

Herschel Space Observatory SPIRE-DPU Virtual Machine

Ref: CNR.IFSI.2003.TR01

Issue: 2.5

Date: 15/11/2005

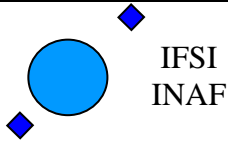
Page: 1 of 37

SPIRE-IFS-DOC-001622

SPIRE-DPU Virtual Machine

ISSUE: 2.5

	Name and function	Date	Signature
Prepared by:	Riccardo Cerulli-Irelli	27/03/2003	
Verified by:			
Approved by:			



Herschel Space Observatory SPIRE-DPU Virtual Machine

Ref:
Issue: 2.5
Date: 15/11/2005
Page: 2 of 37

Distribution List :

K. King	SPIRE	
S.D. Sidher	SPIRE	
J.L. Auguères	SPIRE	
C. Cara	SPIRE	
R. Orfei	IFSI	
A. Di Giorgio	IFSI	
S. Molinari	IFSI	
S. Pezzuto	IFSI	
S. J. Liu	IFSI	

Document Status Sheet:

Issue	Revision	Date	Reason for Change
	Draft 1	11/06/2002	Initial issue
	Draft 2		Added ICALL and TABLE instructions New program figures with many new features.
	Draft 3	23/09/2002	Added ICPT, ICPF, TER13, TER15, TER17, EVNT, TXTBL instructions. New input file to simulate READ data words.
	Draft 4	3/02/2003	Added IRCALL, IRCPT, IRCPF
1.0		27/03/2003	Changed packet definition in sect. 4.3
		10/09/2003	Update definition of TXTBL in sect 3.7
1.1		24/2/2004	Added NAME, VERSION and CVSID in sect 3.7
1.2		19/04/04	Typo corrections More info on VM instructions (paragraph 3.7)
1.2		15/7/2004 18/7/2004	Added LTIM instruction (paragraph 3.7) Added EVERR instruction (paragraph 3.7) Modified TLC packet wit addition of w16[6]: structure ID
2.0		6/9/2004	Added a pre-processor (paragraph 3.7.1, 6.1)
2.1		15/9/2004	Added XREQ new instruction § 3.7 Added TRST new instruction § 3.7 § 4.2 Removed bug on multiple INC at the same level Optimisation is now obsolete and discouraged § 4.1
2.1.1		12/11/2004	Added foot note for opcode 50 – 54
2.2		25/02/2005	Added SVEV new instruction § 3.7 Modified parameters in EVNT and EVERR § 3.7
2.3		9/03/2005	Added RSVEV new instruction § 3.7
2.4		22/04/2005	Added VMSTP new instruction § 3.7
2.5		28/11/2005	Added OVRD new instruction § 3.7

Reference documents

Document Reference	Title
RD1	Contents of a SPIRE VM Table File. Ref: SPIRE-RAL-NOT-001907

Acronyms

CDMS	Central Data Management System
CI	Critical instruction
CNR	Consiglio Nazionale delle Ricerche
CPU	Control Processing Unit
DPU	Digital Processing Unit
FCU	Focal plane Control Unit
FIFO	First In First Out storage element
FIRST	Far InfraRed and Submillimeter Telescope
HK	HouseKeeping
HRS	High Resolution Spectrometer
HW	HardWare
DPU	Digital Processing Unit
I/F	Interface
IFSI	Istituto di Fisica dello Spazio Interplanetario
ISR	Interrupt Service Routine
LCU	Local Oscillator Control unit
LSB	Least Significant Bit(s)
LSU	Local oscillator Source 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
RT	Real Time
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
WBS	Wide Band Spectrometer

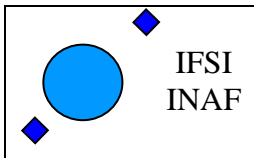
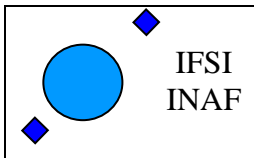


Table of contents

Reference documents	3
1 Introduction	5
2 Reason for a Virtual Machine	5
3 The Virtual Machine	6
3.1 Critical instructions (CI).....	6
3.2 VM structure	6
3.3 VM_Map Table	7
3.4 VM Program.....	7
3.5 VM program exec TC	9
3.6 VM Multitasking	9
3.7 VM Instructions.....	10
3.7.1 The generic pre-processor	12
3.8 Instructions Format	13
4 VM Compiler/Simulator	14
4.1 Compiler.....	14
4.2 VM Simulator.....	18
4.3 Packetiser	19
4.4 Directory structure.....	21
5 Example	22
6 Appendix.....	29
6.1 The Generic PreProcessor	29



1 Introduction

This document describes the special command line interpreter of SPIRE-DPU, implemented in order to control the SS (via the LS I/F) in all the situation where the variations in time distance between commands must be less than few milliseconds. Such a command line interpreter can be seen as a kind of an elementary computer with a simple pseudo assembler commanding language that from now on we call virtual machine (VM). The document describes also the developing SW tools associated with the VM which consists in a compiler, a simulator and a VM-program TC packet generator.

2 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 a 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.

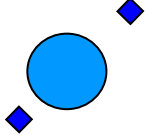
We can build a table as a sequence of two words: command and initial count, and perform always the same two operations inside the ISR:

- 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 greater 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 blocks 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[256]), then we could add other simple features like subroutine etc.

Ok we have come to a Virtual Machine implemented inside the ISR_3 routine.

 <p>IFSI INAF</p>	<h1>Herschel Space Observatory</h1> <h2>SPIRE-DPU Virtual Machine</h2>	<p>Ref: Issue: 2.5 Date: 15/11/2005 Page: 6 of 37</p>
--	--	---

3 The Virtual Machine

3.1 Critical instructions (CI)

The VM is used to send timely synchronized commands to the SS via the LS I/F, each command is transmitted when the HW down counter generates an interrupt. This SS commands are here defined as “critical” instructions (CI), each CI may be followed by a number of non CI which are executed during the same interrupt cycle. The DPU has just one LS I/F which must be used both by the VM and by the LS_Task for non time critical commands like HK request et al. In order to avoid collision on the LS I/F , a VM CI which lock the I/F has been introduced. This CI, which is effectively a mutual exclusive flag (mutex), must be executed at least 2 ms prior the use of the I/F by the VM in order to allow the termination of an HK request to a possible running LS_Task. The last “dummy” CI is a no operation (NOP) instruction, to be used whenever a time gap must be introduced in the program. Typical use of NOP is before a READ instruction.

3.2 VM structure

The main components of the VM are:

- Program area
- VM-CPU clock
- Interpreter routine
- Local variable storage

Program area - This is a 32 bit words table (array of up to 32 Kword) containing the SS commands and local control instructions which forms a VM program. The table effectively represent the program/data memory area of the VM, with the table position (array index) acting as the program counter (PC). The table will contains a number of VM programs with associated tables of constants and subroutines. Each program is identified by the table position (array index) of the entry point.

VM-CPU clock - As mentioned the VM clock is generated by a down counter whose period is dynamically modifiable by the VM. This variable period clock, triggers an interrupt signal (IRQ3) which force the DPU CPU to execute the interpreter routine, thus executing a block of VM instructions.

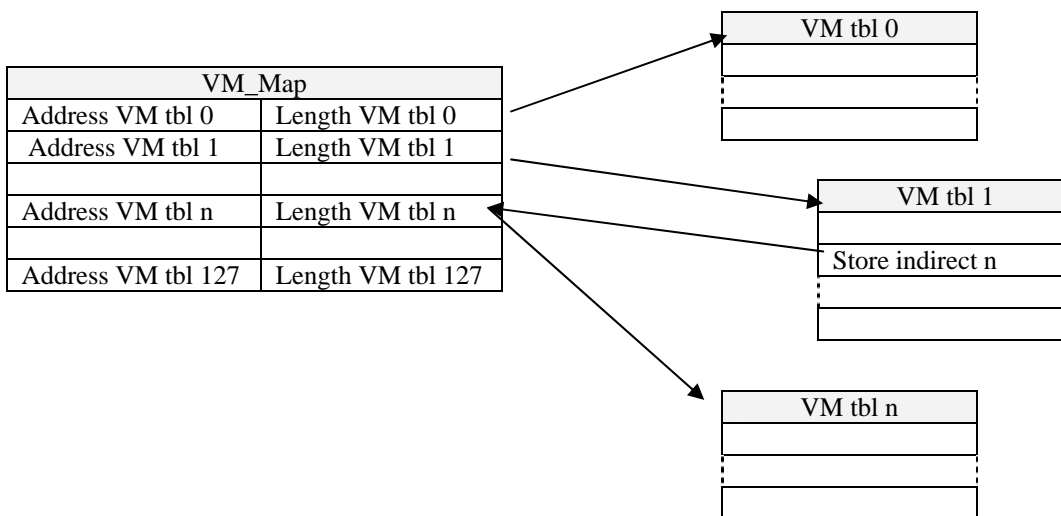
Interpreter routine – The interpreter routine executes a block of VM instructions starting at the present PC up to (excluding) the next “critical” instruction CI (SS command, mutex or NOP instruction). So, for every VM-CPU clock, a block of instructions is executed, the first one (time critical) being a command to SS and the following being some type of DPU internal commands. This scheme effectively minimize the SS commands time jitter.

Local variable – In order to implement simple mathematical operations on SS commands, pass parameters to subroutine and keep track of “for” loops counts, a number of internal “global” registers are implemented. The 256 registers (R[0] ... R[255])¹ are statically defined inside the interpreter routine and are common to the stored VM programs.

¹ Register R[255] is also used as offset in the VM instructions ICPT and ICPF. R[254] may be used by the simulator to mimic the low speed READ port.

3.3 VM_Map Table

The DPU will contains a number (128 TBC) of VM tables (program area), each table with a maximum dimension of 32 Kwords may contain one or more VM program and may reside everywhere in the DPU data memory area. The physical address and the dimension of each VM table being stored in the 128 x 2 (TBC) **VM_Map** table. The VM program “scope” is the actual VM table, but using the “move/store indirect” or “call subroutine indirect” may also span (using the VM_Map) to the other tables.



3.4 VM Program

Each VM table has been divided in 3 sections:

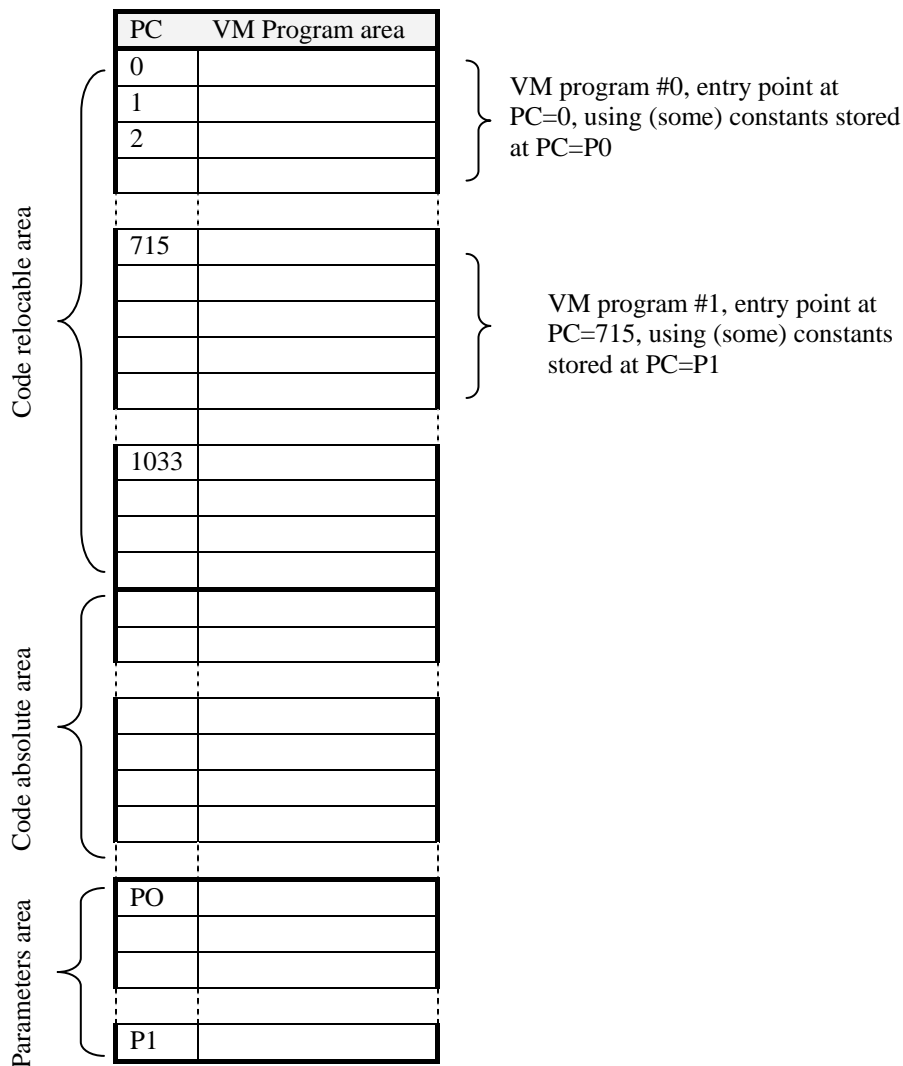
- Code relocable area
- Code absolute (library subroutine) area
- Parameters area

Code relocable area - In this area are stored the different VM programs, each program associated to an observation routine or time critical task. The programs here are completely relocable, to achieve this goal all “JUMP” instructions, with the exception of the “CALL SOUBROUTINE”, are relative.

