SPIRE-IFS-NOT-001130

Note on SPIRE-DPU architecture

Prepared by: Riccardo Cerulli-Irelli

Distribution List :

K. King	SPIRE	
S.D. Sidher	SPIRE	
J.L. Auguères	SPIRE	
C. Cara	SPIRE	
D. Ferrand	SPIRE	
G. Olofsson	SPIRE	
L. Seno	Audionica	
M. Giordano	Audionica	
R. Orfei	IFSI	
A. Di Giorgio	IFSI	
S. Molinari	IFSI	
S. Pezzuto	IFSI	
S. Codella	IFSI	
S. J. Liu	IFSI	

Document Status Sheet:

Document Title:			
Issue	**Revision**	**Date**	**Reason for Change**
Draft 3		26/09/01	Added Tuning task. Added generalized table uploading protocol
Draft 3.1		7/10/01	VM code updated Added in Appendix an example on VM code
Draft 3.2			Chapter formatting Chapter 3.1 expanded Chapter 3.4 initiated Chapter 3.4 – introduced minimum time between SS commands Chapter 3.5 expanded Appendix – Reason for a Virtual Machine
Draft 3.3		21/01/02	Chapter 3.4 Introduction of a VM optimising compiler and simulator.
Draft 3.4		23/01/02	Paragraph 3.4.1 Corrected VM code Introduced paragraph 3.4.2 and3.4.3 for Compiler/Simulator

Reference documents

Document Reference	Title

Acronyms

CDMS	Central Data Management System
CNR	Consiglio Nazionale delle Ricerche
CPU	Control Processing Unit
DCU	Detector Control Unit
DPU	Digital Processing Unit
DRU	Detector Readout Unit
FIFO	First In First Out storage element
FIRST	Far InfraRed and Submillimeter Telescope
HK	HouseKeeping
HW	HardWare
I/F	Interface
IFSI	Istituto di Fisica dello Spazio Interplanetario
ISR	Interrupt Service Routine
LSB	Least Significant Bit(s)
MCU	Mechanism Control Unit
MSB	Most Significant Bit(s)
mutex	Mutual Exclusive flag
NA	Not Applicable
OBS	On-Board Software
OS	Operating System
PC	Program Counter
PDU	Power Distribution Unit
S/C	Spacecraft
SPIRE	Spectral and Photometric Imaging REceiver
SS	Subsystem
SW	SoftWare
TBC	To Be Confirmed
TBD	To Be Defined
TBW	To Be Written
TC	Telecommand
TM	Telemetry

VM	Virtual Machine

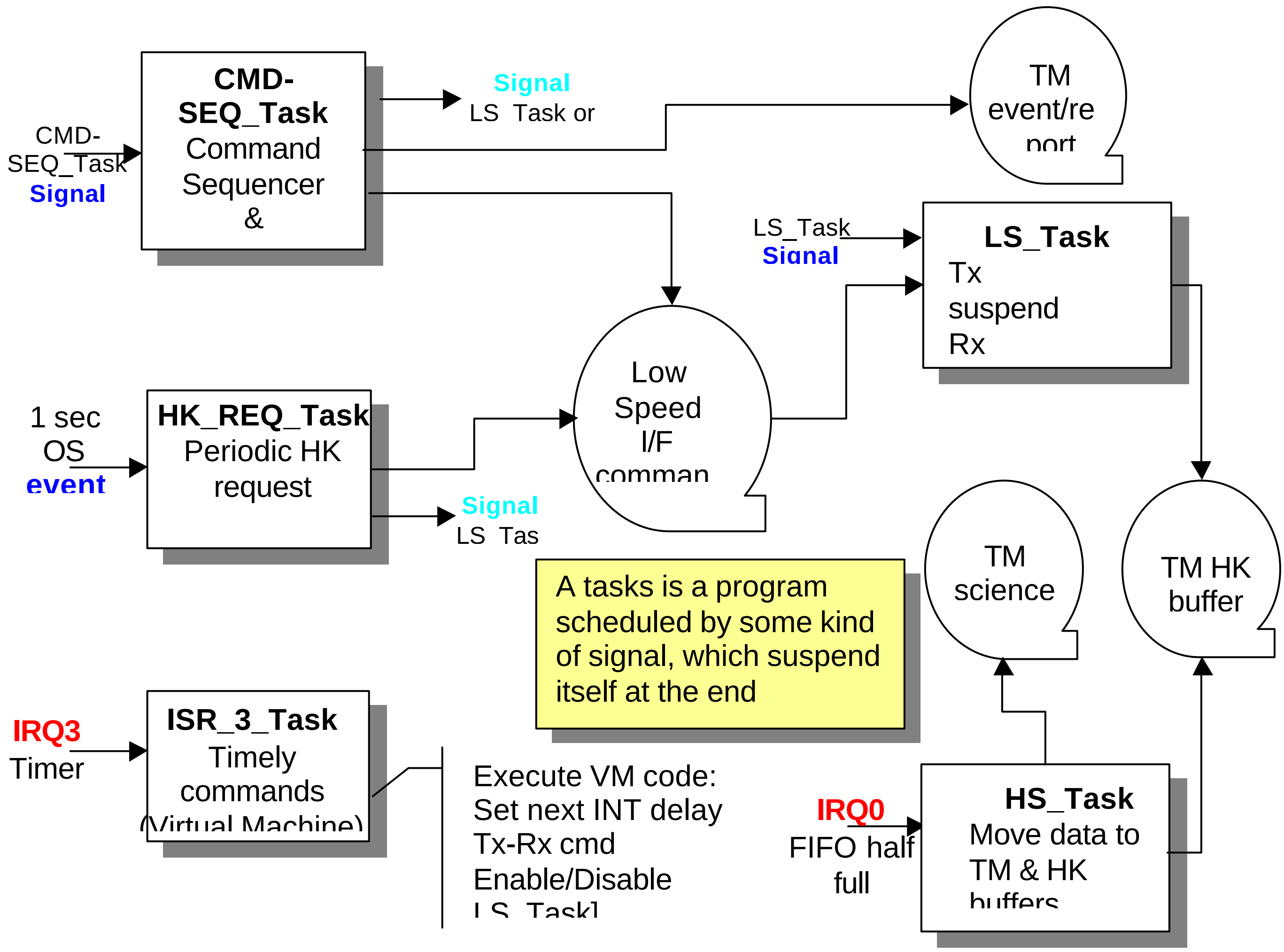
Table of contents

1 Introduction

This note tries to present in an informal format the status of SPIRE DPU OBS architecture.
The note will eventually be translate in the right format and included in the SSD as the "DPU Architecture Description".

