Singularity Design Note

33

# Event Tracing for Singularity

## *How ETS works*

*ETS is Singularity's Event Tracing subsystem, and is similar to the ETW (Event Tracing for Windows) system present in Windows XP and later. ETS provides a lightweight event tracing system for performance monitoring and debugging. This document describes the architecture of ETS, and includes a tutorial on its use.*

### 1. Brief Overview

Events are posted from all over the codebase by using calls of the form:

```
Monitoring.Log(PROVIDER, TYPE, ARGs...)
```

This logs an event to a circular buffer in kernel memory. Currently this is 8MB, split into 6MB for regular events and 2MB for text strings. Size is controlled by `MONITORING_BUFFER_SIZE` etc in Kernel/Native/Monitoring.cpp

`PROVIDER` is a member of the `Monitoring.Providers` enum which allows several bits of code to post events without needing to negotiate unique event IDs. Each provider defines their own event `TYPE`s (as `ushort`s), which are typically from an enum near the top of each provider's code, eg `ThreadEvent` in Thread.cs.

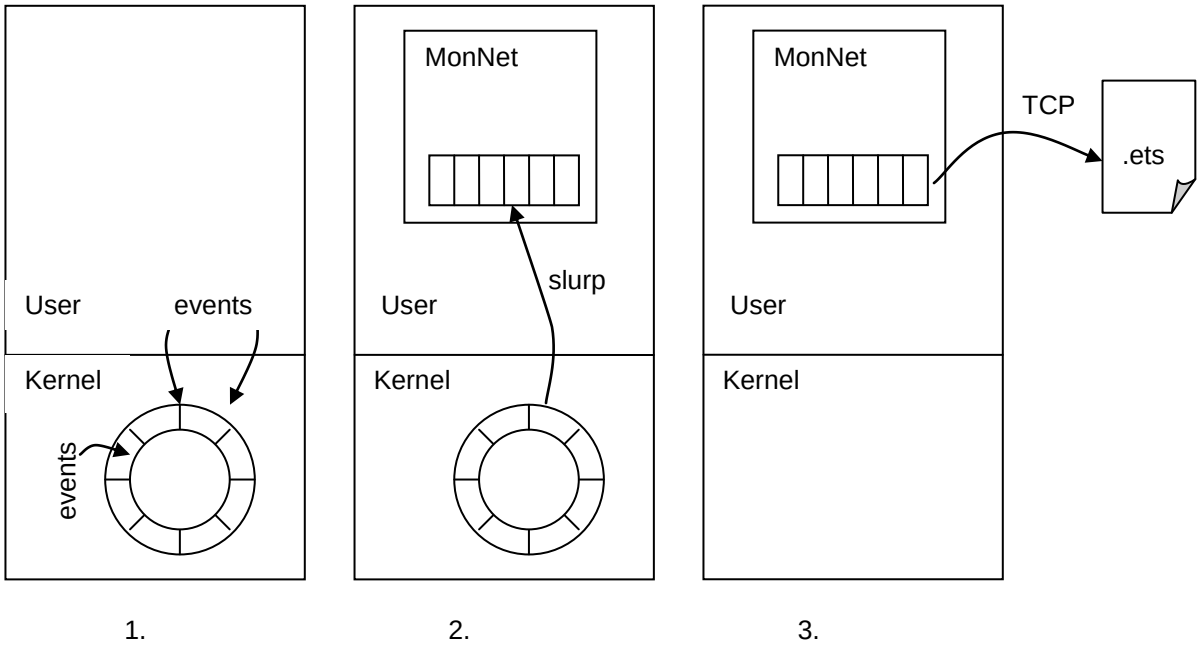
A full listing of all providers together with their events is given in TMF format as "singularity.tmf" in the Windows/ETS/ directory. A thinned out set of events (mainly ignoring WaitHandle synchronisation) is documented in "singularity-lite.tmf".

