Virtual Master 3.2 script writing manual.

By Sven B.

This manual assume you are familiar with Virtual Master. If you are not familiar with Virtual Master, you should read the document **User guide.pdf**. That is a short document explaining the concept of Virtual master and with an introduction guide through the program.

You can find a chapter about news in this release at the bottom of this document.

Table of contents

Create your own rules.

The program is shipped with a sample script file called **demo.ini**. Make a copy of demo.ini and edit it to to create your own rules. You can call the new file anything you want, except it must end with .ini. The file can be edited by a simple text editor like Notepad. Usually you can double click on the file to start Notepad. **Warning:** If you use a text processing system like Word to edit the script, make sure that you save it as a text file. If you save it as a Word file, you will not be able to use it in the program.

For practical reasons the sub is most of the time called "he" in this manual. Most of my examples are for a female slave, so sometimes I write “she”. In reality, the sub can be a man or a woman. The program doesn't care.

While you read this manual, I suggest that you look in the demo script at the same time, to get some examples on how to use the keywords described in this manual.

Script syntax:

The script is divided in sections. Most of the section defines an element the sub will see, like a permission, a report or a punishment.

A few of the sections describe options common to the whole script. These are the General, Init and Events sections.

Each section starts with a name in [ ]. Each section has it own keywords.

Each keyword must be on a separate line. You can start the line with as many blanks as you wish before the keyword.

Here is a description of the syntax of script.

Basic elements

This section describes the basic elements, the status, the reports, the permissions, the jobs, the punishments etc.

It also describes the general stuff that must always be present in a script, and what happens the first time the sub starts the script.

General section, must always be present.

The general section defines general preferences. **It must be present in the script**. These are the most important keywords of the [general] section. There are other keywords, that will be described later.

What must be there

[general]	This line starts the general section.
Master=*Masters name*	Give your virtual Master a name. The name shows on the title bar.
SubName=*sub's name*	Name the sub. If present, the *sub's name* is used in some of the messages. You can have more than one SubName line. If you have, the program will choose a random name.
MinVersion=3.1	Tells which version of the program the script is made for. If you try to run the script on an older version of the program, you will get an error message. Write the version number as you can read it in the About box under the help menu. MinVersion must be present in the script.

Optional keywords

Version=*xxx*	Use this to write in the report which script version is used. Replace *xxx* by any identification of the script you want. It can be a date, but it can also be anything else. If you use an encrypted script, the program will use the encryption date and time as the script version and ignore this keyword.

What happens first time the sub opens the script.

The init section defines what happens first time a new script is used. The Events section have the keyword FirstRun, where you can specify a procedure to run first time a script is used. The FirstRun procedure is called after the Init section is executed and is intented to supplement the Init section. Read more about procedures later.

[init]	This line starts the init section.
NewStatus=*statusname*	This defines the first status to enter
Merits=*startmerits*	The sub will start with *startmerits* merit points. If you do not specifi this, the sub will start with half of MaxMerits.
[events]	This line starts the event section.

FirstRun=*procedure*	Activates *procedure* the very first time a new script is used. See more about procedures later.

Read more about the Events section in the chapter Events.

Status

Status is an important concept of the program. The status should reflect what the sub is currently doing, so VM can react accordingly to that.

What must be there

[status-*name*]	This line starts a status definition. Replace *name* with the name of the status.

Use status

You can use status to decide when something is possible. You do this by adding the keyword PreStatus to a permission, report, confession, procedure, timer, pop up or set.

Prestatus=*status*	Means that sub must be in status *status* for the permission, report etc. to be possible. You can have more than one Prestatus line. Or you can specify several status on the same line by separating them with commas.

When prestatus is used in a report, the report will only be visible in the communication menu when the sub is in the specified status.
When prestatus is used in a confession, the confession will only be visible in the communication menu when the sub is in the specified status.
When prestatus is used in a permission, the permission will only be visible in the communication menu when the sub is in the specified status.
When prestatus is used in a procedure, the procedure will only be executed when the sub is in the specified status.
When prestatus is used in a timer, the timer will only be executed when the sub is in the specified status.
When prestatus is used in a pop up, the pop up will only pop when the sub is in the specified status.
When prestatus is used in a set, the set will only be used when the sub is in the specified status.

ReportsOnly=1	If you add this keyword to a status, all the sub can do is make a report. It will not be possible to ask permission, confess, ask for instructions, report clothing or look at assignments. This is usefull for status where the sub is only

	supposed to to one thing, like going to the toilet or standing in the corner.
Assignments=0	If you add this keyword to a status, the sub will not be able to access the Assignments menu.
Permissions=0	If you add this keyword to a status, the sub will not be able to ask permissions.
Confessions=0	If you add this keyword to a status, the sub will not be able to confess.
Reports=0	If you add this keyword to a status, the sub will not be able to do reports.

Changing status

You can change status in permissions, reports, confessions, procedures, timers or pop ups.

NewStatus=*newstatus*	Status is set to *newstatus*.

If you use NewStatus in jobs and punishments, the program will change to *newstatus* when the job or punishment starts and will automatic change back when it is marked done. You can not use NewSubStatus in jobs and punishments.

Substatus

Sometimes you want to switch to another status for a short period and the return to the original status. One example is when the sub is allowed to go to the toilet. You then want the status to reflect that the sub is in the toilet, and afterwards you want to return to the original status. You can do this in two ways. Either use NewSubStatus instead of NewStatus when switching status, or define the status you temporarily switch to as a substatus. A substatus is a status from which you can return to the previous status. A primary status is a status that is not a substatus.

If you go from a substatus to a primary status using NewStatus=*statusname*, you loose the substatus and will no longer be able to use NewStatus=&LastStatus.

NewSubStatus

You can use NewSubStatus anywhere you can use NewStatus, except in the [init] section.

NewSubStatus=*newstatus*	Status is set to *newstatus*.

Define the status as a substatus

In the status definition add:

SubStatus=1	Means that this status is a substatus, and that you can use &laststatus to return.

Return to the original status

When you want to return to the original status, add this to a permission, report, confession, procedure, timer or pop up:

NewStatus=&LastStatus	Returns to the previous status if you are in a substatus.

If you use NewStatus=&LastStatus when there is no saved LastStatus, then the program can not return to a previous status. There are two ways of handling this. If you have set a default status (see below), the program will go to that status. If not, the program will stay in the current status. In both cases an error line is placed in the report. The sub will not receive an error.

DefaultStatus=*statusname*	Defines the status that the program should return to when no last status exists. This keyword must be put in the [General] section.

Automatic substatus:

Another and more simple way to define a substatus is to use the EndReport keyword. EndReport is useful in simple situations like going to the toilet.

EndReport=*reporttext*	If you use EndReport in combination with NewStatus, an automatic report is created with the name *reporttext* and the new status is automatic used as a substatus. When you use EndReport, you don't even have to define the status, unless you want to use other of the status related keywords. Further more, when you use EndReport, the sub will have no other choices than to use this report to end the status.

The generated report will automatic be used as a quick report. This means that the sub can report it by clicking a button in the main window or by pressing the F9 key.

EndText is an obsolete name for EndReport and works the same way.

Reports

Reports are the reports the sub has to make. This is the way the sub tells the program that something has happened, that he is going to do something that he don't need to ask permission for or that he has finished doing something. Reports are often, but not always, used to change to a new status.

What must be there

[report-*name*]	This line starts a report definition. Replace *name* with the name of the report.

Making relevant reports appear at the top of the report list

Usually reports are shown in the order they are defined. But if the sub is doing something that he must report finished, you usually want that report to be on the top of the report list. When he is doing something else, you don't want the ”finshed” report to show up at all.

OnTop=1	If you add OnTop=1 to a report definition, then the report will be on top of the list when the sub wants to make a report.

Confessions

Specifies the confessions, the sub has to make. Confessions are a lot like reports. The only difference is that they have a separate entry in the Communications menu.

[confession-*name*]	This line starts a confession definition. Replace *name* with the name of the confession.

You can use the same keywords for confessions as for reports.

Automatic confessions

The program generates two special confessions: "Forgot to ask permission" and "Did something though permission was denied". When confessing one of theese, the sub is asked what he has to say for himself. Then he may be punished. You can specify penalties for theese confessions with the keywords ForgetPenalty and IgnorePenalty. These keywords are used in a permission definition to specify a special penalty for forgetting/ignoring this permsssion or in the General section to specify a general penalty for all permissions, except those who have an individual specification.

ForgetPenalty=*points*	If the sub confess that he has forgotten to ask permission, he will be punished with a severity of *points*.
IgnorePenalty=*points*	If the sub confess that he has done something, although he was denied permission,he will be punished with a severity of *points*.

If for some reason you don't want automatic confessions, you can disable them by using these keywords in the [General] section.

ForgetConfession=0	There will be no automatic confessions for forgetting to ask permission.
IgnoreConfession=0	There will be no automatic confessions for doing something that was denied.

If you want control when an automatic confession is made, you can use the events ForgetConfession and IgnoreConfession. See Events.

If you want control when an automatic confession is made for a specific permission, you can use these keywords in the permission. If you use both ForgetProcedure / IgnoreProcedure and events, the ForgetProcedure / IgnoreProcedure is called first and then the event.

ForgetProcedure= *procedure*	If the sub confess that he has forgotten to ask permission, *procedure* is called.
IgnoreProcedure= *procedure*	If the sub confess that he has done something, although he was denied permission, *procedure* is called.

Permissions

Permissions define what the sub has to ask permission for and sets rules for when permission is given.

What must be there

[permission-*name*]	This line starts a permission definition. Replace *name* with the name of the permission.
Pct=*nn*	Gives the sub *nn* % of chance for an immediate permission.

What about the next time the sub asks?

If the program refuses permission, you don't want the sub to be able to just ask again, and then maybe get permission. If permission is granted, you may not want the program to grant permission once again shortly after. If the sub is refused permission to use the toilet, you want to make sure he isn't goint to wait forever. This chapter describes how to control these intervals.

MinInterval=*hh:mm,hh:mm*	Mininterval defines the minimum interval that must pass after the permission is granted before it can be granted again.. Thus mininterval=24:00 insures that this permission is never granted twice within 24 hours. If you specify two values separated by comma, the program will use a random value in that range.
Delay=*hh:mm,hh:mm*	If permission is not granted, delay defines the minimum amount of time to pass before permission can be granted. If you specify two values separated by comma, the program will use a random value in that range. <u>Note:</u> If the sub asks permission again before the calculated time arrives, the program calculates a new time, and if it is later than the old time the new time will be used.
MaxWait=*hh:mm*	Specifies the maximum time the sub can wait. If the sub keeps asking, the delay will continue to extent the time the sub has to wait, but never past the time in the MaxWait parameter (calculated from the first time the sub is denied the permission). Useful for things that can't wait forever, such as going to the toilet.

Telling the sub when waiting time is over

Usually when a permission is denied, you can simply leave it to the sub to guess when permission can be asked again. But in some cases, you may want to tell the sub when waiting time (defined by the Delay parameter) is over.

Notify=1	The program will inform the sub when the delay time has passed.
Notify=0	The program will not inform the sub when the delay time has passed. The sub has to guess when to ask again.

Not specifying Notify is the same as Notify=0.

Punishments.

Punishing the sub

To punish the sub, you can add the following keywords to a report, confession, timer, pop up or procedure. You can also use them in a permission, then the sub is punished when the permission is given.

Punish=*points*	The sub will be punished with a severity of *points*. *Points* can be a value or the name of a counter. One severity point is equal to one merit point. You can give two values separated by comma, then the program will chose a random value between the two.
PunishMessage=*text*	Use PunishMessage to specify the text that is presented to the sub. Ignored if Punish is not used. If you do not code PunishMessage, the system will generate one.
PunishmentGroup=*group,group,*	If a punishment is given, it will be chosen from one of the groups specified. If you do not code PunishmentGroup, the system will pick a random punishment. Note that in combination with the RemindPenalty keyword, you should use RemindPenaltyGroup in stead of PunishmentGroup.

Specifying the punishments the sub can receive

Defines the possible punishments the program can give.

What must be there

[punishment-*name*]	This line starts a punishment definition. Replace *name* with the description of the punishment.

Repetitive punishments

Often you want a punishment to consist of a number of repetitions, like "6 beats with the cane". Use # in the name where you want the program to put in the number of repetitions. Example: "[punishment-# beats with the cane]".

Punishments measured in time

Sometimes you want to measure a punishment in time. You can do this in minutes, hours or days. Examples: "Stand 15 minutes in the cornner", "Wear a gag for for 3 hours", "No TV for 3 days". The program will control that the sub don't finish too fast. Use # in the name where you want the program to put in the time. Example: "[punishment-Stand in the corner # minutes]". And use the ValueUnit keyword as described below.

ValueUnit=*unit*	*unit* must be 'minute', 'hour' or 'day'.

Control the severity of the punishment

value=*number*	*number* tells the severity of one repetition/minute/hour/day of the punishment measured in merit points. The severity of the punishment (givin with the Punish keyword) is devided by *number* to calculate the actual punisment. If you do not code value, 1 is used. You may use decimals, e.g. value=0.2.
max=*number*	*number* gives the maximum number of repetitions/minutes/hours/days allowed for this punishment. If the sub needs more punishment than that, more punishment sessions are scheduled. If you do not code max, 20 is used.
min=*number*	*number* gives the minimum mumber of repetitions/minutes/hours/days given in this punishment. If you do not code min, 1 is used.

Example:

```
[punishment-Stand # minutes in the corner]
  value=2
  max=20
  min=5
[confession-Late home]
  punish=20
[confession-Lied to my Master]
  punish=50
[confession-Overslept]
  punish=8
```

If the sub confesses late home, he will be punished with a severiy of 20. Corner time hase a value of 2, which gives 20 devided by 2 = 10 minutes in the corner.

If the sub confesses to have lied, he will be punished with a severiy of 50. Corner time hase a value of 2, which gives 50 devided by 2 = 25 minutes in the corner. This is above maximum, and the sub will be punished with 20 minutes in the corner (maximum) and given a supplemental punishment.
If the sub confesses to have overslept, he will be punished with a severiy of 8. Corner time hase a value of 2, which gives 8 devided by 2 = 4 minutes in the corner. This is below minimum, so the sub is punished with 5 minutes in the corner (minimum).

Don't use a heavy punishment for minor offences.

