IoTBridge - Debugging Techniques, part4

System Analysis with LTT-ng

Table of Contents

Table 1. Changes

Version	Date (yyyy-MM-dd)	Authors	Changes
0.99.0	2021-11-09	Marc TITINGER	Document creation

Chapter 1. Introduction and Warning

This document addresses part 4 of the Table of Contents in Introduction on debugging techniques

- LTT-ng is a very low intrusiveness backend, to generate traces, using the CTF protocol.
- It differs in use-cases with DLT presented in DLT Traces in that is is not practical for rapid instrumentation, and off-the-shelf (CI) tracing. LTTNG requires a significant setup time, and some filesystem hacking (LD_PRELOAD for instance)
- We are going to use **LTT-ng in Userspace to profile applications**, but setting markers to correlate with the kernel timecharts is also possible
- we ar egoing to usee **LTT-ng in Kernel Space for timecharting**

Chapter 2. LTT-NG Setup for Usermode profiling

The following setup is required

2.1. STEP0 Rebuild the app to test with the proper flags

In the case of Forum extensions, cnage the meson.build to add the following flags:

```
add_global_arguments('-g', '-fno-omit-frame-pointer', '-finstrument-functions', '-
finstrument-functions-exclude-file-list=/usr/include', language : 'cpp')
```

Deploy the image **with symbols** to the target:warning-caption:

```
on the target: systemctl stop com.se.extension.local_modbus_server
on the build pc: scp workspace/sources/gxl-forum-extension-local-modbus-server/oe-
workdir/image/usr/bin/* root@192.168.1.2:/usr/bin
```

if you did well, the profiling hooks are instrumented by gcc:

```
 nm workspace/sources/gxl-forum-extension-local-modbus-server/oe-
workdir/image/usr/bin/local_modbus_server.extension  | grep cyg
         U __cyg_profile_func_enter@@GLIBC_2.4
         U __cyg_profile_func_exit@@GLIBC_2.4
```

2.2. STEP1 On the Host PC, launch a collection relay deamon

You can do this with the folliwing command, provides you did setup the LTT ports, as preconfigured in the dev image:

```
 lttng-relayd -vvv --control-port='tcp://0.0.0.0:5342' --data
-port='tcp://0.0.0.0:5343'
...
DEBUG1 - 10:31:13.486907494 [9880/9883]: [thread] Relay dispatcher started (in
relay_thread_dispatcher() at main.c:985)
DEBUG1 - 10:31:13.486918284 [9880/9882]: [thread] Manage health check started (in
thread_manage_health() at health-relayd.c:247)
DEBUG1 - 10:31:13.486927329 [9880/9884]: [thread] Relay worker started (in
relay_thread_worker() at main.c:3327)
DEBUG3 - 10:31:13.486946037 [9880/9882]: Creating LTTng run directory:
/home/galaxy/.lttng (in create_lttng_rundir_with_perm() at health-relayd.c:94)
```

```
...
```

2.3. STEP2 Setup a tracing session on the target

Create a session on the target:

```
lttng create --set-url=net://192.168.1.1:5342:5343
```

Note that this will load all the ltt modules into the kernel, if at this point, the system reboot, just redo this sequence.

Setup kernel mode events, so that we get a timechart in Tracecompass:

```
lttng add-context --type pid -k
lttng add-context --type tid -k
lttng enable-event -k "sched_*"
lttng enable-event -k "irqs_*"
lttng enable-event -k "workq*"
lttng enable-event -k --syscall "*"
```

Setup Usermode events for profiling:

```
lttng enable-event -u -a --loglevel-only TRACE_DEBUG_FUNCTION
lttng add-context -u -t vpid -t vtid -t procname -t ip
```

You may optimize the trace amount by filtering on the required extension:warning-caption:

```
lbms=`pidof local_modbus_server.extension`
lttng track -k --pid 0,1,$lbms
lttng track --userspace --pid $lbms
```