These files are used by the parser "mag_ets.exe" (in build/ directory) to decode the binary packed event format.

Use Applications/Network/MonNet to export the current contexts of the kernel's log buffer out over the network to your workstation (eg 10.99.99.1) for analysis and visualisation.

The following diagram shows how events flow in the system:




The three stages involved in getting an .ets file from a Singularity machine: 1) events from all over the running system are logged into an in-kernel fixed size circular buffer. 2) later (or in parallel), MonNet slurps those events into a larger buffer local to MonNet. 3) MonNet connected to an external TCP listener and transfers the events as an .ets format binary file.

## 2. Sample Usage

Build, ie vanilla "nmake" from:

1. root
2. Applications\Network

2b. <anything else you need>

3. Distro

Run:

1. Boot singularity
2. ipconfig /dev/nic0 10.99.99.2 255.255.255.0 10.99.99.1
3. run your experiment
4. run netcat (or equivalent) on your workstation:

```
nc -l -p 5000 >justboot.ets  # ie listen for TCP on port 5000
```

5. monnet /s /t:10.99.99.1:5000

This MonNet invocation slurps the events into MonNet (/s), then transfers them (/t) to 10.99.99.1 port 5000 where we've arranged (in step 4) to have a socket listening for the data, writing it to a file called "justboot.ets".

There are two main drawbacks with this (simple) method of using MonNet:

1) you'll get events related to starting MonNet in the kernel buffer, ie at the end of your experiment, and

2) the kernel buffer may not be large enough to hold all the events you're interested in.

While you can increase the size of the kernel buffer, it's accessed using 16-bit indices to allow lock-free algorithms to be used: this means there's a maximum of 65535 events storable in the buffer.

To address this problem, you can use MonNet in the background:

1. Boot singularity

2. ipconfig /dev/nic0 10.99.99.2 255.255.255.0 10.99.99.1

3. run netcat on your workstation

4. MonNet /d /c /w:10 /e /r:300 /s /t:10.99.99.1:5000 &

5. run your experiment

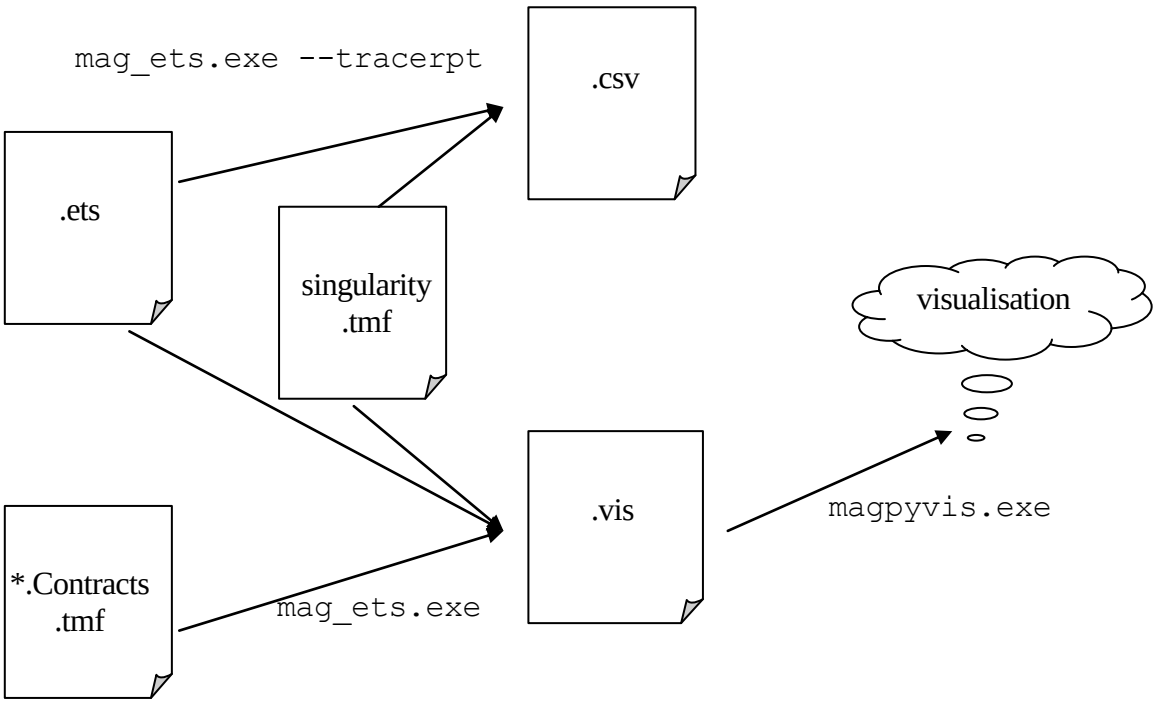
The flags to MonNet says: disable event logging (/d), clear the kernel buffer (/c), wait 10 seconds for experiment to start (/w:10), enable event logging (/e), slurp from kernel buffer for 300 seconds (/r:300 /s), and finally transfer via TCP (/t:10.99.99.1:5000).