2 Subsystem commanding description

With reference to the following figure, here is a description of the identified tasks.
As far as this note is concerned, a task is an infinite repeating program scheduled by some kind of signal, which suspends itself at the end. A new signal will re-schedule the task.



2.1 CMD_SEQ_Task

This task is scheduled in the run queue by a semaphore set by the TMTC_Task, signalling the presence of telecommands. When the task gets the CPU attention, the incoming telecommands are controlled (a TC verification report is generated), exploded in elementary commands and executed/stored. The task, after that, suspends itself until the next semaphore signal.
At this stage there are two are main cathegories of commands:

- CPU internal commands.
- S/S commands.

The DPU internal commands are immediately executed inside the task.

S/S commands are stored in the Low Speed I/F command buffer, a semaphore signal is generated in order to put in the run queue the LS_Task which is in charge of the actual transmission.

Optionally a second highest priority Low Speed I/F command buffer2 can be added in order to cope with immediate commands.

2.2 HK_REQ_Task

The HK_Req_Task handles the periodic HK request. The OS periodically schedules the task in the running queue. This task just reads a table of SS elementary commands, writes the commands on the Low Speed I/F command buffer and sends a signal (increments semaphore counts) to the LS_Task .
The scheduling period of the task is set, enabled/disabled by DPU internal commands.
HK received from the SS on the low speed I/F together with those received via the high speed I/F, are stored and packetized in a circular data buffer as shown for the HS_Task

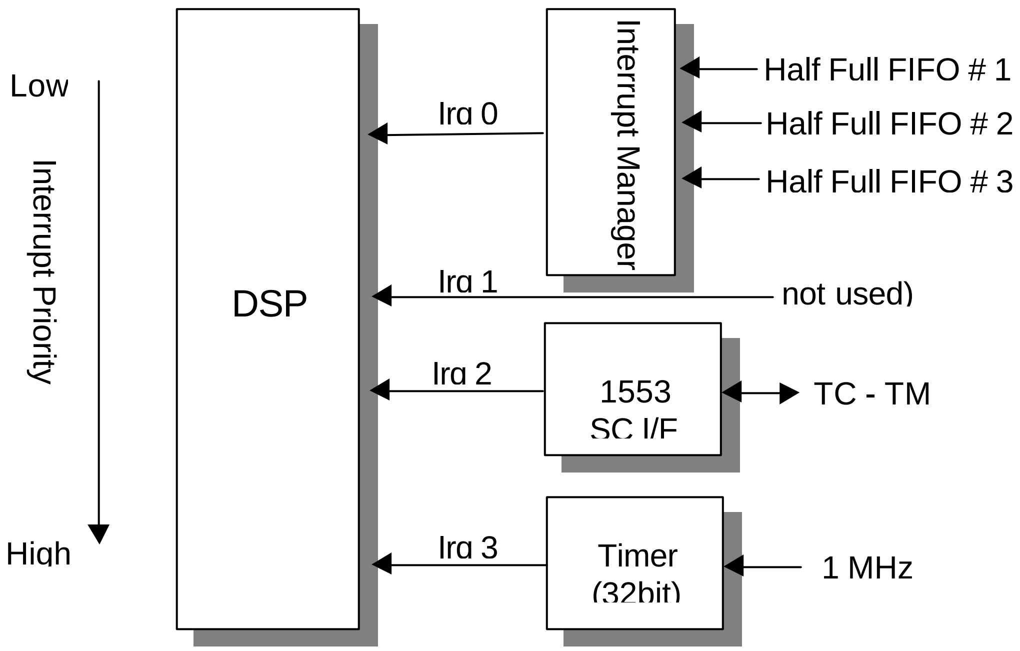
2.3 LS_Task

The LS_Task (together with the ISR_3_Task) is in charge of transmitting and possibly receiving commands/housekeeping to/from the subsystems. The actual timing of the

commands transmission with this task is not predicable due to the multitasking nature of the OS; a jitter of few milliseconds must be taken in account.
The task, scheduled by HK_REQ_Task and CMD_SEQ_Task, checks the command against a fixed table for special cases (flush FIFO/sync etc), checks for the availability (mutex lock) of the low speed I/F port (might be used by ISR_3_Task) and if not available suspends itself until the port is no more busy. The task then writes on the output port the SS command and suspend itself for 1 ms (HIFI 2 ms if it's an HK request) to allow the 100 us transmission time and possibly the HK response word with allowed time-out. If an HK is expected/received, the HK is stored in the TM HK buffer and formatted for TM transmission.