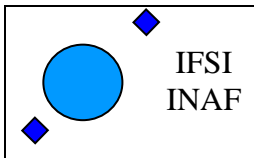
Code absolute area - In this area are stored the “subroutine libraries” of the VM programs. As the “CALL SUBROUTINE” is implemented as a jump to an absolute address, VM instructions here coded are supposed to be relatively stables. Whenever the entry points of the library changes, the VM programs referring to the library must be updated.

It has to be noted that in this context the term absolute refer to the VM table (offset from the beginning of the table), so that each table can still be moved in the DPU memory with no modification to the VM code.

Parameters area - Each observation configuration/execution routine, store in a dedicated fixed portion of this area all the observation parameters.



A number of baseline VM programs, with functionality for the foreseen observation modes, will be stored on the DPU VM tables. These programs, stored in the VM memory area, may be modified/reloaded via TC, thus easing the need for OBS patching. The modification/addition is a simple table upload which can be performed via few TC packets to be compared to the lengthy and possibly dangerous OBS patching procedure. The compiler/simulator program described in the next chapter generates (also) the TC packet of the compiled VM program.



3.5 VM program exec TC

The VM program execution telecommand must indicate:

- I_map – index of the VM_Map table pointing to the VM table with the program
- I_prg – index in the program entry point area of the VM table with the address of the program
- N – number of run time parameters of the VM program to be stored in the first R[256] VM registers
- R[0] – first parameter
-
- R[n-1] – last parameter

3.6 VM Multitasking

In order to implement a VM multitask, SPIRE will use two types of VM: a “Real Time” VM and a “non-Real Time” VM.

The Real Time VM is the one just described which use the hardware down counter as “CPU clock”, the highest priority interrupt line (IRQ3) and direct access to the low speed interface via the lock mechanism (mutex). This VM may execute just one program at the time and is used in time critical tasks.

The second non-Real Time VM is the same as the Real Time one but use the Virtuoso OS sleep instruction to implement the “CPU clock”, and utilise the same LS_Task used by HK and normal commanding via a higher priority queue thus avoiding the lock mechanism. This second VM can execute different programs in a multitasking-like way utilising the multitask feature of Virtuoso OS.

The VM code for the non RT VMs is the same used by the RT one. In order to maintain full compatibility, the sleep time will refer to the time interval between the next critical instruction and the followings.

3.7 VM Instructions

The set of “VM assembler” instructions follows:

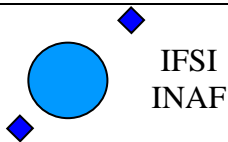
Instr. code (hex)	VM asm Mnemonic	Description	Code type
Critical instructions			
(7)	CMD	Send_Command(addr, code, val) ² Send command code/val to SS addr. Command=0x80000000 (add & 0x7) <<28 (code & 0xFFFF)<<16 val & 0xFFFF	
0	RCMD	Send_Command_Reg(addr, code, reg) ² Send command code/R[reg] to SS addr. Command=0x80000000 (add & 0xF) <<28 (code & 0xFFFF)<<16 R[reg] & 0xFFFF	3
4	RSND	Send_Reg_Command (reg) Send command R[reg] to SS	1
1	MTX	MTX 1/0 => Mutex(On/Off) Lock/Unlock low speed I/F port	1
2	NOP	NOP() No operation	1
Non critical instructions			
8	TIM	Set_Timer(val) ³ Set counter value [us] for next IRQ3. Max value for val is 16,777,215 [us]	1
B	LTIM	Set_Timer(val) ³ Set counter value [ms] for next IRQ3. Max value for val is 4,294,967 [ms]	1
9	RTIM	Set_Timer(R[reg]) ³ Set counter value [us] for next IRQ3	1
A	READ	Read_HK_Reg(reg) Store received HK in R[reg] <i>For simulation purpose, data is read from an optional file or R[254] register (see chapter 3.9)</i>	1
C	OVRD	OVRD 1/0 => Override (On/Off) ⁴ the command inhibition system, up to the next OVRD instruction.	1
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) ⁵ R[reg] = val32	1
13	RADD	Add_To_Reg(reg, val32) ⁵ R[reg] = R[reg] + val32	1
14	RSUB	Sub_To_Reg(reg, val32) ⁵ R[reg] = R[reg] - val32	1
15	RMUL	Multiply_To_Reg(reg, val32) ⁵ R[reg] = R[reg] * val32	1
16	RDIV	Divide_To_Reg(reg, val32) ⁵ R[reg] = R[reg] / val32	1
18	RAND	And(reg, val32) ⁵ R[reg] = R[reg] & val32	1
19	ROR	OR(reg, val32) ⁵ R[reg] = R[reg] val32	1
1A	RSHR	Reg_Shift_Right(reg, val) R[reg] >>= val	2
1B	RSHL	Reg_Shift_Left(reg, val) R[reg] <<= val	2
1F	XREQ	Indexed_Reg_Equate(reg1, reg2) R[R[reg1]] = R[R[reg2]]	2
20	RREQ	Reg_Equate(reg1, reg2) R[reg1] = R[reg2]	2
21	RRAD	Add_Register_To_Register(r1, r2, r3) R[r1] = R[r2] + R[r3]	4
22	RRSB	Sub_Register_To_Register(r1, r2, r3) R[r1] = R[r2] - R[r3]	4
23	RRMP	Multiply_Register_To_Register(r1, r2, r3) R[r1] = R[r2] * R[r3]	4
24	RRDV	Divide_Register_To_Register(r1, r2, r3) R[r1] = R[r2] / R[r3]	4
30	JMPR	Jmp_Relative(vmAddr) PC = PC + vmAddr	1
31	RJPR	Jmp_Relative_Reg(reg) PC = PC + R[reg]	1

² Here the C language syntax is used (<<n => left shift n positions, & => AND, | => OR, 0xhh => Hex constant).

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 The simulator mark with “*” the “overridden” commands

⁵ These instructions are coded as two consecutive 32 bit words, the second containing the plain value of “val32”. Do not put this opcode after a “skip” (RSZ, RSGT, RSLT) instruction.



Herschel Space Observatory SPIRE-DPU Virtual Machine

Ref:
Issue: 2.5
Date: 15/11/2005
Page: 11 of 37

Instr. code (hex)	VM asm Mnemonic	Description	Code type
32	JPNZ	JumpNZ(reg, vmAddr) If (R[reg] !=0) PC = PC + vmAddr	2
33	RSZ	Skip_Reg_Zero(reg) If (R[reg] ==0) PC = PC + 2	1
34	RSGT	Skip_Reg_GT(reg1,reg2) If (R[reg1] > R[reg2]) PC = PC + 2	2
35	RSLT	Skip_Reg_LT(reg1,reg2) If (R[reg1] < R[reg2]) PC = PC + 2	2
40	CALL	Call_Subr(vmAddr). Up to 16 nested subroutine. <i>PC = vmAddr and remember the present PC</i>	1
41	RET	Return() Return from subroutine	1
48	WRT	Write(reg) Write R[reg] to DPU frame/HK	1
49	RMOV	Move_To_Reg(reg,[vmAddr]) R[reg]=val32[vmAddr] <i>Copy the value stored at address vmAddr to R[reg]</i>	2
4A	RRMV	Move_To_Reg(reg,[reg1]) R[reg]=val32[R[reg1]] <i>Copy the value stored at address R[reg1] to R[reg]</i>	2
4B	RSTO	Store_From_Reg(reg,[vmAddr]) val32[vmAddr]= R[reg] <i>Copy the value stored in R[reg] at address vmAddr</i>	2
4C	RRST	Store_From_Reg(reg,[reg1]) val32[R[reg1]] = R[reg] <i>Copy the value stored in R[reg] at address R[reg1]</i>	2
50	TER13	Send_TC_ExecPkt_13() ⁶ Send telecommand execution packet 1,3	1
51	TER15	Send_TC_ExecPkt_15(stepNo) ⁶ Send telecommand execution packet 1,5 with stepNo	1
52	TER17	Send_TC_ExecPkt_17() ⁶ Send telecommand execution packet 1,7	1
53	EVNT	Send_Event(Nreg, reg) ⁶ Send event with R[reg] = Event ID R[reg+1] = parameter #1 R[reg+Nreg-1] = parameter #(Nreg-1)	2
55	EVERR	Send_Exception_Event(Nreg, reg) ⁶ Send Exception event with R[reg] = Event ID R[reg+1] = parameter #1 R[reg+Nreg-1] = parameter #(Nreg-1)	2
54	TXTBL	Transmit_Table(VM_Map_Idx) ⁶ Signal to the OBS to transmit the data stored at address VM_Map[VM_Map_Idx] in a TM frame [Auxiliary science Data Report (21,1) APID5 SID=0x020B]. The first word is the TM frame length (including itself) set to zero by OBS when the operation is completed.	1
56	SVEV	Set_Virtuoso_Event(EventNo) Set the Virtuoso OS event No EventNo	1
57	RSVEV	Set_Virtuoso_Event_From_Reg(reg) Set Virtuoso OS event No R[reg]	1
58	VMSTP	Stop_VM(Val) ⁵ Terminate VM number Val (Val=0 is the real time VM)	1
Indirect instructions via VM_Map table			
60	ICALL	Call_Subr(VM_Map_Idx, Offset) ⁷ . Up to 16 nested subroutine. <i>Call subroutine at address specified in VM_Map[VM_MapIdx] plus Offset</i>	2
61	ICPT	Copy_To_ExtMem(VM_Map_Idx, [reg], n) <i>Copy n (<256) words from local address R[reg] to external address specified in VM_Map[VM_Map_Idx] plus offset defined by R[255]</i>	4
62	ICPF	Copy_From_ExtMem(VM_Map_Idx, [reg], n) <i>Copy n (<256) words from external address specified in VM_Map[VM_Map_Idx] plus offset defined by R[255] to local address R[reg]</i>	4
63	IRCALL	Call_Subr([reg], Offset) ⁷ . Up to 16 nested subroutine. <i>Call subroutine at address specified in VM_Map[R[reg]] plus Offset</i>	2

⁶ In the real-time VM, these instructions must be interleaved with at least one critical instruction (i.e. no more than one opcode in the range 50 – 54 can be used before a critical instruction).

⁷ The “Offset” value must be resolved in the current compilation unit (VM table).

Instr. code (hex)	VM asm Mnemonic	Description	Code type
64	IRCPT	Copy_To_ExtMem([reg1], [reg2], n) <i>Copy n (<256) words from local address R[reg2] to external address specified in VM_Map[R[reg1]] plus offset defined by R[255]</i>	4
65	IRCPF	Copy_From_ExtMem([reg1], [reg2], n) <i>Copy n (<256) words from external address specified in VM_Map[R[reg1]] plus offset defined by R[255] to local address R[reg2]</i>	4
80	END	End End current VM program	1
Pseudo instructions			
	INC	Include source file (up to 3 nested INC)	
	EQU	Store at the current address the constant parameter	
	DEF	Set constants	
	ORG	Address of code	
	TABLE	Table(n) ⁸ . The parameter n<128, must be numeric . <i>This instruction forces the compilation unit to be stored in VM_Tbl=n</i>	
	NAME	Name(string) ⁸ . The string parameter (no blank char allowed), is the CLName stored in the VM Table file (RD1).	
	VERSION	Version(string) ⁸ . The string parameter (no blank char allowed), is the CLVersion stored in the VM Table file (RD1)	
	CVSID	CvsId(string) ⁸ . The string parameter (no blank char allowed), is the CLCvSID stored in the VM Table file (RD1).	
	_Label	Label referred by loop/jmp	
Debug instructions			
	COM	COM text string Comment printed during the simulation	
	ROUT	ROUT 0, 4, 72 Print contents of R[0], R[4], R[72] during simulation	
	TRST	Reset to 0 the local elapsed time on the simulator file	

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

The table notation is:

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

Val32 32 bit numeric constant.