Successfull enabling will show the following output for lttng status:

```
Tracing session auto-20211109-145031: [inactive]
    Trace path: tcp4://192.168.1.1:5342/auto-20211109-145031 [data: 5343]

=== Domain: Kernel ===

Channels:

- channel0: [enabled]

    Attributes:
      Event-loss mode:  discard
```

```
      Sub-buffer size:  1048576 bytes
      Sub-buffer count: 4
      Switch timer:     inactive
      Read timer:       200000 µÊs
      Monitor timer:    1000000 µÊs
      Blocking timeout: 0 µÊs
      Trace file count: 1 per stream
      Trace file size:  unlimited
      Output mode:      splice

    Statistics:
      Discarded events: 0

    Event rules:
      * (loglevel: TRACE_EMERG (0)) (type: tracepoint) [enabled]
      * (type:syscall) [enabled]
      workq* (loglevel: TRACE_EMERG (0)) (type: tracepoint) [enabled]
      irqs_* (loglevel: TRACE_EMERG (0)) (type: tracepoint) [enabled]
      sched_* (loglevel: TRACE_EMERG (0)) (type: tracepoint) [enabled]

=== Domain: UST global ===

Buffer type: per UID

Channels:

- channel0: [enabled]

    Attributes:
      Event-loss mode:  discard
      Sub-buffer size:  524288 bytes
      Sub-buffer count: 4
      Switch timer:     inactive
      Read timer:       inactive
      Monitor timer:    1000000 µÊs
      Blocking timeout: 0 µÊs
      Trace file count: 1 per stream
      Trace file size:  unlimited
      Output mode:      mmap

    Statistics:
      Discarded events: 0

    Event rules:
      * (loglevel == TRACE_DEBUG_FUNCTION (12)) (type: tracepoint) [enabled]
```

2.4. STEP3 Configure the service, to LD_PRELOAD the profiling hooks

On the target, in **/lib/systemd/system/com.se.extension.local_modbus_server.service**

add this to the Unit section:

```
Environment="LD_PRELOAD=/usr/lib/liblttng-ust-cyg-profile.so.0"
```

Note: you must reload/restart the service, after the ltt-ng session is created!

Note: you can check that you instrumentation is taken into account wiuth the command:

```
root@bpas-mp:~# lttng list -u
UST events:

PID: 9201 - Name: /usr/bin/local_modbus_server.extension
      lttng_ust_cyg_profile:func_exit (loglevel: TRACE_DEBUG_FUNCTION (12)) (type:
tracepoint)
      lttng_ust_cyg_profile:func_entry (loglevel: TRACE_DEBUG_FUNCTION (12)) (type:
tracepoint)
      lttng_ust_tracelog:TRACE_DEBUG (loglevel: TRACE_DEBUG (14)) (type: tracepoint)
      lttng_ust_tracelog:TRACE_DEBUG_LINE (loglevel: TRACE_DEBUG_LINE (13)) (type:
tracepoint)
      lttng_ust_tracelog:TRACE_DEBUG_FUNCTION (loglevel: TRACE_DEBUG_FUNCTION (12))
(type: tracepoint)
      lttng_ust_tracelog:TRACE_DEBUG_UNIT (loglevel: TRACE_DEBUG_UNIT (11)) (type:
tracepoint)
      lttng_ust_tracelog:TRACE_DEBUG_MODULE (loglevel: TRACE_DEBUG_MODULE (10)) (type:
tracepoint)
      lttng_ust_tracelog:TRACE_DEBUG_PROCESS (loglevel: TRACE_DEBUG_PROCESS (9))
(type: tracepoint)
      lttng_ust_tracelog:TRACE_DEBUG_PROGRAM (loglevel: TRACE_DEBUG_PROGRAM (8))
(type: tracepoint)
      lttng_ust_tracelog:TRACE_DEBUG_SYSTEM (loglevel: TRACE_DEBUG_SYSTEM (7)) (type:
tracepoint)
      lttng_ust_tracelog:TRACE_INFO (loglevel: TRACE_INFO (6)) (type: tracepoint)
      lttng_ust_tracelog:TRACE_NOTICE (loglevel: TRACE_NOTICE (5)) (type: tracepoint)
      lttng_ust_tracelog:TRACE_WARNING (loglevel: TRACE_WARNING (4)) (type:
tracepoint)
      lttng_ust_tracelog:TRACE_ERR (loglevel: TRACE_ERR (3)) (type: tracepoint)
      lttng_ust_tracelog:TRACE_CRIT (loglevel: TRACE_CRIT (2)) (type: tracepoint)
      lttng_ust_tracelog:TRACE_ALERT (loglevel: TRACE_ALERT (1)) (type: tracepoint)
      lttng_ust_tracelog:TRACE_EMERG (loglevel: TRACE_EMERG (0)) (type: tracepoint)
      lttng_ust_tracef:event (loglevel: TRACE_DEBUG (14)) (type: tracepoint)
      lttng_ust_lib:unload (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      lttng_ust_lib:debug_link (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      lttng_ust_lib:build_id (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
```

```
      lttng_ust_lib:load (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      lttng_ust_statedump:end (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
      lttng_ust_statedump:debug_link (loglevel: TRACE_DEBUG_LINE (13)) (type:
tracepoint)
      lttng_ust_statedump:build_id (loglevel: TRACE_DEBUG_LINE (13)) (type:
tracepoint)
      lttng_ust_statedump:bin_info (loglevel: TRACE_DEBUG_LINE (13)) (type:
tracepoint)
      lttng_ust_statedump:start (loglevel: TRACE_DEBUG_LINE (13)) (type: tracepoint)
```