The idea here is that the 10 seconds gives you a chance to start the experiment and have it warm up, without those events getting added to the kernel buffer. Then logging is enabled and events are pulled up to MonNet's private buffer (which can be much larger than the kernel buffer). Finally, after 300 seconds slurping stops and MonNet's private buffer is transferred over the network.

This allows you to control quite accurately which events will appear in the log.

## 3. Analysing .ets files




ETS commands, their inputs, and their products.

### 3.1. Converting to .csv

To view the contents of an .ets file, you can convert it to a textual .csv file:

```
mag_ets --tracerpt -t Windows\ETS\singularity.tmf -e justboot.ets -o
justboot.csv
```

This is very similar to ETW's tracerpt program.

A sample run produces output which looks like:

```
Parsing TMF ...

    ... Windows\ETS\singularity.tmf

...done

Parsing Contract Message Tag mapfiles ...

...done

.....................................................

Stats:

Processed 0 buffers, 56231 events

Dropped 0 events, 64 unknown events
```

```
Unknown event types (guid, version, type):

('20', 0, 11) 64
```

The .csv file has the following format:

`#timestamp,EIP,CPU,TID,PID,Provider,Type,UserData`

For example:

`6544408122948,34311e,0,d1,1,endpoint,DeliverHook,ChannelId=313,MessageTag=67,ContractTag=96854120`

The fields have the following meaning:

Timestamp: Processor.CycleCount have at time event is recorded in kernel buffer

EIP: program counter value where event was posted (ie site of `Monitoring.Log()` call)

CPU: CPU number which posted the event (always 0 for uni-processor machines)

TID: hex thread ID which posted the event

PID: hex process ID which posted the event

Provider: name of provider, from .tmf file

Type: name of event type, from .tmf file

UserData: KEY=VALUE pairs of extra information, again from .tmf file

The .csv file will probably start with a number of SysInfo provider events: these are fake events inserted by MonNet just before slurping which describe the state of the system, eg CPU clock speed, which processes and threads are active and for processes which image they have loaded (if any).

### 3.2. Visualising .ets files

A better way to see what was going on in the .ets file is to parse the raw events and generate a .vis file.

```
mag_ets         -t         Windows\ETS\singularity-lite.tmf         -c
Contracts\obj\Prototype.MarkSweep\__All.Contracts.tmf  -e  justboot.ets
-o justboot.vis
```