You may have some punishments that are always severe, and which you don't want to use with minor offences.

MinSeverity=*points*	If the severity asked for is less than *points*, this punishment will not be used and another punishment will be chosen.

Use some punishment more often than others

When the sub must be punished, the program will search through all possible punishments and select one (or sometimes more). It is possible to influence this selection and have some punishments turn op more often than others. You do this by weighting the punishments. The higher weight, the more likely is the punishment to be chosen. A punishment with a weight of 3 is chosen 3 times more often than one with weight of 1.

Weight=*weight*	Tells the weight of the punishment. *Weight* can be a positive number or a counter. Or it can be two values separated by comma, and the program will chose a random number between the two values. If you don't code weight, the punishment will have a weight of 1.

Make the "punishment fit the crime" - Punishment groups

A punishment can belong to one or more groups, allowing the Master to control which punishments can be given for each offence, to "make the punishment fit the crime". If the program gives a punishment on the basis of a section where punishmentgroup(s) are defined, it will choose a punishment from one of the listed groups. If the sub declines the punishment, he/she will be given a new punishment from the groups.

Group=*group,group,..*	List one or more punishment groups that the punishment belongs to. You can add a number of grups separated by commas, or you can add more than one Group keywords.
GroupOnly=1	If you use GroupOnly=1, then this punishment wil only be used if it is selected from a punishment group. In other words, it will only be used if there is a PunishmentGroup keyword where the punishment is generated.
PunishmentGroup=*group,group,*	If this punishment results in a new punishment, i.e. for being late, the new punishment will be chosen from one of the groups specified.

Use the keyword PunishmentGroup in the same section where you have the keyword Punish to tell which group(s) to use. See "Punishing the sub" above.

"Long-running" punishments

A "long-running" punishment is a punishment that can be done while the sub go on with other tasks. Maybe even other punishments. It need not actually run for a long time, though it is usually connected with time. Examples: "No TV for 3 days", "Wear a butt plug for 3 hours", "Wear a gag for 17 minutes".

ValueUnit=day	If you use ValueUnit=day the punishment will always be considered long-running.
LongRunning=1	Tells that this is a "long-running" assignment, even if ValueUnit is not day.

Other punishment keywords

MustStart=1	Means that the sub must report when starting the punishment. That it is not possible to report a punishment done, if it is not started. Ignored if ValueUnit=day, hour or minute or if LongRunning=1 is specified, as the sub must always start those.
Accumulative=1	Means that if a punishment that is already in the assignments list when it is given, a new punishment will not be created, but the existing punishment will be increased. However, if the increased punishment will exceed the Max keyword, a new assignment will be created instead.

See also Assignment keywords (jobs and punishments)

Punishments forbidding the sub to do something

You can have a punishment forbid something the sub has to ask permission for. If the sub asks permission while the punishment is started, it will always be denied.

Forbid=*permission*	While this punishment is active, the sub will not be allowed *permission*. *Permission* must be defined as a normal permission. Example: Forbid=Watch TV

Controlling punishments.

In the [general] section you can add some keywords to generally control the behaviour of punishments.

MaxDecline=*number*	The sub is only allowed to decline a punishment *number* times. If you do not specify this keyword, it is set to 3. Every time the sub declines, the punishment is increased by 20%.
MinPunishment=*number*	When the sub is punished, it will minimum be with a severity of *number* points, even for minor offences. If you do not specify this, a value of 1 is used.
MaxPunishment=*number*	A punishment can maximum be of severity *number*. If you do not specify *number* or if you specify MaxPunishment=0, there is no limit.
AskPunishment=*min,max*	When the sub asks for a punishment, *min* and *max* defines the minimum and maximum severity given, measured in merit points. If you don't code AskPunishment, the sub can not ask for punishments.
AskPunishmentGroup=*group,group,*	When the sub asks for a punishment, it will be chosen from one of the groups specified.

Jobs

Jobs are something you want the sub to do regularly. It can be simple housework, it can be some training or anything you want.

What must be there

[job-*name*]	This line starts a job definition. Replace *name* with the name of the job.

Giving further instructions

Text=*instructions*	The instructions for the job.

Repeating the job with an interval

Interval=*min,max*	The job must be done regularly with intervals of minimum *min* and maximum *max* days. Interval=0 or not specifying Interval means that the job will not be scheduled automatically. *Min* and *max* can be an interval or a counter.

When you start a new script, or when you introduce a new job into an existing script, you may not want to wait the first occurrence of the job to be schedules faster that the interval specifies. Then you can use the FirstInterval keyword.

FirstInterval=*min,max*	Tells how soon the first occurrence of the job must be finished.

Jobs on specific days

You can use the Run keyword to have the job run daily or on specific weekdays. If you have more than one run keyword, the job must be done on all appropriate days.

Run=daily	The job must be done every day.
Run=*weekday*	The job must be done on every *weekday*, where weekday is Monday, Tuesday, Wednesday, Thursday, Friday, Saturday or Sunday.
Norun=*weekday*	The job must be done on every *weekday*. Use this to make exceptions from a Run=Daily.

There's one disadvantage of using the run keyword. The sub can not be notified earlier than the same day. Regardless of what you write in respit, the sub will not be notified earlier than the last midnight before the job must be done.

When to do the job

EndTime=*hh:mm*	Specifies the time of day where the sub must have completed the job. Use 24hour clock, eg. EndTime=14:00 means 2 PM.

What if the job is not done on time?

Usually, if the sub doesn't finish on time, you simply want to remind him and maybe punish him for being late. These keywords works for punishments as well.

RemindInterval=*hh:mm*	If the sub is late doing the job, he/she is reminded and punished every *hh:mm* hours and minutes. You can specify RemindInterval on jobs and punishments and in the [General] section. The interval specified in the [General] section will be used for any job or punishment that does not have RemindInterval specified. If you do not specify the interval, it is set to 24 hours. You can code two values separated by comma. The program will then choose a random value between the two values.
RemindPenalty=*severity*	Every time the sub is reminded about being late, he is punished with a severity of *severity*. You can specify a single number, or two numbers separated by comma. If you use two numbers, the program will pick a random number between them. You can specify RemindPenalty on the job or in the [general] section.
RemindPenaltyGroup=*punishmentgroup*	If you use RemindPenalty, you can use RemindPenaltyGroup to tell which group the punishment should be taken from. See Punishing the sub.
LateMerits=*points*	If the job is not done on time, the subs merit points are reduced with *points*. When the job is done, the points are returned. You can use this if you do not wish to punish the sub for being late, but still want to show that you are not satisfied.

Sometimes, if a job is not done on time, you don't want it done at all.

ExpireAfter=*hh:mm*	If the job has not been done after *hh:mm* hours and minutes after it should be finished, it expires and is removed from the assignment list. You can code to values, ExpireAfter=*hh:mm,hh:mm*, and the program will pick a random time between the two values.

ExpirePenalty=*n*	If the job expires, the sub is punished with a severity of *n*. You can code two values, ExpirePenalty=*n1,n2*, and the program will pick a random number between *n1* and *n2*.
ExpireProcedure=*procedurename*	When the job expires, the procedure *procedurename* is called.

Other job keywords

Respit=*hh:mm*	Specifies the time (in hours and minutes) in advance of the deadline that the sub will be informed about the job, he/she has to do. That is, the time the sub has to find a suitable time to do the job.
Estimate=*hh:mm*	The job is estimated to *hh:mm* hours and minutes. Maximum is 23:59.
DeleteAllowed=1	Allows the sub to delete the job from the assignmentlist.
DeleteProcedure=*procedurename*	When the job is deleted the procedure *procedurename* is called.

See also Assignment keywords (jobs and punishments)

More job control

Jobs to run only one time

Sometimes you may want a job to run only one time. You can do that by adding the OneTime keyword to the job definition.

OneTime=1	Means that the sub will only be asked to run this job one time.

Mark a job done, abort it or delete it

Sometimes you want a report or another action to mark a job as done, without requiring the sub do do it himself. For example, you may have a job to go jogging. When the sub reports back from the running, you want to automatic mark the job as done.

In other cases you want to abort a job. For example, the sub is standing in the corner as a punishment when someone comes to the door. The sub report that guests have arrived, and you want to make sure the punishment is aborted so the sub must start over another time.

You may also want to delete a job from the assignments list.

You do this with the keywords MarkDone, Abort and Delete, which can be used in permissions, reports, confessions, procedures, timers or pop ups.

MarkDone=*jobname*	Marks the job *jobname* done. *Jobname* must include the “job-” prefix. The job will be removed from the assignment list.
Abort=*jobname*	Aborts the job *jobname,* if it is active. *Jobname* must include the “job-” prefix. The job will be returned to the assignment list as not started.
Delete=*jobname*	Deletes the job *jobname* from the assigment list. *Jobname* must include the “job-” prefix.

MarkDone, Abort and Delete can be used for punishments as well.

<u>Add a job to the assignment list</u>

You may want to add a job to the assignment list from the script, instead of letting the program control it.

You do it with the keyword Job, which can be used in permissions, reports, confessions, procedures, timers or pop ups.

Job=*jobname*	Announces the job *jobname* to the sub and adds it to the assignment list. *Jobname* must <u>not</u> include the “job-” prefix.

Assignment keywords (both jobs and punishments)

Assignment keywords are keywords there are common for jobs and punishments.

AddMerit=*points*	When the job is done, *points* points are added to the subs merit points. *Points* can be a value or the name of a counter.
LongRunning=1	Tells that this is a "long-running" assignment. A long-running assignment can be going on during other activities.
MustStart=1	Means that the sub must report when starting the assignment. It is not possible to report an assignment done if it is not started.
StartFlag=*flagname*	Sets the flag *flagname* when the assignment is started and removes it again when assignment is done or aborted.
NewStatus=*newstatus*	Status is set to *newstatus* when the assignment is started and set back when the assignment is done.
AnnounceProcedure=*procedurename*	When the job is announced or the punishment given, the procedure *procedurename* is called.
BeforeProcedure=*procedurename*	When the assignment is starting the procedure *procedurename* is called <u>before</u> the assignment is started.
StartProcedure=*procedurename*	When the assignment is started the procedure *procedurename* is called <u>after</u> the assignment is started.
DoneProcedure=*procedurename*	When the assignment is marked done the procedure *procedurename* is called.

Control when assignments may pop up with messages

When it's time for a job to announce itself to the sub, it pops up with a message box and sounds the alarm. When the deadline for an assignment (job or punishment) has been passed, the sub is reminded and possible punished. Again a dialogue box pops up with an alarm. These dialogue boxes may not be practical when the sub is sleeping, is away or maybe have guests.

To prevent these messages from popping up at undesired times, use the InterruptStatus keyword in the [general] section.

InterruptStatus=*statuslist*	Assignment pop ups will only occur in the status listed in the *statuslist*. *Statuslist* can be one status name or a list separated by commas.

Merits

The merits show how well the sub is doing. A high merit score means that the sub is doing well, and should be allowed privileges. A low merit score means that the sub is not behaving well, and should have less privileges.

Setting the limits

In the [General] section you define the limits of the merits and how they are displayed.

MinMerits=*points*	The lowest merit score displayed will be *points*. A score below this will not be shown on the slider bar, except the bar will turn black. The score will still be shown as a number.
MaxMerits=*points*	The highest merit score possible will be *points*. If the merit score gets above this level, it is truncated to *points*.
Yellow=*points*	If the merit score gets below *points*, the slider bar will become yellow. When the merits are above *points*, the slider bar will be green.
Red=*points*	If the merit score gets below *points*, the slider bar will become red.

Defining the start score

In the [Init] section you can tell what the merit score should be when the sub opens the script for the first time.

Merits=*points*	The sub will start with *points* merit points.

Adjusting the merit score

The merit points are automatic reduced when the sub is punished and restored when the punishment is done. You can influence the merit score in other ways. You can add merit points when the sub has finished his jobs. You can deduct merit points when the sub is confessing bad behaviour or asking for privileges. You can give the sub an "allowance" of a certain number of points per day.

Giving and taking merit points

You can use the AddMerit keyword in permissions, reports, confessions, procedures, timers or pop ups. You can also use AddMerit in jobs, meaning that when the job is done, merit points are added.

AddMerit=*points*	Add *points* points to the subs merit points. *Points* can be a value or the name of a counter. *Points* can be negative.

AddMerits=*points*	Same as AddMerit.
SetMerit=*points*	Set the merit score to *points*, no matter what it was when you before. Use this with care. This is only intended for very special scripts.
SetMerits=*points*	Same as SetMerit.

Giving a daily "allowance"

You can give the sub an "allowance" of a certain number of points per day. Use DayMerits in the [General] section.

DayMerits=*points*	Add *points* points to the subs merit points each day. *Points* can be negative. Points are only given on days where the program have been opened.

Using the merit score

You can use the merit score to decide whether the sub gets a permission. See Permissions based on the merit score.

Hiding the merit score

In the [General] section you can add the keyword HideMerits to hide the merit score from the sub, if you wish to do so.

HideMerits=1	Hide the merit score and slider bar.

Advice on how to use merits

Balancing the merits is one of the most difficult tasks in writing a script. You easoly risk that the score gets way too high or the opposite, that the sub can not get the score up where it should be. It's my experience that it is best to use the maximum score as the standard. Which means that whenever the score is below maximum, the sub must work to get it up. If the sub gains too many merits when already at maximum, they are simply discarded.

Prevent Cheating

If you modify the demo script to make your own script, it will be fairly easy to cheat by reading the report and by modifying the status file. There are two ways to prevent this, depending on your situation.

If you're writing a script for another person:

1. Add the line
   ReportPassword=xxxxx
   to the [general] section. Replace xxxxx with a password of your choice.
2. Encrypt the script (using the File menu) and give the encrypted script (with the extension of .vmi) to the sub. Do not give the sub access to the original script.

These two actions will encrypt the report and the status file and password protect the report. Note that you will need the program to read the report yourself.

If you're writing a script for yourself:

1. Add the line
   Restrict=1
   to the [general] section.

This will encrypt the status file and it will omit information from the report that the sub should not know. Thus making it safe to read the report.

Autoencrypt the script

If you're writing a script for another person, you may want to automatic encrypt the script every time you start the program. This way you don't need to remember to do it manually, and you are always sure your .vmi file is up to date.

1. Add the line
   AutoEncrypt=1
   to the [general] section.

Instructions and clothing

This section describes how to

- give the sub clothing instructions
- give the sub other instructions
- define how you want the sub to report what he is wearing.