2.4 ISR_3_Task

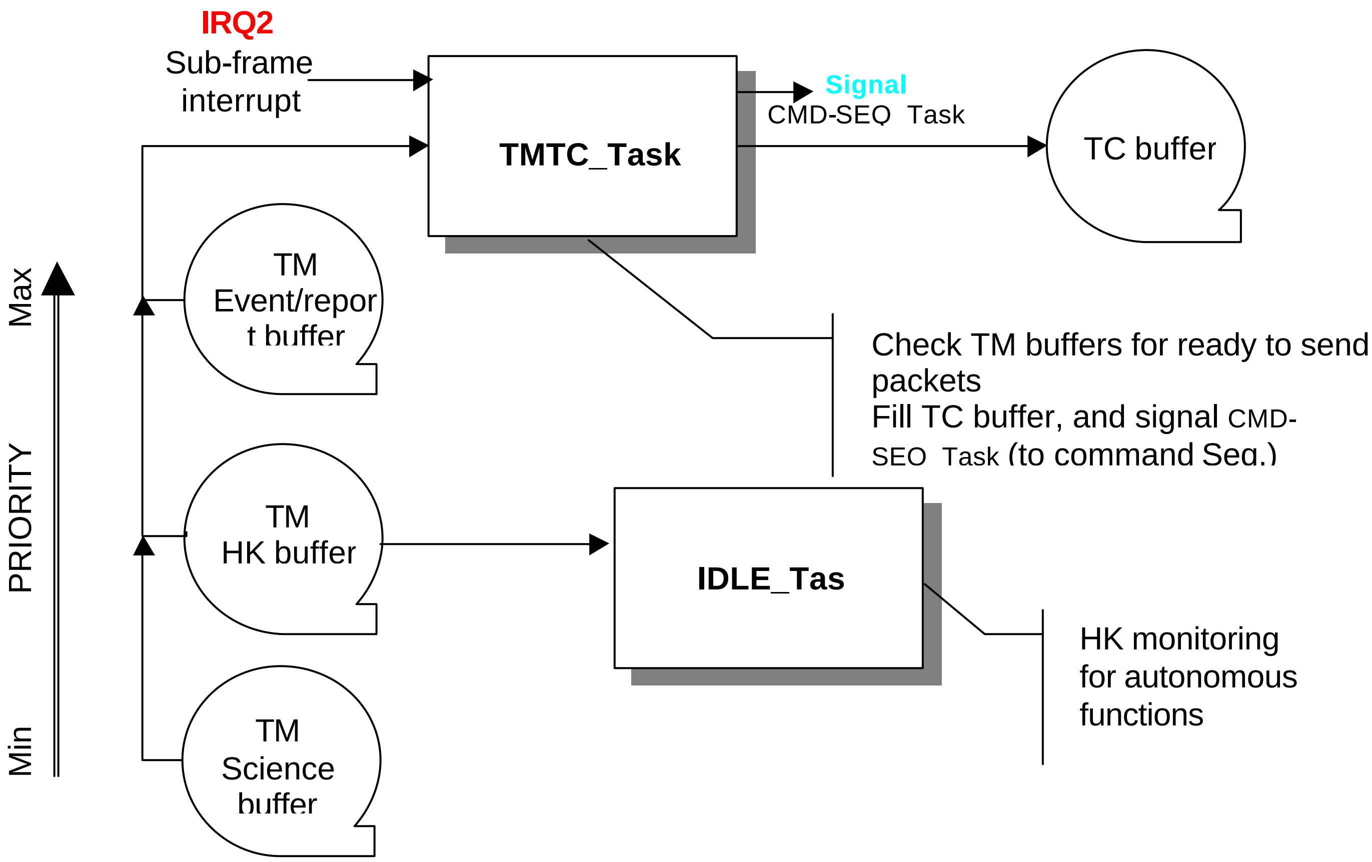
This task allows the transmission of commands to the SS at a fixed time with a maximum jitter of 10us (TBC). The task, interrupt driven, is started (and possibly terminated) by an DPU internal command which enable/disable the highest priority interrupt (Irq 3) driven by a 1 MHz clocked HW timer.



The tasks read from a preloaded table (exec_table) the time to the next command and the command to transmit, it then set the timer and transmits the command to the SS.
In order to avoid collision on the low speed I/F with the LS_Task, a special (internal) command is foreseen to lock/unlock (via a mutex) the low speed I/F. The locking command will precede the SS commands of at least 100 us in order to allow the possible contemporary (just started) transmission of a command via the LS_Task.

2.5 HS_Task

This task collects science and HK data on the high speed I/F, format the data in "packet transmission ready" and stores the data in the right buffer.
The data on the high speed I/F are temporary stored on four 8Kword deep FIFOs, the "half FIFO full" signal associated to each FIFO generates an interrupt (IRQ 0) which in turn schedules the HS_Task on the run queue.
The main point here is that, due to the asynchronous operation of the FIFOs, the actual timing of the incoming data is lost, in other words no cause/effect between commands (on low speed I/F) and received data (on high speed I/F) is possible, at least in a simple efficient and reliable way.



2.6 TMTC_Task

This task checks on the 3 record descriptor buffers if any Event/Report, HK or science TM packet (in this sequence) is ready for transmission, and transmit the packets at S/C generated (64 per seconds) interrupts. The same task signals to the CMD_SEQ_Task the possible presence of a TC.

2.7 IDLE_Task

This low priority task is executed when no other task is active. The time spent inside this task is an indication of the load charge of the DPU/SPU.

3 Detailed tasks description

3.1 CMD_SEQ_Task

This task begins the execution on arrival of a TC packet signalled by the TMTC_Task. The first operation is the TC verification (RD1 3.1) with the generation of the acceptance report TM packet stored in the highest priority (Event/Report) buffer.
The received TC packets can be subdivided in 2 categories:

- DPU commands to update on board tables
- DPU internal commands

3.1.1 On Board Tables

The SPIRE instrument will be completely controlled by a set of preloaded and TC modifiable 32 bit words tables. The tables fully control the SS configuration and observation modes, and are stored in data memory in contiguous locations in order to simplify a possible relocation.
A preliminary list of tables include:

- SS configuration tables
- VM programs

As a general rule, consecutive selected data (commands) from the configuration tables will be sent to the SS via the LS_Task, VM programs will be transmitted/interpreted in a timely ordered sequence by means of the ISR3_Task. The DPU uses observation parameters internally.
A unified generalized protocol to update the various tables is foreseen.
Each updating block of data will contain a 4-byte header followed by the relevant data

Len	Field	Description
8	Table number (ID)	Configuration, observation, VM prog….. etc
8	N. of data items which follows	
4	Type of data	0=8 bit , 1=16, 2=24, 3=32 bit
12	Offset from beginning of the table	0=8 bit , 1=16, 2=24, 3=32 bit
8..32	Data	8,16,24 or 32 bit words as defined above
	…	
	Data	

If data type is 0 (8 bit items) an even number of data must follow.

Depending on the table number some different block header or implicit rule might be used, an example is an implied 16 bit left shift for HK request commands (stored on DPU as 32 bit words with the 16 LSB all '1').

A type 8 "function management" is foreseen with one or more data blocks as above.

This generalized method has the advantage to permit whatever reconfiguration might be needed from the ground, besides being extremely simple as OBS implementation. The TC bandwidth should be reduced due to the possibility to upload only part of the relevant tables.