This should produce output like the following:

```
Parsing TMF ...

    ... Windows\ETS\singularity-lite.tmf

...done

Parsing Contract Message Tag mapfiles ...

    ... Contracts\obj\Prototype.MarkSweep\__All.Contracts.tmf
```

```
...done

Writing visualisation output to justboot.vis

.....Found Shell event: 'netstack
&' at time

: 1079855362440

.........Found Shell event: 'ipconfig /dev/nic0 10.99.99.2
255.255.255.0 10.99.9

9.1' at time: 1106483455418

......

Stats:

Processed 0 buffers, 56231 events

Dropped 0 events, 13209 unknown events

Unknown event types (guid, version, type):

('12', 0, 2) 289

('10', 0, 10) 3953

('11', 0, 2) 5579

('20', 0, 11) 64

('13', 0, 2) 9

('11', 0, 1) 31

('9', 0, 2) 89

('9', 0, 1) 3195

Found Shell event: 'monnet 10.99.99.1 5000' at time: 1121610922437

Found 0 requests
```

Then to visualise the .vis file:

```
magpyvis justboot.vis
```

See later for a section describing the UI.

### 3.3. Contract and message names

As you can see from Section 3.1, the endpoint message events are not very human-readable. This is because we only post a contract ID and message tag to keep the events small. This means we need to (1) lookup to contract ID to get its name, and once we know which contract this is, we can (2) map the message tag to a message name.

(1) The contract names are queried from the runtime type system by MonNet just before it slurps the kernel buffer. It does this in a really gorey manner by getting the typeof() all the contracts it knows about, and posting them as SysInfo/ContractName events. This clearly should be fixed, but needs support from the typesystem.

(2) The message tag names are got when you build a tree by running ILDASM over the contract DLLs just after they are built and using an awk script to generate .tmf from the Tags enum each contract has. These extra .tmf files are called the contract message tag map files; they are concatenated to form `__All.Contracts.tmf`, as used in the command lines quoted previously.

Having described how the setup works currently (2 May 2006), here's how I would like it to work in the future. There are really three separate issues:

1) **getting a stable identifier for the contract while running in the ReceiveHook() or SendHook() function**. This identifier could be the contract type's MD5; currently we use the result of SharedHeapService.GetType() on the endpoint's shared heap allocation. However, I haven't been able to work out how to get to the underlying type's MD5 value.

2) **mapping from the stable identifier (eg MD5) to a friendly name for presentation purposes**; this is where a central type registry would be useful: it would record each type's MD5 and type name as each type is registered, and be capable of being queried for this information from MonNet. Posting this information as ETS events isn't quite enough, because the event log is a circular buffer in-memory and thus older events get overwritten. We'd lose the registration events for the first few types that were introduced to the system. The current way we get the mapping is terrible: MonNet needs to know ahead of time all the typeof(ContractName) that we care about resolving; it gets their type handles and fully qualified names, and posts them as synthetic "sysinfo" events which are prepended to the raw ETS log just before it is shipped off the singularity box. The central type registry would solve this issue in a much nicer fashion.

3) **mapping from message tags to friendly names**; knowing the contract name isn't quite enough: we also need to know the names of the messages exchanged by endpoints adhering to the contract. Again, we currently do this is an inelegant manner: we ILDASM the DLLs defining the contract types, looking for the "Tags" enum giving names and values. We dump this info (as TMF format since our tool already can read this). The problem with this approach is that it is not always obvious where the contracts of interest are being build; and it doesn't work well with private contracts (ie those where the contract is defined in the application itself rather than under Contracts/ in a DLL). I'm not sure what a better approach might be however.

# 4. How do I... ?

## 4.1. Set up the ETS applications?

The ETS applications, `mag_ets.exe` and `magpyvis.exe`, have dependencies on external libraries and these libraries need to be installed in your path in order to work. The DLLs can be got from the windows MSI for python, available from:

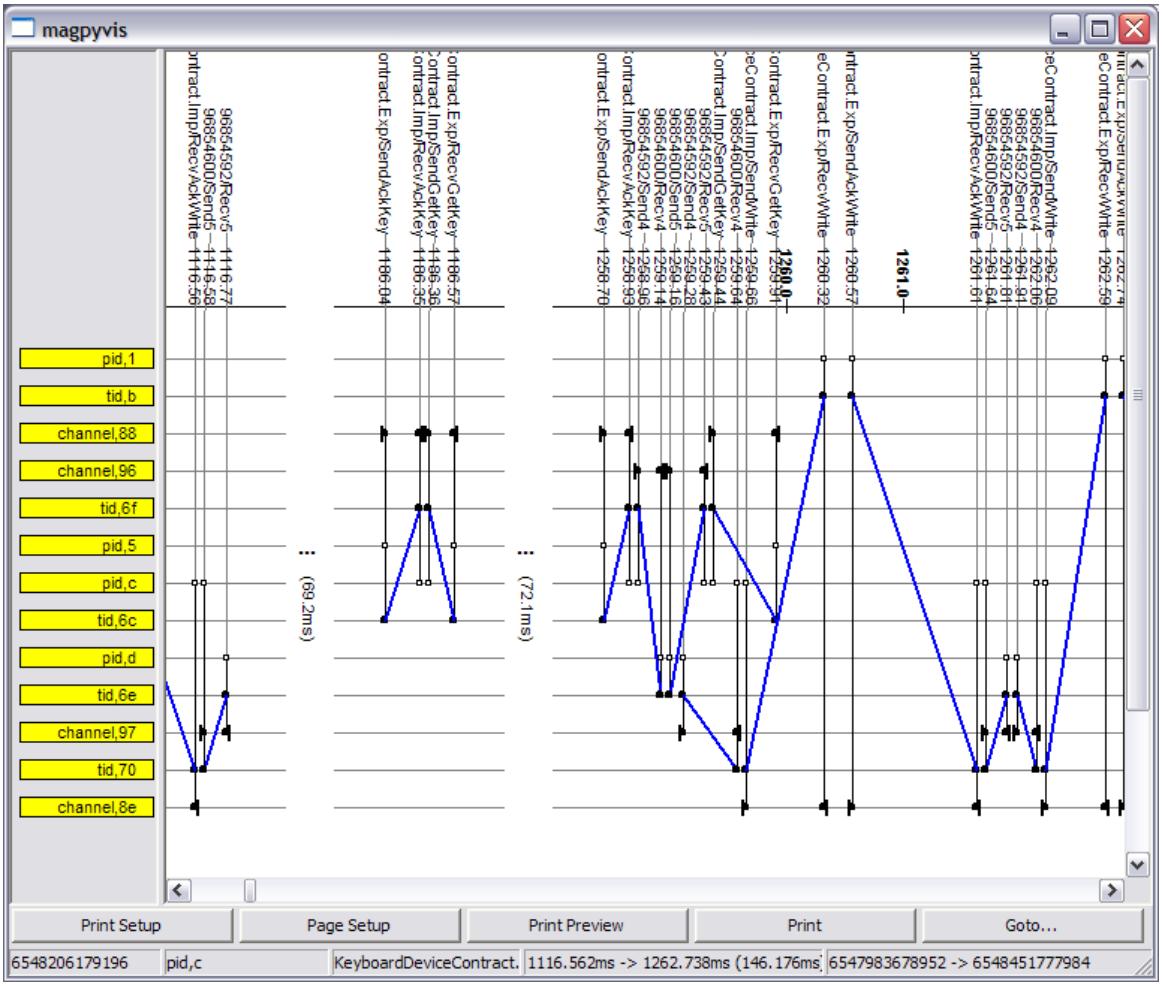
http://www.python.org/download/releases/2.4.4/

The executable in the build directory is built against 2.4.1 but that's not available any more. Hopefully 2.4.4 will be compatible.

The wx Python DLLs are from:
http://wxpython.org/download.php
The build of magpysvis.exe used version 2.6.1.0 (ansi).

