



User Script Management

Abstract:

This technical note describes a safe and easy methodology to cleanly start and stop user applications, in the SIXNET IPm, through the use of hard and soft resets and startup scripts.

This information is applicable to IPm firmware v1.6 and I/O Tool Kit v2.2 or later.

Basic Scripting

This section attempts to explain some of the scripting issues that are specific to the SIXNET IPm. Although the IPm does indeed run Linux as its operating system, scripts are handled a little differently. Embedded systems have limited resources (processing power, available memory, disk space, etc). We needed to reduce some of the overhead that is typical of a Linux workstation. Changes were also made to simplify the process for users. The goal of this section is to provide the user with an understanding of how the IPm processes user scripts, and also to provide some basic guidelines for writing user scripts.

Users are encouraged to write their scripts so that they are independent. Just as in writing programs, care should be taken so that scripts do not interfere with other scripts or programs. A single IPm could be running multiple vendors' software. Ideally, each vendor application should operate independently.

Simple scripts, like the one described below, will work for most. Some users will need more control over their programs. Those users are encouraged to write scripts that use conditionals and check the return status of their programs. If a program depends on another program exiting successfully prior to it being started, it would be a good idea to check the status of the exiting program prior to starting the next program.

Anatomy of a script

For the purpose of this document, Bourne shell scripts will be used. There is native support for other scripts, namely perl. The IPm uses Ash, which is a Bourne shell derivative. It is important to recognize that Ash, Bash, and Korn, are Bourne shell derivatives. Each shell may interpret syntax a little differently. Just because a script works fine in a PC does not mean that it is supposed to work in an IPm. Most Linux distributions use Bash as their Bourne shell. Bash and Ash are different and it may be necessary to modify a perfectly good working script to work in an IPm. It is recommended that users reference the 'ash' manual or online material specific to 'ash'.

An example of a very basic script:

```
#!/bin/sh
/usr/local/bin/myprog
```

The first line, '#!/bin/sh', is very important. It tells the system where to find the interpreter. The second line runs the program 'myprog'.

Notice that the full path to 'myprog' is provided. This is desirable for a couple of reasons. First, there is not any question what version of 'myprog' is being run. If the full path was not given, and a version of 'myprog' existed in /usr/bin, the version of 'myprog' in /usr/bin would be run. The PATH environment



variable determines the search order for applications. Secondly, the script is more efficient as a result of not having to process additional commands.

```
#!/bin/sh
cd /usr/local/bin
./myprog
cd /
```

The above mentioned script is equivalent to the first script. However, it has two additional commands (cd /usr/local/bin and cd /) that need to be parsed and interpreted. On a Linux PC, the additional commands may appear to not affect the performance of the system. However, the additional commands will be noticeable in an IPm, especially if there are many user scripts doing similar operations.

Making scripts run in Linux

Linux and Windows create text files differently. Linux places a single 'line feed' at the end of each line. Windows places a 'carriage return' followed by a 'line feed' at the end of each line. Linux does not particularly deal well with Windows text files. The following command will put a script (text file) in the proper format:

```
# dos2unix -u myscript.sh
```

The command listed below will remove any 'carriage returns' from the file 'myscript.sh'. This will put 'myscript.sh' in Unix file format.

It may also be necessary to set the script as executable. The following command should be sufficient for most applications:

```
# chmod 755 myscript.sh
```

To '&' or not to '&', that is the question

Ampersands are used to force programs that normally run in the foreground, into the background. A process is said to be in the foreground when it is interacting with the user. A process that does not require interaction with the user to run is called a background process. While a background process runs, the user can continue using other programs or commands. If a program requires input from the keyboard, it will not work in the background.

Most programs that run in an IPm will be running in the background. Some programs are written in such a way that they are automatically put in the background. An '&' is not needed. Other programs that were written to run in the foreground will need the '&'.

The following two lines are equivalent. The space between the command and the '&' is not necessary. However, it does make it more readable.

```
/usr/local/bin/myprog &
/usr/local/bin/myprog&
```



Typically, adding an ‘&’ to the end of a program, that automatically runs in the background, does not have any effect. By adding an ‘&’, you are telling the parent process to start in the background. Once the child has been spawned, the parent exits in the background.

Using ‘sleep’ (or ‘usleep’)

The use of ‘sleep’ is not recommended in any startup or shutdown script. Special care must be taken when calling sleep or usleep within any script. If a program must sleep, it is best to incorporate the sleep in the program itself. If a user program is using the SIXNET I/O database or tag database, it *must* sleep for 5 seconds prior to making any calls. This gives the databases enough time to populate the appropriate tables and come online. Because many applications are started up in parallel during the boot process, the order of processes in the process table is not necessarily deterministic.