3.1.2 DPU Internal Commands

This set of commands, fixes the operating mode of the DPU and are in general responsible for setting the SS configuration and executing the observation.
The internal commands may be divided in three categories:

- Instrument specific commands
- Observation parameters (# of data blocks to transmit etc)
- System commands

A preliminary set of DPU internal instrument specific commands follows:

Mnemonic	Parameters	Description
Run	Addr	Runs VM program beginning at address (index) Addr. (ISR_3_Task)
Stop		Halt the current running VM program
TblLoad	Tbl#, Addr, N	Transmits to SS via the LS_Task the N 32 bit words (commands) of table Tbl#, beginning at address (index) Addr
TblDump	Tbl#, Addr, N	Transmits in TM the N 32 bit words of table Tbl#, beginning at address (index) Addr
FifoFlush	N	Flushs FIFO number N [1 … 3]. If N=0 flush all
Sync		Sends broadcast DPU time to SS
Exec	P1,P2 … Pn	Executes some specialized function/Task, which need an "hard coded" function as for the tuning procedure. (TUNING_Task)

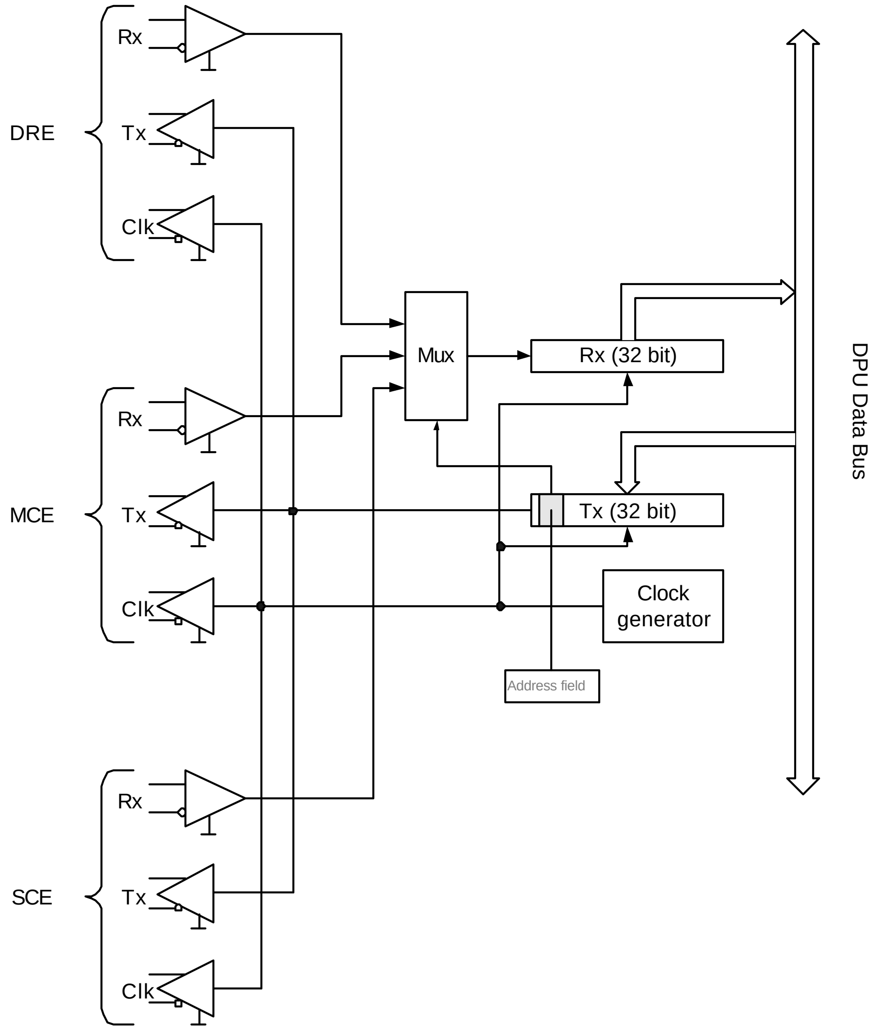
The preliminary lists of DPU internal system commands follows:

- Time sync commands: The time received by the S/C is compared with the DPU internal clock and a constant offset is generated. The deviation of ICT time from the S/C time will be of the order of a ms.
- Housekeeping transmission rate: The rate sets the execution period of the HK_REQ_Task
- HK enable/disable

3.2 HK_REQ_Task

3.3 LS_Task

This task sends commands and HK requests to the SS via the low speed serial I/F.
Commands and HK requests written to the out buffer (Tx) are sent to all the SS, the address field of an HK request set the multiplexer (Mux) to the addressed SS in order to receive the requested HK on the input buffer (Rx).



3.4 ISR_3_Task

The ISR_3_Task is actually a Virtual Machine executing one group of instructions (coded in the Exec_table) per each INT3 request. The Exec_Table is actually a one column 32 bit word vector containing commands to SS, timer setting (int3), mutex, loop and other Virtual Machine "assembler" instruction, operating as an absolute program.
A number of baseline VM programs, with functionality for the foreseen observation modes, will be stored on the DPU/DPU. These programs, stored in a fixed memory location, will be modified/reloaded via TC, thus easing the need for OBS patching. A program can be as simple as a loop calling a preloaded subroutine.

VM Program

PC	Instructions
0	Set_Timer(1)
1	Mutex(1)
2	…
3	Call_Subr(100)
4	…
…	…
…	…
100	Send_Command_Reg(addr,cod,reg)
101	….
102	…
103	Return
…	…

Subroutine 100

For each INT3 request, the block of instructions from the current VM program counter (PC - index vector value) up to (excluding) the next SS command or mutex or NOP instruction, are executed. A set of VM internal registers R[32] TBC, are foreseen in order to execute loop, to pass parameters with subroutine etc. Each register is a 32 bit unsigned integer.

3.4.1 Virtual Machine instructions

A preliminary set of "VM assembler" instructions follows:

Instr. code (hex)	VM asm Mnemonic	Description	Code type
(7)	**CMD**	**Send_Command(addr, code, val)**[1] Send command code/val to SS addr	
0	**RCMD**	**Send_Command_Reg(addr, code, reg)**[1] Send command code/R[reg] to SS addr	3
1	**MTX**	**Mutex(OnOff)**[2] Lock/Unlock low speed I/F port	1
2	**NOP**	**NOP()** No operation	1
8	TIM	Set_Timer(val)[3] Set counter value (us) for next IRQ3	1
A	READ	Read_HK_Reg(reg) Store received HK in R[reg]	
10	RINC	Increment_Register(reg) R[reg] = R[reg] + 1	1
11	RDEC	Decrement_Register(reg) R[reg] = R[reg] - 1	1
12	RSET	Set_Register(reg, val32)[4] R[reg] = val32	1[4]
13	RADD	Add_To_Reg(reg, va32l)[4] R[reg] = R[reg] + val32	1[4]
14	RSUB	Sub_To_Reg(reg, val32)[4] R[reg] = R[reg] – val32	1[4]
15	RAND	And(reg, val32)[4] R[reg] = R[reg] & val32	1[4]
16	ROR	OR(reg, val32)[4] R[reg] = R[reg] | val32	1[4]

[1] Assuming all commands can be divided in 3 fields. If this is not the case "code" disappears.
[2] May be forced in the program, but the compiler insert automatically this instruction whenever is needed based on the optimisation level.
[3] This time is the interrupt period valid after the next instruction. The minimum interrupt period is the maximum value between the time used by the I/F to transmit a command (100 us) and the actual duration of the ISR3. For the time being let's fix it to 1 ms. This period is the minimum period between two SS commands
[4] These instructions are coded as two consecutive 32 bit words, the second containing the plain value of "val32".