This section describes the basic of instructions and clothing. In the section about advanced script writing, you will find more information.

Definitions

First some definitions of the words I will use in this section.

Option	An option is a single line in the instructions you want to give the sub.
Choice	A choice is a set of options, from which VM will choose one.
Set	A set is a collection of choices and other sets, that naturally belong together. In a set VM will choose one option from each choice in the set and one option from each choices in the other sets that is contained in this set.
Instruction	An instruction is a collection of all the choices possible in a certain situation. It is what you want the sub to ask instructions for. It can be how to sleep, how to eat or anything else you want the sub to ask instructions for.
Clothing instruction	A clothing instruction is exactly the same as an instruction, it is just about what the sub must wear in a certain situation.
Check	Checks can be used to compare clothing instructions with a cloth report to check if the sub is dressed as described.

Checks are described in the section Advances cloth issues.

Instructions

Instructions can be used for giving instructions for specific situations. The program works with two kinds of instructions, Clothing Instructions and Other Instructions. They work identically, they just appear in different menus.

How to use instructions

You can use instructions in two ways. One is that the sub asks for instructions for a specific situation using the Communication menu. Another is by use of the Clothing and Instructions keywords. The Clothing and Instructions keywords can be used in permissions, reports, confessions, procedures, timers or pop ups.

Clothing=*name*	This line displays clothing instructions for *name*.
Instructions=*name*	This line displays other instructions for *name*.

Defining the instructions - What must be there

[clothing-*name*]	This line starts a clothing situation definition. Replace *name* with the name of the situation the sub has to ask clothing instructions for.
[instructions-*name*]	This line starts an instruction definition. Replace *name* with the name of the situation the sub has to ask instructions for.

When can the sub ask for instructions?

You can control whether the sub can ask for instructions or instructions are only shown by the script.

Askable=1	The sub can ask for instructions. This is what happens if you don't code Askable.
Askable=0	The sub can not ask for instructions. Instructions are only shown when the script uses the Instructions= or Clothing= keyword.

Defining a choice

Choice=new	Tells VM that a new choice is starting here
Option=*optionname*	Defines the options to choose between. Add one line for each option. An Option=* means that no instructions will

	be given from this choice, if it is selected. An option starting with % will not be shown to the sub.
Weight=*n*	Optional. Use this if you do not want an equal chance for each option to be chosen. Weight=2 will give last option twice as large a change to get selected. Weight=3 three times. Etc.
OptionSet=*setname*	Optional. Tells that if the last option before this line is chosen, then the set *setname* will be used. Se below about sets.

What is nothing is chosen?

If for some reason nothing is chosen, you can define a text to be shown. This can happen if you chose an * or an item with % in all choices, or if you use sets (see later).

None=*text*	If no choices are shown at all, the program will show *text*.

Examples:
none=You must be naked
none=There are no special instructions

When does the program show a new selection?

It is up to you how often the program will change selection for the same instruction.

Change=daily	Means that the sub will get the same answer all day, but a new answer the next day. This is useful for most purposes. If you do not specify Change, Change=Daily is used.
Change=program	Means that every time you use Instructions= or Clothing= in a report, procedure etc., the program will generate a new answer. If the sub asks later for the same instructions, he will get the same answer.
Change=always	Means that every time the sub presses the Ask button, he will get a new answer. This is mostly useful for testing a script. If you use this option in real life, the sub can keep pressing the Ask button until he gets an answer he likes. A better way to test your script is to use the Test menu. Se the chapter "Testing your script".

Using sets to control the choices

Often you want to select choices depending on what have already been chosen. Or you may want your choices to depend on some other circumstances. You do this by using sets. Instead of a Choice keyword and one or more Option keywords, you add a Set keyword to tell that you have put the choices available in a set. See below how to define a set.

Set=*setname*	Tells that the options will be picked from the set *setname* (see below). Use as many lines as necessary. You can mix set lines and choice/option lines, and the result will be shown in the sequence you use.
Weight=*n*	Optional. Use this if you use Select=random and do not want an equal chance for each set to be chosen. Weight=2 will give last set twice as large a change to get selected. Weight=3 three times. Etc.

More control of what is chosen

You now know that you build instructions of choices and sets. If you don't son anything, all choices and sets are used in the sequence they are written. However, you have some further possibilities to control what is chosen. You can use the Select keyword.

Select=all	The program will use all the set and choice keywords between this select and the next select. The same as if you don't use select at all.
Select=first	The program will only use the first possible set or choice between this select and the next select. A choice keyword is always possible. A set may be impossible, if you have used If, IfNot, IfChosen or IfNotChosen. See below.
Select=random	The program will choose a random set or choice keyword between this select and the next select.

Sets

Sets are collections of choices. Sets can be used or not used depending on what else is chosen. Sets are useful for conditional instructions and to avoid repeating the same instructions again and again. The same set may be used in several instructions. Sets can use other sets, but the set that is referred to must be defined before the set that refers to it.

[set-*name*]	Defines the beginning of a set.
Option=new	Works like in instructions (see above).
Set=*setname*	Works like in instructions (see above).
Weight=*n*	Works like in instructions (see above).
OptionSet=*setname*	Works like in instructions (see above).
Select=...	Works like in instructions (see above).

If sets are not always to be used

IfChosen=*precondition,precondition..*	If any of the words in the precondition list exist in the items chosen in former choices, the program will use this set. If none of the words exists in the chosen items, the set will not be used. Consider using OptionSet instead of IfChosen, as it will give better control.
IfNotChosen=*precondition,precondition..*	If none of the words in the precondition list exists in the items chosen in former sets, the program will use this set. If any of the words exist in the chosen items, the set will not be used. Consider using OptionSet instead of IfNotChosen, as it will give better control.
If=*flagname,flagname...*	The set will only be used if all the flags in the list are set. You may have more than one If line. If there are more than one line, the set will be used if any line is true.
NotIf=*flagname,flagname...*	If all of the flags in the list are set, the set will not be used. You may have more than one NotIf line. If there are more than one line, the set will not be used if any line is true.

An old way to define choices

This is an older way of defining choices. It is not recommended, as it is less flexible than the way described above. It can be used in both instructions, clothing instructions and sets.

Choice=*option,option,option...*	This is the simple form. The program will chose one *option* randomly. You can have as many choice lines as you wish, the program will chose one item from each line.
Choice=*option*,*weight,option,weight..*	Use this form if you do not want an equal chance for each *option* to be chosen. The list alternates between items and the weight of that item. E.g.: "choise=skirt and blouse,2,slacks and blouse,2,dress,1" will tell the program to choose skirt and blouse 2 days out of 5, slacks and blouse 2 days out of 5 and a dress 1 day out of 5. You can omit some or all of the weights, then the option will be given a weight of 1. Maximum weight for an option is 255.

Cloth reports

The sub can report what he is wearing. The sub can chose to make the report himself via the communications menu, or he can be ordered to make the report.

The principle is that you (the script writer) specifies what you want to know, and the sub gives the wanted information. As a script writer you define some cloth types and what you want to know about each. For each type you define a number of attributes that the sub must describe. Each attribute can either be a simple text field, or a list the sub can chose from.

Defining the cloth types

[clothtype-*type*]	This line starts a type definition.
Attr=*attributename*	This line defines an attribute about the type. This could be style, length, color, size etc. Whatever you want to know.
Value=*value*	If you want the sub to chose attribute values from a list, you code a number of Value keyword after the Attr keyword it describes. A Value keyword is always referring to the preceding Attr keyword.
Value=?	Means that although the sub can chose values from a list, he is also allowed to write a different value, not on the list.

Ordering the sub to make a cloth report.

You can order the sub to make a cloth report in permissions, reports, confessions, procedures, timers, pop ups or sets.

ClothReport=*text*	Brings up the cloth report window with *text* as the headline.
ClothReport=1	Brings up the cloth report window with a standard headline.
ClearCloth=1	Removes all the subs current clothes. The next time the sub is asked to make a cloth report, he will start with nothing selected. ClearCloth dows not remove the clothes available, just the current clothes (below “You are wearing”).

Importing and exporting clothes

The clothes that the sub enter in the system is stored in the status file, along with everything else about the sub. When the sub deletes the status file, it is all lost. When running seriously, you should not delete the status file and thus restart the program, as this means that all punishments and assignments are removed and the merit score is reset.

However, when testing the program, you might want to keep the clothes you have entered. And there may be situations where you want to allow the sub to delete the status file and restart the program without loosing the clothes.

In the test menu you will find the options Export clothes and Import clothes. You can use Export clothes to create a file containing the clothes you have entered. Then after a restart, you can use Import Clothes to import the clothes to your new s file. See The test menu.

If you want to allow the sub to export and import clothes, you can add ClothExport=1 to the [general] section.

ClothExport=1	Adds Export clothes and Import clothes to the File menu.

Advanced script writing

This section describes more advanced script writing.

Changing the name of elements

Sometimes you want to give a different name to a permission, report etc. than the one written in the header between the [and] brackets. One reason can be that you want to define more than one element that for the sub looks the same. Another reason can be that you want to refer to short names in your script and still give the sub a long and meaningful name.

You change the name with the Title keyword. You can use it in reports, permissions, confessions, timers, popups, jobs, punishments and procedures.

Title=*new_name*	New_name is what is presented to the sub.

Examples:

```
[permission-sleep-workday]
title=Go to sleep
if=sonday
if=monday
if=tuesday
if=wednesday
if=thursday

[permission-sleep-weekend]
title=Go to sleep
if=friday
if=saturday
```

Procedures

You may want a lot of reports or confessions to behave the same way. Instead of defining a lot of very similar actions, you can define a procedure to do the wanted action and then have the reports call the procedure. This approach makes it much easier to change the behaviour of all the procedures at once. This is only one example of how procedures can be used.

Another reason for using reports is conditional actions. You may want something to happen only under special circumstances. You can put the actions in a procedure and then use flags to decide whether you want the actions done.

Procedures can do the same thing as reports, but the sub can not start a procedure. A procedure is started from reports, permissions, timers, popups, jobs, punishments or questions. Almost everywhere you define some action the program must do, you can call a procedure.

Note: <u>A procedure can not call itself.</u> Neither directly or indirectly. Indirectly means procedure1 call procedure2 that tries to call procedure1. Succeeding attempts to call the same procedure will be ignored and you will get a message in the report. The reason for this is to protect you from setting the program in an indefinite loop by accident.

What must be there

[procedure-*name*]	This line starts a procedure definition. Replace *name* with the name of the procedure.

What can you do in a procedure?

You can use the same keywords for procedure as for reports, including Procedure= to call another procedure.

How to use a procedure?

A procedure can be started from reports, permissions, timers, popups, jobs, punishments or questions. Almost everywhere you define some action the program must do.

Procedure=*procedurename*	The procedure *procedurename* is called. You can have more than one procedure line.

You can use variables in the procedure name when calling a procedure. Note that if you do, there is no check that the procedure exists. And if you end up calling a procedure that doesn't exist, the procedure call is ignored.

Conditional actions

You can use If, NotIf, NotBefore, NotAfter, NotBetween and PreStatus to specify that the procedure shall not be executed, even if it is called. See chapters about Flags and Status.

You can combine conditional actions with several procedure calls to determine which procedure is called. If you have more than one Procedure keyword in sequence, you can use the select keyword to tell which procedure(s) you want executed. Use select before the first Procedure keyword which must be affected.

Select=All	All the following procedures are called.
Select=First	Only the first possible of the following procedures are called. It is possible to call a procedure, if there are no If, NotIf, NotBefore, NotAfter, NotBetween or PreStatus keywords in the procedure that makes the procedure not calable.
Select=Random	The program will select a random procedure from the following procedures. Only the selected one is called.

If you do not specify Select, all procedures will be called.

Example:

```
Select=All
procedure=procedure1
procedure=procedure2
Select=Random
procedure=procedure3
procedure=procedure4
procedure=procedure5
Select=First
procedure=procedure6
procedure=procedure7
procedure=procedure8
Select=All
procedure=procedure9
procedure=procedure10
```

In the above example, the following will happen in this sequence:

1. Procedure 1 is called
2. Procedure 2 is called
3. Either Procedure3, procedure4 or Precedure5 is called. The program will choose randomly.
4. Either Procedure6, procedure7 or Precedure8 is called. The program will choose the first possible procedure.
5. Procedure 9 is called.
6. Procedure 10 is called

Weighting procedures with Select=Random

If you use Select=Random, you may use some procedure to be chosen more often than others. You can control that by weighting the procedures.

Procedure=procedurename,*weight*	The procedure *procedurename* is called. Used with Select=Random, the procedure will have the weight specified by *weight*. *weight* must be an integer between 1 and 255.

Example:

```
Select=Random
procedure=procedure1,1
procedure=procedure2,3
procedure=procedure3,1
```

In the above example, procedure2 have a weight if 3, which is three times what the others have. Which means that procedure 3 will be called three times as often as procedure 1 or procedure 2.

Pop ups

Pop ups define actions to pop up randomly. Use pop ups to check that the sub is home when he/she is supposed to be, to give random instructions to the sub or for whatever else you want.

How to define a pop up

[popup-*name*]	This line starts a popup definition.

You can mostly use the same keywords that can be used for a report. Use the keyword to give a message or an order to the sub. Use Prestatus, If and Notif, NotBefore, NotAfter and NotBetween to control when a pop up can be invoked. Se the chapter Flags later.

How to activate pop ups

Activate pop ups by specifying the following keyword on a status definition:

PopupInterval=*min,max*	If PopupInterval is present in the definition of the current status, the program will periodically invoke a Pop up. *min* and *max* defines the minimum and maximum time between popups. *min* and *max* are written as hh:mm (hours and minutes) or hh:mm:ss (hours, minutes and seconds).

How to enable and disable pop ups by use of flags

Normally it's the status that defines whether pop ups are active. But maybe under certain circumstances you may want to disable pop ups, even in a status where they are usually enabled. Do this by setting a flag when you want to disable the pop ups. Or by only setting a flag when you want pop ups enabled.

Use this in a status definition:

PopupIf=*flagname*	Only if *flagname* is set, popups will occur.
NoPopupIf=*flagname*	If *flagname* is set, no popups will occur.

