Processing Task-Streams

RJ Cappallo 99.8.30

Mark 4 correlator processing streams provide a mechanism for specifying and controlling multiple (up to 4) correlation streams. Each stream will consist of a number of correlation tasks, which are constrained to be processed in the order they appear in, within the stream. Conductor will read task-stream files and generate correlation control messages, based upon commands in the files. Initially the stream files will be written by hand, but eventually they should be machine-generated by a task-stream composer, invoked, perhaps, via the operator interface.

Example:

ekey 2704
mode C32-16 * this is the correlator standard single-slice mode
channels 0x3FFF * enables the standard 14 Mk3 channels

task 98123-093544a:EFOQ

fork *the following lines (until **fend**) will be given to another stream
task 98233-093900:EF
task 98123-094500:EF
fend

slice 2 * force following scans to be correlated in slice 2

task 98123-093900:OQ
task 98123-094330:OQ

task 98123-095200:EFOQ

slice 0 1 2 3 * otherwise following task would fail due to resource lack
task 98123-100414a:EFOQKY:A32-16

ekey 2713
task 94005-102102

Notes:

1. A minimal stream file would have one **ekey** statement and one or more **task**'s.
2. A maximum of 4 concurrent streams can be active at any one time. Stream #'s are in range 0..3, and specify a *logical* stream (in contradistinction to the physical slice #'s).

3. Stream 0 is the operator stream; commands are entered into it in realtime, by the operator. Streams 1..3 are all "fed" from files, which the operator initiates by typing in their filename, which is then sent to conductor.
4. Initially, conductor will allocate correlator resources, and thus implement stream synchronization on a first-come, first-serve basis. Whenever a stream is halted due to lack of resources a message will be sent to opera.
5. The operator will be allowed to specify a stream priority order. At completion of tasks, lower priority streams will be suspended if there is a higher-priority stream waiting on resources.
6. Sub-netting is handled by the **fork-fend** construct (see example). When conductor encounters a **fork** in the stream, it creates a temporary file of all the statements between **fork** and **fend** and passes it along to a free thread via a message. This simple artifice allows the subnets to be processed in parallel.
7. A null station_list in a **task** statement indicates that all participating stations are to be processed.
8. The experiment key (**ekey**) can be specified globally, but overridden optionally within each task command.
9. Similarly, the correlator mode key (**mode**) can be overridden on a task-by-task basis. If **mode** is not present at all, then the value from the evex file is used.
10. The default length of the accumulation period in milliseconds is specified within the experiment (.evex) file, but it can be overridden within a specific **task** command.
11. The mirror flags are also specified within the experiment file, but overridden locally within a **task** command.
12. Any characters from an asterisk to the end of a line are considered comments, and are ignored during parsing.
13. The **slice** command allows the user to constrain physical slice allocation within the correlator to the listed slices, which are delimited by spaces.
14. The **channels** command specifies a bit-mapped mask to enable control of active SU channels. For example, **channels** 0x4080 would enable channels 14 and 7.

Syntax:

ekey <exper_key>
mode <mode_key>
task <scan_name>[:<station_list>][:<mode_key>][:<exper_key>]
[:<ap_length>][:<mirrors>]
fork
fend

slice <0..3>[<0..3>][<0..3>][<0..3>]
channels <hex_mask_value>