17	RREQ	Reg_Equate(reg1,reg2) R[reg1] = R[reg2]	2
21	JMPR	Jmp_Relative(vmAddr) PC = PC + vmAddr	1
23	RJPR	Jmp_Relative_Reg(reg) PC = PC + R[reg]	1
25	JPNZ	JumpNZ(reg, vmAddr) If (R[reg] !=0) PC = PC + vmAddr	2
26	RSZ	Skip_Reg_Zero(reg) If (R[reg] ==0) PC = PC + 1	1
27	RSGT	Skip_Reg_GT(reg1,reg2) If (R[reg1] > R[reg2]) PC = PC + 1	2
28	RSLT	Skip_Reg_LT(reg1,reg2) If (R[reg1] < R[reg2]) PC = PC + 1	2
30	CALL	Call_Subr(vmAddr) PC = vmAddr (remember the present PC)	1
31	RET	Return() Return from subroutine	1
41	WRT	Write(reg) Write R[reg] to ICU frame	1
50	END	End End current VM program	1
	DEF	Set constants	
	ORG	Address of code	
	_Label	Label referred by loop/jmp	
	COM	COM textstring Comment printed during the simulation	

It has to be noted that in order to make the VM program as relocable as possible, all jump instructions, with the exclusion of the Call Sub, are relative to the PC.

The table notation is:

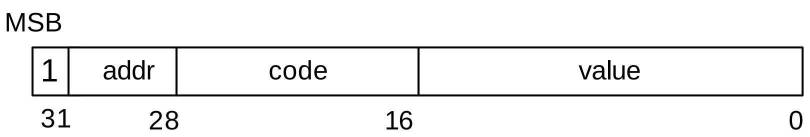
Val 16 or 24 bit numeric constant possibly defined in a DEF statement.

Reg VM internal registers index. Numeric constant between 0 and 31 possibly defined in a DEF statement.

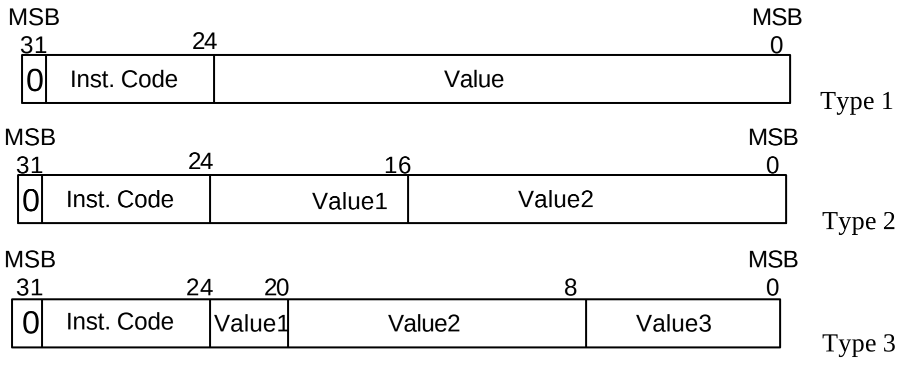
VmAddr Signed 16 bit numeric constant indicating the relative address displacement in a Jump instruction. It may be coded as a _label mnemonic, in this case the relative address displacement is computed by the compiler.

The preliminary instruction coding is as follows:

1. First (MSB) bit=1 then it is a plain command to the SS, as the first bit (start bit) is always set.
   Here we assume that the data content of the command can be splitted in two fields (code and value). The MSBit of addr field indicate cmd/hk request.



2. First (MSB) bit=0 then it is a coded 32 bit instruction with:



A VM assembler compiler/simulator program is provided in order to simplify the on ground coding of the observation programs.

3.4.2 VM Compiler

The compiler resolve all the mnemonic labels and constant in a VM program and produce the absolute VM code. The compiler optimiser try also to take care of the MTX instructions which enable/disable the low speed I/F usage by the LS_Task.
The compiler run time arguments are:

Switch	Description	Default value
o0, o1, o2	Different levels of optimisation	o0 = No optimisation
v0, v1, v2, v3	Verbose level for terminal display	v0 = minimum display
prg filename	VM program input file	prg spire.vm
lst filename	Compiler/Simulator output filename	lst list.txt
t1 n	Start time for the simulation display [us]	t1 0
t2 n	End time for the simulation display [us]	t2 100000000
pc n	Address (program counter) of the program entry point	pc 0

The compiler optimisation level 1, check for a TIM instruction and, based on a threshold value (10 ms), test/insert a correct MTX instruction using the following criteria:

```
If TIM > threshold and last MTX=1
      Then check/insert MTX=0
If TIM < threshold and last MTX=0
      Then check/insert MTX=1
```

The compiler optimisation level 2, add to the level 1 optimisation a check for any “unprotected” (MTX=0) CMD/RCMD instruction, and protect the command with a double TIM-MTX couple using the following criteria:

```
If exist a CMD/RCMD instruction while MTX=0 and TIM=oldtim
      Then modify to:
            TIM 2000    (1 ms is chosen as the minimum TIM value)
            MTX 1
            CMD/RCMD xxx  (original instruction)
```

```
TIM oldtim
MTX 0
```

The TIM-MTX instructions inserted by the optimiser are prefixed by A_

Example:

No optimisation	Optimisation level 1	Optimisation level 2
MTX 0	MTX 0	MTX 0
…	…	…
TIM 1000	TIM 1000	TIM 1000
CMD aaa	A_MTX 1	A_MTX 1
CMD bbb	CMD aaa	CMD aaa
CMD ccc	CMD bbb	CMD bbb
TIM 100000	CMD ccc	CMD ccc
TIM 1000	TIM 100000	TIM 100000
CMD ddd	A_MTX 0	A_MTX 0
…	TIM 1000	TIM 1000
…	A_MTX 1	A_MTX 1
…	CMD ddd	CMD ddd
…	…	…
MTX 1	MTX 1	MTX 1
…	…	…
TIM 110000	TIM 110000	TIM 110000
CMD xxx	A_MTX 0	A_MTX 0
TIM 1000	**CMD xxx**	A_TIM 2000
CMD aaa	TIM 1000	A_MTX 1
CMD bbb	A_MTX 1	**CMD xxx**
…		A_TIM 110000
		A_MTX 0
		TIM 1000
		MTX 1