2.5. STEP4 Setup our device for the activity you wan tto trace, and start tracing

On the target:

```
lttng start
... do stuff ...
lttng stop
```

Once capture is done, ctrl+C the relay deamon on the host PC, trace data is stored in your home on the PC:

```
 tree lttng-traces/
lttng-traces/
└── bpas-mp
    └── auto-20211109-145031
        ├── kernel
        │   ├── channel0_0
        │   ├── channel0_1
        │   ├── index
        │   │   ├── channel0_0.idx
        │   │   └── channel0_1.idx
        │   └── metadata
        └── ust
            └── uid
                └── 101
                    └── 32-bit
                        ├── channel0_0
                        ├── channel0_1
                        ├── index
                        │   ├── channel0_0.idx
                        │   └── channel0_1.idx
                        └── metadata
```

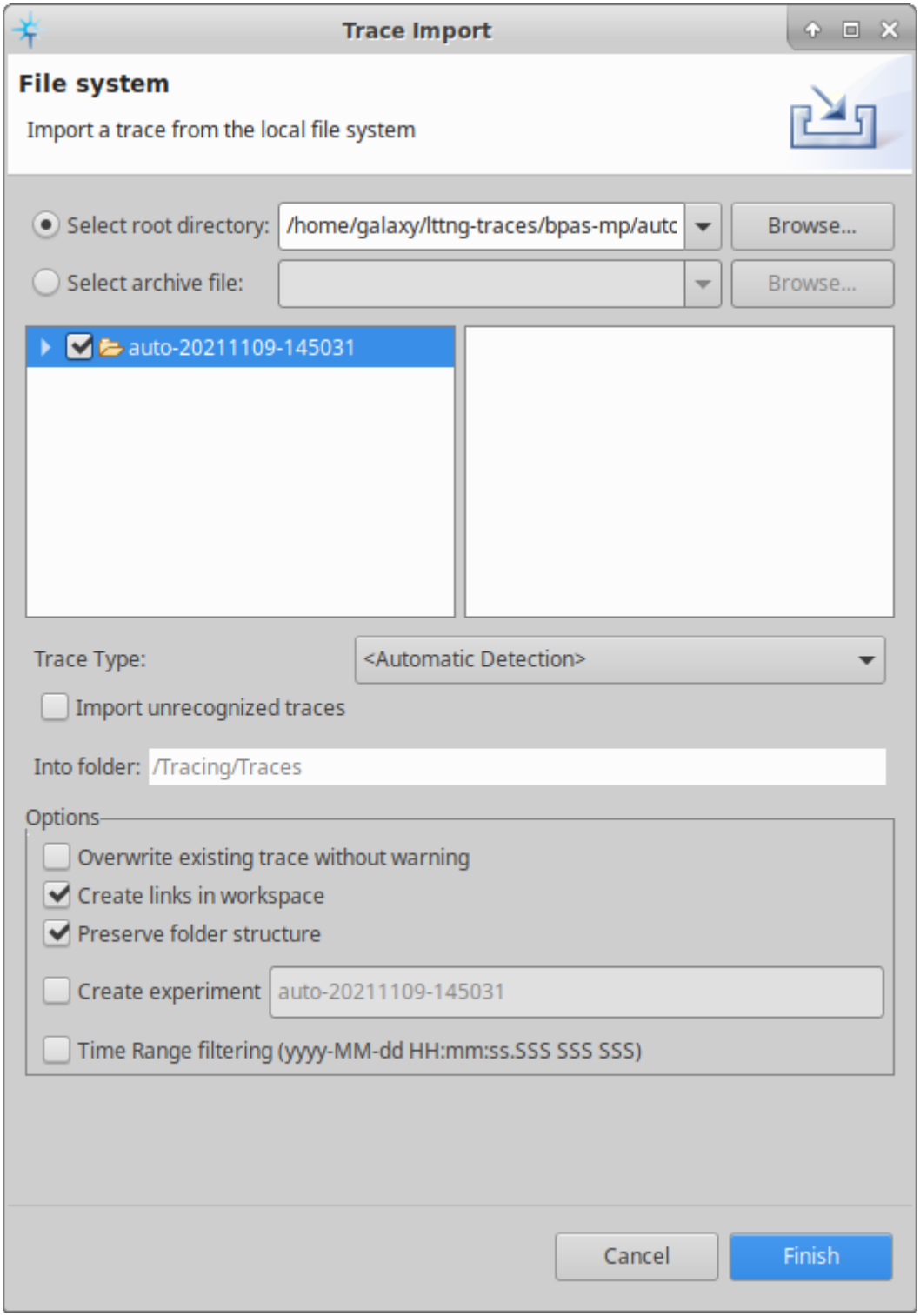
Chapter 3. Analysis with Tracecompass

Download and install tracecompass, note that you will need a JVM uptodate, not the one for iotb-yocto.

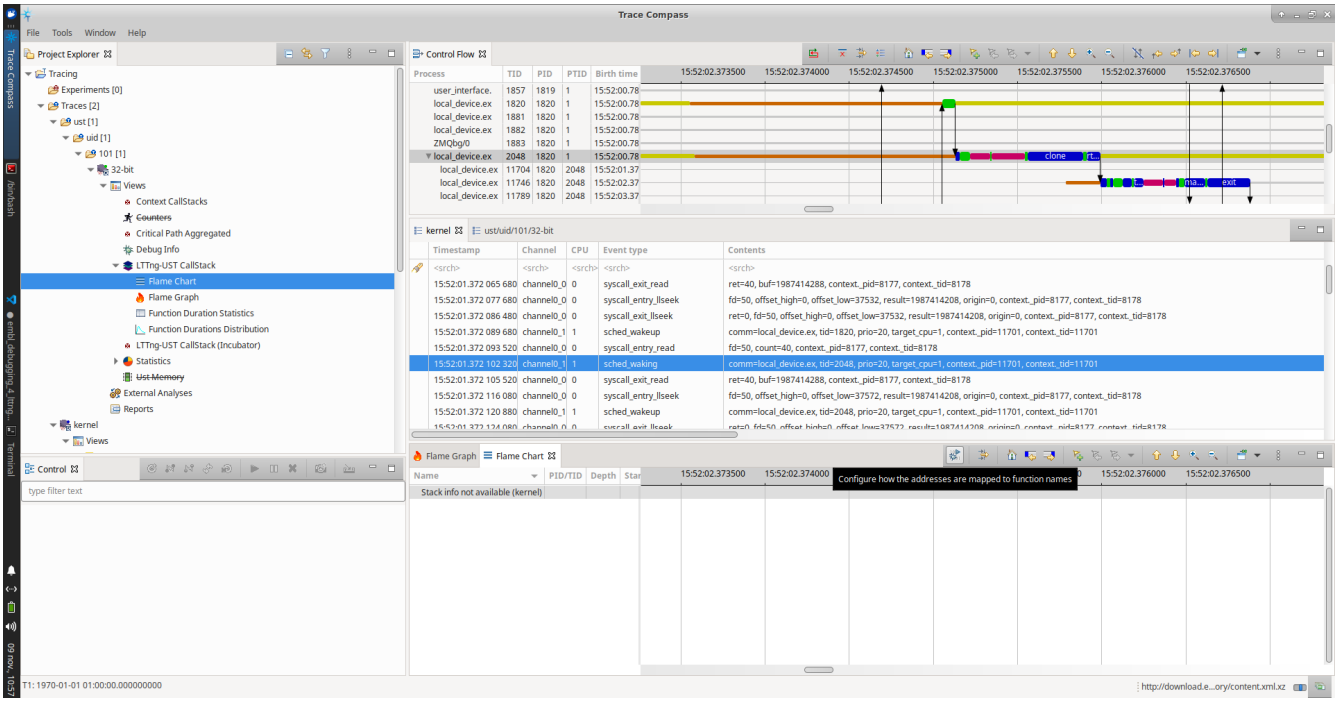
3.1. Importing the traces and setting up symbol sysroot

