the system functions will return error codes if the parameters are incorrect or the context does not allow the function to be executed. These checks will be skipped completely if RTM/Z80 is compiled without the DEBUG setting. Therefore, the main benefit of executing RTM/Z80 code in DEBUG mode is the “safe” environment provided, enabling the identification of incorrect use of the system

facilities, with the downside of being slower compared with executing the same code without DEBUG mode. It is advisable to start testing an RTM/Z80 application always in DEBUG mode, to catch all the possible programming errors, and only then to switch to the faster non-DEBUG mode.

- **C_LANG** – if selected, all RTM/Z80 functions will be available from programs written in the C language. If deselected, only Z80 assembler programs may use the RTM/Z80 functions.

- **IO_COMM** – if selected, support for the communications I/O is provided. If deselected, no such support will be provided.

- **WATSON** – if selected, support for breakpoints execution and code for saving the "snapshot" image on the second bank of 64 KB RAM is provided, making possible to launch the WATSON utility after reaching a breakpoint or after the system shutdown. This option should be used only with RC2014 hardware configurations that provide 128 KB RAM (2 x 64 KB), for example with the SC108 , the 32 KB ROM + 128KB RAM Memory Module or the 512 KB ROM + 512 KB RAM module..

- **BDOS** – if selected, support for CP/M disk I/O is added

- **LPT** – if selected, support for Parallel Line Printer I/O is added

- **DS1302** – if selected, support for the DS1302 real time clock is added (only for Z80ALL)

- **PS2** - if selected, support for the PS/2 keyboard is added (only for Z80ALL)

- **CMD** – if selected, a console utility task is provided, offering the following commands:

    - ACT – list of active tasks

```
>act
Active tasks:
TCB: 4200H Priority: 240 Free stack:1A7H
TCB: 4400H Priority: 10 Free stack:1E3H
TCB: 7F16H Priority: 0 Free stack:4AH
>
```

    - TAS – list of all tasks

```
>tas
TCB: 7F16H Priority: 0 Free stack:4AH, running
TCB: 4000H Priority: 250 Free stack:C1H, waiting for semaphore: 7FF4H
TCB: 4200H Priority: 240 Free stack:16FH, running
TCB: 4400H Priority: 10 Free stack:1E3H, running
>
```

    - MEM – status of the dynamic memory

```
>mem
Block of size 80H at address 4000H owned by task with TCB 7F16H
Block of size 40H at address 4080H owned by task with TCB 4000H
Block of size 10H at address 40C0H owned by task with TCB 4200H
Block of size 200H at address 4200H owned by task with TCB 7F16H
Block of size 200H at address 4400H owned by task with TCB 7F16H
Available blocks of size 10H : 40D0H
```

```
Available blocks of size 20H : 40E0H
Available blocks of size 100H : 4100H
Available blocks of size 200H : 4600H
Available blocks of size 800H : 4800H
Available blocks of size 1000H : 5000H
Total free dynamic memory : 1B30H
>
```

- MAP – map of the dynamic memory

```
>map
Dynamic memory map (* = allocated 10H block)
     _____________+100_____________+200_____________+300_____________
4000|*************                  *********************************|
4400|********************************                                 |
4800|                                                                 |
4C00|                                                                 |
5000|                                                                 |
5400|                                                                 |
5800|                                                                 |
5C00|                                                                 |
     _____________+100_____________+200_____________+300_____________
>
```

- HEX – load (and optionally execute) a .HEX file.

```
>hex
```

The system issues the message:

```
Ready to read HEX file (timeout=10 sec)
```

, and the user must 'feed' on the screen the .HEX file (by example copying the file and pasting it to the screen). The following outcomes are possible:

  - The file is correctly loaded, and the user is asked if he wants to execute-it
  - The file is corrupted (no end-of-file record)
  - No dynamic memory is available for the read buffer
  - 10 seconds passed and no file was provided (time-out)
  - The checksum is incorrect

- EXI – terminates CMD execution

- STP addr – stops the task with TCB=addr

- RES addr – resumes the task with TCB=addr

- PRI addr,pr – sets pr as new priority for the task with TCB=addr

- RRB on (or off) – sets the round robin mode

- SHD – shuts down RTM/Z80

NOTE: (pay attention at the **bss** segment length – your application read-write RAM bss segment, added to the RTM/Z80's bss, must fit into the range of addresses 0D000H – 0DF00H)

The development of RTM/Z80

I started developing RTM/Z80 using Z80SIM, on Windows, under CygWin.