Use some pop ups more often than others

When it is time for a pop up, the program will search through all possible pop ups and select one. It is possible to influence this selection and have some pop ups turn op more often than others. You do this by weighting the pop ups. The higher weight, the more likely is the pop up to be chosen. A pop up with a weight of 3 is chosen 3 times more often than one with weight of 1.

Weight=*weight*	Tells the weight of the pop up. *Weight* can be a positive number or a counter. Or it can be two values separated by comma and the program will choose a random value between the two. If you don't code weight, the pop up will have a weight of 1.

How to change the sound when a pop up occur

You can change the sound that occurs when a pop up occurs. You can set up a general sound for all pop ups, and you can set individual sounds in individual pop ups. The sound must be in a .wav file.

Use the keyword Alarm in the [general] section to set up an alarm file for all pop ups which don't have an individual alarm file.

Use the keyword Alarm in the [popup-*name*] section to set up an alarm file for this pop up only.

alarm=*wave file*	The name of a sound file to be used as an alarm when a pop up occurs.

Use the keyword PopupAlarm in the [status-*name*] section or the [popupgroup-*name*] section to set up an alarm file for this status or group.

PopupAlarm=*wave file*	The name of a sound file to be used as an alarm when a pop up occurs.

The program is looking for a sound file in this sequence:

1. The pop up
2. The pop op group
3. The status
4. The [General] section
5. It uses the default name *alarm.wav*.

The first filename found is used. If the file doesn't exist, there will be no sound.

How to control response times for pop up.

You may want to ensure that the sub reacts to a pop up within a specified time. To do this, see Response time for pop ups.

How to use pop up groups

Normally it's the status that defines how pop ups are used. But you may have several status with the same pop up settings. Writing and maintaining the pop up keywords for several status may be tedious.

You can restrict a pop up to be used in a certain status by using the Prestatus keyword. But if you have a lot of pop ups which all are to be used in certain status, your script may be rather complex.

Pop groups can help you with these problems. You can define pop up groups for pop ups that are to be used in the same way. You can connect status and pop ups to the pop up groups. A status can only belong to one pop up group, whereas a pop up can belong to several groups.

If a status belong to a pop up group, it will use all the pop up settings from the status. Pop up control keywords in the status itself will be ignored.

If a pop up belong to one or more pop up groups, those groups will be compared to the pop up group of the current status. Only if the current status belong to one of the groups, will the pop up be used.

Defining pop up groups

[Popupgroup-*name]*	Starts a pop up group definition.
PopupInterval=*min,max*	Same as if you used it in a status definition.
PopupIf=*flagname*	Same as if you used it in a status definition.
NoPopupIf=*flagname*	Same as if you used it in a status definition.

The following keywords are described in Response time for pop ups.

PopupMinTime=*hh:mm*
PopupQuickPenalty1=*points*
PopupQuickPenalty2=*ratio*
PopupQuickMessage=*text*
PopupMaxTime=*hh:mm*
PopupSlowPenalty1=*points*
PopupSlowPenalty2=*ratio*
PopupSlowMessage=*text*
PopupQuickProcedure=*procedurename*
PopupSlowProcedure=*procedurename*

Using pop up groups

Use this in a status definition:

Popupgroup=*name*	Tells that this status belongs to the pop up group *name*. A status can only belong to one pop up group.

Use this in a pop up definition:

Group=*name*	Tells that this pop up belongs to the pop up group *name*. You can have more Popupgroup keywords, if the pop up belongs to more than one group.

Timers

Timers define actions to happen on time. You can use timers give the sub an order on a fixed time (e.g. go to bed) or to do something that the sub don't even notice, like setting or removing a flag (see later). You can use timers for anything that must happen on time.

Note that timers does not automatic play an alert sound. You must use the Sound keyword if you want to play a sound. See Playing sounds.

What must be there

[timer-*name*]	This line starts a timer definition. Replace *name* with the name of the timer.
start=*hh:mm,hh:mm*	Tells the timer to activate on a random time between the first and the second time. Times are in hours and minutes (24 hour clock).
end=*hh:mm*	If the end time arrives and the timer has not been activated, the timer will not start, but will wait till next day.

Using status with timers

Often you only want the timer be activated when the sub is in a specific status. Use the keyword PreStatus to specify in which status the timer can be activated. If the sub is in a corrent status when the time is up, the timer will be activated. If not, the timer will wait until the sub gets into that status. If the time specified by the keyword End is reached without the status have been reached, the timer will wait til next day.

PreStatus=*status,status...*	When the time arrives, the timer will only be activated if current status is in the list of *status,status.....*

Using timers with flags

You can use If and Notif, NotBefore, NotAfter and NotBetween to control when a timer can be activated. If one of these conditions gives that the timer should not be activated, the timer will wait till next day.

Timers and weekdays.

If you want a timer to only be activated on certain weekdays, you can use the automatic flags (se the chapter Flags).

If=Sunday	The timer will only be activated on Sundays.

NotIf=Saturday	The timer will not be activated on Saturdays.

Writing messages to the sub

There are different ways of giving messages to the sub. You can show a simple message in a popup box with the Message keyword. You can do that in a permission, report, confession, procedure, timer or pop up.

Message=*messagetext*	The *messagetext* will appear in a pup up box.

If you have more than one message statement following each other, you can control how many is shown to the sub. Use the Select keyword before the first Mrocedure keyword which must be affected.

Select=All	All the following messages are shown.
Select=Random	The program will select a random message from the following Message keywords.

If you do not code a Select keyword, all message will be shown.

<u>Example:</u>

```
Select=All
Message=This message will be shown.
Message=This message will also be shown.
Select=Ransom
Message=This message may be shown.
Message=Or This message may be shown.
Select=All
Message=This message will always be shown.
```

Another way of giving information to the sub is to write in the message area in the main window. You do this by the Text keyword. While pop up boxes generated from the Message keyword are a one time incidents, the message area is a static area. The text here will stay for while. For this reason it can only appear in elements that defines a period of time. You can use the text keyword in status, flags, jobs and punishments.

Text=*textline*	The *textline* will appear in the message area. You can have more than one text line. They will all show.
Text=%instructions	This will show the last instructions given by the Instructions= keyword. It will <u>not</u> show instructions that the sub asked for.

Text=%clothing	This will show the last clothing instructions given by the Clothing= keyword. It will not show clothing instructions that the sub asked for.

You can have a fixed message at the top and/or bottom of the message area. You do that by adding one or both of the following keywords to the [General] section.

TopText=*textline*	The *textline* will appear at the top of the message area. You can have more than one TopText line. They will all show.
BottomText=*textline*	The *textline* will appear at the bottom of the message area. You can have more than one BottomText line. They will all show.

Asking questions to the sub

You may want to ask questions to the sub. There are different ways of doing this, depending on how you want to react to the answer.

Input, the really simple way

The simplest way of asking a question is to use the Input keyword. It can be used in in a permission, report, confession, procedure, timer or pop up.

Input=*question*	The sub gets a pop up box with headline *question* and with a field to type a reply. The reply is recorded in the report.

The sub can chose to not reply to a question phrased by an Input keyword. However, you can catch that situation and react on it. Do that by adding the NoInputProcedure keyword along with the Input keyword.

NoInputProcedure=*procedurename*	If the sub doesn't answer the question, the procedure *procedurename* is called. You can use this procedure to punish the sub or do whatever you wish.

Input to variables, a bit more complex

You can ask the sub to type a text or a number which can be stored in a variable. You do that by using one of the keywords Input$ or Input". Read more in the chapter Variables.

Advanced questions, how to define them

Advanced questions are used when you want to ask the sub a question and have the program react on the answer.

[question-*name*]	This line starts a question definition. Replace *name* with the name of the question
Phrase=*question*	Defines the question the sub is asked.
?*answer*=*procedure*	All lines starting with a question mark (?) defines a possible answer to the question the sub is asked. The sub is presented with a list of answers and must choose one. When an answer is chosen the corresponding procedure will be activated. See the chapter Procedures.
?*answer*=*	If you write an asterisk (*) instead of a procedure name, the program does nothing when this answer is chosen.

Advanced questions, how to ask them

You ask the question by using the keyword Question in a permission, report, confession, procedure, timer or pop up.

Question=*question*	If you write an asterisk (*) instead of a procedure name, the program does nothing when this answer is chosen.

Duration keywords, being too quick or too slow

These keywords deals with an interval of time. They can be used in status, flags, jobs and punishments. They can also be used in timers to describe the response time. See later about response time for pop ups. You can specify the duration as hours and minutes (hh:mm) or in the special duration syntax. See Specifying intervals and durations later.

MinTime=*hh:mm*	The interval must be at least *hh:mm* hours and minutes. You can use MinTime=*hh:mm,hh:mm* and the program will pick an interval between the two values. You can use a time variable instead of *hh:mm*.
QuickPenalty1=*points*	If MinTime is not reached then the sub is punished with the value of *points*. Use this to give a minimum punishment for being early and still make sure that each minute counts. Used only if MinTime is specified.
QuickPenalty2=*ratio*	The penalty for being early is *ratio* merit points for each minute late. *Ratio* may have decimals, e.g. 0.8 is valid. Used only if MinTime is specified. *ratio* is reduced when the sub is very early, see Punishments for details.
QuickMessage=*text*	If you are not satisfied with the standard message when the sub is early, you can write your own message here. If you use a number sign (#) in the text, it will be replaced with the time the sub is early. If you use a ¤ or a percent sign (%) in the text, it will be replaced with the time the sub has used.
QuickProcedure=*procedurename*	The procedure *procedurename* is called when the sub is early. Use it for any action you want to take. The procedure is called after the sub is punished. If you specify QuickProcedure and you do not specify QuickPenalty1 or QuickPenalty2 or QuickMessage, the sub will not receive a message from the program.
MaxTime=*hh:mm*	The interval must not exceed *hh:mm* hours and minutes. You can use MaxTime=*hh:mm,hh:mm* and the program will pick an interval between the two values. You can use a time variable instead of *hh:mm*.
SlowPenalty1=*points*	If MaxTime is exceeded then the sub is punished with the value of *points*. Use this to give a minimum punishment for being late and still make sure that each minute counts. Used only if MaxTime is specified.
SlowPenalty2=*ratio*	The penalty for being late is *ratio* merit points for each minute early. *Ratio* may have decimals, e.g. 0.8 is valid.

	Used only if MaxTime is specified. *ratio* is reduced when the sub is very late, see Punishments for details.
SlowMessage=*text*	If you are not satisfied with the standard message when the sub is late, you can write your own message here. If you use a number sign (#) in the text, it will be replaced with the time the sub is early. If you use a ¤ or a percent sign (%) in the text, it will be replaced with the time the sub has used.
SlowProcedure=*procedurename*	The procedure *procedurename* is called when the sub is late. Use it for any action you want to take. The procedure is called after the sub is punished. If you specify SlowProcedure and you do not specify SlowPenalty1 or SlowPenalty2 or SlowMessage, the sub will not receive a message from the program.
MinTimeProcedure=*procedurename*	The procedure *procedurename* is called when the mininum time specified with MinTime is up. You can use this to give a message to the sub. You can use MinTimeprocedure without MinTime in punishments with ValueUnit=minute or ValueUnit=hour. Then *procedurename* will be called when time is up.
MaxTimeProcedure=*procedurename*	The procedure *procedurename* is called when the maximum time specified with MaxTime is up. You can use this to give a message to the sub.

Using MinTime and MinTimeProcedure with jobs and punishments

If you use MinTime with a job or punishment, and the sub finishes before the mintime specified, the job or punishment will not be completed. The reason for this is to make it more difficult to cheat.

If you use MinTimeProcedure in a punishment with ValueUnit=minute or ValueUnit=hour without specifying MinTime. The program will know the minimum time. You can use this to alert the sub when time is up.

Response time for pop ups.

If you want to measure the response time for pop ups, you can add the following keywords to the status or pop up group, where you put the PopupInterval keyword.

PopupMinTime=*hh:mm*
PopupQuickPenalty1=*points*
PopupQuickPenalty2=*ratio*

PopupQuickMessage=*text*
PopupMaxTime=*hh:mm*
PopupSlowPenalty1=*points*
PopupSlowPenalty2=*ratio*
PopupSlowMessage=*text*
PopupQuickProcedure=*procedurename*
PopupSlowProcedure=*procedurename*

Response time for timers and pop ups.

When you use MaxTime in a timer or PopupMaxTime in a status, the program will display the message "Yes, Master". It is the response time of this message that is checked. If you want another message, you can use the keyword PopupMessage to describe your own message. PopupMessage can be used in timers, pop ups or in the [General] section. The message you specify in the [General] section is used for all timers or pop ups where you don't code PopupMessage.

PopupMessage=*message*	Use this message in stead of "Yes, Master".

Advanced permission use

This chapter is about advanced ways of controlling whether the sub will be granted permissions. And about how to take action when permission is denied.

Permissions based on time

DenyBefore=*hh:mm*	If the subs asks for permission before *hh:mm*, permission is denied. *hh:mm* is using the 24-hour clock.
DenyAfter=*hh:mm*	If the subs asks for permission after *hh:mm*, permission is denied. *hh:mm* is using the 24-hour clock.
DenyBetween=*time1,time2*	If the subs asks for permission between time1 and time2, permission is denied. *Time1* and *time2* is in the format hh:mm using the 24-hour clock.

Permissions based on flags

DenyIf=*flagname*	Permission is denied if the flag *flagname* is set.
PermitIf=*flagname*	Permission is given if the flag *flagname* is set.

Permissions based on the merit score

DenyBelow=*points*	If the subs merit score is below *points* points, permission is denied. *Points* can be a number or a counter.
DenyAbove=*points*	If the subs merit score is above *points* points, permission is denied. *Points* can be a number or a counter.

More variation in permission limits

You can generate even more random permissions or you can control the permissions by use of counters.

Pct=*value1*,*value2*	The program will pick a random number between *value1* and *value2* and use that as the permission percentage. *Value1* and *value2* can be numbers or counters.

Variable permissions based on the merit score

Sometimes you want the chance of getting a permission to vary depending on the merit score.

Pct=Var	Tells that you will use a variable percentage of chance to get permission.
HighMerits=*points*	If the subs merit score is above *points* points, the sub has the chance of getting permission which is specified by HighPct.
HighPct=*number*	Use together with HighMerits. If the subs merit score is above the number given in HighMerits, the percentage chance of getting will be *number*%. If you use HighMerits without HighPct, the chance will be 100%.
LowMerits=*points*	If the subs merit score is below *points* points, the sub has the chance of getting permission which is specified by LowPct. LowMerits must be greater than zero.
LowPct=*number*	Use together with LowMerits. If the subs merit score is below the number given in LowMerits, the percentage chance of getting will be *number*%. If you use LowMerits without LowPct, the chance will be 0%.