Reg VM internal registers index. Numeric constant between 0 and 255 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.

3.7.1 The generic pre-processor

An open source “C/C++ like” pre-processor (GPP by Denis Auroux), can be optionally used. The pre-processor is executed before the compiler (substitutions with #define are executed before the substitution with DEF).

Using GPP, the INC and DEF instructions can be substituted by the “standard” #include and #define. There are two difference in DEF versus #define:

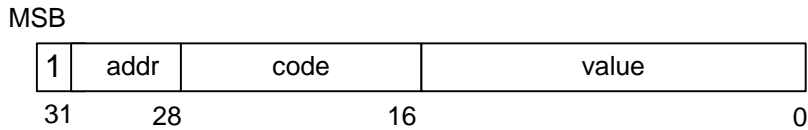
- DEF (as the VM compiler) is case independent, #define is not. This means that it doesn't resolve symbolic names with different character case from the #define
- The symbolic names resolved by DEF are still kept in the compiler and simulator output in order to facilitate the debugging. The symbolic names resolved by #define are substituted in the compiler and in the simulator files.

⁸ Must be in the main source file

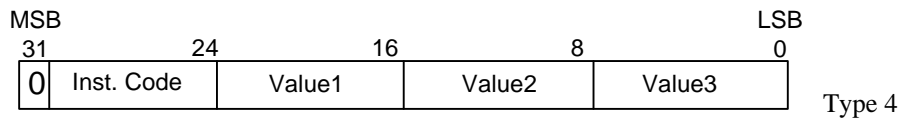
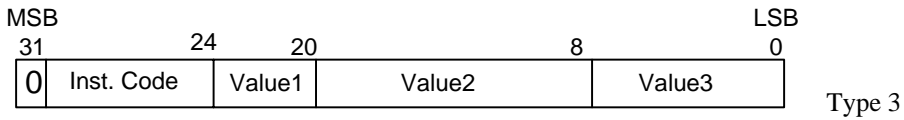
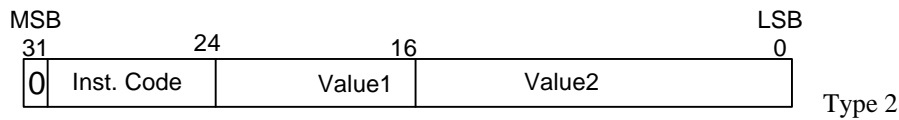
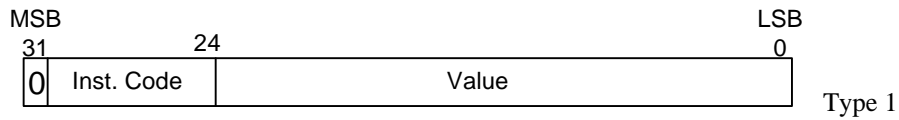
3.8 Instructions Format

The present 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.

4 VM Compiler/Simulator

4.1 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.

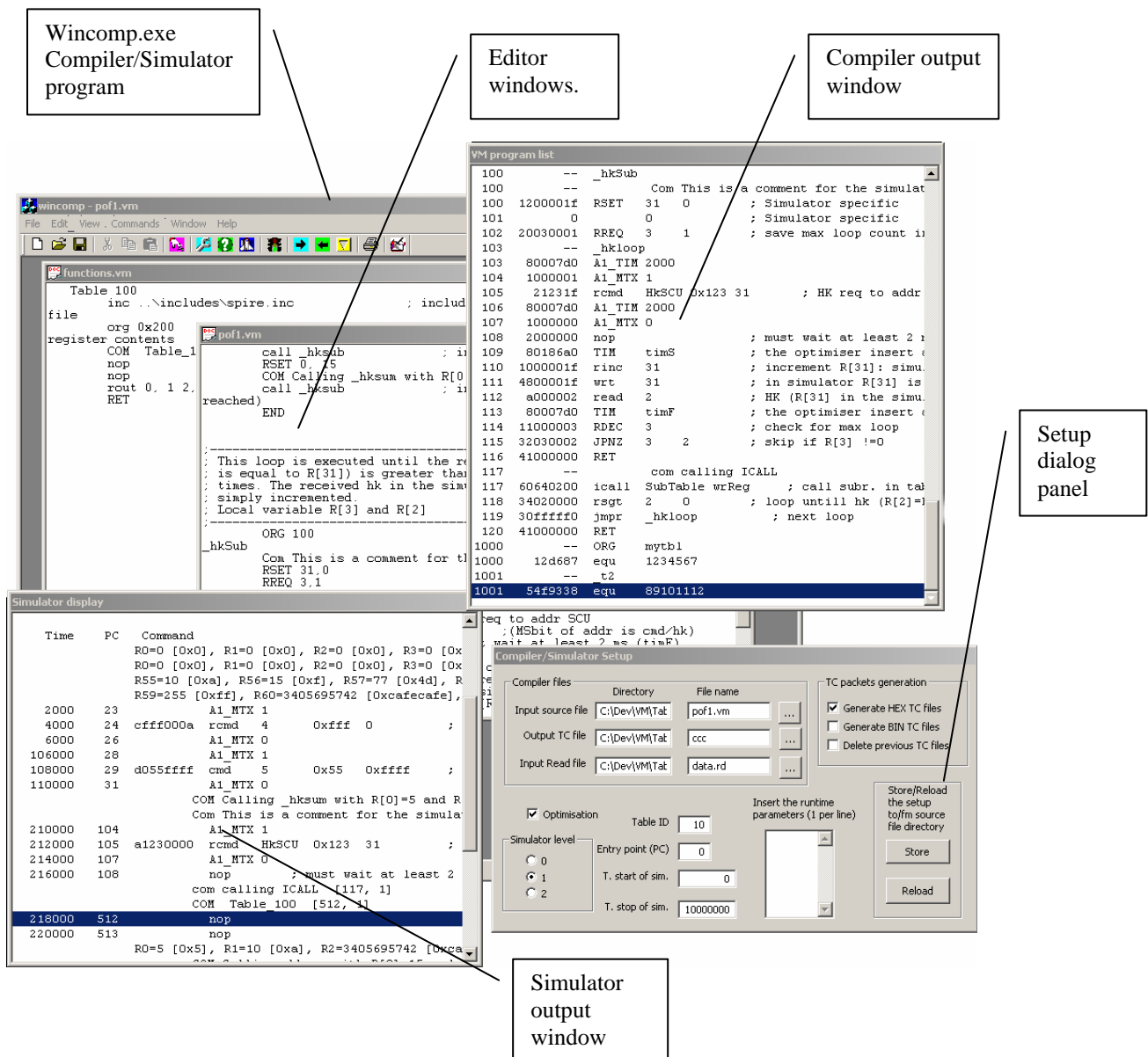
The VM program syntax is:

- Code is case insensitive.
- Hexadecimal constants are prefixed with 0x
- Comments begin with “;” and can appear also after an instruction.
- Labels begin with “_”.

The compiler/simulator program consists of a MDI simple editor, a dialog box used to set the program parameters and two list windows with the compiler and simulator output.

The program should run on every Win98, WinNT, Win2000, WinXp computer.

The figure below shows the compiler/simulator program



The screenshot shows the Wincomp.exe Compiler/Simulator program interface. The main window is titled "wincomp - pof1.vm" and contains several panes:

- Wincomp.exe Compiler/Simulator program:** The main application window.
- Editor windows:** A list of open files including "functions.vm" and "pof1.vm".
- Compiler output window:** A window titled "VM program list" showing the compiled assembly code with addresses and instructions.
- Setup dialog panel:** A "Compiler/Simulator Setup" dialog box with fields for "Input source file", "Output TC file", "Input Read file", "Optimisation", "Table ID", "Simulator level", "Entry point (PC)", "T. start of sim.", "T. stop of sim.", and "Store/Reload" buttons.
- Simulator output window:** A window titled "Simulator display" showing a table of execution time, PC, and commands.

VM program list (Compiler output window):

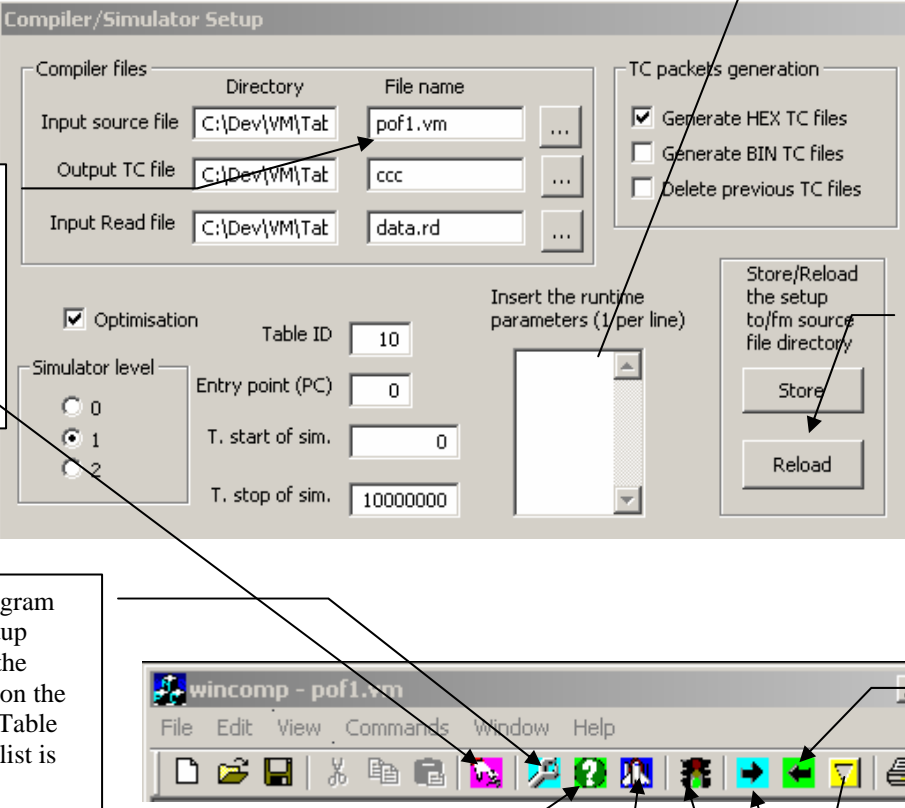
```

100 -- _hkSub
100 -- Com This is a comment for the simul
100 1200001f RSET 31 0 ; Simulator specific
101 0 ; Simulator specific
102 20030001 RREQ 3 1 ; save max loop count in
103 -- _hkloop
103 80007d0 A1_TIM 2000
104 1000001 A1_MTX 1
105 21231f rcmd HKSCU 0x123 31 ; HK req to addr
106 80007d0 A1_TIM 2000
107 1000000 A1_MTX 0
108 2000000 nop ; must wait at least 2
109 80186a0 TIM timS ; the optimiser insert
110 1000001f rinc 31 ; increment R[31]: simul
111 4800001f wrt 31 ; in simulator R[31] is
112 a000002 read 2 ; HK (R[31] in the simul
113 80007d0 TIM timF ; the optimiser insert
114 11000003 RDEC 3 ; check for max loop
115 32030002 JPNZ 3 2 ; skip if R[3] !=0
116 41000000 RET
117 -- com calling ICALL
117 60640200 icall SubTable wrReg ; call subr. in tab
118 34020000 rsgt 2 0 ; loop untill hk (R[2])=
119 30fffff0 jmptr _hkloop ; next loop
120 41000000 RET
1000 -- ORG mytb1
1000 12d687 equ 1234567
1001 -- t2
1001 54f9338 equ 89101112
  
```

Simulator display (Simulator output window):

Time	PC	Command
2000	23	A1_MTX 1
4000	24	cfff000a rcmd 4 0xfff 0 ;
6000	26	A1_MTX 0
106000	28	A1_MTX 1
108000	29	d055ffff cmd 5 0x55 0xffff ;
110000	31	A1_MTX 0
210000	104	COM Calling _hksum with R[0]=5 and R
212000	105	COM This is a comment for the simul
214000	107	A1_MTX 0
216000	108	cmd HKSCU 0x123 31 ;
218000	512	nop
220000	513	nop

The following figure shows/describes few details of the setup panel and toolbar.