First, I used HiTech's ZAS as assembler, but I was quickly confronted with its limitations regarding the size of the source files being assembled.

The biggest ZAS problem is related to its size (38KB), and because of this ZAS is unable to assemble large source files (too small free space remains for the symbols).

Therefore, I took Hector Peraza's ZSM4 and modified-it to output code object files compatible with HiTech's OBJ format; the result was Z80AS (see https://github.com/Laci1953/Z80AS) , an assembler compatible with ZAS.

Compared to HiTech's ZAS assembler, Z80AS has some advantages:

- can compile larger source files
- supports the undocumented Z80 instructions
- has more pseude operators (coonditionals, listing control)
- better MACRO facilities (REPT, IRP, IRPC, LOCAL)
- better support for expression evaluation, including extensive use of parantheses and well-defined operator precedence

The RTM/Z80 sources must be compiled only with Z80AS (HiTech's ZAS fails because of the large source files size).

Later, migrating on RC2014, using CP/M as a development platform, I was confronted with another set of problems.

First, I needed a decent text editor, so I adapted Miguel Garcia's TE, allowing-it to use more than 64KB RAM (128KB or 512KB), then adding TAB and arrow keys to move the cursor. The result is an improved text editor (see https://github.com/Laci1953/RC2014-CPM/tree/main/te) , able to edit large files in the RAM memory.

Then, when trying to build RTM/Z80 test applications written in the C language, I become aware of the HiTech's C compiler's limitations (it failed to compile larger C source files).

Therefore, I was constraint to modify the HiTech's C compiler, allowing it to use more than 64KB RAM (see https://github.com/Laci1953/HiTech-C-compiler-enhanced).

The result was an enhanced C compiler, running on Z80 computers provided with 128KB/512KB RAM, accepting larger C source files to be compiled.

Currently, I made most of my RTM/Z80 maintenance work on Bill Shen's Z80ALL standalone home-brew computer (25MHz Z80, 128KB RAM, VGA screen with 48 x 64 chars, PS/2 keyboard).

Building and running an RTM/Z80 application on Z80SIM

RTM/Z80 applications may be built and executed on Z80 simulated environments (Z80 emulators).

I installed Z80SIM on Cygwin (a Linux-like) running under Windows 10, and stored all the necessary software tools and RTM/Z80 source files on one of its 4MB disk.

Z80SIM CP/M implementation provides a 10 milliseconds real-time clock, on interrupts, making it a best fit for building and executing a multitasking system on a Z80 simulator.

About running RTM/Z80 applications on Z80SIM, some important issues must be mentioned.

First of all, be aware that running RTM/Z80 applications on Z80SIM comes with some important limitations, mainly related to the fact that no interrupt-based serial I/O is possible.
In fact, in the Z80SIM version of RTM/Z80 the “SIO input/output” interrupts are simulated!

More exactly:

- after a CON_Read call, the next Wait call is used to “loop” until all characters are entered
- the CON_Write call “loops” until all characters are written on the terminal

It is true that these “loops” are calling the “real” interrupt routines, but it is only a compromise!

Also, all CON_Read and CON_Write calls must use Semaphores (and be coupled with Wait calls for the specified Semaphore).

The asynchronous serial I/O communications calls are also affected; simulated calls of SIO input/output interrupts must be used in order to solve the issue. This does not mean, however, that these functions cannot be used at all (e.g. loading .HEX files - using CMD HEX command - can be done also on the CP/M implementation).

Another issue is related to the fact that on Z80SIM, the real-time clock interrupts come at each 10 ms, not at 5 ms as in the case of RC2014, and because this the RTM/Z80 “timer-related” functions are affected.

The use of **GetHost** function may solve this issue, but some code must be used to handle the difference between the 5 ms and 10 ms tics while computing “number of tics per second”.

That being said, the implementation of RTM/Z80 on CP/M still ensures the proper functioning of all system facilities.

Note: at RTM/Z80 StartUp, a small check verifies if CP/M runs under Z80SIM; in case Z80SIM is not detected, a message error is issued and CP/M is re-booted.

An example of how to build and run an RTM/Z80 application on CP/M is presented.