<u>Example:</u>
```
[Permission-Do something]
Pct=Var
HighMerits=800
HighPct=90
LowMerits=300
LowPct=20
```

The above example works like this:
- If the merit score is above 800, the sub will have 90% chance of getting permission.
- If the merit score is below 300, the sub will have 20% chance of getting permission.
- If the merit score is between 300 and 800, the subs chance of getting a permission will vary between 20% and 90%. The higher merit score, the better chance.

Use a procedure to decide whether permissions should be granted

You can call a procedure a procedure when permission is asked and set a special flag to tell if permission should be granted or denied. You can also chose to cancel the request.

BeforeProcedure=*name*	When permissions is asked, the procedure *name* is called before it is decided whether to give permission.

The procedure can set one of these flags:

zzPermit	Set this flag if permission must be given.
zzDeny	Set this flag if permission must be denied.
zzCancel	Set this flag if you want to cancel the request. Canceling the request will not influence what happens next time the permission is asked, and the sub will not get a permission denied message.

If non of these flags are set, the usual rules determine if permission is given.

Example:

```
[permission-Use the toilet]
beforeProcedure=BeforeToilet

[procedure-BeforeToilet]
question=BeforeToilet

[question-BeforeToilet]
text=Is it really urgent?
?Yes, it is very urgent=ToiletYes
?No, not really yet=ToiletNo
?Sorry, wrong permission=ToiletCancel

[procedure-ToiletYes]
setflag=zzPermit

[procedure-ToiletNo]
setflag=zzDeny

[procedure-ToiletCancel]
setflag=zzCancel
```

Take action when a permission is denied

When a permission is denied, it's possible for you to take some actions.

DenyProcedure=*name*	When permissions is denied, the procedure *name* is called.
DenyFlag=*flagname*	When permissions is denied, the flag *flagname* is set. You can have more that one flag, separate the names by commas.
DenyMessage=*message*	When permissions is denied, display the message.
DenyLaunch=*filename*	When permissions is denied, the file *filename* is launched.
DenyStatus=*statusname*	When permissions is denied, the the program switch to status *statusname*.

Sign in

The sub can be asked to sign in at regular intervals. Sign in means using the program in some way. If not anything else, click on a Sign in button in the lower right corner of the screen.

You can use this to make sure the sub is home and awake, or simply to keep the sub on his/her toes.

Sign in is connected to a status. Use the SigninInterval keyword in a status definition.

SigninInterval=*hh:mm:ss*	The sub must sign in at least once every hh:mm:ss (hours, minutes and seconds). Instead of *hh:mm:ss* you can use a time variable. You can use two values separated by a comma. The program will then pick a random time between the two values. The program will pick a new value every time the sub signs in.

Time keywords, being late or early

These keywords deal with the time of day. They specify the latest or earlies time the sub are allowed to do something, like make a report or ask a permission. They can be used in reports, confessions, procedures and permissions.

Latest=*hh:mm*	The report etc. should be made before the time given in *hh:mm* (hours and minutes, 24 hour clock).
LatePenalty1=*points*	Use this to give a minimum punishment for being late. Used only if Latest is specified.
LatePenalty2=*ratio*	The penalty for being late is *ratio* merit points for each minute late. *Ratio* may have decimals, e.g. 0.8 is valid. *ratio* is reduced when the sub is very late. Used only if Latest is specified.
LateProcedure=*procedure*	When the sub is late, the procedure *procedure* will be called.
Earliest=*hh:mm*	The report etc. should be made after than the time given in *hh:mm* (hours and minutes, 24 hour clock).
EarlyPenalty1=*points*	Use this to give a minimum punishment for being early. Used only if Earliest is specified.
EarlyPenalty2=*ratio*	The penalty for being late is *ratio* merit points for each minute too early. *Ratio* may have decimals, e.g. 0.8 is valid. *ratio* is reduced when the sub is very early. Used only if Earliest is specified.
EarlyProcedure=*procedure*	When the sub is early, the procedure *procedure* will be called.

The severity of the punishment given is calculated by multiplying penalty2 by the number of minutes and adding penalty1

Specifying Latest and Earliest

Latest and Earliest can be specified by using two values. The program will then pick random value between the two, each time it is needed.

Latest and Earliest can be specified by using a time variable instead of a constant time. If the value of the time variable is less that 24 hours it is treated as a time of day. However, if the value is above 24 hours, it is treated as a date and time.
See Time variables.

Controlling menus based on time

Sometimes you want to control which permissions, reports and confessions are available in menu, based on time. You may want to control in the same way whether procedures can be called or timers or pop ups are allowed to spring.

Use these keywords in permissions, reports, confessions, procedures, timers or pop ups.

NotAfter=*hh:mm*	The report etc. is <u>not</u> available after the time given in *hh:mm* (hours and minutes, 24 hour clock).
NotBefore=*hh:mm*	The report etc. is <u>not</u> available before the time given in *hh:mm* (hours and minutes, 24 hour clock).
NotBetween=*hh:mm,hh:mm*	The report etc. is <u>not</u> available between the time given in the first *hh:mm* and the time given in the last *hh:mm*.

Prevent simultaneous assignments

If the sub have several long-running assignments (jobs or punishments) of the same kind in the assignment list, the program will prevent him from starting them simultaneously. Examples: Wear a gag for 4 hours and Wear a gag for 3 hours.
But if the sub has different assignments, that can't possibly be carried out simultaneously, the program have no chance of knowing this, unless you tell it. Examples: Wear a ball gag for 4 hours and Wear a ring gag for 3 hours. In this case, you must tell the program that the punishments are mutually exclusive. You do this by using the Resource keyword with same resource name on both jobs or punishments.

Resource=*name,name..*	*Name* is the name that you chose to describe the "resource". You can have several names separated by comma.

Example:

```
[punishment-Wear a ball gag for # hours]
resource=Gag
[punishment-Wear a ring gag for # hours]
resource=Gag
[job-Wear a gag for 2 hours]
resource=Gag
```

Events

Events are things that happens, which you may want to react on and which are not controlled in other ways. You can specify a procedure to start on each event.

[events]	This line starts the event section.
FirstRun=*procedure*	Activates *procedure* the very first time a new script is used.
OpenProgram=*procedure*	Activates *procedure* each time the program is opened.
CloseProgram=*procedure*	Activates *procedure* each time the program is closed.
StartFromPause=*procedure*	Activates *procedure* when the program is opened after a pause activated by PgmAction=Pause. This procedure is called before the OpenProgram event.
DeleteStatus=*procedure*	Activates *procedure* just before the status file is deleted..
BeforeNewReport=*procedure*	Activates *procedure* just before a new report is made. Use this to send a mail with the last part of the old report before a new report is started.
AfterNewReport=*procedure*	Activates *procedure* just after a new report is made.
BeforeClothReport=*procedure*	Activates *procedure* just before a cloth report is made.
AfterClothReport=*procedure*	Activates *procedure* just after a cloth report is made.
CheckOn=*procedure*	Called every time VM finds that the sub is not wearing something he should. Use the string variable $zzCheck to tell what it is. See Check the subs cloth report.
CheckOff=*procedure*	Called every time VM finds that the sub is wearing something he is not supposed to wear. Use the string variable $zzCheck to tell what it is it is. See Check the subs cloth report.
CheckAll=*procedure*	Called after a cloth report is made if the checks showed a difference between what the sub is wearing and what he is supposed to wear. Use the counter #zzClothFaults to tell how many violations there were. See Check the subs cloth report.
PunishmentGiven=*procedure*	Activates *procedure* just after a punishment is given, except when it is asked for by the sub. The counter #zzpunishment will contain the severity of the punishment.
PunishmentAsked=*procedure*	Activates *procedure* just after the sub asks for a punishment. The counter #zzpunishment will contain the severity of the punishment.
PunishmentDone=*procedure*	Activates *procedure* just after a punishment is done (finished).
JobAnnounced=*procedure*	Activates *procedure* just after a job is announced to the sub.

JobDone=*procedure*	Activates *procedure* just after a job is done (finished).
Signin=*procedure*	Activates *procedure* whenever the sign in button is pressed.
MailFailure=*procedure*	Activates *procedure* if for some reason a mail can not be sent.
AutoAssignEnd=*procedure*	*procedure* is called when an autoassign period ends. It is not called immediately when the time is up, it is called when the active assignment is finished and you're back in a status with autoassign=1.
Minimize=*procedure*	*procedure* is called when the program is minimized.
Restore=*procedure*	*procedure* is called when a mimimized program is restored.
ForgetConfession=*procedure*	*procedure* is called when the sub makes a confession that he has forgotton to ask a permission. See Automatic confessions.
IgnoreConfession=*procedure*	*procedure* is called when the sub makes a confession that he has done something though permission was denied. See Automatic confessions.
MeritsChanged=*procedure*	*procedure* is called every time the merits are changed. Note: If you change the merits in *procedure*, it will not be recalled. When in *procedure*, you can use the variables #zzBeforeMertis to show the merits before change, #zzAddMerits to show the change and #zzMerits to show the merit points after the change. *procedure* is called after the merits are changed.

Starting assignments - Keeping the sub busy

You may have periods where you want the program to keep the sub busy, to tell him exactly what to do. In VM this is called "automatic assignments".

In a period of automatic assignments the program will pick the first assignment (the one with the first deadline) that can be done in the time available. It will order the sub to do it now. When the sub reports it finished, the VM will chose the next one. This will go on until the time is up or there are no more assignments that fit into the tile available.

VM will never start an assignment that is marked as long running, as the long running assignments are meant to be simultaneous with something else.

How to do it

There are three or four things you need to do:

1. You need to make sure that your jobs and punishments have an estimate. That's the only way VM can now how long the assignment will take. You need not specify an estimate on punishments with ValueUnit=minute or ValueUnit=hour. If VM can not estimate an assignment, it will not use it.
2. (Optional) You may define some jobs that VM can use, if no other assignments are available.
3. You need to tell in which status VM is allow to do automatic assignments.
4. You need to start the period and tell how long it will be.

<u>Estimating jobs and punishments</u>

In jobs and punishments:

Estimate=*hh:mm*	Tells how long time a job is estimated to take. Estimate is not necessary in punishments with ValueUnit=minute or ValueUnit=hour.

<u>Defining which status to use</u>

In status:

AutoAssign=1	Tells that in this status is allowed to use automatic assignments if automatic assignments are started.

Defining extra jobs

In jobs:

AutoAssign=*n*	Tells that in this job may be started if no other assignments are available. *n* is usually 1, but may be a higher number. The higher the number, the more likely the job is to be chosen. A job with AutoAssign=2 is chosen twice as often as a job with AutoAssign=1.

Starting automatic assignments

Use on of these keywords to start a period with automatic assignments. They can be used in reports, permissions, confessions, procedures, timers, pop ups or timers.

StartAutoAssign=time,*hh:mm*	Starts automatic assignments and let them run until a specific time of day. *hh:mm* specifies the time when automatic assignments are to stop. *hh:mm* is hours and minutes, it used the 24-hour clock.
StartAutoAssign=interval,*hh:mm*	Starts automatic assignments and let them run for a fixed period. *hh:mm* specifies how long automatic assignments will be used.
StartAutoAssign=ask,*text*	Starts automatic assignments and ask the sub how much time is available. *text* is the message shown to the sub. If *text* contains a comma, it must be enclosed in ". If you omit *text*, a standard text is used.

Knowing what happens

You can set up events that tells you when the period is over or when there are no assignments to do.

In [events]:

AutoAssignEnd=*procedure*	*procedure* is called when an autoassign period ends. It is not called immediately when the time is up, it is called when the active assignment is finished and you're back in a status with autoassign=1.
AutoAssignNone=*procedure*	*procedure* is called when there are no asssignments to do and the automatic assignment period is not finished.

Examples

[Status-Normal]

```
AutoAssign=1

;Work from you get home from work until 6 PM
[report-Home from work]
prestatus=On job
newstatus=Normal
StartAutoAssign=time,18:00

;Work one hour after dinner
[report-Finished diner]
prestatus=Dinner
newstatus=Normal
StartAutoAssign=interval,01:00

[report-I have some time available]
prestatus=Normal
StartAutoAssign=ask,"Then do some work for me, how much time do you have?"

;Define some events
[events]
autoassignEnd=autoEnd
autoassignNone=autoNone

[procedure-autoEnd]
message=OK, you may relax now.

[procedure-autoNone]
message=Nothing to do right now, I will be back.

; Define some extra jobs in case there are no assignments left
[job-Clean the toilet]
AutoAssign=1
Estimate=00:15

[job-Stand in the corner 10 minutes]
AutoAssign=1
Estimate=00:10
```

```
[job-Kneel on the floor 5 minutes]
AutoAssign=1
Estimate=00:5

[job-Write 50 times obey]
title=Write 50 times: "I will always obey my Master"]
AutoAssign=1
Estimate=00:30
```

Advances cloth issues

Check the subs cloth report

You can let VM check if the sub is wearing what she is supposed to wear. Do this with the Check and CheckOff keywords.

The concept

- In a clothing instruction you can use Check=stockings to tell that the sub must wear stockings.
- In a clothing instruction you can use CheckOff=panties to tell that the sub are not allowed to wear panties.
- In the cloth type stockings, you use Check=stockings.
- In the cloth type panties, you use Check=panties.
- To set up the requirements, you must use Clothing=*xxxx* to enable the requirements.

If you have done as above, and the sub makes a cloth report that does not include the cloth type stockings, then VM will display a message saying "You are supposed to wear stockings". If the sub makes a cloth report that includes the cloth type panties, then VM will display a message saying "You are not supposed to wear panties".

If you want to define your own actions instead of just a message, you can do so. You may want to punish the sub, lower the merit points or do something else.

The requirements are only activated when you use the Clothing keyword in a report, permission, procedure or anywhere else. If the sub asks for instructions himself, no requirements are set up. The reason for this is simple: The sub can ask for any instructions at any time. This is not the same as he is suddenly required to wear what he asks for. Only when the master orders a specific clothing, are the requirements activated.