Compiler/Simulator Setup

Compiler files

Directory	File name
Input source file: C:\Dev\VM\Tab	pof1.vm
Output TC file: C:\Dev\VM\Tab	ccc
Input Read file: C:\Dev\VM\Tab	data.rd

TC packets generation

- Generate HEX TC files
- Generate BIN TC files
- Delete previous TC files

Store/Reload the setup to/from source file directory

Store
Reload

Simulation Parameters

- Optimisation
- Table ID: 10
- Entry point (PC): 0
- T. start of sim.: 0
- T. stop of sim.: 10000000

Simulator level

- 0
- 1
- 2

wincomp - pof1.vm

File Edit View Commands Window Help

Toolbar Buttons:

- File icon
- Folder icon
- Save icon
- Cut icon
- Copy icon
- Paste icon
- Print icon
- Run icon (pink)
- Stop icon (red)
- Help icon (question mark)
- Clear icon (eraser)
- Generate TC icon (lightning bolt)
- Hide/Show icon (eye)
- Clear compiler icon (trash)
- Clear simulator icon (trash)
- Clear both icon (trash)

Parameters copied to VM registers beginning from R[0]

The input program directory and file name may be also forced to the file in the active editor window, with the pink toolbar button.

The setup in this panel may be saved/restored in the source file directory. Starting the program, the setup stored in the last visited directory is loaded. Ending the program, the actual setup is stored in the current source file directory.

Clear the simulator output window

Compile the program named in the setup panel and store the "binary" output on the specified table (Table ID). The output list is appended in the compiler output window

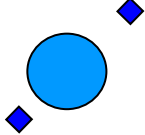
Simulate the program stored in the specified table beginning at offset "Entry point (PC)". The optional "n" run time parameters are loaded in the VM internal registers R[0]... R[n-1]. The output list, is appended in the simulator output window.

Generate the telecommands (TC) packets in the specified directory

Clear the compiler output window

Clear the compiler and simulator output windows. Compile, simulate and generate the TC packets.

Hide/Shows the other windows

 <p>IFSI INAF</p>	<h1>Herschel Space Observatory</h1> <h2>SPIRE-DPU Virtual Machine</h2>	<p>Ref: Issue: 2.5 Date: 15/11/2005 Page: 16 of 37</p>
--	--	--

The input files of the program are:

Filename.vm, ... Filenamex.vm: source program files.

DataFile.rd: optional file with input data to the READ instruction. If the file exists, each READ instruction found during the simulation, read a new number from that file. If the file is not present or contains less numbers than then READ instruction, the contents of register R[254] is used instead.

This is an example of formats allowed in the read data file.

```

;----- READ data file-----
; Comments begin with # or ;
; Number in decimal or "c-hex" format: (0xff)
; More then a number per each line,
; number separator are: space commas
;-----
0xa 15 ; My comment

77, 0xcafecafe,,12
;
0xff ;comment

```

The output files of the program are:

In the same directory as the input source files (specified in the dialog box)

Filename.lst: This file list the compiled program. The file name is the same as the input program filename with extension ".lst".

Filename.sim: This file list the simulator output. The file name is the same as the input program filename with extension ".sim".

In the output directory specified in the dialog box:

outfilnam0.txt, outfilnam .txt outfilnamn.txt: files with the TC packets of the compiled VM program in Hex format

outfilnam0.bin, outfilnam1.bin outfilnamn.bin: files with the TC packets of the compiled VM program in binary format

The compiler optimisation level 1 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

```

IMPORTANT NOTE

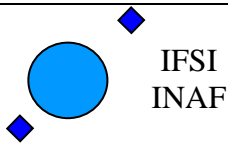
The "Optimisation" fails if the runtime flow of the program is modified (by a jump or a call to a routine), so the "Optimisation" option must be considered obsolete and its use discouraged.

In any case the simulator flag any "unprotected" command.

The TIM-MTX instructions inserted by the optimiser are prefixed by A1_ (TIM-MTX).

Example:

No optimisation	Optimisation level 1
MTX 0	MTX 0
TIM 30000	TIM 30000
...	
	A1_MTX 1
	A1_TIM 2000
	CMD aaa
	A1_MTX 0
CMD aaa	A1_TIM 30000
CMD ccc	A1_MTX 1
TIM 100000	A1_TIM 2000
...	CMD ccc
...	A1_MTX 0
...	A1_TIM 30000
	TIM 100000



4.2 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. Two VM program instructions are interpreted only by the simulator:

Comment instructions:

COM comment string
inserted in the input program, are listed by the simulator as:
COM comment string [addr,n]

Display internal register

ROUT n1 n2 ... nx
Display on the simulator out list the value of register R[n1], R[n2],... R[nx]. The following instruction
rout 0, 1 2,31
encountered at address 126 for the 6th time, generate on the simulator list:
R0=5 [0x5], R1=10 [0xa], R2=0 [0x0], R31=6 [0x6], [126, 6]

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

The simulator output file format is controlled by the run time switch s0-2 (radio button objects on the dialog window):
s0 -> (default value) only command to SS (in hex) are listed with relative time and PC for each CMD RCMD MTX NOP instructions.

Time	RelTime	PC	Command
2000	2000	10	
4000	4000	11	d7000000
6000	6000	12	db000000
8000	8000	15	e4000009
10000	10000	16	e8000009
12000	12000	18	
112000	112000	20	
			T Reset [21, 1]
114000	0	69	fc000003

s1 -> as for s0 but the input text and comment for the above command is also shown. If a WRT instruction is encountered, the content of the addressed register is also shown.

Time	RelTime	PC	Command
2000	2000	10	mtx LOCK ; lock LS I/F
4000	4000	11	d7000000 CMD HR_H SEL_HRB0 ; select
6000	6000	12	db000000 CMD HR_V SEL_HRB0 ; select
8000	8000	15	e4000009 CMD WB_H RST_WB ; reset WBS_H
10000	10000	16	e8000009 CMD WB_V RST_WB ; reset WBS_V
12000	12000	18	mtx unlock
112000	112000	20	mtx LOCK
			T Reset [21, 1]
114000	0	69	fc000003 CMD BR BSTR_WB ; start WBS H&V

s2 -> as for s1 but the input text and comment for all instructions is also shown.

```

Time RelTime      PC      Command
  0         0        8          CALL   _SetVar      ; compute prg parameters
  0         0       71          rmov   hrs_int HIF_T_ACC_HRS
.....
.....
  0         0      118          rrsb   wbs_int wbs_int 10
  0         0      119          RET
  0         0        9          tim    FAST        ; 2 ms between HRS commands
2000      2000     10          mtx    LOCK        ; lock LS I/F
4000      4000     11  d7000000  CMD    HR_H   SEL_HRB0   ; select
6000      6000     12  db000000  CMD    HR_V   SEL_HRB0   ; select
6000      6000     13          jpnz   i_wbs   _c1
8000      8000     15  e4000009  CMD    WB_H   RST_WB    ; reset WBS_H
10000     10000    16  e8000009  CMD    WB_V   RST_WB    ; reset WBS_V
10000     10000    17          tim    100000     ; Wait 100000 ms after WBS
12000     12000    18          mtx    unlock
12000     12000    19          tim    FAST        ; Prepare again for
112000    112000    20          mtx    LOCK
112000     0      21  T Reset [21, 5]
          RSZ   i_wbs      ; No WBS

```

4.3 Packetiser

At the end of the compiler and simulation phase, two group of TC packet files (one file per packet) are generated. The packet file format is: big endian 16 bit words hexadecimal (*.txt) and binary (*.bin), each group of files has the order number included in the name.

Example:

```

vmTC_0.txt
vmTC_1.txt
vmTC_2.txt
vmTC_0.bin
vmTC_1.bin
vmTC_2.bin
vmTbl.hex

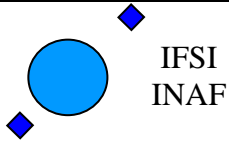
```

The packet structure is as follows:

```

----- PACKET HEADER (48 bits) -----
w16_0=   Packet ID
        Version Number (3)
        Type (1)
        Data field header (1)
        PID (7)
        PCAT (4)
w16_1=   Packet Sequence control
        Sequence flag (2)
        Sequence count (14)
w16_2=   Packet Length = (Number of octets in Packet Data Field) - 1
----- Packet data field -----
w16_3=   Packet Data field header
        PUS (3)
        Checksum type (1)
        ACK (4)
        Pkt Type (8)           = 8
w16_4=   Packet Data field header
        Pkt SubType (8)       = 4
        Pad (8)

```



Herschel Space Observatory SPIRE-DPU Virtual Machine

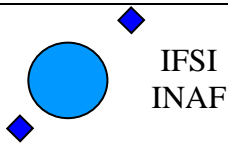
Ref:
Issue: 2.5
Date: 15/11/2005
Page: 20 of 37

w16_5= Application data
.....
w16_n= Application data
w16_n+1=CRC (16) of full packet

===== SPIRE APP DATA =====

w16_5= Function ID (8 MSB)
Activity ID (8 LSB)
w16_6= Table ID
w16_7= Offset from beg of table (8 MSB)
w16_8= N. of 32 bit words data items
w16_9= Data
.....
w16_n = CRC

In the same directory the file **vmTbl.txt** is generated. This hex file contains the VM program code to be included (as initial program) on the OBS at compile time.



4.4 Directory structure

The compiler executable needs always the files:

```
wincomp.exe           // executable
spiresyntax.h        // implemented instructions
```

in the same directory. In the dialog box must be specified the directory, absolute or relative to the compiler program, of the source files and TC packet files. The include files are always in the same directory as the source files, the generated output list and simulator file will be generated in the same directory with the same name of the source file and extension .lst and .sim..

Here follows the directory structure utilized for the compilation of the program in figure in paragraph 4.1 (input files are underlined).

Directory of C:\VM_Comp

```
02/04/2002  12:32      <DIR>      ..
02/04/2002  12:32      <DIR>      .
26/04/2002  12:00      <DIR>      TC_SPkt
26/04/2002  11:44      <DIR>      VM_SProg
23/04/2002  14:28                4,143 spiresyntax.h
26/04/2002  11:43                598,016 wincomp.exe
          2 File(s)
```

Directory of C:\VM_Comp\VM_SProg

```
26/04/2002  11:44      <DIR>      ..
26/04/2002  11:44      <DIR>      .
25/04/2002  13:56                2,714 spire.vm
25/04/2002  13:56                1,580 spire.inc
26/04/2002  12:00                30,202 spire.lst
24/04/2002  13:04                1,418 spire.sim
          4 File(s)
```

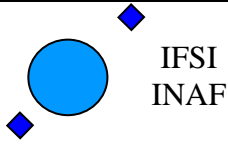
Directory of C:\VM_Comp\TC_SPkt (if it doesn't exists, this directory is automatically generated)

```
02/05/2002  13:02      <DIR>      .
02/05/2002  13:02      <DIR>      ..
02/05/2002  13:02                721 vmTbl.hex
02/05/2002  13:02                109 vmTC_0.txt
02/05/2002  13:02                589 vmTC_1.txt
02/05/2002  13:02                97 vmTC_2.txt
02/05/2002  13:02                36 vmTC_0.bin
02/05/2002  13:02                196 vmTC_1.bin
02/05/2002  13:02                32 vmTC_2.bin
          7 File(s)          1,780 bytes
```

5 Example

As an example of the a VM program, here is the implementation of the “Total Power” measurement in HIFI. The measuring routine is based on the following algorithm:

	Control command	LS command	VM code	comment
1.	Set Mutex			
2.	hrsloop=0			
3.	WBS count =10			this is HIF_N_PERIODS
4.	Label WBS loop			
5.		Reset WBS-H Reset WBS-V	E400-0009 E800-0009	
6.	Timer = ???			HIF_T_DEL_WBS ?
7.		Start WBS H&V	FC00 003	
8.	HRS count =8			this is HIF_R_HRS
9.	Label HRS loop			
10.	hrsbuf = (hrsbuf + 1) mod 2			
11.		Select HRS-H buffer	hrsbuf=0: D700 0000 hrsbuf=1: D710 0000	
12.		Select HRS-V buffer	hrsbuf=0: DB00 0000 hrsbuf=1: DB10 0000	
13.		Start HRS H&V	FF80 0000	
14.	timer = 100 ms			this is HIF_T_ACC_HRS
15.	Reset Mutex			Time for HK-collection
16.	timer = 0.2 ms			
17.	Set Mutex			
18.		Stop HRS H&V	FF90 0000	
19.		Start Transfer HRS-H	D740 0000	
20.		Start Transfer HRS-V	DB40 0000	
21.		Reset readout buffer H	hrsbuf=0: D730 0000 hrsbuf=1: D720 0000	
22.		Reset readout buffer V	hrsbuf=0: DB30 0000 hrsbuf=1: DB20 0000	
23.	Decrement HRS count			
24.	conditional Jump to HRS loop			
25.		Stop WBS H&V	FC00 0005	
26.		Start Transfer WBS-H	E400 0006	
27.		Start Transfer WBS-V	E800 0006	
28.	Decrement WBS count			
29.	conditional Jump to WBS loop			
30.	Reset Mutex			

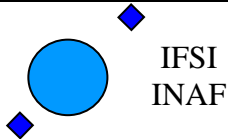


Herschel Space Observatory SPIRE-DPU Virtual Machine

Ref:
Issue: 2.5
Date: 15/11/2005
Page: 23 of 37

Here follows the VM program source of the HIFI total power measurement:

```
-----  
; Case insensitive  
; Comments begin with a ;  
; Labels begin with an _  
-----  
;  
; HIFI - Total Power  
; Ver 1.1  
-----  
                INC hifi.inc          ; include file with constant's definition for HIFI  
  
                DEF lock 1           ; for mutex. Lock the LS I/F  
                DEF unlock 0         ; for mutex. release the LS I/F  
                DEF slow 100000      ; 100 ms timer  
                DEF fast 2000        ; 2 ms timer  
  
                ORG EntryPointTbl     ; begin of programs entry points table  
                EQU 8                ; TotPow begin at 8  
                EQU 512              ; next program  
                EQU 1024             ; next program  
  
                ORG 8                ; address of main program  
                TIM fast             ; timer period at 2 ms  
                MTX lock             ; lock LS I/F  
                RMOV 0 wbCnt         ; R[0]=10. WBS loop counter  
                RSET 2 _c2           ; in R[2] last address of table  
                RREQ 3 2            ; in R[3] last address of table  
  
                CMD wb_h,rst_wb      ; reset WBS_H  
                CMD wb_v,rst_wb      ; reset WBS_V  
_wbLoop CMD     br,bstr_wb          ; start WBS H&V  
                RMOV 1 hrCnt         ; R[1]=8. HRS loop counter  
  
_hrLoop        RINC 3                ; increment R[3]  
                RSGT 3 2             ; Skip next instr if R[3] > R[2]  
                JMPR _intbl          ; Skip next opcode (2 instruction code)  
                RSET 3 _c1           ; R[3]= address of begin of table  
_intbl         RRMV 4 3             ; move value stored at address=R[3] in R[4]  
                RCMD hr_h, 4         ; select HRS_H buffer  
                RCMD hr_v, 4         ; select HRS_V buffer  
                CMD br, bstr_hr      ; start HRS  
                TIM slow             ; wait 100 ms  
                MTX unlock           ; release SL I/F  
                TIM fast             ; timer period at 2 ms  
                MTX lock             ; lock SL I/F  
                CMD br, bstp_hr      ; stop HRS  
                CMD hr_h, stt_hr     ; start transfer HRS_H  
                CMD hr_v, stt_hr     ; start transfer HRS_V  
                RINC 3              ; increment R[3]  
                RRMV 4 3             ; move value stored at address=R[3] in R[4]  
                RCMD hr_h, 4         ; reset HRS_H buffer  
                RCMD hr_v, 4         ; reset HRS_V buffer  
                RDEC 1               ; decrement HRS loop counter  
                JPNZ 1, _hrLoop      ; if R[1]>0 go to _hrLoop  
  
                CMD br, bstp_wb      ; stop WBS  
                CMD wb_h, stt_wb     ; start transfer WBS_H  
                CMD wb_v, stt_wb     ; start transfer WBS_V  
                RDEC 0               ; decrement WBS loop counter  
                JPNZ 0, _wbLoop      ; if R[0]>0 go to _wbLoop  
                MTX unlock           ; release SL I/F  
                END  
  
_c1            EQU sel_hrb0  
                EQU rst_hrb0  
                EQU sel_hrb1  
_c2            EQU rst_hrb1  
  
; ----- Parameters area -----  
; Here I store the program parameters. May be changed by TC.  
; This section can be omitted, the parameters are stored  
; by the OBS on reception of "configure/start measure" TC
```



IFSI
INAF

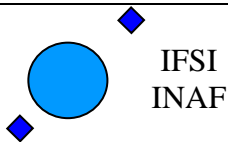
Herschel Space Observatory SPIRE-DPU Virtual Machine

Ref:
Issue: 2.5
Date: 15/11/2005
Page: 24 of 37

ORG wbCnt
EQU 10
ORG hrCnt
EQU 8

and the hifi.inc definitions file

```
-----  
; Include file with constants definitions  
; up to 3 deep nested include files  
-----  
;  
; HIFI definitions  
-----  
  
;----- VM Program area definition -----  
DEF EntryPointTbl 0 ; begin of VM progams entry points table  
DEF ParArea0 4096 ; begin of TotPower parameter area  
DEF wbCnt 4096 ; location of WBS loop for Tot Pow  
DEF hrCnt 4097 ; location of HRS loop for Tot Pow  
  
;----- Subsystems address -----  
DEF LSU, 0  
DEF FCU, 3  
DEF HR_H 5  
DEF HR_V, 6  
DEF WB_H, 9  
DEF WB_V, 0xA  
DEF LCU, 0xC  
DEF BR, 0xF ; Broadcast address  
  
; WBS definition (Val26)  
DEF BSTR_WB 3 ; Broadcast Start WBS H&V  
DEF BSTP_WB 5 ; Broadcast Start WBS H&V  
DEF RST_WB 9 ; reset WBS  
DEF STT_WB 6 ; Start transfer WBS  
  
; HRS definition (Val26)  
DEF BSTR_HR 0x3800000 ; Broadcast start HRS H&V  
DEF BSTP_HR 0x3900000 ; Broadcast stop HRS H&V  
DEF STT_HR 0x3400000 ; Start transfer HRS  
DEF SEL_HRB1 0x3100000 ; HRS select buffer 1  
DEF SEL_HRB0 0x3000000 ; HRS select buffer 0  
DEF SEL_HRB1 0x3100000 ; HRS select buffer 1  
DEF RST_HRB0 0x3300000 ; reset readout buffer 0  
DEF RST_HRB1 0x3200000 ; reset readout buffer 1  
  
; Chopper definition (FCU)  
DEF CHOP_0 0x3105555 ; FCU Chopper pos 0  
DEF CHOP_1 0x310AAAA ; FCU Chopper pos 1  
DEF CHOP_2 0x310AAAA ; FCU Chopper pos 2  
DEF CHOP_3 0x3105555 ; FCU Chopper pos 3
```

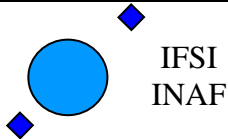
Herschel Space Observatory SPIRE-DPU Virtual Machine

Ref:
Issue: 2.5
Date: 15/11/2005
Page: 25 of 37

Here is the compiler output list file (comments manually tabulated for this document):

VM program file: VM_HProg\hifitotpov.vm
Compilation time: Thu May 02 13:02:29 2002
Optimisation level= 1
Simulation level = 1
Start address (PC)= 8

Addr	opCode	Instruction	
0		INC hifi.inc	; include file with constant's definition for HIFI
0		DEF EntryPointTbl 0	; begin of VM programs entry points table
0		DEF ParArea0 4096	; begin of TotPower parameter area
0		DEF wbCnt 4096	; location of WBS loop for Tot Pow
0		DEF hrCnt 4097	; location of HRS loop for Tot Pow
0		DEF LSU 0	
0		DEF FCU 3	
0		DEF HR_H 5	
0		DEF HR_V 6	
0		DEF WB_H 9	
0		DEF WB_V 0xA	
0		DEF LCU 0xC	
0		DEF BR 0xF	; Broadcast address
0		DEF BSTR_WB 3	; Broadcast Start WBS H&V
0		DEF BSTP_WB 5	; Broadcast Start WBS H&V
0		DEF RST_WB 9	; reset WBS
0		DEF STT_WB 6	; Start transfer WBS
0		DEF BSTR_HR 0x3800000	; Broadcast start HRS H&V
0		DEF BSTP_HR 0x3900000	; Broadcast stop HRS H&V
0		DEF STT_HR 0x3400000	; Start transfer HRS
0		DEF SEL_HRB1 0x3100000	; HRS select buffer 1
0		DEF SEL_HRB0 0x3000000	; HRS select buffer 0
0		DEF SEL_HRB1 0x3100000	; HRS select buffer 1
0		DEF RST_HRB0 0x3300000	; reset readout buffer 0
0		DEF RST_HRB1 0x3200000	; reset readout buffer 1
0		DEF CHOP_0 0x3105555	; FCU Chopper pos 0
0		DEF CHOP_1 0x310AAAA	; FCU Chopper pos 1
0		DEF CHOP_2 0x310AAAA	; FCU Chopper pos 2
0		DEF CHOP_3 0x3105555	; FCU Chopper pos 3
0		DEF lock 1	; for mutex. Lock the LS I/F
0		DEF unlock 0	; for mutex. release the LS I/F
0		DEF slow 100000	; 100 ms timer
0		DEF fast 2000	; 2 ms timer
0		ORG EntryPointTbl	; begin of programs entry points table
0	8	EQU 8	; TotPow begin at 8
1	200	EQU 512	; next program
2	400	EQU 1024	; next program
8		ORG 8	; address of main program
8	80007d0	TIM fast	; timer period at 2 ms
9	1000001	MTX lock	; lock LS I/F
10	49001000	RMOV 0 wbCnt	; R[0]=10. WBS loop counter
11	12000002	RSET 2 _c2	; in R[2] last address of table
12	32	_c2	; in R[2] last address of table
13	20030002	RREQ 3 2	; in R[3] last address of table
14	e4000009	CMD wb_h rst_wb	; reset WBS_H
15	e8000009	CMD wb_v rst_wb	; reset WBS_V
16		_wbLoop	; start WBS H&V
16	fc000003	CMD br bstr_wb	; start WBS H&V
17	49011001	RMOV 1 hrCnt	; R[1]=8. HRS loop counter
18		_hrLoop	; increment R[3]
18	10000003	RINC 3	; increment R[3]
19	34030002	RSGT 3 2	; Skip next instr if R[3] > R[2]
20	30000003	JMPR _intbl	; Skip next opcode (2 instruction code)
21	12000003	RSET 3 _c1	; R[3]= address of begin of table
22	2f	_c1	; R[3]= address of begin of table
23		_intbl	; move value stored at address=R[3] in R[4]
23	4a040003	RRMV 4 3	; move value stored at address=R[3] in R[4]
24	500004	RCMD hr_h 4	; select HRS_H buffer
25	600004	RCMD hr_v 4	; select HRS_V buffer
26	ff800000	CMD br bstr_hr	; start HRS
27	80186a0	TIM slow	; wait 100 ms
28	1000000	MTX unlock	; release SL I/F
29	80007d0	TIM fast	; timer period at 2 ms
30	1000001	MTX lock	; lock SL I/F



Herschel Space Observatory SPIRE-DPU Virtual Machine

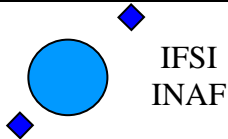
Ref:
Issue: 2.5
Date: 15/11/2005
Page: 26 of 37