Let’s take for example **birds.c** (see the source code in ***RTM/Z80 demo applications***)

First, the file **config.mac** must be edited to set the desired system configuration; here is the result:

```
J>type config.mac
DEBUG           equ 0   ;1=verify task SP, task TCB, dealloc, lists
SIM             equ 1   ;1=Runs under Z80SIM, 0=Runs on RC2014(SC108+SC110)
DIG_IO          equ 0   ;1=RC2014 Digital I/O module
CMD             equ 0   ;1=CON CMD task
RSTS            equ 0   ;1=use RST for list routines (not for CP/M)
WATSON          equ 0   ;1=Watson will be used
C_LANG          equ 1   ;1=Functions called from C language
IO_COMM         equ 0   ;1=Support for async communications I/O
…
```

Then, we build the RTM/Z80 components and store them into an object code library:

```
J>submit make
J>submit makelib
```

Next, we compile **birds.c**, link-it to the RTM/Z80 library and build **birds.com** as result:

```
J>c -o birds.c rand.as rt.lib
```

Now, **birds.com** is ready to run, on Z80SIM's CP/M.

When building RTM/Z80 applications intended to be run on Z80SIM's CP/M, special care must be given to the RTM/Z80 CP/M version's memory map:

```
;Memory map for CP/M version
;
;      0100H - 7B00H       sys code & data, apps code & data
;      7B00H - 7D00H       HEX loader buffers
;      7D00H - 7E00H       CleanReqB
;      7E00H - 7F00H       SIO receive buffer
;      7F00H - 8000H       Tasks vectors, sys data
;      8000H - 9FFFH       Dynamic Memory
;      A000H - DC00H       reserved for ZSID.COM
;      DC00H - FFFFH       reserved for CP/M
```

The code and data space for an RTM/Z80 application in CP/M is limited to 100H – 7B00H.
Of course, the application can allocate extra space for its data using the 8KB dynamic memory.

Building and running an RTM/Z80 application on Z80 computers

An example of how to build and run an RTM/Z80 application on RC2014 or Z80ALL is presented.

We have two options:

- **CP/M cannot be used on RC2014** - we will build the application on Z80SIM and load it on RC2014 as a .HEX file using the SCM monitor
- **we have CP/M installed on RC2014 or Z80ALL** – we will build the application on the local CP/M and run-it as a .COM file

First scenario: CP/M cannot be used on RC2014

We must build the RTM/Z80 application on Z80SIM.

Let's take for example **tbdos.as** (TESTS folder).

The file **config.mac** must be edited to set the desired system configuration; here is the result:

```
J>type config.mac
DEBUG        equ 0  ;verify task SP, task TCB, dealloc, lists
SIM          equ 0  ;1=Runs under Z80SIM, 0=Runs on RC2014(SC108+SC110)
DIG_IO       equ 1  ;RC2014 Digital I/O module
CMD          equ 0  ;CON CMD task
RSTS         equ 1  ;use RST for list routines (not for CP/M)
WATSON       equ 1  ;Watson will be used (only for CP/M)
C_LANG       equ 0  ;Functions called from C language
IO_COMM      equ 0  ;Support for async communications I/O
BDOS         equ 1  ;1=BDOS disk file support
RAM128K      equ 1  ;0=only 64K RAM, 1= 2 x 64K RAM available
```

Then we build the system files:

```
J>submit make
…
```

We build the system library:

```
J>submit makelib
…
```

We assemble the application:

```
j>z80as tbdos.as
```

Then we link the application to the system files:

```
J>link
Link> -x -z -Pzero=0,text,ram=0D000H,bss -os.obj tbdos.obj rt.lib
```

We build the .HEX file:

```
j>objtohex s.obj tbdos.hex
```

To run the **tbdos** application, we need first to load the code into SC2014.

First, let's start RC2014; the SCM prompt will be displayed:

```
Small Computer Monitor - S3
*
```

We will then copy_paste the RTM/Z80 bootstrapper **hexboot.hex** to the terminal app , let's say, TeraTerm, used to connect with SC2014:

```
J>type hexboot.hex
:20E00000F33166E22141E211E3DF010500EDB021ABE1114DDF019600EDB0DD21ABE1114DD4

...
(copy hex file on clipboard)
```

NOTE: If your RC2014 is not provided with 128KB RAM, you must modify, in the hexboot.as and cpmboot.as, the value of the constant RAM128K to 0, then assemble both files, and build the hexboot.hex.