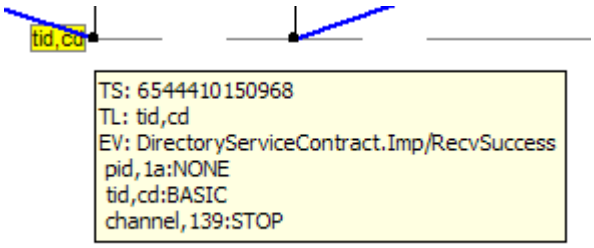
## 4.2. Use magpyvis?




On x-axis is time, on y-axis are various resources like threads, CPUs, IRQs, channels, waithandles. Vertical lines are labelled events which happened at some time, and are bound to the resources they used. Fat blue lines join contract send and receive events together.

Left-click near an event pops up a tooltip:

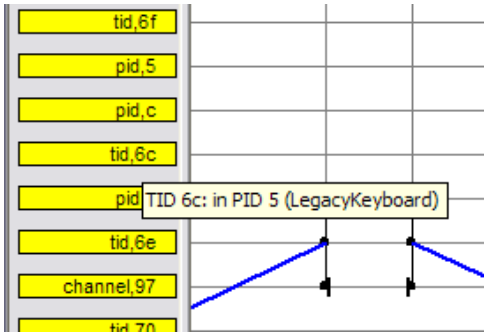
 




giving the event's timestamp, and all its attributes together with their binding mode (NONE, BASIC, START, STOP). Left-click also sets the "current timestamp" to the location clicked. Subsequent clicks will print to stdout the time delta between the previous and current clicks.

The statusbar at the bottom shows: current timestamp, closest event summary, timerange visible (both in ms and timestamps). All real-world time units rely on a valid SysInfo/CpuSpeed event in the .vis file header, and it is printed to stdout on startup so check it looks sane.

Right-click on a timeline highlights it by drawing it in fat grey; click again to unhighlight.

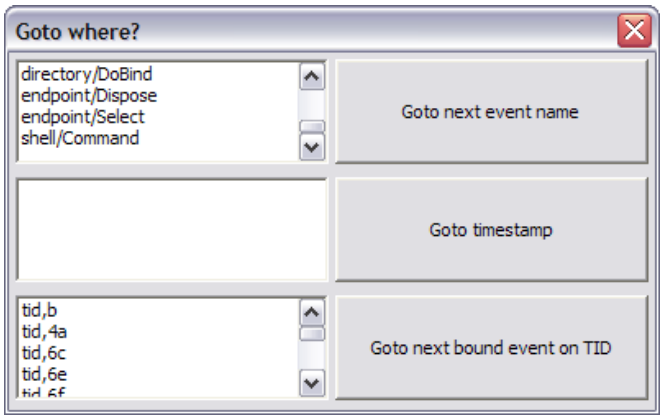



Left-click on PIDs or TIDs in the yellow legend area pops up a tooltip describing the containing PID and associated image name (assuming SysInfo events are available).

Mouse-wheel-roll-up zooms the timescale in (event get wider appart), mouse-wheel-roll-down zooms the timescale out (events get closer together)

The "Goto..." button pops up a modal dialog allowing you to search forwards from the "current timestamp" for the next named event, or the next event bound to a tid timeline.




It also allows you to jump to any absolute timestamp (eg cut-n-pasted from a logfile). It beeps and prints a warning if a search fails; on success it scrolls the window to put the found event or timestamp just visible at the left-hand-side of the window.

#### 4.2.1. I want more detail! What are the…

1) …red lines and text?

The red *__vertical__* lines and text highlight certain events (with x-coord proportional to the event's time); think of it as syntax highlighting. The colours are as follows:

If event name starts with X, we colour it Y:

Pcap purple

Diskio red

Tcpip green

Httpreq red

W3server dark green

Thread grey

Thread/Create orange

Perfinfo grey

Processor red

Channelservice sea green

Endpoint sea green

Endpointcore sea green

Cassini purple

"contract" anywhere in the name makes it blue.

(Not all of these event name prefixes are singularity-related, clearly).