3.4.3 VM Simulator

The simulator section of the compiler program, is a modified version of the OBS VM section, the simulator control any “unprotected” CMD/RCMD instruction and output (on the out list file) a timeline of the SS commands. Comment instructions:

COM comment string

inserted in the input program, are listed by the simulator as:

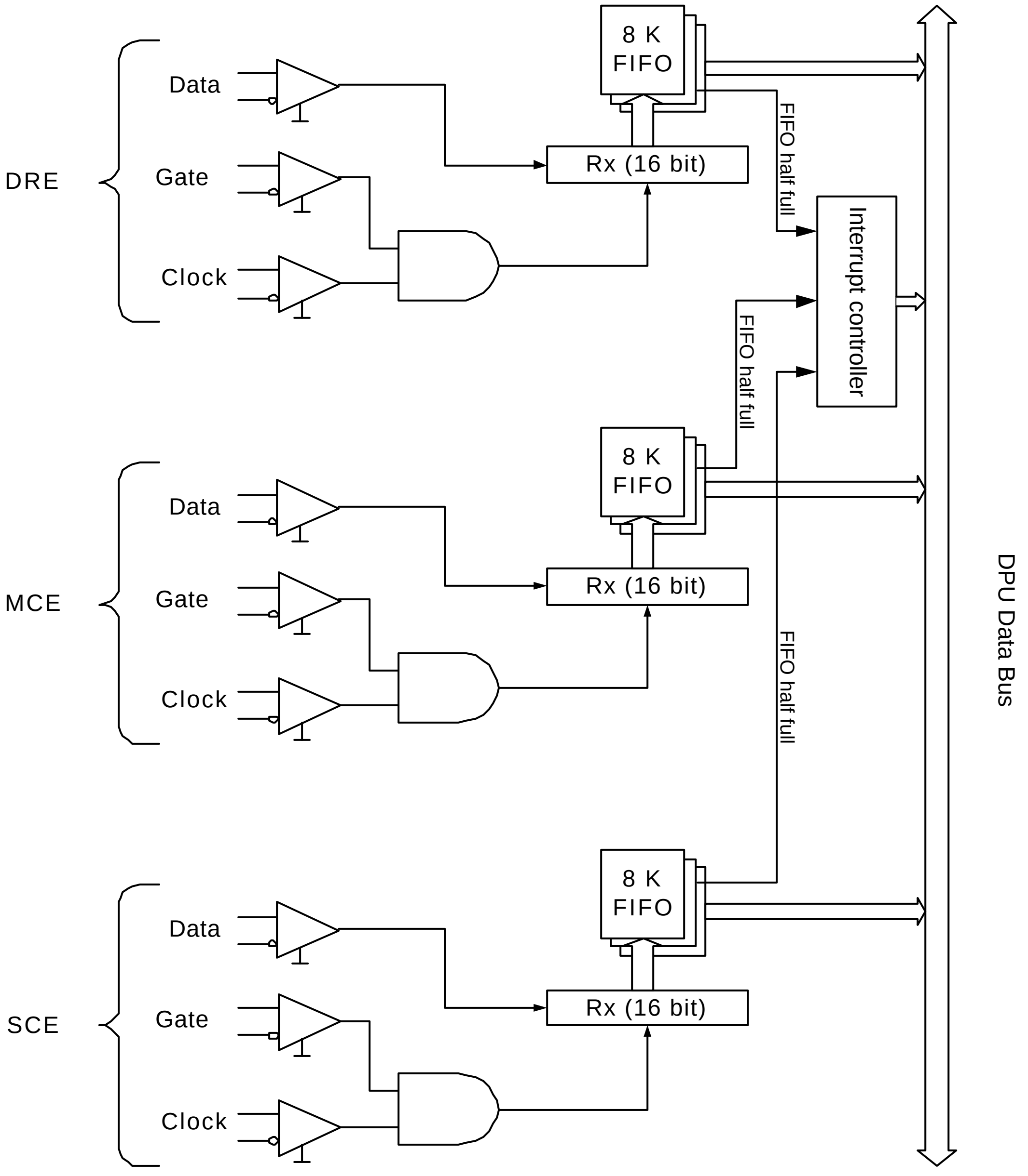
COM comment string [addr,n]

with addr = address of the next istruction
n = auto incrementing number counting # of comment occurrence.

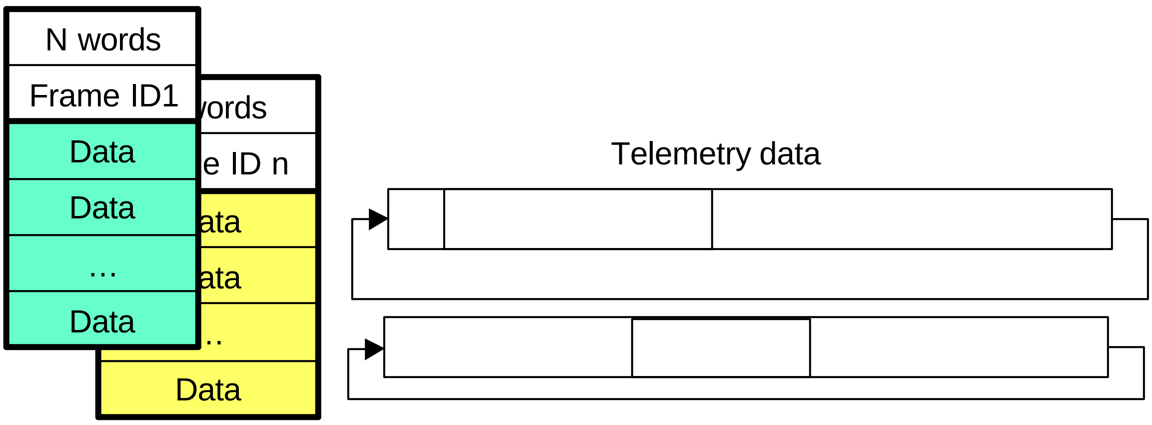
In the appendix a test program compilation/simulation is provided.