How to define the cloth report.

In a instruction or set definition, you can use the following keywords.

Check=*name*	Means that if the last option before this line is chosen, then the sub is supposed to wear *name*. *name* is the name displayed to the sub in the message "You are supposed to wear *name*".
CheckOff=*name*	Means that if the last option before this line is chosen, then the sub is not supposed to wear *name*. *name* is the name

displayed to the sub in the message “You are not supposed to wear *name*”.

Examples:

```
[set-underwear]
    choice=new
        option=panties
            check=panties
        option=string panties
            check=panties
            check=string panties
        option=no panties
            check=no panties
    choice=new
        option=pantyhose
        option=stockings
            check=stockings
        option=bare legs
            checkOff=stockings
            checkOff=pantyhose
```

How to define the cloth type

In a clothtype definition you can use the Check keyword. Naturally, there is no CheckOff keyword in cloth types. Here you only tell what the sub is wearing, not what she is not wearing.

Check=*name*	Means that the cloth type is equal to *name*. *name* must be the same name as used in a Check or CheckOff. If you use Check after a value, it applies only to that value. If you use it before the first Attr keyword, it applies to the whole cloth type.

Examples:

```
[clothtype-panties]
    check=panties
    attr=color
    attr=style
        value=normal
        value=string
            check=string panties
        value=?
    attr=description

[clothtype-stockings]
    check=stockings
    attr=color
    attr=description
```

How to activate / deactivate the requirements

The requirements are only activated when you use the Clothing keyword in a report, permission, procedure or anywhere else. If the sub asks for instructions himself, no requirements are set up. The reason for this is simple: The sub can ask for any instructions at any time. This is not the same as he is suddenly required to wear what he asks for. Only when the master orders a specific clothing, are the requirements activated.

You can deactivate the requirements by using a new Clothing keyword or by using the keyword ClearCheck. The Clothing and ClearCheck keywords can be used in permissions, reports, confessions, procedures, timers or pop ups

Clothing=*name*	This line displays clothing instructions for *name*. It also activate checking of required closes.
ClearCheck=1	Deactivates the checking of the clothes.

Examples:

```
[permission-Go to work]
Pct=100
Clothing=On job
ClothReport=What are you wearing?
Newstatus=On job

[report-Home from job]
Prestatus=On job
```

```
NewStatus=Home
ClearCheck=1
```

How do define your own actions

Basically the program just inform the sub that he/she is not correct dressed. If you want to take further actions, like punishing the sub, you can use three events. See Events.

CheckOn=*procedure*	Called every time VM finds that the sub is not wearing something he should. Use the string variable $zzCheck to tell what it is.
CheckOff=*procedure*	Called every time VM finds that the sub is wearing something he is not supposed to wear. Use the string variable $zzCheck to tell what it is it is.
CheckAll=*procedure*	Called after a cloth report is made if the checks showed a difference between what the sub is wearing and what he is supposed to wear. Use the counter #zzClothFaults to tell how many violations there were.

Example 1:

```
[events]
CheckOn=CheckOn
CheckOff=CheckOff

[procedure-CheckOn]
PunishMessage=You are supposed to wear {$zzCheck}.
Punish=30

[procedure-CheckOff]
PunishMessage=You are not supposed to wear {$zzCheck}.
Punish=30
```

Example 2:

```
[events]
CheckAll=CheckAll

[procedure-CheckAll]
PunishMessage=You are wearing {#zzClothFaults} wrong items.
set#=#punishment,#zzClothFaults
multiply#=#punishment,10
```

Punish=#punishment

Setting flags in cloth reports

You can set flags when the sub makes a cloth report. This gives you the opportunity to code your own checks of what the sub is wearing. The flags are set when the sub makes a cloth report, and removed the next time a cloth report is made. Then of course new flags are set.

Use this in clothtype definitions.

Flag=*flagname*	Tells that you want the flag *flagname* raised when this is selected. If the flag keyword is before the first attribute, it is raised if the cloth type is selected. If the flag keyword comes after a Value keyword, it is raised when this value is selected.

You can also let the program automatic set flags depending on the cloth types the sub is wearing. The flags are the name of the cloth type suffixed with "_on". If clothtype "panties" is chosen, the flag "panties_on" is set.

In the [general] section you add

AutoClothFlags=1	Tells that you want automatic flags for cloth types.

Examples:

```
[clothtype-panties]
    flag=underwear_on
    attr=color
    attr=style
        value=normal
        value=string
            flag=string_on
        value=?
    attr=description
```

If the sub reports wearing string panties, the flags "underwear_on" and "string_on" are set. If the sub reports wearing normal panties, only the flag "underwear_on" is set.

If you have AutoClothFlags=1 in the [general] section, the flag "panties_on" will also be set in both cases.

Setting flags in instructions and clothing instructions

You can set flags when you give instructions to the sub. This gives you the opportunity to react on the instructions later in the program. The flags are set when you give instructions to the sub using the Instructions or Clothing keywords. the sub makes a cloth report, and removed the next time instructions are given. Then of course new flags are set.

Note: Flags are not removed and set when the sub asks for instructions. The reason is that the sub can ask for any instructions a any time. And you do not want this to mess with the flags you have set.

Use these keywords in sets:

Flag=*flagname*	Tells that you want the flag *flagname* raised when this set is selected.
OptionFlag=*flagname*	Tells that you want the flag *flagname* raised when this option is selected. Must be coded immediately after the option you want to react on.

Example:

```
[set-pants]
flag=Pants
select=all
Choice=new
        Option=Slacks
                OptionFlag=Slacks
        Option=Jeans
                OptionFlag=Jeans
        Option=Shorts
                OptionFlag=Shorts
```

When the sub is instructed to wear pants, the flag Pants are always set and one of the flags Slacks, Jeans or Shorts are set.

Send mails from the program

You can send mails from the program in two ways. You can send a mail with a subject when necessary, and you can let the program send reports automatically.

Note: To send mail you need a SMTP server (outgoing mail server). Usually your ISP (Internet Service Provider) offers you the service of a SMTP server. The problem is that because of the growing spam thread, your ISP will probably have some security requirements. Some of these requirements can be provided by VM, some can not. You will have to test it, there is no guarantee that you will succeed.

To begin with you must have some information for the SMTP server. Your ISP can provide you with this information. All mail programs (including Outlook Express and Windows Mail) use the same information, so you should be able to find the relevant information in your mail program. Except for the password.

What must be there

In the [general] section you must add these keywords:

smtp=*outgoing-mail-server*	The name of the SMTL server or outgoing mail server.
masterEmail=*email-address*	The email-address you want the mails sent to.
subEmail=*email-address*	The subs email-address to be used as the sender and reply-to address. Note: Some ISPs requires this address to be an address the server knows. In those cases you can't use a webmail address like Hotmail, Yahoo mail or GMail.

If authentication is required

Most ISPs requires the mail program authenticate itself. If yours does, you must use the following keywords (still in the [general] section).

smtpUser=*user*	The user name the server expects
smtpPassword=*password*	The password for the user

Optional

masterEmail2=*email-address*	An extra email-address you want all mails sent to. This address will receive a copy of all mails sent to the master.
smtpPort=*user*	If your ISP requires you to use a non-standard port-number for outgoing mail, you can write it here.

How to send a message

You can send messages from permissions, reports, confessions, procedures, timers or pop ups.

MasterMail=*subject*	This keyword will send a mail to the master with the subject *subject*. Included in the mail will be an extract from the report.
MasterAttach=*filename*	If you want to attach one or more files to the mail, use this keyword. Write one MasterAttach keyword for each file you want to attach.

How to automatically send a created report

You can have the program send the reports automatic when the sub chooses “Make new report” from the menu or when the the program creates a new report because of MakeNewReport in the Script. See Make a new report.

Do it by adding this keyword to the [general] section:

AutoMailReport=1	This keyword will send a mail to the master with the report attached every time a new report is made.

Line writing as jobs or punishments

You can order the sub to write lines. You can do that either as a punishment or a job. The sub will be given a number of lines to write. It can be one line to be written a number of times, or the program can chose between a number of lines.

For jobs you must define the number of lines to write. For punishments you can let the the program decide based on the value keyword and the severity of the punishment.

What must be there

Use these keywords in either a job or punishment definition. A # in the header or the title is replaced by the number of lines to write.

Type=Lines	This keyword tells that the job or punishment is a line writing assignment.
Line=*textline*	This keyword defines the line the sub must write. Replace *textline* with the phrase the sub must write. You can have as many lines as you like.
Required in jobs:	
Linenumber=*number*	The number of lines to write.

The same or different lines

If you define more than one line, you must tell the program if you want the sub to alternate between the different lines or you want him to write the same line over and over again.

Select=Random	The program will pick a random line when the job is announced or the punishment is given, and the sub must write that line the specified number of times. Note that this choice will change the name of the punishment or job that is announced to the sub to “Write # times: *textline*” where # is the number of times and *textline* is the line to write.
Select=All	The program will alternate randomly between the possible lines, so the sub will never know which line to write next.

Especially for line writing punishments

When defining a line writing punishment you must use the Value keyword to give the value of one line. This is used to determine how many lines to write.

Examples:

<u>Line writing job examples:</u>

```
[job-lines1]
Title=Write # lines about your Master
NewStatus=Writing lines
Type=Lines
Linenumber=20
Select=All
Line=I am devoted to my Master.
Line=I love how my Master dresses me.
Line=I have no free will.
Line=I have no secrets for my Master.
Line=Life is good when being controlled.
Line=I hate to disappoint my Master.
Interval=8,16
```

This example the sub must write 20 lines alternating between the lines.

```
[job-lines2]
NewStatus=Writing lines
Type=Lines
Linenumber=20
Select=Random
Line=I am devoted to my Master.
Line=I love how my Master dresses me.
Line=I have no free will.
Line=I have no secrets for my Master.
Line=Life is good when being controlled.
Line=I hate to disappoint my Master.
Interval=8,16
```

This example the sub must write 10 identical lines. A title is not necessary, as the program will supply one.

Use FTP

If you have an FTP server (File Transfer Protocol), you can use it to let the program deliver reports to the FTP server and to have the program retrieve new versions of the script or the program itself (VirMst3.exe) from the FTP server.

Using FTP requires a bit more knowledge than using mail, but you can avoid all the problems involved when using mail.

If you use FTP, the program will connect to the FTP server in these situations:

- When the program starts.
- When a new report is made, either by the sub or by the script.

When the program is connected to the FTP server, it will perform the actions you have specified. See below.

What must be there

You must define a special section for the FTP keywords.

[ftp]	This defines the start of the FTP section.
URL=*address*	The address of the FTP server. Often it looks this: ftp.xxxxxx.com
ftpUser=*userid*	The user-id that the program shall use to log on the the server.
ftpPassword=*password*	The password that the program shall use to log on the the server.
ftpServerType=*type*	The type of server you are connectiong to. *Type* must be **windows**, **dos**, **unix**, **multinet** or **vms**. Use unix for a Linux server. Usually your server will be Windows or Unix (Linux).

Using a subdirectory

You may not want to keep the files in the root of your server. You can specify a directory path where your files must be placed.

ftpDir=path	The path where the files should be placed.

Automatic update the script and/or the program

VM can automatic retrieve new versions of the script and the program itself from the server. Each time the program connects to the FTP site, it checks if a new version is available. When you have tested a new script, you upload it to the FTP site using a FTP program, and VM will find it and retrieve it to the subs machine.

UpdateScript=Restart	Tells the program to retrieve new scripts and to restart VM if a new script is found, to make it active immediately.
UpdateScript=Update	Tells the program to retrieve new script, but not restart VM. The script will become active next time VM is restarted.
UpdateScript=No	Tells the program not to retrieve new scripts. This is the same as not coding the UpdateScript keyword.
UpdateProgram=Restart	Tells the program to retrieve new programs and to restart VM if a new version of the program is found. When retrieving a new program, you must restart. The program can not be replaced without a restart.
UpdateProgram=No	Tells the program not to retrieve new program versions. This is the same as not coding the UpdateProgram keyword.

Uploading reports

You can let VM upload the report to the FTP server whenever the sub uses the “Make new program” menu or you use MakeNewProgram=1 in the script.

SendReports=1	This will upload all reports to the FTP server.

Other FTP options

FtpLog=1	This will show all FTP communication in the report. This is useful for testing the FTP-connection and identifying problems, but should be turned off when you are using the script for real. Turn it off by changing to FtpLog=0 or removed the line.
TestFtp=0	This will suppress uploading and downloading via FTP unless you run from an encrypted script or have

Restrict=1 in the [General] section. This is useful when you are testing other parts of the script.

An FTP example:

```
[ftp]
  URL=ftp.virmst.eu
  ftpUser=myuser
  ftpPassword=mypassword
  ftpDir=/virmst/slaves
  ftpServerType=unix
  UpdateScript=Restart
  UpdateProgram=Restart
  SendReports=1
  ftpLog=0
  testFtp=0
```

Using a web cam

The program can control a web camera. Using a web camera can increase the accountability and gives a new way of monitoring the sub, when Virtual Master is used in a long distance relationship. The web cam can be controlled with a few but powerful keywords.

You can use email or FTP to send the pictures to yourself.

Use of a web cam requires DirectX 9.0 to be installed.

The program have bin tested with a Logitech Quickcam, but I imagine that most cameras can be used.

Monitoring a status

One way to use a web cam is to check randomly that the sub is doing what he/she is supposed to do. If the sub is ordered to stand in the corner, write an essay, scrub the floor or any other activity that occurs in a room with a web cam, you can let the program take pictures at random intervals to make sure that the order is followed.

Use these keywords in a status definition:

CameraInterval=*min*,*max*	Tells how often the program should take pictures. Specify a minimum and a maximum interval on the form hh:mm:ss or just hh:mm.
PointCamera=*text*	Write a text that directs the sub to point the camera in the desired direction. A live picture will be shown to the sub, to make him/her able to see what the camera is pointing at.

<u>Example:</u>

[status-Standing in the corner]

PointCamera=Point the camera at the corner

CameraInterval=00:03,00:10

This will ask the sub to point the camera at the corner when the status starts. And it will take pictures at random intervals with a minimum of 3 minutes and a maximum of 10 minutes between each picture.

Having the sub pose for the camera

You can have the sub pose for the camera. Use this to verify that the sub is dressed as ordered or for any other reason you wish.