Now, on TeraTerm screen, we paste the clipboard contents:

```
Small Computer Monitor - S3
*
(paste the clipboard contents…)
*Ready
```

We start the bootstrapper:

```
*gE000
```

The bootstrapper now waits to load the RTM/Z80 application HEX file:

```
Ready to read RTM/Z80 HEX file:
```

Next, we need to copy-paste **tbdos.hex** to the terminal app used to connect with SC2014 and then boot RTM/Z80:

```
J>type tbdos.hex
:20000000C33C1D0000000000C3461D0000000000C36D1D0000000000C30E1D000000000063

...
(copy hex file on clipboard)
```

Now, on TeraTerm screen, we paste the clipboard contents:

```
Ready to read RTM/Z80 HEX file:
(paste the clipboard contents…)
```

The bootstrapper asks for optional breakpoints to be set (when reaching a breakpoint, a full 64KB copy (snapshot) of the code and data is saved, and you may use the Watson utility to investigate the aftermath of the application run):

```
Breakpoint (4 hex digits, .=no more breakpoints to set): .
```

…then asks if we want to boot RTM/Z80:

```
Boot? (Y/y=yes) : Y
Booting RTM/Z80...
(tbdos is running)
```

At RTM/Z80 shutdown, control is passed to the SCM monitor.

Second scenario: CP/M can be used on RC2014 or Z80ALL

We will use CP/M to build the RTM/Z80 system and its applications.

Let's suppose we have all our RTM/Z80 related files on drive D.

The file **cpmboot.obj** must be present. If not already present, we will assemble **cpmboot.as** (RESOURCES folder):

```
D>z80as cpmboot.as
```

The file **config.mac** must be edited to set the desired system configuration; here is the result:

```
D>type config.mac
DEBUG        equ 0  ;verify task SP, task TCB, dealloc, lists
SIM          equ 0  ;1=Runs under Z80SIM, 0=Runs on RC2014(SC108+SC110)
DIG_IO       equ 1  ;RC2014 Digital I/O module
CMD          equ 0  ;CON CMD task
RSTS         equ 1  ;use RST for list routines (not for CP/M)
WATSON       equ 1  ;Watson will be used (only for CP/M)
C_LANG       equ 0  ;Functions called from C language
IO_COMM      equ 0  ;Support for async communications I/O
BDOS         equ 1  ;1=BDOS disk file support
RAM128K      equ 1  ;0=only 64K RAM, 1= 2 x 64K RAM available
```

We then build the RTM/Z80 components:

```
D>submit make
…
```

We build the system library:

```
J>submit makelib
…
```

We will assemble **tbdos.as** and link-it to RTM/Z80 system object codes to obtain **tbdos.com**:

```
D>z80as tbdos.as
D>link
-x -z -C100h -Pboot=0E300H/100H,zero=0/,text/,ram=0D000H/,bss/ -otbdos.com cpmboot.obj tbdos.obj
rt.lib
```

NOTE: if EXTM512=1, then the "ram" psect must be set to 8000H (…,ram=8000H/…)

Then, **tbdos** can be executed as a .COM file:

```
D>tbdos
Booting RTM/Z80...
(tbdos is running)
```

At RTM/Z80 shutdown, control is passed to the SCM monitor.

For programs written in the C language, the LINK command line must contain also **libc.lib**.

For example, to build **birds.com**:

```
D>c -v -o birds.c
D>link
  -x -z -C100H -Pboot=0E300H/100H,zero=0/,text/,data/,ram=0D000H/,bss/ -Obirds.com \
  cpmboot.obj birds.obj rand.obj rt.lib libc.lib
```

Setting breakpoints

When executing an RTM/Z80 application under CP/M, it is possible, as an option, to set a breakpoint.

For this, in the command line, the name of the executable must be followed by a valid 4-digits hexadecimal number.

For example, suppose we have an application named TEST.
The command:

```
D>test 1d4f
```

…will set a breakpoint at the address 1D4FH before starting TEST.COM

When the execution will reach the address 1D4FH, the RTM/Z80 will be shut down, signalling that a breakpoint was reached.

Then, we may use the Watson tool to investigate (see Ch. Watson):

```
D>watson
RTM/Z80 probably reached a breakpoint…
Watson at your service!
:r PC=1D4F …
```

NOTE: in both scenarios, it is important to use the appropriate LINK command parameters !