The red *__horizontal__* lines are colour coded according to the timeline (ie resource) that they represent:

Cpu black double width

Disk black double width

Wait blue

Interrupts red

(default is grey)

All of these colours are hard-coded in the python source code, so if you have the source you can easily change them.

2) …little bold vertical bars with directional nib?

These are *bindings* between events and the timelines. Think of them as little squares at the intersection of the event (in the x-direction) and the timeline (y-direction). The presence of a square marks a binding between one attribute from an event and the timeline for that attribute. The kind of square tells you the kind of binding:

hollow = informational only,

solid = basic binding,

solid with bold bar on LHS = start binding,

solid with bold bar on RHS = stop binding

3) …blue rectangles?

The blue/lilac horizontal rectangles around the CPU and thread timelines represents which thread was currently running on which CPU – ie the scheduler behaviour. They are drawn based on context switch and irq events.

4) …little squares?

See event bindings (item 2 above). An event can have more than one because (eg) Processor/Resume event has 3 attributes: source tid, dest tid, and CPU. We bind the source tid as informational only (hollow square), dest tid (ie the TID that's being resumed) as a start binding (meaning that this TID timeline starts being active at this event's timestamp), and the CPU the thread is running on as a start binding because that CPU starts being active with that dest TID at the event time.

The event binding business is probably not so important if all you want is a rough idea of what's going on in the system without wanting to parse requests out of the event stream.

### 4.3. Add more tracing?

Add a name and value for your new provider to the `Provider` struct in Kernel/Singularity/Monitoring.cs. Define a new enum in your facility, named following the convention: ProviderEvent (eg the `Process` provider defines a `ProcessEvent` enum). Allocate names and values to your provider's events as you see fit. Add calls to one of the overloads of `Monitoring.Log()` to post your events from strategic places in your facility, eg when starting some activity, finishing it, making a scheduling decision, hitting/missing in a local cache, etc. The different overloads allow you to post extra arguments with your event, for example the PID of a newly created process together with the parent's PID. You can post up to 5 arguments of UINT32 data; if you need less just post zeros and don't describe those arguments in the TMF. Posting strings is a little more expensive than non-string events, so try to post strings rarely, eg on file open rather than each file operation. Finally, write TMF to describe your events and their arguments.

### 4.4. Write TMF event descriptions?

If you add events which are posted manually by your code, then you'll need to write the .tmf to describe their argument layout yourself. Take a look at Windows/ETS/Singularity.tmf for inspiration.

For example, the endpoint provider is described there as follows; my comments in green inline:

```
8 Endpoint
```

8 is the provider ushort, "Endpoint" is the name to print to .csv or .vis files

```
#version 0
```

All events so far are version 0; it's a param when posting the event

```
#type Dispose           1

{

    ChannelId, ItemLong

}
```

This declares an event with id 1 (ushort), named "Dispose". It has one argument, named "ChannelId" of type ItemLong (types used to be defined on MSDN but they seem to have pulled the article!). The type controls both how many bytes of argument are expected, and how to display them.

```
#type Select            2

{

}

#type RetrieveHook      6

#type DeliverHook       7

{
```

```
    MessageTag, ItemLong

    ChannelId, ItemLong

    ContractTag, ItemULong

}
```

This declares two events, id 6 and 7, named “RetrieveHook” and “DeliverHook”, which share the same argument layout.

```
#type Notify           10

{

    ChannelId, ItemLong

}
```

You can add your TMF to the main singularity.tmf file or put your definitions into a separate file and give multiple “–t TMFFILENAME” arguments to `mag_ets.exe`

As for strings in general, they are not great to post, since they can be quite large and need out-of-band storage.

However, to post them use the special overload of Monitoring.Log() which takes 5 arguments and a string. The parameters are just junk, but need to be there. They also need to be declared in the TMF.

Take a look at the Cassini event posting code and the tmf used to parse it. The other place that posts a string is the image loader.