3.5 HS_Task

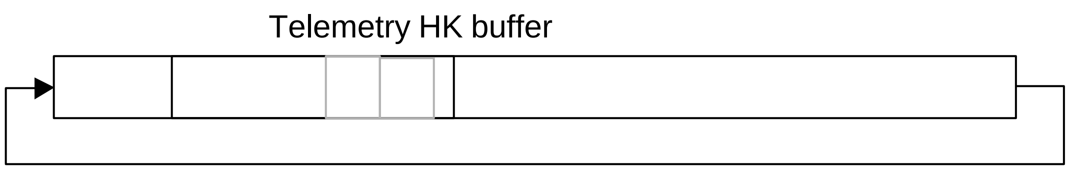
This task collect science and HK data on the high speed I/F, format the data in “packet transmission ready” and store the data on the right buffer.
In order to allow asynchronous collection of science, the 16 bit data words on the high speed I/F are temporary stored on four 8Kword deep FIFOs, at a rate of 16 us per word.
The “half FIFO full” signal associated to each FIFO generate an interrupt (IRQ 0) which in turn schedule the HS_Task (through a Virtuoso event) on the run queue.



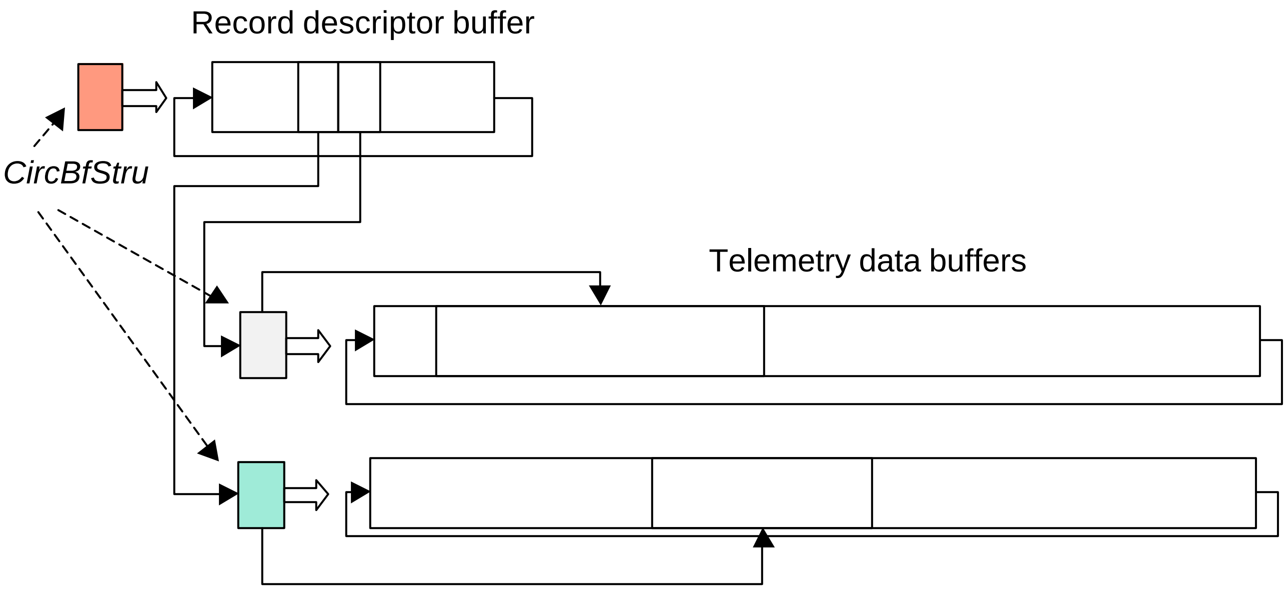
Data coming from FIFOs must be organized in frames, each frame consisting of a header (stripped before transmission) followed by data.
The header must contains at least the number of words which follows and the ID of the frame.
Each frame is stored in a separate circular buffer depending on the frame ID.



Some frames might contain HK data, the HK will overwrite the appropriate locations of the SPIRE common HK current buffer.



The presence of a TM data packet ready for transmission is signalled by a new entry (of type RecDesStru) in the science record descriptor buffer.



3.6 TMTC_Task

3.7 IDLE_Task

Appendix

Reason for a Virtual Machine

The driving requirement for the VM is the time sequence constraint between SS commands during an observation. The time sequence jitter on the SS commands (LS I/F) goes from seconds down to 10us.
Consider the following example:

Cmd1 @ T

Cmd2 @ T + t1 +-5ms = T2

Cmd3 @ T2 + t2 +- 100ms = T3

Cmd4 @ T3 + t3 +- 5us = T4

It is clear that, in a multi-task OS as Virtuoso, the only way to achieve the 10us and probably the 10 ms constraint is via an Interrupt Serviced Routine (with a high priority interrupt). It is also evident that once it has been decided to implement the interrupt environment, every command in the sequence should be sent via interrupt, so that all the commands will have the same (10 us) jitter in the time sequence.
The HW problem to generate the sequence of different period interrupts, is solved by using the DPU programmable 32 bit (1 MHz clock) down counter. This down counter starts decrementing its content from the last preset initial value, and generates an interrupt on zero value. Then the counter restarts again the cycle, beginning from the last preset initial value loaded before the zero count.
Now we have a mechanism, which forces the execution of a routine (ISR_3) at pre-defined time intervals. Entering the routine, the relevant SS command must be sent. In order to preserve the time jitter constraint, this command must be already prepared (in a table).
After the command is sent (written in the low speed serial output I/F), we might want to change the down counter initial count for the next interrupt, the only time constraint now is to exit from the ISR before the present terminal count. This new “initial count” value will be stored in some table, let’s say we store this value in the same table with the command sequence.
We can build a table as a sequence of two words: command and initial count, and perform always the same two operations inside the ISR:

- Increment the table pointer and send the command stored at the current table location

- Increment the table pointer and preset the initial count stored at the current table location

This scheme is not the most efficient in the case when a series of commands can be equally spaced in time and use the same initial count with no need to rewrite it. Moreover we have to disable/enable the LS_Task, depending on the interval time between the SS observation commands (HK are collected via LS_Task), as an example we might decide that every time the delay between two commands is grater than 10ms we want to enable LS_Task. So we have to build a table that is interpreted inside the ISR: every time an interrupt occurs a number of actions (table instructions beginning at the current pointer) is performed, the first one (time critical) being a command to SS and the following being some type of DPU internal commands.
Now we have come to a long table containing all the SS and DPU observation commands already somehow interpreted by an OBS routine (ISR_3). The first thing to note is that the commands are repeated in block as in a computer loop, so why not to add an DPU internal loop command to the table? Well to do so we must also define some local variable (register R[32]), then we could add other simple features like subroutine etc.
Ok we have come to a Virtual Machine implemented inside the ISR_3 routine.