```
31 ff900000 CMD br bstp_hr ; stop HRS
32 d7400000 CMD hr_h stt_hr ; start transfer HRS_H
33 db400000 CMD hr_v stt_hr ; start transfer HRS_V
34 10000003 RINC 3 ; increment R[3]
35 4a040003 RRMV 4 3 ; move value stored at address=R[3] in R[4]
36 500004 RCMD hr_h 4 ; reset HRS_H buffer
37 600004 RCMD hr_v 4 ; reset HRS_V buffer
38 11000001 RDEC 1 ; decrement HRS loop counter
39 3201ffeb JPNZ 1 _hrLoop ; if R[1]>0 go to _hrLoop
40 fc000005 CMD br bstp_wb ; stop WBS
41 e4000006 CMD wb_h stt_wb ; start transfer WBS_H
42 e8000006 CMD wb_v stt_wb ; start transfer WBS_V
43 11000000 RDEC 0 ; decrement WBS loop counter
44 3200ffe4 JPNZ 0 _wbLoop ; if R[0]>0 go to _wbLoop
45 10000000 MTX unlock ; release SL I/F
46 50000000 END
47 _c1
47 3000000 EQU sel_hrb0
48 3300000 EQU rst_hrb0
49 3100000 EQU sel_hrb1
50 _c2
50 3200000 EQU rst_hrb1
4096 ORG wbCnt
4096 a EQU 10
4097 ORG hrCnt
4097 8 EQU 8
```

Here is the simulator output list file (comments manually tabulated for this document):

Begin simulation from t1= 0 up to t2= 1000000

Time	PC	Command
2000	9	MTX lock ; lock LS I/F
4000	14	e4000009 CMD wb_h rst_wb ; reset WBS_H
6000	15	e8000009 CMD wb_v rst_wb ; reset WBS_V
8000	16	fc000003 CMD br bstr_wb ; start WBS H&V
10000	24	d7000000 RCMD hr_h 4 ; select HRS_H buffer
12000	25	db000000 RCMD hr_v 4 ; select HRS_V buffer
14000	26	ff800000 CMD br bstr_hr ; start HRS
16000	28	MTX unlock ; release SL I/F
116000	30	MTX lock ; lock SL I/F
118000	31	ff900000 CMD br bstp_hr ; stop HRS
120000	32	d7400000 CMD hr_h stt_hr ; start transfer HRS_H
122000	33	db400000 CMD hr_v stt_hr ; start transfer HRS_V
124000	36	d7300000 RCMD hr_h 4 ; reset HRS_H buffer
126000	37	db300000 RCMD hr_v 4 ; reset HRS_V buffer
128000	24	d7100000 RCMD hr_h 4 ; select HRS_H buffer
130000	25	db100000 RCMD hr_v 4 ; select HRS_V buffer
132000	26	ff800000 CMD br bstr_hr ; start HRS
134000	28	MTX unlock ; release SL I/F
234000	30	MTX lock ; lock SL I/F
236000	31	ff900000 CMD br bstp_hr ; stop HRS
238000	32	d7400000 CMD hr_h stt_hr ; start transfer HRS_H
240000	33	db400000 CMD hr_v stt_hr ; start transfer HRS_V
242000	36	d7200000 RCMD hr_h 4 ; reset HRS_H buffer
244000	37	db200000 RCMD hr_v 4 ; reset HRS_V buffer
246000	24	d7000000 RCMD hr_h 4 ; select HRS_H buffer
248000	25	db000000 RCMD hr_v 4 ; select HRS_V buffer
250000	26	ff800000 CMD br bstr_hr ; start HRS
252000	28	MTX unlock ; release SL I/F
352000	30	MTX lock ; lock SL I/F
354000	31	ff900000 CMD br bstp_hr ; stop HRS
356000	32	d7400000 CMD hr_h stt_hr ; start transfer HRS_H
358000	33	db400000 CMD hr_v stt_hr ; start transfer HRS_V
360000	36	d7300000 RCMD hr_h 4 ; reset HRS_H buffer
362000	37	db300000 RCMD hr_v 4 ; reset HRS_V buffer
364000	24	d7100000 RCMD hr_h 4 ; select HRS_H buffer
366000	25	db100000 RCMD hr_v 4 ; select HRS_V buffer
368000	26	ff800000 CMD br bstr_hr ; start HRS
370000	28	MTX unlock ; release SL I/F
470000	30	MTX lock ; lock SL I/F
472000	31	ff900000 CMD br bstp_hr ; stop HRS
474000	32	d7400000 CMD hr_h stt_hr ; start transfer HRS_H
476000	33	db400000 CMD hr_v stt_hr ; start transfer HRS_V
478000	36	d7200000 RCMD hr_h 4 ; reset HRS_H buffer

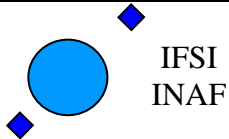


Herschel Space Observatory SPIRE-DPU Virtual Machine

Ref:
Issue: 2.5
Date: 15/11/2005
Page: 27 of 37