Important memory constraints

The RTM/Z80 system, application code (text segment) and read-only data (data segment) must fit into the range of addresses 0000H to 0D000H (52 K bytes).

The size of RTM/Z80 system code alone varies between 4K and 8K bytes, according to the selected configuration; this means the largest application code size is between 44 to 48 K bytes.

The application read-write data (bss segment) must be smaller than 3 K bytes (the exact limit is 0A00H, however, the application may access also the 8 K bytes dynamic memory area, to get access to more space for its read-write buffers).

Using RTM/Z80 in a ROM/RAM configuration

The RTM/Z80 system can be configured to run on RC2014 with system code stored on (E)EPROM. There are several hardware configurations which could be used.

First of all, a Z80 CPU must be present (e.g. a simple, plain Z80 board, as the Karl Brokstad's Z80 modules) .

Then, a serial communication board must be present, provided with Z80 SIO and CTC clock.

And, of course, a board containing (E)EPROM and RAM must be added.

Possible options:

- Steve Cousins's SC108 (Z80, 32KB EPROM, 128KB RAM) + SC110 (SIO+CTC)
- Any Z80 CPU board + Phillip Stevens's Memory Module (32KB EPROM, 128KB RAM) + SC110 (SIO+CTC)
- Any Z80 CPU board + a 512KB EPROM 512KB RAM module + SC110 (SIO+CTC)

The RTM/Z80 system will be copied on RAM at boot-time (if stored on 32KB EPROM) or executed directly on ROM (if stored on 512KB EPROM), then the user will be asked to load his application.

As an option, if the hardware contains also a CF board, on the EPROM may be stored also a CP/M booter.

Here, two options are possible:

1. With 64MB CF cards, an enhanced CP/M may be used as an option (it will load at 0DA00H), offering an extra 2 and a half KB of TPA area (very important if you need to assemble or compile large files!)
2. With 128MB CF cards, the classic CP/M (loaded at 0D000H) will be used

This way, the user will have a complete development & execution media.

Instead of booting Steve Cousins SCM, the user will be presented with the following options:

```
Press 0 to boot CP/M,
or 1,2,3,4 to boot an RTM/Z80 version,
or 5 to start Watson :
```

Thus, the user has the possibility to boot CP/M, develop RTM/Z80 applications, then boot RTM/Z80 and execute them, using then the Watson tool to inspect the aftermath of an application run.

An example of how to build and run an RTM/Z80 application for the 32KB ROM of SC108 board is presented. To build-it, the CP/M will be booted from the EPROM.

To build an RTM/Z80 application, the **.sym** files produced at RTM/Z80 build time must be used.

For example, when building RTM/Z80 version #1, the **sc108r1.sym** file is created by the linker.

Using the utility **symtoas.com,** we build the source file **sc108r1.as** , containing the RTM/Z80 API definitions:

```
D>symtoas sc108r1.as sc108r1.sym
```

To build applications, the **sc108r1.obj** file is needed, it can be obtained by assembling the source file:

```
D>z80as sc108r1.as
```

We will compile **rtmdemo.c**, link-it to the RTM/Z80 API, and use the **objtohex.com** command to build **rtmdemo.hex** (the app code must be built to be loaded and executed at 2800H):

```
D>c  -c -o rtmdemo.c
D>z80as rand.as
D>link
link> -x -z -Ptext=2800H,data,bss -os.obj jp.obj rtmdemo.obj rand.obj sc108r1.obj\
link> csv.obj libc.lib
D>objtohex s.obj rtmdemo.hex
```

The **jp.obj** must be placed first in the list of object files; it contains only a “jp _main” (see below)

```
JP.AS:
psect text
global _main
jp _main
```

Next, we boot RC2014; on the TeraTerm screen we will have the following message:

```
Press 0 to boot CP/M,
or 1,2,3,4 to boot an RTM/Z80 version,
or 5 to start Watson :
```

(see Ch. Configuring RTM/Z80)

If we choose to boot an RTM/Z80 version, the following message will be displayed:

```
Ready to read HEX file:
```

…and we will then paste the **rtmdemo.hex** file contents:

```
(paste the clipboard contents…)
```

The bootstrapper asks for optional breakpoints to be set:

```
Breakpoint (4 hex digits, .=no more breakpoints to set): .
```

…then asks if we want to boot RTM/Z80:

```
Boot? (Y/y=yes) : Y
Booting...
RTM/Z80 Demo program
...
```

NOTE : if RTM/Z80 was configured with the CMD task, you need first to exit from CMD, using:

```
>EXI<CR>
```

It is possible to set some breakpoints in the application; when reaching a breakpoint (or at shutdown), the system will store the current full 64KB image of RAM into the second 64 KB RAM bank, then reboot.