Sample VM code

The following VM program (file name spire.vm) has been compiled with the command:

compiler o2 v1

```
;-----------------------------------
; Case insensitive
; Comments begin with a ;
; Labels begin with a _ and are alone in a line
;-----------------------------------
      def timF 2000            ; Fast command sequence (2 ms)
      def timS 100000          ; Slow command sequence (100 ms)
      def CmdDCU 4             ; Address for DCU commands
      def CmdMCU 5             ; Address for MCU commands
      def CmdSCU 6             ; Address for SCU commands
      def CmdBRC 7             ; Address for Broadcast commands
      def HkDCU  0             ; Address for DCU commands
      def HkMCU  1             ; Address for MCU commands
      def HkSCU  2             ; Address for SCU commands

      def r0val 0xa            ; constants in dec or hex

      TIM   timS               ; the optimiser 01 switch insert a MXT
      RSET  0,r0val            ; R[0]=0xa
                               ; here the optimiser o2 switch take care of
                               ; the TIM/MTX for the next two commands
      rcmd 4,0xfff,0           ; command to addr 0 (MSbit of addr is cmd/hk)
      cmd  5,0x55,0xffff       ; command to addr 1 (MSbit of addr is cmd/hk)
      RSET 0,5
      RSET 1, 10
      TIM timF
      COM Calling _hksum with R[0]=5 and R[1]=10 ; comment line for the
simulator
      call _hksub              ; in the sub loop 6 times
      RSET 0, 15
      COM Calling _hksum with R[0]=15 and R[1]=10
      call _hksub              ; in the sub loop 10 times (R[1] limit
reached)
      END
```

```
;-------------------------------------------------------------------
; This loop is executed until the received hk (in the simulator
; is equal to R[31]) is greater than R[0], up to a maximum of R[1]
; times. The received hk in the simulator is R[31], this value is
; simply incremented.
; Local variable R[3] and R[2]
;-------------------------------------------------------------------
      ORG 100
_hkSub
      Com This is a comment for the simulator
      RSET 31,0              ; Simulator specific
      RREQ 3,1               ; save max loop count in local var R[3]
_hkloop
      rcmd  HkSCU,0x123,31   ; HK req to addr SCU
                             ;(MSbit of addr is cmd/hk)
      nop                    ; must wait at least 2 ms (timF)
                             ; to have the hk reply
      TIM   timS             ; the optimiser insert a MXT
      rinc 31                ; increment R[31]: simulator specific
      read 2                 ; HK (R[31] in the simulator) in R[2]
      TIM   timF             ; the optimiser insert a MXT
      RDEC  3                ; check for max loop
      JPNZ  3, 2             ; skip if R[3] !=0
      RET
      rsgt 2,0               ; loop untill hk (R[2]=R[31]) <= R[0]
      jmpr _hkloop           ; next loop
      RET
```

Here follows the Compiler /Simulator output stored in the default list file "list.txt". Note the A_TIM and A_MTX instructions inserted by the optimizer.

```
VM program file spire.vm
Optimisation level= 2
Start address (PC)= 0

Addr   opCode     Instruction
----  --------  --------------------
   0            def    timF 2000
   0            def    timS 100000
   0            def    CmdDCU 4
   0            def    CmdMCU 5
   0            def    CmdSCU 6
   0            def    CmdBRC 7
   0            def    HkDCU 0
   0            def    HkMCU 1
   0            def    HkSCU 2
   0            def    r0val 0xa
```

```
  0  80186a0  TIM    timS
  1 12000000  RSET   0    r0val
  2        a         r0val
  3  80007d0  A_TIM  2000
  4  1000001  A_MTX  1
  5   4fff00  rcmd   4    0xfff 0
  6  80186a0  A_TIM  100000
  7  1000000  A_MTX  0
  8  80007d0  A_TIM  2000
  9  1000001  A_MTX  1
 10 d055ffff  cmd    5    0x55 0xffff
 11  80186a0  A_TIM  100000
 12  1000000  A_MTX  0
 13 12000000  RSET   0    5
 14        5         5
 15 12000001  RSET   1    10
 16        a         10
 17  80007d0  TIM    timF
 18  1000001  A_MTX  1
 19           Calling _hksum with R[0]=5 and R[1]=10
 19 30000064  call   _hksub
 20 12000000  RSET   0    15
 21        f         15
 22           Calling _hksum with R[0]=15 and R[1]=10
 22 30000064  call   _hksub
 23 50000000  END
100           ORG    100
100           _hkSub
100           This is a comment for the simulator
100 1200001f  RSET   31   0
101        0         0
102 17030001  RREQ   3    1
103           _hkloop
103   21231f  rcmd   HkSCU 0x123 31
104  2000000  nop
105  80186a0  TIM    timS
106  1000000  A_MTX  0
107 1000001f  rinc   31
108  a000002  read   2
109  80007d0  TIM    timF
110  1000001  A_MTX  1
111 11000003  RDEC   3
112 25030002  JPNZ   3    2
113 31000000  RET
114 27020000  rsgt   2    0
115 21fffff4  jmpr   _hkloop
116 31000000  RET
```

```
Begin simulation from t1= 0  up to t2= 100000000

    Time    PC   Command
    2000     5  cfff000a
  106000    10  d055ffff
Calling _hksum with R[0]=5 and R[1]=10  [19, 1]
This is a comment for the simulator [100, 1]
  210000   103  a1230000
  316000   103  a1230001
  422000   103  a1230002
  528000   103  a1230003
  634000   103  a1230004
  740000   103  a1230005
Calling _hksum with R[0]=15 and R[1]=10  [22, 1]
This is a comment for the simulator [100, 2]
  846000   103  a1230000
  952000   103  a1230001
 1058000   103  a1230002
 1164000   103  a1230003
 1270000   103  a1230004
 1376000   103  a1230005
 1482000   103  a1230006
 1588000   103  a1230007
 1694000   103  a1230008
 1800000   103  a1230009
Found END. Normal end of execution

Simulation: total No of errors: 0
```