```
480000 37 db200000 RCMD hr_v 4 ; reset HRS_V buffer
482000 24 d7000000 RCMD hr_h 4 ; select HRS_H buffer
484000 25 db000000 RCMD hr_v 4 ; select HRS_V buffer
486000 26 ff800000 CMD br bstr_hr ; start HRS
488000 28 MTX unlock ; release SL I/F
588000 30 MTX lock ; lock SL I/F
590000 31 ff900000 CMD br bstp_hr ; stop HRS
592000 32 d7400000 CMD hr_h stt_hr ; start transfer HRS_H
594000 33 db400000 CMD hr_v stt_hr ; start transfer HRS_V
596000 36 d7300000 RCMD hr_h 4 ; reset HRS_H buffer
598000 37 db300000 RCMD hr_v 4 ; reset HRS_V buffer
600000 24 d7100000 RCMD hr_h 4 ; select HRS_H buffer
602000 25 db100000 RCMD hr_v 4 ; select HRS_V buffer
604000 26 ff800000 CMD br bstr_hr ; start HRS
606000 28 MTX unlock ; release SL I/F
706000 30 MTX lock ; lock SL I/F
708000 31 ff900000 CMD br bstp_hr ; stop HRS
710000 32 d7400000 CMD hr_h stt_hr ; start transfer HRS_H
712000 33 db400000 CMD hr_v stt_hr ; start transfer HRS_V
714000 36 d7200000 RCMD hr_h 4 ; reset HRS_H buffer
716000 37 db200000 RCMD hr_v 4 ; reset HRS_V buffer
718000 24 d7000000 RCMD hr_h 4 ; select HRS_H buffer
720000 25 db000000 RCMD hr_v 4 ; select HRS_V buffer
722000 26 ff800000 CMD br bstr_hr ; start HRS
724000 28 MTX unlock ; release SL I/F
824000 30 MTX lock ; lock SL I/F
826000 31 ff900000 CMD br bstp_hr ; stop HRS
828000 32 d7400000 CMD hr_h stt_hr ; start transfer HRS_H
830000 33 db400000 CMD hr_v stt_hr ; start transfer HRS_V
832000 36 d7300000 RCMD hr_h 4 ; reset HRS_H buffer
834000 37 db300000 RCMD hr_v 4 ; reset HRS_V buffer
836000 24 d7100000 RCMD hr_h 4 ; select HRS_H buffer
838000 25 db100000 RCMD hr_v 4 ; select HRS_V buffer
840000 26 ff800000 CMD br bstr_hr ; start HRS
842000 28 MTX unlock ; release SL I/F
942000 30 MTX lock ; lock SL I/F
944000 31 ff900000 CMD br bstp_hr ; stop HRS
946000 32 d7400000 CMD hr_h stt_hr ; start transfer HRS_H
948000 33 db400000 CMD hr_v stt_hr ; start transfer HRS_V
950000 36 d7200000 RCMD hr_h 4 ; reset HRS_H buffer
952000 37 db200000 RCMD hr_v 4 ; reset HRS_V buffer
954000 40 fc000005 CMD br bstp_wb ; stop WBS
956000 41 e4000006 CMD wb_h stt_wb ; start transfer WBS_H
958000 42 e8000006 CMD wb_v stt_wb ; start transfer WBS_V
960000 16 fc000003 CMD br bstr_wb ; start WBS H&V
962000 24 d7000000 RCMD hr_h 4 ; select HRS_H buffer
964000 25 db000000 RCMD hr_v 4 ; select HRS_V buffer
966000 26 ff800000 CMD br bstr_hr ; start HRS
968000 28 MTX unlock ; release SL I/F
1068000 30 MTX lock ; lock SL I/F
```

Simulation: total No of errors: 0
Exceeded max time. Normal end of execution



Herschel Space Observatory SPIRE-DPU Virtual Machine

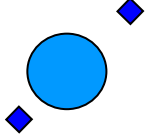
Ref:
Issue: 2.5
Date: 15/11/2005
Page: 28 of 37

Here follows the three TC packets to upload the program.

vmTC_0.txt	vmTC_1.txt		vmTC_2.txt
1c00	1c00	3403	0003
c000	c000	0002	0050
001d	00bd	3000	0004
0008	0008	0003	0060
0400	0400	1200	0004
0510	0510	0003	1100
0000	0000	0000	0001
0000	0000	002f	3201
0000	0000	4a04	ffeb
0303	032b	0003	fc00
0000	0008	0050	0005
0000	0800	0004	e400
0008	07d0	0060	0006
0000	0100	0004	e800
0200	0001	ff80	0006
0000	4900	0000	1100
0400	1000	0801	0000
elfb	1200	86a0	3200
	0002	0100	ffe4
	0000	0000	0100
	0032	0800	0000
	2003	07d0	5000
	0002	0100	0000
	e400	0001	0300
	0009	ff90	0000
	e800	0000	0330
	0009	d740	0000
	fc00	0000	0310
	0003	db40	0000
	4901	0000	0320
	1001	1000	0000
	1000	0003	d06b
	0003	4a04	

The following table shows the VM program code to be stored on the ICU OBS.

vmTbl.txt		
----- block	0x34030002,	0x00600004,
start address 0	0x30000003,	0x11000001,
0x00000008,	0x12000003,	0x3201ffeb,
0x00000200,	0x0000002f,	0xfc000005,
0x00000400,	0x4a040003,	0xe4000006,
----- block	0x00500004,	0xe8000006,
start address 8	0x00600004,	0x11000000,
0x080007d0,	0xff800000,	0x3200ffe4,
0x01000001,	0x080186a0,	0x01000000,
0x49001000,	0x01000000,	0x50000000,
0x12000002,	0x080007d0,	0x03000000,
0x00000032,	0x01000001,	0x03300000,
0x20030002,	0xff900000,	0x03100000,
0xe4000009,	0xd7400000,	0x03200000,
0xe8000009,	0xdb400000,	----- block
0xfc000003,	0x10000003,	start address 4096
0x49011001,	0x4a040003,	0x0000000a,
0x10000003,	0x00500004,	0x00000008,

 <p>IFSI INAF</p>	<p>Herschel Space Observatory SPIRE-DPU Virtual Machine</p>	<p>Ref: Issue: 2.5 Date: 15/11/2005 Page: 29 of 37</p>
--	---	--

6 Appendix

6.1 The Generic PreProcessor

In the following are few sections of the GPP Generic Preprocessor (by Denis Auroux) manual.

The only modifications to the Auroux program are:

- Transported from C to C++ (hopefully without bugs)
- Comments are not stripped out
- Wildcard matching (globbing) is not implemented

The GPP is internally called with parameters: `-o outFile -z -n +c ; \n inFile`

- **GPP 2.22 — Generic Preprocessor**

N.B. — The latest version of GPP and this manual are available from the [GPP home page](#).

- **DESCRIPTION**

GPP is a general-purpose preprocessor with customizable syntax, suitable for a wide range of preprocessing tasks. Its independence from any programming language makes it much more versatile than `cpp`, while its syntax is lighter and more flexible than that of `m4`.

GPP is targeted at all common preprocessing tasks where `cpp` is not suitable and where no very sophisticated features are needed. In order to be able to process equally efficiently text files or source code in a variety of languages, the syntax used by GPP is fully customizable. The handling of comments and strings is especially advanced.

Initially, GPP only understands a minimal set of built-in macros, called *meta-macros*. These meta-macros allow the definition of *user macros* as well as some basic operations forming the core of the preprocessing system, including conditional tests, arithmetic evaluation, wildcard matching (globbing), and syntax specification. All user macro definitions are global—*i.e.*, they remain valid until explicitly removed; meta-macros cannot be redefined. With each user macro definition GPP keeps track of the corresponding syntax specification so that a macro can be safely invoked regardless of any subsequent change in operating mode.

In addition to macros, GPP understands comments and strings, whose syntax and behavior can be widely customized to fit any particular purpose. Internally comments and strings are the same construction, so everything that applies to comments applies to strings as well.

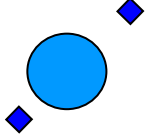
- **EVALUATION RULES**

Input is read sequentially and interpreted according to the rules of the current mode. All input text is first matched against the specified comment/string start sequences of the current mode (except those which are disabled by the 'i' modifier), unless the body being evaluated is the contents of a comment/string whose modifier enables macro evaluation. The most recently defined comment/string specifications are checked for first. Important note: comments may not appear between the name of a macro and its arguments (doing so results in undefined behavior).

Anything that is not a comment/string is then matched against a possible meta-macro call, and if that fails too, against a possible user-macro call. All remaining text undergoes substitution of argument reference sequences by the relevant argument text (empty unless the body being evaluated is the definition of a user macro) and removal of the quote character if there is one.

Note that meta-macro arguments are passed to the meta-macro prior to any evaluation (although the meta-macro may choose to evaluate them, see meta-macro descriptions below). In the case of the `#mode` meta-macro, GPP temporarily adds a comment/string specification to enable recognition of C strings ("...") and prevent any evaluation inside them, so no interference of the characters being put in the C string arguments to `#mode` with the current syntax is to be feared.

On the other hand, the arguments to a user macro are systematically evaluated, and then passed as context parameters to the macro definition body, which gets evaluated with that environment. The only exception is when the macro definition is empty, in which case its arguments are not evaluated. Note that GPP temporarily switches back to the mode in which the macro was defined in order to evaluate it, so it is perfectly safe to change the operating mode between the time a macro is defined and the time when it is called. Conversely, if a user macro wishes to work with

 <p>IFSI INAF</p>	<h1>Herschel Space Observatory</h1> <h2>SPIRE-DPU Virtual Machine</h2>	<p>Ref: Issue: 2.5 Date: 15/11/2005 Page: 30 of 37</p>
--	--	--

the current mode instead of the one that was used to define it it needs to start with a *#mode restore* call and end with a *#mode save* call.

A user macro may be defined with named arguments (see *#define* description below). In that case, when the macro definition is being evaluated, each named parameter causes a temporary virtual user-macro definition to be created; such a macro may be called only without arguments and simply returns the text of the corresponding argument.

Note that, since macros are evaluated when they are called rather than when they are defined, any attempt to call a recursive macro causes undefined behavior except in the very specific case when the macro uses *#undef* to erase itself after finitely many loop iterations.

Finally, a special case occurs when a user macro whose definition does not involve any arguments (neither named arguments nor the argument reference sequence) is called in a mode where the short user-macro end sequence is empty (e.g., *cpp* or *TeX* mode). In that case it is assumed to be an *alias macro*: its arguments are first evaluated in the current mode as usual, but instead of being passed to the macro definition as parameters (which would cause them to be discarded) they are actually appended to the macro definition, using the syntax rules of the mode in which the macro was defined, and the resulting text is evaluated again. It is therefore important to note that, in the case of a macro alias, the arguments actually get evaluated twice in two potentially different modes.

- **META-MACROS**

These macros are always predefined. Their actual calling sequence depends on the current mode; here we use *cpp*-like notation.

- **#define** *x y*

This defines the user macro *x* as *y*. *y* can be any valid GPP input, and may for example refer to other macros. *x* must be an identifier (i.e., a sequence of alphanumeric characters and '_'), unless named arguments are specified. If *x* is already defined, the previous definition is overwritten. If no second argument is given, *x* will be defined as a macro that outputs nothing. Neither *x* nor *y* are evaluated; the macro definition is only evaluated when it is called, not when it is declared.

It is also possible to name the arguments in a macro definition: in that case, the argument *x* should be a user-macro call whose arguments are all identifiers. These identifiers become available as user-macros inside the macro definition; these virtual macros must be called without arguments, and evaluate to the corresponding macro parameter.

- **#defeval** *x y*

This acts in a similar way to *#define*, but the second argument *y* is evaluated immediately. Since user macro definitions are also evaluated each time they are called, this means that the macro *y* will undergo *two* successive evaluations. The usefulness of *#defeval* is considerable as it is the only way to evaluate something more than once, which may be needed to force evaluation of the arguments of a meta-macro that normally doesn't perform any evaluation. However since all argument references evaluated at define-time are understood as the arguments of the body in which the macro is being defined and not as the arguments of the macro itself, usually one has to use the quote character to prevent immediate evaluation of argument references.

- **#undef** *x*

This removes any existing definition of the user macro *x*.

- **#ifdef** *x*

This begins a conditional block. Everything that follows is evaluated only if the identifier *x* is defined, and until either a *#else* or a *#endif* statement is reached. Note, however, that the commented text is still scanned thoroughly, so its syntax must be valid. It is in particular legal to have the *#else* or *#endif* statement ending the conditional block appear only as the result of a user-macro expansion and not explicitly in the input.

- **#ifndef** *x*

This begins a conditional block. Everything that follows is evaluated only if the identifier *x* is not defined.

- **#ifeq** *x y*

This begins a conditional block. Everything that follows is evaluated only if the results of the evaluations of *x* and *y* are identical as character strings. Any leading or trailing whitespace is ignored for the comparison. Note that in *cpp*-mode any unquoted whitespace character is understood as the end of the first argument, so it is necessary to be careful.

- **#ifneq** *x y*

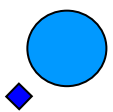
This begins a conditional block. Everything that follows is evaluated only if the results of the evaluations of *x* and *y* are not identical (even up to leading or trailing whitespace).

- **#else**

This toggles the logical value of the current conditional block. What follows is evaluated if and only if the preceding input was commented out.

- **#endif**

This ends a conditional block started by a *#if...* meta-macro.

 <p>IFSI INAF</p>	<h1>Herschel Space Observatory</h1> <h2>SPIRE-DPU Virtual Machine</h2>	<p>Ref: Issue: 2.5 Date: 15/11/2005 Page: 31 of 37</p>
--	--	--

- **#include** file

This causes GPP to open the specified file and evaluate its contents, inserting the resulting text in the current output. All defined user macros are still available in the included file, and reciprocally all macros defined in the included file will be available in everything that follows. The include file is looked for first in the current directory, and then, if not found, in one of the directories specified by the *-I* command-line option (or */usr/include* if no directory was specified). Note that, for compatibility reasons, it is possible to put the file name between "" or <>.

The order in which the various directories are searched for include files is affected by the *-nostdinc*, *-nocurinc* and *-curdirinclast* command-line options.

Upon including a file, GPP immediately saves a copy of the current operating mode onto the mode stack, and restores the operating mode at the end of the included file. The included file may override this behavior by starting with a *#mode restore* call and ending with a *#mode push* call. Additionally, when the *-m* command line option is specified, GPP will automatically switch to the cpp compatibility mode upon including a file whose name ends with either '.c' or '.h'.

- **#exec** command

This causes GPP to execute the specified command line and include its standard output in the current output. Note that, for security reasons, this meta-macro is disabled unless the *-x* command line flag was specified. If use of *#exec* is not allowed, a warning message is printed and the output is left blank. Note that the specified command line is evaluated before being executed, thus allowing the use of macros in the command-line. However, the output of the command is included verbatim and not evaluated. If you need the output to be evaluated, you must use *#defeval* (see above) to cause a double evaluation.

- **#eval** expr

The *#eval* meta-macro attempts to evaluate *expr* first by expanding macros (normal GPP evaluation) and then by performing arithmetic evaluation and/or wildcard matching. The syntax and operator precedence for arithmetic expressions are the same as in C; the only missing operators are <<, >>, ?:, and the assignment operators.

POSIX-style wildcard matching ('globbing') is available only on POSIX implementations and can be invoked with the *=~* operator. In brief, a '?' matches any single character, a '*' matches any string (including the empty string), and '[...]' matches any one of the characters enclosed in brackets. A '[...]' class is complemented when the first character in the brackets is '!'. The characters in a '[...]' class can also be specified as a range using the '-' character—e.g., '[F-N]' is equivalent to '[FGHIJKLMN]'.

If unable to assign a numerical value to the result, the returned text is simply the result of macro expansion without any arithmetic evaluation. The only exceptions to this rule are the comparison operators *==*, *!=*, *<*, *>*, *<=*, and *>=* which, if one of the sides does not evaluate to a number, perform string comparison instead (ignoring trailing and leading spaces). Additionally, the *length(...)* arithmetic operator returns the length in characters of its evaluated argument.

Inside arithmetic expressions, the *defined(...)* special user macro is also available: it takes only one argument, which is not evaluated, and returns 1 if it is the name of a user macro and 0 otherwise.

- **#if** expr

This meta-macro invokes the arithmetic/globbing evaluator in the same manner as *#eval* and compares the result of evaluation with the string "0" in order to begin a conditional block. In particular note that the logical value of *expr* is always true when it cannot be evaluated to a number.

- **#elif** expr

This meta-macro can be used to avoid nested *#if* conditions. *#if ... #elif ... #endif* is equivalent to *#if ... #else #if ... #endif #endif*.

- **#mode** keyword ...

This meta-macro controls GPP's operating mode. See below for a list of *#mode* commands.

- **#line**

This meta-macro evaluates to the line number of the current input file.

- **#file**

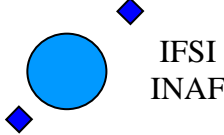
This meta-macro evaluates to the filename of the current input file as it appears on the command line or in the argument to *#include*. If GPP is reading its input from stdin, then *#file* evaluates to 'stdin'.

- **#error** msg

This meta-macro causes an error message with the current filename and line number, and with the text *msg*, to be printed to the standard error device. Subsequent processing is then aborted.

- **#warning** msg

This meta-macro causes a warning message with the current filename and line number, and with the text *msg*, to be printed to the standard error device. Subsequent processing is then resumed.

	<h1>Herschel Space Observatory</h1> <h2>SPIRE-DPU Virtual Machine</h2>	Ref: Issue: 2.5 Date: 15/11/2005 Page: 32 of 37
---	--	--

The key to GPP's flexibility is the `#mode` meta-macro. Its first argument is always one of a list of available keywords (see below); its second argument is always a sequence of words separated by whitespace. Apart from possibly the first of them, each of these words is always a delimiter or syntax specifier, and should be provided as a C string delimited by double quotes (" "). The various special matching sequences listed in the section on syntax specification are available. Any `#mode` command is parsed in a mode where `"..."` is understood to be a C-style string, so it is safe to put any character inside these strings. Also note that the first argument of `#mode` (the keyword) is never evaluated, while the second argument is evaluated (except of course for the contents of C strings), so that the syntax specification may be obtained as the result of a macro evaluation.

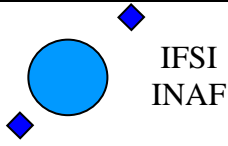
The available `#mode` commands are:

- **#mode save / #mode push**
Push the current mode specification onto the mode stack.
- **#mode restore / #mode pop**
Pop mode specification from the mode stack.
- **#mode standard name**
Select one of the standard modes. The only argument must be one of: default (default mode); cpp, C (cpp mode); tex, TeX (tex mode); html, HTML (html mode); xhtml, XHTML (xhtml mode); prolog, Prolog (prolog mode). The mode name must be given directly, not as a C string.
- **#mode user "s1" ... "s9"**
Specify user macro syntax. The 9 arguments, all of them C strings, are the mode specification for user macros (see the `-U` command-line option and the section on syntax specification). The meta-macro specification is not affected.
- **#mode meta {user | "s1" ... "s7"}**
Specify meta-macro syntax. Either the only argument is *user* (not as a string), and the user-macro mode specifications are copied into the meta-macro mode specifications, or there must be seven string arguments, whose significance is the same as for the `-M` command-line option (see section on syntax specification).
- **#mode quote ["c"]**
With no argument or `""` as argument, removes the quote character specification and disables the quoting functionality. With one string argument, the first character of the string is taken to be the new quote character. The quote character can be neither alphanumeric nor `'_'`, nor can it be one of the special matching sequences.
- **#mode comment [xxx] "start" "end" ["c" ["c"]]**
Add a comment specification. Optionally a first argument consisting of three characters not enclosed in `" "` can be used to specify a comment/string modifier (see the section on syntax specification). The default modifier is *ccc*. The first two string arguments are used as comment start and end sequences respectively. The third string argument is optional and can be used to specify a string-quote character. (If it is `""`, the functionality is disabled.) The fourth string argument is optional and can be used to specify a string delimitation warning character. (If it is `""`, the functionality is disabled.)
- **#mode string [xxx] "start" "end" ["c" ["c"]]**
Add a string specification. Identical to `#mode comment` except that the default modifier is *sss*.
- **#mode nocomment / #mode nostring ["start"]**
With no argument, remove all comment/string specifications. With one string argument, delete the comment/string specification whose start sequence is the argument.
- **#mode preservelf { on | off | 1 | 0 }**
Equivalent to the `-n` command-line switch. If the argument is *on* or *1*, any newline or whitespace character terminating a macro call or a comment/string is left in the input stream for further processing. If the argument is *off* or *0* this feature is disabled.
- **#mode charset { id | op | par } "string"**
Specify the character sets to be used for matching the `\o`, `\O` and `\i` special sequences. The first argument must be one of *id* (the set matched by `\i`), *op* (the set matched by `\o`) or *par* (the set matched by `\O` in addition to the one matched by `\o`). *"string"* is a C string which lists all characters to put in the set. It may contain only the special matching sequences `\a`, `\A`, `\b`, `\B`, and `\#` (the other sequences and the negated sequences are not allowed). When a `'-'` is found inbetween two non-special characters this adds all characters inbetween (e.g. `"A-Z"` corresponds to all uppercase characters). To have `'-'` in the matched set, either put it in first or last position or place it next to a `\x` sequence.

• EXAMPLES

Here is a basic self-explanatory example in standard or cpp mode:

```
#define FOO This is
#define BAR a message.
#define concat #1 #2
```

```
concat(FOO,BAR)
#ifeq (concat(foo,bar)) (foo bar)
This is output.
#else
This is not output.
#endif
```

Using argument naming, the *concat* macro could alternatively be defined as

```
#define concat(x,y) x y
```

In TeX mode and using argument naming, the same example becomes:

```
\define{FOO}{This is}
\define{BAR}{a message.}
\define{\concat{x}{y}}{\x \y}
\concat{\FOO}{\BAR}
\ifeq{\concat{foo}{bar}}{foo bar}
This is output.
\else
This is not output.
\endif
```

In HTML mode and without argument naming, one gets similarly:

```
<#define FOO|This is>
<#define BAR|a message.>
<#define concat|#1 #2>
<#concat <#FOO>|<#BAR>>
<#ifeq <#concat foo|bar>|foo bar>
This is output.
<#else>
This is not output.
<#endif>
```

The following example (in standard mode) illustrates the use of the quote character:

```
#define FOO This is \
  a multiline definition.
#define BLAH(x) My argument is x
BLAH(urf)
\BLAH(urf)
```

Note that the multiline definition is also valid in *cpp* and *Prolog* modes despite the absence of quote character, because `\` followed by a newline is then interpreted as a comment and discarded.

In *cpp* mode, C strings and comments are understood as such, as illustrated by the following example:

```
#define BLAH foo
BLAH "BLAH" /* BLAH */
'It's a /*string*/ !'
```

The main difference between *Prolog* mode and *cpp* mode is the handling of strings and comments: in *Prolog*, a `'...'` string may not begin immediately after a digit, and a `/*...*/` comment may not begin immediately after an operator character. Furthermore, comments are not removed from the output unless they occur in a `#command`.

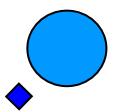
The differences between *cpp* mode and default mode are deeper: in default mode `#commands` may start anywhere, while in *cpp* mode they must be at the beginning of a line; the default mode has no knowledge of comments and strings, but has a quote character (`\`), while *cpp* mode has extensive comment/string specifications but no quote character. Moreover, the arguments to meta-macros need to be correctly parenthesized in default mode, while no such checking is performed in *cpp* mode.

This makes it easier to nest meta-macro calls in default mode than in *cpp* mode. For example, consider the following HTML mode input, which tests for the availability of the `#exec` command:

```
<#ifeq <#exec echo blah>|blah
> #exec allowed <#else> #exec not allowed <#endif>
```

There is no *cpp* mode equivalent, while in default mode it can be easily translated as

```
#ifeq (#exec echo blah
) (blah
)
\#exec allowed
#else
\#exec not allowed
```

 <p>IFSI INAF</p>	<h2>Herschel Space Observatory</h2> <h3>SPIRE-DPU Virtual Machine</h3>	<p>Ref: Issue: 2.5 Date: 15/11/2005 Page: 34 of 37</p>
--	--	--

```
#endif
```

In order to nest meta-macro calls in cpp mode it is necessary to modify the mode description, either by changing the meta-macro call syntax, or more elegantly by defining a silent string and using the fact that the context at the beginning of an evaluated string is a newline character:

```
#mode string QQQ "$" "$"
#ifeq $#exec echo blah
$ $blah
$
\#exec allowed
#else
\#exec not allowed
#endif
```

Note, however, that comments/strings cannot be nested ("..." inside \$...\$ would go undetected), so one needs to be careful about what to include inside such a silent evaluated string. In this example, the loose meta-macro nesting introduced in version 2.1 makes it possible to use the following simpler version:

```
#ifeq blah #exec echo -n blah
\#exec allowed
#else
\#exec not allowed
#endif
```

Remember that macros without arguments are actually understood to be aliases when they are called with arguments, as illustrated by the following example (default or cpp mode):

```
#define DUP(x) x x
#define FOO and I said: DUP
FOO(blah)
```

The usefulness of the *#defeval* meta-macro is shown by the following example in HTML mode:

```
<#define APPLY|<#defeval TEMP|<##1 \#1>><#TEMP #2>>
<#define <#foo x>|<#x> and <#x>>
<#APPLY foo|BLAH>
```

The reason why *#defeval* is needed is that, since everything is evaluated in a single pass, the input that will result in the desired macro call needs to be generated by a first evaluation of the arguments passed to APPLY before being evaluated a second time.

To translate this example in default mode, one needs to resort to parenthesizing in order to nest the *#defeval* call inside the definition of APPLY, but need to do so without outputting the parentheses. The easiest solution is

```
#define BALANCE(x) x
#define APPLY(f,v) BALANCE(#defeval TEMP f
TEMP(v))
#define foo(x) x and x
APPLY(\foo,BLAH)
```

As explained above the simplest version in cpp mode relies on defining a silent evaluated string to play the role of the BALANCE macro.

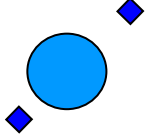
The following example (default or cpp mode) demonstrates arithmetic evaluation:

```
#define x 4
The answer is:
#eval x*x + 2*(16-x) + 1998%x
```

```
#if defined(x)&&!(3*x+5>17)
This should be output.
#endif
```

To finish, here are some examples involving mode switching. The following example is self-explanatory (starting in default mode):

```
#mode push
#define f(x) x x
#mode standard tex
\f{blah}
\mode{string}{"$" "$"}
\mode{comment}{"/*" "*/"}
${f{urf}}$/* blah */
\define{FOO}{bar/* and some more */}
```

 <p>IFSI INAF</p>	<h1>Herschel Space Observatory</h1> <h2>SPIRE-DPU Virtual Machine</h2>	<p>Ref: Issue: 2.5 Date: 15/11/2005 Page: 35 of 37</p>
--	--	--

```
\mode{pop}
f($FOO$)
```

A good example where a user-defined mode becomes useful is the GPP source of this document (available with GPP's source code distribution).

Another interesting application is selectively forcing evaluation of macros in C strings when in cpp mode. For example, consider the following input:

```
#define blah(x) "and he said: x"
blah(foo)
```

Obviously one would want the parameter *x* to be expanded inside the string. There are several ways around this problem:

```
#mode push
#mode nostring ""
#define blah(x) "and he said: x"
#mode pop
```

```
#mode quote ""
#define blah(x) `and he said: x`
```

```
#mode string QQQ "$$" "$$"
#define blah(x) $$"and he said: x"$$
```

The first method is very natural, but has the inconvenience of being lengthy and neutralizing string semantics, so that having an unevaluated instance of 'x' in the string, or an occurrence of '/*', would be impossible without resorting to further contortions.

The second method is slightly more efficient because the local presence of a quote character makes it easier to control what is evaluated and what isn't, but has the drawback that it is sometimes impossible to find a reasonable quote character without having to either significantly alter the source file or enclose it inside a *#mode push/pop* construct. For example, any occurrence of '/*' in the string would have to be quoted.

The last method demonstrates the efficiency of evaluated strings in the context of selective evaluation: since comments/strings cannot be nested, any occurrence of "" or '/*' inside the '\$\$' gets output as plain text, as expected inside a string, and only macro evaluation is enabled. Also note that there is much more freedom in the choice of a string delimiter than in the choice of a quote character.

Starting with version 2.1, meta-macro calls can be nested more efficiently in default, cpp and Prolog modes. This makes it easy to make a user version of a meta-macro, or to increment a counter:

```
#define myeval #eval #1
```

```
#define x 1
#defeval x #eval x+1
```

• ADVANCED EXAMPLES

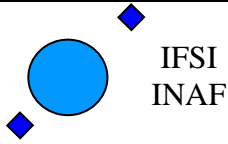
Here are some examples of advanced constructions using GPP. They tend to be pretty awkward and should be considered as evidence of GPP's limitations.

The first example is a recursive macro. The main problem is that (since GPP evaluates everything) a recursive macro must be very careful about the way in which recursion is terminated in order to avoid undefined behavior (most of the time GPP will simply crash). In particular, relying on a *#if/#else/#endif* construct to end recursion is not possible and results in an infinite loop, because GPP scans user macro calls even in the unevaluated branch of the conditional block. A safe way to proceed is for example as follows (we give the example in TeX mode):

```
\define{countdown}{
  \if{#1}
  #1...
  \define{loop}{\countdown}
  \else
  Done.
  \define{loop}{}
  \endif
  \loop{\eval{#1-1}}
}
\countdown{10}
```

Another example, in cpp mode:

```
#mode string QQQ "$" "$"
```



```
#define triangle(x,y) y \  
  $#if length(y)<x$ $#define iter triangle$ $#else$ \  
  $#define iter$ $#endif \  
$ iter(x,*y) \  
triangle(20)
```

The following is an (unfortunately very weak) attempt at implementing functional abstraction in GPP (in standard mode). Understanding this example and why it can't be made much simpler is an exercise left to the curious reader.

```
#mode string "" "" "" \  
#define ASIS(x) x \  
#define SILENT(x) ASIS() \  
#define EVAL(x,f,v) SILENT( \  
  #mode string QQQ "" "" "" \  
  #defeval TEMP0 x \  
  #defeval TEMP1 ( \  
    \#define \TEMP2(TEMP0) f \  
  ) \  
  TEMP1 \  
  )TEMP2(v) \  
#define LAMBDA(x,f,v) SILENT( \  
  #ifneq (v) () \  
  #define TEMP3(a,b,c) EVAL(a,b,c) \  
  #else \  
  #define TEMP3(a,b,c) \LAMBDA(a,b) \  
  #endif \  
  )TEMP3(x,f,v) \  
#define EVALLAMBDA(x,y) SILENT( \  
  #defeval TEMP4 x \  
  #defeval TEMP5 y \  
  ) \  
#define APPLY(f,v) SILENT( \  
  #defeval TEMP6 ASIS(\EVA)f \  
  TEMP6 \  
  )EVAL(TEMP4,TEMP5,v)
```

This yields the following results:

```
LAMBDA(z,z+z) \  
=> LAMBDA(z,z+z)
```

```
LAMBDA(z,z+z,2) \  
=> 2+2
```

```
#define f LAMBDA(y,y*y) \  
f \  
=> LAMBDA(y,y*y)
```

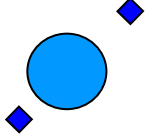
```
APPLY(f,blah) \  
=> blah*blah
```

```
APPLY(LAMBDA(t,t t),(t t) \  
=> (t t) (t t)
```

```
LAMBDA(x,APPLY(f,(x+x)),urf) \  
=> (urf+urf)*(urf+urf)
```

```
APPLY(APPLY(LAMBDA(x,LAMBDA(y,x*y)),foo),bar) \  
=> foo*bar
```

```
#define test LAMBDA(y,`#ifeq y urf \  
y is urf#else \  
y is not urf#endif
```

 <p>IFSI INAF</p>	<h1>Herschel Space Observatory SPIRE-DPU Virtual Machine</h1>	<p>Ref: Issue: 2.5 Date: 15/11/2005 Page: 37 of 37</p>
--	---	--

)
APPLY(test,urf)
=> urf is urf

APPLY(test,foo)
=> foo is not urf

- **AUTHOR**

GPP was written by Denis Auroux <auroux@math.mit.edu>. Since version 2.12 it has been maintained by Tristan Miller <psychonaut@nothingisreal.com>.

- **COPYRIGHT**

Copyright © 1996–2001 Denis Auroux.

Copyright © 2003, 2004 Tristan Miller.

Permission is granted to anyone to make or distribute verbatim copies of this document as received, in any medium, provided that the copyright notice and this permission notice are preserved, thus giving the recipient permission to redistribute in turn.

Permission is granted to distribute modified versions of this document, or of portions of it, under the above conditions, provided also that they carry prominent notices stating who last changed them.