At this stage, the user may opt to load the Watson tool, in order to inspect the application’s data & code, frozen at the breakpoint execution (see Ch. Watson).

Debugging an RTM/Z80 application

As mentioned before, it is important to run first our RTM/Z80 application using an RTM/Z80 built with the DEBUG option ON.

This way, all the API call parameter values will be verified and, if incorrect, the call will not be executed, eliminating the risk to crash the RTM/Z80 system because of a wrong parameter value.

In case of executing the application on RC2014, if the DIG_IO option was selected and if the Digital I/O module is present, the following events will be shown as a led turned ON:

LED nr.0: ON=RTM/Z80 is running
LED nr.1: ON/OFF every one second
LED nr.2: ON=(if DEBUG option ON) stack warning (a Task stack space is below 20H)
LED nr.3: ON=(if DEBUG option ON) wrong TCB address used in API parameter value
LED nr.4: ON=(if DEBUG option ON) wrong Semaphore address used in API parameter value
LED nr.5: ON=(if DEBUG option ON) no more free memory (dynamic memory is full)
LED nr.6: ON=SIO A External Status Change
LED nr.7: ON=SIO A Special Receive Condition

If the DEBUG option is turned OFF, no such checks will be performed, leading to potential issues related to RTM/Z80 API called with wrong parameter values (e.g. calling Wait with a wrong Semaphore address will store some data at wrong RAM memory addresses, possibly leading to application and/or system crash).

Also, after executing an RTM/Z80 application, using the WATSON utility is recommended to inspect the aftermath of the application run, or to view the status of the system/application after a breakpoint was reached (see chapters WATSON & Running applications on RC2014).

WATSON - Investigating the results of an RTM/Z80 application run

WATSON is a tool designed to investigate the results of executing an RTM/Z80 application on the RC2014 homebrew Z80 computer.

The RC2014 must be provided with 128KB of RAM, on the SC108 board (32 KB ROM and 2 x 64KB RAM) or the Phillip Stevens ‘s Memory Module (32KB ROM and 2 x 64KB RAM) or the 512KB RAM/ROM module.

The application written to be run under RTM/Z80 is loaded and executed on the first bank of 64KB RAM. When the application executes the Shutdown function (or when a breakpoint is reached), a full 64KB copy (snapshot) of the code and data is moved to the second bank of 64KB RAM and the RC2014 is re-booted.

Then, WATSON can be loaded and executed, to investigate the results of the application run.

The functions provided by WATSON will made possible to examine the user data, code, registers, RTM/Z80 system data.

The addresses used as parameters for these functions will be the “real” addresses of data or code from the RTM/Z80 application that has been run.

WATSON will “load” the data/code from the snapshot and will display it exactly as it was after the RTM/Z80 shutdown or breakpoint.

The following functions are provided:

- Memory
- List
- Semaphore
- TCB
- Active task list
- All task list
- Current active task
- Dynamic memory status
- Queue
- Mailbox
- Timer list
- Timer
- Disassemble
- Registers
- Exit

The functions syntax is simple: after WATSON displays its prompt (a colon :) each function is called using a single letter (possibly followed by a hexadecimal address) followed by <CR>. The command line can be “edited” using the BACKSPACE key.

A full list of the functions will be printed using as input a question mark :?<CR>

An example of using WATSON is presented below. Let's consider the following simple application:

```
                psect   text
        _main:
                ld      bc,1E0H
                ld      hl,_Task
                ld      e,10H
                call    __StartUp
                ret
        _Task:
                ld      bc,4
                call    __Balloc
                ld      bc,5
                call    __Balloc
                ld      bc,6
                call    __Balloc
                ld      bc,7
                call    __Balloc
                ld      bc,8
                call    __Balloc
                call    __MakeSem
                ld      (sem),hl
                call    __MakeTimer
                ld      (timer),hl
                ld      de,(sem)
                ld      bc,10
                xor     a
                call    __StartTimer
                ld      bc,1007H
                call    __MakeQ
                ld      (queue),hl
                ld      bc,50H
                call    __MakeMB
    ld      (mailbox),hl
                 call   __Shutdown
```

After running the application, let's start WATSON.