When the sub is ordered to pose, he/she will be shown a live picture from the camera and a count down from 6 seconds. This gives the sub 6 seconds to pose. When the 6 seconds has passed, a picture is taken. The sub is asked whether the pictures is OK. If not, a new count down is given. This continues until the sub reports that the picture is OK. All pictures will be send to the master, if automatic send is used. The reason for the short time frame is that the sub will not have time to change clothes.

I suggest you use posing in two ways:

- Let the sub pose after changing clothes or maybe after each cloth report.
- Have pop ups at random intervals that orders the sub to pose.

This keyword can be used in reports, confessions, permissions, procedures, timers and pop ups.

PoseCamera=*text*	Orders the sub to pose. *Text* is the order that is shown to the sub.

Example 1:

[report-I have changed clothes]

ClothReport=What are you wearing now?

PoseCamera=Pose for the camera.

Example 2:

[popup-pose]

PoseCamera=Pose for the camera.

How to send pictures

You can send pictures taken with the PoseCamera keyword, as mail attachments or upload them via FTP. Even pictures marked as “Not OK” will be sent. This is to give you as much information as possible.

If you do send mails in the script, all pictures will automatic be attached to the next mail sent.

To upload pictures via FTP, you must add SendPictures=1 to the [FTP] section of the script.

SendPictures=1	If you put this keyword in the [FTP] section of the script, all pictures will be uploaded via FTP next time a connection is made.

Other options

Playing sounds

You can let the program play sound files using the Sound keyword. The sound files must be in the Wave format (.WAV). You can use the Sound keyword in reports, confessions, permissions, procedures, timers and pop ups.

sound=*wave file*	The name of a sound file to be played

Showing pictures

You can show a picture to the right of the text in the main window. The status controls if a picture is shown or which picture. Use the Picture keyword in a status definition. Only JPG files are supported, and the file extension must be .jpg or .jpeg. Some GIF files will work, and must have an extension of .gif. But there is no guarentee, and a .gif file may cause the program to fail.

Picture=*picture file*	The name of an image file. Files with extensions other than .jpg, .jpeg and .gif are ignored. You can have a number of picture keywords for a status. If you have, the program will select a random picture. You can also use variables in picture names.

Examples:

```
[status-cleaning]
Picture=cleaning1.jpg
Picture=cleaning2.jpg
Picture=cleaning3.gif
Picture=work*.jpg
Picture=c:\images\cleaning\*.jpg
Picture={$picname}
Picture={$picture}.jpg
```

Starting files and other programs

You can launch a file or start a program from within VM.

Launch=*filename*	Will open the *filename* using the program that Windows has registered as the program to open this file type.

Launch=*program*,*parameters*	Will start the program *program* with the parameters *parameters*. You can omit *parameters* if they are not relevant. Parameters can be a file to be opened by *program* or whatever *program* needs.

Writing to the report file

It is possible to write your own lines to the report file. You can do it in reports, confessions, permissions, procedures, timers and pop ups.

WriteReport=*text*	Writes *text* to the report. Use this if you want to make a note to yourself in the report, or to add an eye-catcher. You can have more than one WriteReport line. If you do, all lines will be shown in the report.

Make a new report

It is possible to make a new report, just as if the sub had chosen Make new report from the File menu. You can do it in reports, confessions, permissions, procedures, timers and pop ups.

MakeNewReport=1	Does exactly the same as if the sub had chosen Make new report from the File menu.
MakeNewReport=2	Does the same as MakeNewReport=1, except it will not show a message box to the sub.

If you use the mail facility, you can have the program send the reports automatic when the sub chooses “Make new report” from the menu or when the the program creates a new report because of MakeNewReport in the Script.

Do it by adding this keyword to the [general] section:

AutoMailReport=1	This keyword will send a mail to the master with the report attached every time a new report is made.

See also Send mails from the program.

The rules menu

The Rules menu where the sub can see which rules he must obey, is maintained automatic. You can, however, control it in a few ways.

Omitting permissions, reports and instructions from the Rules menu

There may be reports, permissions or instructions that you don't want to appear in the Rules menu.

Rules=0	Use this in a report, permission or instruction to tell that this report/permission/instruction should not appear in the Rules menu.

Creating your own rules

You can add your own rules to the Riles menu. These rules are only instructions to the sub, they have no effect on how the program functions.

[rule-*name*]	Use this header to define a rule.
Text=*textline*	Use this to write extra text lines to explain the rule to the sub. You can have many lines with the Text keyword.

Minimizing or closing the program

You can minimize or close the program by use of the PgmAction keyword. You can use it in reports, confessions, permissions, procedures, timers and pop ups.

PgmAction=Close	When this keyword is encountered, the program will close.
PgmAction=Minimize	When this keyword is encountered, the program will be minimized.

Setting the program on stand by

You can set the program on stand by. When the program is on stand by, all deadlines will be pushed ahead. You can use this to allow the sub to go on vacation or close the program for some time for other reasons.

Note that stand by does not affect started activities. Long running assignments, status with a MinTime or MaxTime, flags with an ExpireProcedure etc. will not be affected.

To set the program on stand by, use this keyword. You can use it in reports, confessions, permissions, procedures, timers and pop ups.

PgmAction=Pause	When this keyword is encountered, the program will close and it will be on stand by until it is opened again.

Quick reports

A quick report is a report that the sub can make quickly. Either by clicking a large button on the screen or by pressing a function key (F9-F12). You can define one global quick report and up to 2 local quick reports attached to a status. You will automatic get one quick report when you have a started task that is not long running.

The global quick report

The global quick report is defined in the [General] section. You can use it to quickly close the program in case of unexpected guests or for whatever purpose you want. The global quick report can be called by pressing the F12 key.

Add this keywords to the [General] section:

QuickReport=*reportname*	Tells that *reportname* should be used as the quick report. *Reportname* must be defined as a normal report.
QuickLabel=*text*	Optional. Defines the text shown on the quick report button. If you omit QuickLabel, the name of the report will be used.

The assignment quick report

You will automatic get an assignment quick report button when you have started an assignment that is not long running. The assignment quick report can be called by pressing the F11 key.

You need not do anything. However, you may find that the generated label for the button is too long, and may want to define a shorter one.

Add this keyword to the job or punishment definition:

DoneLabel=*label*	Tells that the assignment button should be labelled *label.*

Sometimes you may want to hide the assignment quick report button and disable the F11 functionality.

DoneButton=0	This will hide the button and disable F11.

The local quick reports

You can have 1 or 2 local quick reports for each status. The local quick reports are defined in the status sections. Use it in status where the sub always or most of the time will be using the same report to end the status.

Add this keyword to the status definition:

QuickReport=*reportname*	Tells that *reportname* should be used as the quick report. *Reportname* must be defined as a normal report.

If you use EndReport to define a report to end a status, it will automatic become a quick report.

Changing font size

If you wish, you can change the font size of some of the text boxes and buttons. You need to use the [Font] section like this:

[font]

TextSize=*size*	Sets the general font size in the program to *size*. If you do not code this keyword, the size will be 8.
ButtonSize=*size*	Sets the size of the labels of buttons to *size*. If you do not code this keyword, the size will be 8.

The size of menus will follow your Windows setting, and can not be changed here.

Hiding the program

You may need to hide the program on the computer, even when it is active, so that by passers will not see it. Use this if others have access to your computer and you don't want them to know that you are using Virtual Master.

Setting a password on the program

You can set a password on the program, so that every time it is opened it will ask for a password before displaying anything.

Another way of using the password is to make the slave feel more submissive. If he has to enter “i am a slave” every time he starts the program, it may help making him feel more like a slave.

In the [general] section add

OpenPassword=*password*	Sets the password for the program to *password*.

Example:

```
[general]
OpenPassword=i am a slave
```

Minimize to the system tray

When you minimize the program, you may want it to go to the system tray (the icons on the right side of the task bar) and not to the task bar itself. Add these keywords to the [General] section.

UseIcon=1	Minimizes to the system tray and not to the task bar. If you use OpenPassword, the program will ask for the password when it is restored.
StartMinimized=1	If you use this keyword, the program will always start minimized.

How to handle pop ups when minimized

MinimizePopup=Ignore	If you use this keyword, pop ups are not possible when the program is minimized.
MinimizePopup=Restore	If you use this keyword, a minimized program will be restores if a pop up occurs.

Variables

Variables are used as the programs "memory". They provide you with a tool to save some information and use it later.

There are three kind of variables:

- **Flags** are conditions that can be true or false.
- **String variables** are text strings, that is words.
- **Counters** are numbers.

Flags

A flag is a condition, that can be true (the flag is set) or false (the flag is removed). Flags can (like status) be used to control which permissions, reports etc. are possible. They can also be used to control how long a condition is true (e.g. how long the sub may wear panties).

Set and remove flags:

You can set and remove flags in reports, confessions, procedures, timers and pop-ups. You can also set a flag when a permission is granted.

SetFlag=*flagname*	Sets the flag *flagname* when the status begins. You can specify several flags separated by commas. You can have more than one SetFlag line.
RemoveFlag=*flagname*	Removes the flag *flagname* when the status begins. You can specify several flags separated by commas. You can have more than one SetFlag line.

Use flags:

Just like PreStatus, you can use flags to decide when something is possible. Sometimes you can also usem to decide what happens, like if a permission is granted or denied. You can use If and NotIf in permissions, reports, confessions, procedures, timers, pop-ups and sets. Combined with procedures, you can control almost anything the program can do.

If=*flagname,flagname...*	This is only possible if all the flags in the list are set. You may have more than one If line. If there are more than one line, it is possible if any line is true.
NotIf=*flagname,flagname...*	If all of the flags in the list are set, this is not possible. You may have more than one NotIf line. If there are more than one line, it will be impossible if any line is true.
DenyIf=*flagname,flagname...*	Can be used in permissions. If all the flags in the list are set, the sub is denied permission. There may be more than one DenyIf line, if any line is true then the sub is denied permission.

PermitIf=*flagname,flagname...*	Can be used in permissions. If <u>all</u> the flags in the list are set, permission is given. There may be more than one PermitIf line, if <u>any</u> line is true then permission is given.

Define flag attributes:

You need not define flags to use them. You can use the above keywords without having defined the flag specifically. But you can define a flag with some special characteristics (attributes), if you need to.

[flag-*name*]	This line starts a flag definition.
text=*text*	When the flag is set, show *text* in the instruction window along with the status instructions. You can have more than one *text=* line.
Duration=*hh:mm*	When *hh:mm* hours and minutes has passed, the flag is automatically removed. You can add two values, Duration=*hh:mm,hh:mm*, and the program will pick a random value between the two.
ExpireMessage=*text*	If the flag is removed because the duration is up, the message *text* will be shown.
ExpireProcedure=*procedure*	If the flag is removed because the duration is up, the procedure *procedure* will be called.
SetProcedure=*procedure*	When the flag is set, the procedure *procedure* will be called.
RemoveProcedure=*procedure*	When the flag is removed, the procedure *procedure* will be called. The RemoveProcedure is not called when a flag expires. Instead the ExpireProcedure is called.

Predefined flags:

Some flags are set automatically. They are the names of the day and month: monday, tuesday, january, february etc. Note that it is always the English names that are used. They are not translated. A flag indicating the day of the month is also set, it is called "day*nn*", where *nn* is the day. Examples: day01 is the first day of the month, day15 is the 15th etc. In the report you can see which flags are set automatic when the program starts and after midnight.

String variables

String variables are another kind of memory than flags. Where flags are simply yes/no's, string variables are words to be remembered. You can ask the sub for some text or you can use some of the predefined variables.

The name of a string variable must start with $.

Setting a string variable:

set$=$*name*,*text*	The value of the variable $*name* is set to *text*.
input$=$*name*,*question*	The sub is asked the question *question*, and the answer is placed in the variable $*name*.

Using a string variable:

You can use string variables in texts, messages etc. you write in the script. Simply write the variable name enclosed in { }. Remember the $ sign in the beginning of the name.

You can compare two string variables and use the result as if it were a flag. See Using variables in If and NotIf.

Removing a string variable:

drop$=*$name*	Removes $*name* completely.

Examples of string variables:

```
input$=$FirstName,What is your first name?
input$=$LastName,What is your last name?
set$=$name,{$FirstName} {$LastName}
message=Hello {$name}.
input=Where do you live, {$name}?
```

Predefined string variables:

There are some predefined variabels you can use. Predefined variables always start with $zz.

$zzSubname	gives the name of the sub as specified with the Subname keyword in the [General] section. If you have specified more than one subname, $zzSubname will chose one at random.
$zzMaster	gives the name of the master as specified with the Master keyword in the [General] section.
$zzDate	gives the date.
$zzTime	gives the time.
$zzReportFile	gives the filename of the last report made.
$zzPVersion	gives the version of the program.
$zzSVersion	gives the version of the script.
$zzName	gives the name of the object*. Name is what is in the header record between [and]. Usually you should not use $zzName in text to the sub. Use $zzTitle instead.
$zzTitle	gives the title of the object*. Title is the name that the sub sees. If there is no Title keyword in the object, the name is used.
$zzReport	gives the name of the last or current report given. Consider using $zzTitle instead of this.
$zzPermission	gives the name of the last or current permission asked. Consider using $zzTitle instead of this.
$zzStatus	gives the name of the current status. Consider using $zzTitle instead of this.
$zzAnswer	gives the name of the last answer from a question
$zzCheck	gives the name of the check that caused a CheckOn or CheckOff event

*) An object is a status, report, permission, assignment, job or whatever you are working on right now, where the variable needs to be translated.

Counters

Counters are used to store numbers. You can do simple calculations on counters.

The name of a counter must start with #.

Setting a counter:

set#=#*counter,value*	Sets the counter #*counter* equal to *value*. *Value* can be a number or another counter.
input#=#*name*,*question*	The sub is asked the question *question*, where the answer must be a number. The answer is placed in the variabel $*name*.

Calculating on counters:

set#=#*counter,value*	Sets the counter #*counter* equal to *value*. *Value* can be a number or another counter.
add#=#*counter,value*	Adds *value* to #*counter*. *Value* can be a number or another counter. The result is stored in #*counter*.
subtract#=#*counter,value*	Subtracts *value* from #*counter*. *Value* can be a number or another counter. The result is stored in #*counter*.
multiply#=#*counter,value*	Multiplies #*counter* with *value*. *Value* can be a number or another counter. The result is stored in #*counter*.
divide#=#*counter,value*	Divides #*counter* with *value*. *Value* can be a number or another counter. The result is stored in #*counter*.