Take the following script:

```
#!/bin/sh
sleep 5
/usr/local/bin/myprog &
/usr/local/bin/myotherprog &
```

Assume that a user wants to start ‘myprog’ after a 5 second delay. The above mentioned script will indeed wait five seconds before starting ‘myprog’. However, ‘myotherprog’ will also wait that same 5 seconds. It is true that this could be rectified by starting ‘myotherprog’ prior to the 5 second sleep. However, since the sleep is occurring in a user startup script, any subsequent user scripts will be forced to wait the same 5 seconds. This behavior will slow the startup time by 5 seconds and has the potential to cause conflicts.

i.e. Assume that an IPm has 3 programs (A, B, and C). Each program may have been written by a different person. Each program is started by a separate startup script. Program ‘A’ is started immediately. Program ‘B’ is started second and has a five second delay in the script. Program ‘C’ is started last. If program ‘A’ expects program ‘C’ to come online within 3 seconds, program ‘A’ may fail because program ‘C’ needs to wait for program ‘B’s 5 second delay. Subsequently, program ‘C’ may fail. By placing a 5 second sleep in program ‘B’, and eliminating the 5 second sleep in the user script, all three programs would have started normally. Only program ‘B’ would have slept for 5 seconds. Program ‘C’ would have started almost immediately and the system’s startup time would remain relatively unchanged.

Script Management

The IPm uses a simplified scheme, with pieces borrowed from the popular ‘BSD’ and ‘SYSV’ script schemes (although it does not support the full range of either of these more complex schemes.) The IPm scheme is fairly easy to understand, as it has only a small number of scripts to read and the order in which things are started is quite clear. Similar to the BSD-style scheme, it is intended to be fast, simple and efficient. However, the IPm uses some of the security features of the SYSV-style scheme, where users are encouraged to create individual scripts as apposed to editing the master scripts.

Users should not be editing master scripts (rc.init, rc3.user, softreset, and hardreset) unless they really know what they are doing. Users have always been able to edit the existing rc3.user, softreset, and hardreset scripts.



Such editing is dangerous, as incorrect startup and shutdown scripts could produce undesirable behavior. A simple mistake by the user can lead to an unstable system. Those files are also volatile in the sense that they will be over-written when a firmware load is done. Any changes made by a user will be lost.

Implementation

User scripts can use any filename, however, it is recommended that users respect the naming conventions outlined below. In order for the script to be recognized, it must end with the **‘.sh’** file extension. The **‘.sh’** file extension typically identifies a Bourne or Bourne-like (Ash, Bash, Korn, etc) shell script. If the user script is written in Perl, it still must have a **‘.sh’** file extension.

It is important to understand that a script’s precedence is determined by alphabetical order. The scripts located in the directories listed below will be executed in alphabetical order. If a script must run prior to another, it is important that the filename of the first script precedes the second.

Startup Scripts

Startup scripts are run at boot time only. Startup scripts will be run after a hard reset, but will not be run after a soft reset.

All user startup scripts should be located in **/etc/user.d**. /etc/rc3.user will search /etc/user.d for user startup scripts.

All startup script filenames should start with the letter **‘S’** (Startup).
i.e. /etc/user.d/Smyscript.sh

Hard Reset Scripts

Hard reset scripts **kill** all user processes. The Linux kernel is shutdown and restarted.

All hard reset scripts should be located in **/etc/hardreset.d**. /etc/hardreset will search /etc/hardreset.d for user hard reset scripts.

All shutdown script filenames should start with the letter **‘K’** (Kill).
i.e. /etc/hardreset.d/Kmyscript.sh

Soft Reset Scripts

Soft reset scripts **kill** and **restart** selected user processes. The Linux kernel is not restarted.

Soft resets are a little more complicated as they really are a two part process. The first step is to shutdown or kill the currently running processes. The second step is to restart the processes that were previously shut down. If either script, or part, does not exist, the soft reset will fail.

All soft reset scripts should be located in **/etc/softreset.d** and **/etc/softreset.d/restart**. /etc/softreset will first search /etc/softreset.d for shutdown scripts, and then it will search /etc/softreset.d/restart for startup scripts.



/etc/softreset.d contains scripts, or links to scripts, that shut down user processes.

All shutdown script filenames should start with the letter 'K' (Kill).

i.e. /etc/softreset.d/Kmyshutdown.sh

It is possible to create a symbolic link to a startup script in /etc/hardreset.d. This prevents having to maintain what would be duplicate scripts. If changes needed to be made to a startup script, only the actual file would need to be updated. The link would always point to the most up-to-date version of the startup script. Using a link will also save space in flash. The following command will create the associated symbolic link:

```
# ln -s /etc/hardreset.d/Kmyscript.sh Kmyshutdown.sh  
i.e. /etc/softreset.d/Kmyshutdown.sh -> /etc/hardreset.d/Kmyscript.sh
```

/etc/softreset.d/restart contains scripts, or links to scripts, that start user processes.

All reset script filenames should start with the letter 'R' (Restart).

i.e. /etc/softreset.d/restart/Rmyrestart.sh

It is possible to create a symbolic link to a startup script in /etc/user.d. This prevents having to maintain what would be duplicate scripts. If changes needed to be made to a startup script, only the actual file would need to be updated. The link would always point to the most up-to-date version of the startup script. Using a link will also save space in flash. The following command will create the associated symbolic link:

```
# ln -s /etc/user.d/Smyscript.sh Rmyrestart.sh  
i.e. /etc/softreset.d/restart/Rmyrestart.sh -> /etc/user.d/Smyscript.sh
```