Here, we have the following options:

- In case the application was executed under CP/M on RC2014, after the application shutdown we will simply run WATSON.COM from CP/M
- In case the application was executed using HEXBOOT, we will load WATSON.HEX using the same HEXBOOT facility
- In case the RTM/Z80 system was booted from EPROM (and the application was executed using HEXBOOT), we will boot next Watson from the EPROM

```
Watson at your service!
:?
M<addr><CR>    Memory display (*)
L<addr><CR>    Double linked list display
S<addr><CR>    Semaphore display
T<addr><CR>    TCB display
A<CR>          Active tasks list display
O<CR>          All tasks list display
C<CR>          Current active task display
D<CR>          Dynamic memory display
Q<addr><CR>    Queue display
B<addr><CR>    Mailbox display
K<CR>          Display list of Timer Control Blocks
J<addr><CR>    Display Timer Control Block
E<addr><CR>    Disassemble (*)
R<CR>          Registers
X<CR>          Exit
where <addr> = 4 hexa digits
(*) autorepeat at <CR>
```

```
:m016d
016D 46 41 D6 40 66 41 06 4C 21 00 7F E5 21 3A 0C CD FA.@fA.L!...!:..
017D C9 0F E1 11 04 00 01 02 00 E5 21 42 02 CD C9 0F ..........!B....
018D E1 CD B2 10 E5 21 3A 0C CD C9 0F E1 19 E5 21 59 .....!:.......!Y
019D 02 CD C9 0F E1 CD B2 10 E5 21 3A 0C CD C9 0F E1 .........!:.....
:k 4146
:q4166 Write pointer=4506 Read pointer=4506 Buffer=4506 Size=00E0 Batch size=20
:s40d6 TCB waiting: Counter:0000
:l7f04->4200->7F16
:t4200 Size=0200 Priority=10 SP=43F2 Status=Active
:a 4200 7F16
:o 7F16 4000 4200
:c 4200
:d
Available blocks of size 0010: 4150
Available blocks of size 0020: 4C20
Available blocks of size 0040: 4C40
Available blocks of size 0080: 4C80
Available blocks of size 0100: 4D00
Available blocks of size 0200: 4E00
Available blocks of size 0800: 5800
Total free dynamic memory : 0BF0
Block of size=0080 at address=4000 owned by TCB=7F16
Block of size=0040 at address=4080 owned by TCB=4000
Block of size=0010 at address=40C0 owned by TCB=4200
Block of size=0010 at address=40D0 owned by TCB=4200
Block of size=0020 at address=40E0 owned by TCB=4200
Block of size=0040 at address=4100 owned by TCB=4200
Block of size=0010 at address=4140 owned by TCB=4200
Block of size=0020 at address=4160 owned by TCB=4200
Block of size=0080 at address=4180 owned by TCB=4200
Block of size=0200 at address=4200 owned by TCB=7F16
Block of size=0100 at address=4400 owned by TCB=4200
Block of size=0100 at address=4500 owned by TCB=4200
Block of size=0200 at address=4600 owned by TCB=4200
Block of size=0400 at address=4800 owned by TCB=4200
Block of size=0020 at address=4C00 owned by TCB=4200
Block of size=0800 at address=5000 owned by TCB=4200
:b4c06 TCB waiting: Mails list: Message size=50
:j4146 Tics:000A Sem:40D6
:e0100
0100: 01 00 02     ...   LD BC,0200
0103: 21 0B 01     !..   LD HL,010B
0106: 1E 10        ..    LD E,10
0108: CD 39 1D     .9.   CALL 1D39
010B: 01 00 00     ...   LD BC,0000
010E: CD 78 29     .x)   CALL 2978
0111: 01 01 00     ...   LD BC,0001
0114: CD 78 29     .x)   CALL 2978
0117: 01 02 00     ...   LD BC,0002
011A: CD 78 29     .x)   CALL 2978
011D: 01 03 00     ...   LD BC,0003
:r AF= F3E1 BC= 3031 DE= CD0F HL= 08B2 AF'=0F28 BC'=4C21
DE'=CD0D HL'=0FC9 IX =000E IY =05C3 SP =AF00 PC =000E
:x
```