Generating a random value:

You can assign a random value to a counter.

random#=#*counter,maxvalue*	Sets the counter #*counter* to a random value between 1 and *maxvalue* (both included). *Maxvalue* can be a number or another counter.
random#=#*counter,minvalue,maxvalue*	Sets the counter #*counter* to a random value between *minvalue* and *maxvalue* (both included). *Minvalue and Maxvalue* can be a number or another counter.

Removing a counter:

drop#=#*counter*	Removes #*counter* completely.

Using a counter:

You can use counter variables in texts, messages etc. you write in the script. Simply write the variabel name enclosed in { }. Remember the # sign in the beginning of the name.

You can compare two counters or a counter with a number and use the result as if it were a flag. See Using variables in If and NotIf.

Examples of counters:

```
set#=#MonthPerYear,12
input#=#age,How old are you?
set#=#month,#age
multiply#=#month,#MonthPerYear
message={#age} years equals {#month} month.
set#=#left,100
subtract#=#left,#age
message=In {#left} years you will become 100 years old.
```

Predefined counters:

There are some predefined counters you can use. Predefined counters always start with #zz.

#zzMerits	gives the subs merit points.
#zzMaxMerits	gives the maximum possible merit number (top of the bar).
#zzMinMerits	gives the minimum merit number (bottom of the bar).
#zzYellow	gives the number of merits below which the bar turns yellow.
#zzRed	gives the number of merits below which the bar turns red.
#zzBlack	gives the same as #zzMinMerits.
#zzBeginMerits	gives the merit points at the beginning of the current day.
#zzDay	gives the day of the month (1-31).
#zzMonth	gives the month (1-12).
#zzYear	gives the year.
#zzLeapYear	gives 1 if you're in a leap year and 0 if not. Add this value to 28, and you have the number of days in February.
#zzHour	gives the hour of the day (0-24).
#zzMinute	gives the minutes (0-59).

#zzSecond	gives the seconds (0-59).
#zzSecondsPassed	gives the seconds passed since midnight.
#zzDaysPassed	gives the number of days passed since 1899-12-31.
#zzClothFaults	gives the number of checks that failed and have caused a CheckAll event.
#zzPunishmentSeverity	Gives the severity of the last punishment given.

These variables are only available when in a relevant context:

#zzPunishmentNumber	Can be used in a BeforeProcedure, StartProcedure, DoneProcedure, AbortProcedure and DeleteProcedure for a punishment to give the number of iterations, minutes, hours or days. In other words, what # in the punishment name is translated to.
#zzLate	gives the number of minutes late in a SlowProcedure.
#zzEarly	gives the number of minutes early in a QuickProcedure.
#zzBeforeMerits	gives the subs merit points before change. Can only be used in a MeritsChanged event.
#zzAddMerits	In a MeritsChanged event gives the number that the merit points have changed. Or gives the number of merit points given when an assignment is done, a report is made etc.

Time variables

Time variables are used to store dates, times and time intervals. You can do simple calculations on time variables.

The name of a time variabel must start with !.

Time constants

In the following is mentioned time constants. A time constant can be written in one of the following ways:

- hh:mm
- hh:mm:ss
- nd

where hh is hours in the 24 hour clock, mm is minutes, ss is seconds, n is the number of days and d is the letter “d”.

Examples:

• 08:00	8 am or 8 hours.
• 13:00	1 pm or 13 hours.
• 14:30	2:30 pm or 14 hours 30 minutes.
• 00:02	2 minutes.
• 00:02:30	2 minutes 30 seconds.
• 2d	2 days

Setting a time variable:

set!=!*variable,value*	Sets the time variable #*variable* equal to *variable2*. V*alue* must be a time constant or another time variable.

Calculating on time variables:

set!=!*variable,value*	Sets the time variable #*variable* equal to *value*. V*alue* must be a time constant or another time variable.
add!=!*variable, value*	Adds *value* to *!variable*. V*alue* must be a time constant or another time variable. The result is stored in *!variable*.
subtract!=!*variable, value*	Subtractss *value* from *!variable*. V*alue* must be a time constant or another time variable. The result is stored in *! variable*.

multiply!=!*variable, value*	Multiplies *!variable* with *value*. *Value* must be a number or a counter. The result is stored in *!variable*.
divide!=!*variable, value*	Divides *!variable* with *value*. *Value* must be a number or a counter. The result is stored in *!variable*.

Rounding a time variable:

You round a time variable.

round!=!*variable,base*	Rounds the time variable *!variable* to size specified by *base*. *base* must be a time variable or constant.

Examples:

• round!=!hours,01:00	Rounds !hours to a whole number of hours.
• round!=!used,00:10	Rounds !used to a multiply of 10 minutes.

Generating a random value:

You can assign a random value to a time avriable.

random!=!*variable,!mintime,!maxtime*	Sets the time variable *!variable* to a random value between !*mintime* and !*maxtime*. *!mintime and ! maxtime* must be time variables or constants.

Removing a time variable:

drop!=*!variable*	Removes *!variable* completely.

Using a time variable:

You can use counter variables in texts, messages etc. you write in the script. Simply write the variabel name enclosed in { }. Remember the ! sign in the beginning of the name.

There are different ways of showing a time variable in a text. It can be formattet as a date, a time, date and time or an interval. VM will try to guess how you wished it formatted depending on the size of the variable. However, you may wish to decide the format yourself. You decide the format like this:

• {!variable,d}	will always be shown as a date.
• {!variable,t}	will always be shown as a time of the day.
• {!variable,i}	will always be shown as an interval.

You can compare two time variables or a time variable with a time constant and use the result as if it were a flag. See Using variables in If and NotIf.

Predefined time variables:

There are some predefined time varfiables you can use. Predefined time variables always start with !zz.

!zzDate	gives the date.
!zzTime	gives the time of the day.
!zzDateTime	gives the date and time of the day (same as zzNow).
!zzNow	gives the date and time of the day (same as zzDateTime).
!zzDate	gives the date.

These variables are only available when in a relevant context:

!zzMinTime	gives the minimum time of an assignment.
	gives the minimum date and time when an assignment to be done (!zzStartTime + !zzMinTime).
!zzMaxTime	gives the maximum time of an assignment.
!zzMaxTimeEnd	gives the maximum date and time when an assignment must be done (!zzStartTime + !zzMaxTime).
!zzStartTime	gives the time an assignment is started.
!zzRunTime	gives the time passed since an assignment was started.
!zzDeadline	gives the time when an assignment must be finished.
!zzRemindTime	gives the time when the sub next time will be reminded of an assignment.
!zzLate	gives the time late in a SlowProcedure.
!zzEarly	gives the time early in a QuickProcedure.

Using variables in If and NotIf

You can compare two string variables and use the result as if it were a flag.

You can compare two counters or a counter with a number and use the result as if it were a flag.

If the comparison is true the program will act as if a flag was set. If the comparison is false the program will act as if a flag was not set.

Examples:

```
if=#try<3
if=$answer=yes
DenyIf=#zzmerits<=40      (same as DenyBelow=#zzmerits,40)
```

You can use the following operators:

=	Equal to (for string variables ignoring case)
<>	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

You can use the following extra operators vith string variables only:

==	Exactly equal to (case sensitive)
[	Part of (ignoring case)
[[	Part of (case sensitive)

Examples:

```
set$=$Master,Master
set$=$line,master
if=$Master=$Line          returns true
if=$Master==$Line         returns false

set$=$line,Yes master
if=$Master[$Line          returns true     ($Line contains “master”)
if=$Master[[$Line         returns false    ($Line does not contain “Master”)
if=$Line[$Master          returns false    ($Master does not contain “Yes master”)
```

Divide your script in parts

If you have a very large script, you may want to divide it into different parts, each containing a specific subject. You can do that by using the **%include** keyword. You can add %include anywhere in the script.

%include=*filename*	copies the file named by *filename* into the script at the position of the %include line. %include must always begin a new line.

Note: If you encrypt a script containing %include, the program will generate one encrypted script containing all the files from the original script. Thus you need only distribute one file to the sub. However, if you chose to run the unencrypted script, all the files must be available to the sub. Otherwise, the sub will not be able to start Virtual Master.

Foreign language translation

The program can be translated to a language of your own choice. To translate do the following steps:

1. Write your script file in your preferred language. Remember not to translate the keywords, only the text that the sub sees may be translated.
2. Choose a name for your language file, where you will do the translations. I suggest your use the language followed by **.txt**. Examples: frence.txt, francais.txt, german.txt, deutch.txt.
3. In the [general] section of your script, add the line *language=filename* (where *filename* is the name you chose).
4. Run the program. It will still be in English. Try as many different situations as possible.
5. Close the program.
6. Edit your language file. You will find only one section with the name [missing].
7. Change [missing] to [language].
8. Change the text on the right side of the equal signs (=) to your own language.
9. When finished translating, run the program. You will find that it is translated. If the programs discovers a new phrase, that hasn't been translated, it will add a new [missing] section to the language file and add the new phrases. Your job is to translate the phrase and move the lines from the [missing] to the [language] section. You can leave the empty [missing] section if you want.

Notes:

- There may only be one [language] section in the language file. So you only change the name once.
- You will notice an ampersand (&) in many of the texts that are menu entries and button labels. This indicates that the letter after the ampersand (&) will be underlined and used as a windows hot key (used with the Alt key). Move the ampersands to some suitable letters in your language.
- Sometimes I add new functionality and forget to call the language module. Which means that some phrases will not be translated. If you find some phrases that are not translated, mail me, and I will correct the mistake.
- I know that I have a problem with Yes and No buttons, which are not translated. There's no need to write me about these. Maybe some day in the future I will solve this problem.

Testing your script

The test menu

You can add a Test menu to the menu line, by adding the keyword TestMenu=1 to the [General] section. The Test menu will help you test your scripts.

If you run from an encrypted script or if you have Restrict=1 in the [General] section, the test menu will always be hidden.

NOTE: When you test a script, always use a different folder from when you run the program, if you make scripts for yourself. Once you have manipulated with time, you can not turn it back without deleting the status file.

TestMenu=1	Adds the Test menu to the menu line, unless you run from an encrypted script or have Restrict=1.

The test menu

Add	Simulate that time passes. Note: This simulation does not work for pop ups. To test pop ups, use Quick pop ups (see below). It will work for almost anything else.
Randomize	If this is checked, all instructions and clothing instructions will generate a new result, even if you have Change=Daily or Change=Program.
Quick pop ups	Will make pop ups pop up every 5 seconds, if you are in a status that allows pop ups. Use this to test your pop ups.
Always permit	Permissions will always be permitted.
Always deny	Permissions will always be denied.
Change status	Can be used to change to any status possible.
Launch job	Will add a specific job to the assignment list, thus giving you the possibility to test it.
Give punishment	Let you test a punishment punishment.
Export clothes	Export the subs entered clothes to a file.
Import clothes	Import clothes from a file made by Export clothes.

Avoiding Internet connection when testing

You may not be interested in sending mails or using FTP when testing your script.

Add the line
TestMail=0
to the [General] section of your script. This will suppress sending of mails unless you run from an encrypted script or have Restrict=1 in the [General] section.

Add the line
TestFtp=0
to the [FTP] section of your script. This will suppress uploading and downloading via FTP unless you run from an encrypted script or have Restrict=1 in the [General] section.

Avoiding entering the open password when testing

You may not be interested in typing the open password when testing your script. Add the line

TestPassword=0

to the [General] section of your script. This will ignore the openpassword unless you run from an encrypted script or have Restrict=1 in the [General] section.

Making reports only for test usage

You can make reports that only works when not in sub mode.

Add the line
Test=0
to the report. This will ignore the report if you run from an encrypted script or have Restrict=1 in the [General] section.

Debug help

This is not for common use. In case of problems with the status file, you may be asked to supply some statistical information about it to help me finding the problem. In that case, add the line

Statistics=1

to the [General] section of your script. This will give you a new menu entry call Statistics.

What's new

What's new in 3.1

A lot! Version 3 have a complete rewrite of syntax compared to version 2 and 1. Thus it is not possible to use scripts made for version 1 or 2 with version 3 of the program. Because of the great changes, I have not made a list of what is changed.

What's new in 3.2

- Use of FTP.
- Use of a webcam.
- Line writing as a job or punishment.
- QuickReport in status definitions. Automatic quick reports for assignments.
- Weight can be used in punishments and pop ups.
- You can define your own rules to show in the Rules menu.
- You can now use Latest and Earliest in permissions, confessions and procedures.
- You can now use special flags in a BeforeProcedure on a permission to tell the program whether to permit, allow or cancel the permission. The special flags are zzDeny, zzPermit and zzCancel.
- New keyword for status and popup groups:
    - PopupAlarm
- New keywords for jobs:
    - FirstInterval to make first interval shorter than the normal interval.
    - OneTime to make the job run only once.
    - AnnounceProcedure is called when the job is announced.
- New keywords for permissions:
    - ForgetProcedure
    - IgnoreProcedure
    - ForgetConfession=0 and IgnoreConfession=0 in the [General] section-
- New keyword for punishments
    - AnnounceProcedure is called when the punishment is given.

- New keywords for instructions and clothing instructions:
    - Rules to control whether the instructions are shown in the Rules menu.
    - Askable to control whether the sub can ask for instructions.
- Use %include to divide your script into parts.
- Text=%instructions and Text=%clothing will show latest (clothing) instructions in the message area.
- New events:
    - StartFromPause
    - JobAnnounced
    - ForgetConfession
    - IgnoreConfession
    - MeritsChanged
- Variable improvements:
    - Time variables
    - A new Random# keyword to generate random values.
    - New string operators: ==, [and [[.
    - New standard variables.
- Several keywords have been changed so you can specify one or two values. If you specify two values the program will choose a random value between the two. One or both values can be replaced by variables.
- The restriction that a procedure can not be called twice have been changed a bit. A procedure can still not call itself, but a procedure can call another procedure twice. If it happens, you will now get a message in the report.
- Test menu improvement:
    - Change status
    - Launch job
    - Give punishment

About Foreign language translation:

The way that foreign language translation works internally, have been changed between 3.1 and 3.2. This means that you may have to retranslate some phrases. The reason for this change is to make it easier for myself to maintain the foreign language facility in the future.