RTM/Z80 Demo applications

See the source files:

https://github.com/Laci1953/RTM-Z80/blob/main/DEMO/msort.c

https://github.com/Laci1953/RTM-Z80/blob/main/DEMO/rtmdemo.c

https://github.com/Laci1953/RTM-Z80/blob/main/DEMO/getxfile.c

https://github.com/Laci1953/RTM-Z80/blob/main/DEMO/putxfile.c

https://github.com/Laci1953/RTM-Z80/blob/main/DEMO/Birds.c

The executables are stored here:

https://github.com/Laci1953/RTM-Z80/tree/main/DEMO

Contents of the RTM/Z80 project on GitHub

Here is the description of the GitHub RTM/Z80 project (folders):

- SOURCES: Z80 assembly language source files of the RTM/Z80 system (to be assembled using Z80AS)
- WATSON: Z80 assembly language files, parts of the WATSON utility
- ROM:
    - SC108 folder: contains files used to build the RTM/Z80 ROM version for the SC108 board
    - MM folder: contains files used to build the RTM/Z80 ROM version for the 32KB ROM + 128KB RAM board
    - M512 folder: contains files used to build the RTM/Z80 ROM version for the 512KB ROM + 512KB RAM board
    - CPM folder: contains the CP/M booter to be stored in the ROM
- TESTS: various tests that I used to test RTM/Z80 plus the submit files used to build them
- RESOURCES: various sources for utility-type programs plus submit files used to build the demo applications, plus the HEXBOOT.HEX file that must be used to load an RTM/Z80 application on SC108
- DEMO: the demo applications, sources and executables

Porting RTM/Z80 to other hardware

RTM/Z80 can be easily modified to be run on another hardware.

However, there are some constraints to be considered:

- the processor must be compatible with Z80
- interrupt mode 2 must be used
- a real time clock, on interrupts
- a serial asynchronous device, on interrupts
- at least 4 KB EPROM space
- at least 8 KB RAM space

The list below contains all the places where hardware ports are used or hardware-dependent constants are configured:

- CONFIG.MAC
    - RAM128K set to 0 means only (up to) 64 KB RAM will be used; in this case, WATSON must be also set to 0 and all the macros related to selecting low/up banks of RAM will be ignored
    - Real time clock ports (CTC_1…CTC_3) must be modified
    - Serial asynchronous ports (SIO_*) must be modified
    - BMEM_BASE must be set to the RAM address of the 8K dynamic memory
    - _main must be set to the start addr of the app, in case of a RAM/ROM configuration
    - TICS_PER_SEC must be set to obtain 1 second (multiplied with the clock interrupt interval)
- BOOT.AS contains code used only in RAM/ROM configurations
- CPMDATA.AS is relevant only for Z80SIM
- IO.AS contains the serial asynchronous driver code; a ring-based buffer is used for serial communications related functions
- RTCLK.AS contains the real time clock driver
- RTSYS1A.AS
    - _InitInts contains the interrupts initialization code
    - StopHardware contains the code to stop using interrupts
    - SIO_*_TAB contains data used when initializing the serial device (SIO)
    - INTERRUPTS is the interrupts vector
    - SIO_buf is the ring buffer used for serial communications related functions
- SNAPSHOT.AS contains code used only in case of 128 KB RAM systems

Acknowledgements

I want here to thank all those who helped me, directly or without even knowing it, to finish this work:

- Stephen C. Cousins, for his essential contribution related to building my RC2014 computer; without his expertise, patience and wisdom, I surely would have failed to make-it run. He kindly allowed me to use part of his SCM code (the Z80 disassembler). He was the first who reviewed this manual and advised me for making it better
- Phillip Stevens, for his advices related to the use of his RC2014 Memory Module
- Karl Albert Brokstad, for helping me to setup his Compact Flash Storage Module for RC2014
- Bill Shen, for the kindness with which he made one of his Z80ALL computers available to me
- Donald E. Knuth, for his excellent book (The art of computer programming), who inspired me 40 years ago to start coding multitasking software systems
- Udo Munk, for his outstanding Z80SIM, used to develop this project
- The (unknown to me) programmers who contributed to build the HiTech Z80 C compiler, ZAS assembler, LINK linker and all the auxiliary tools

… and all others who offered me sources of inspiration in their articles about multitasking software.