For using the callstack view and, you may refer to https://archive.eclipse.org/tracecompass/doc/stable/org.eclipse.tracecompass.doc.user/LTTng-UST-Analyses.html#Call_Stack_View

go to "files/import" and select the trace session folder:

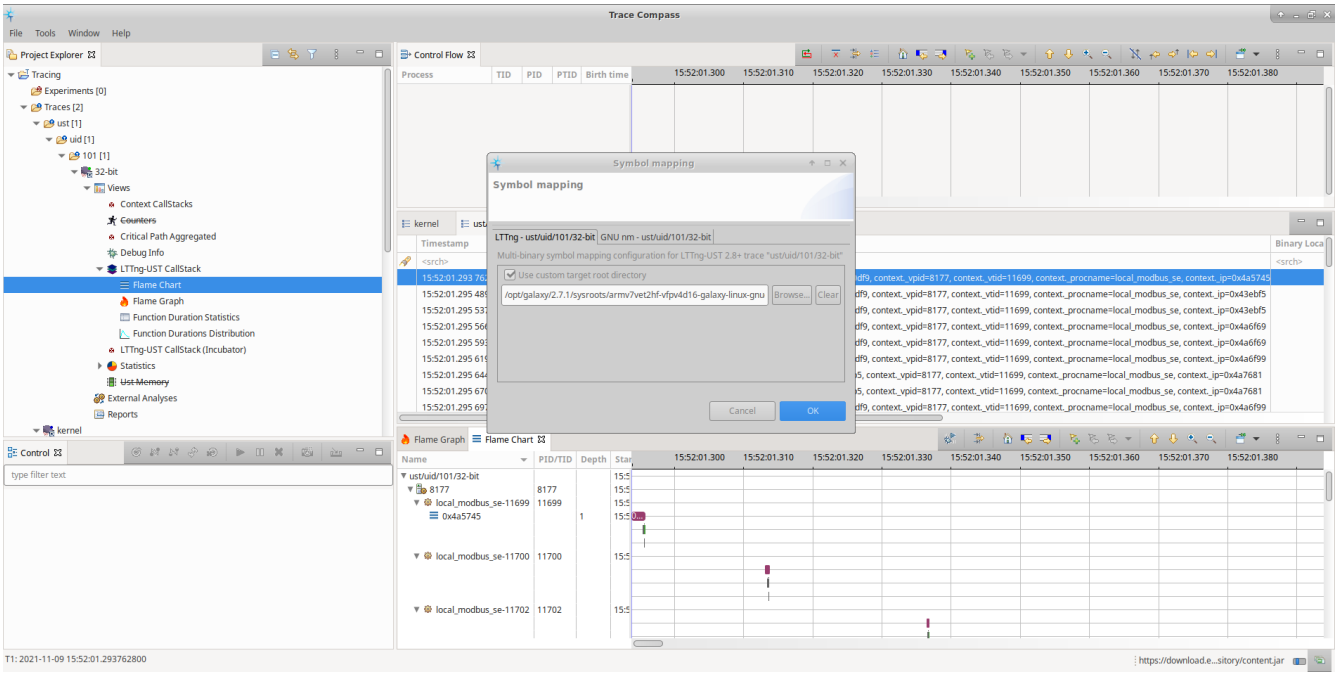


you should have a trace session in TC, with both kernel and usermode traces:



NOTE	**after import, double clikc on "kernel" and "32bits" icons in explorer, so views are computed/initialized**

you need to setup the sysroot as with GDB: select the ust event windows, and click on the "bug" icon in the flamegraph plugin, setup the sysroot:



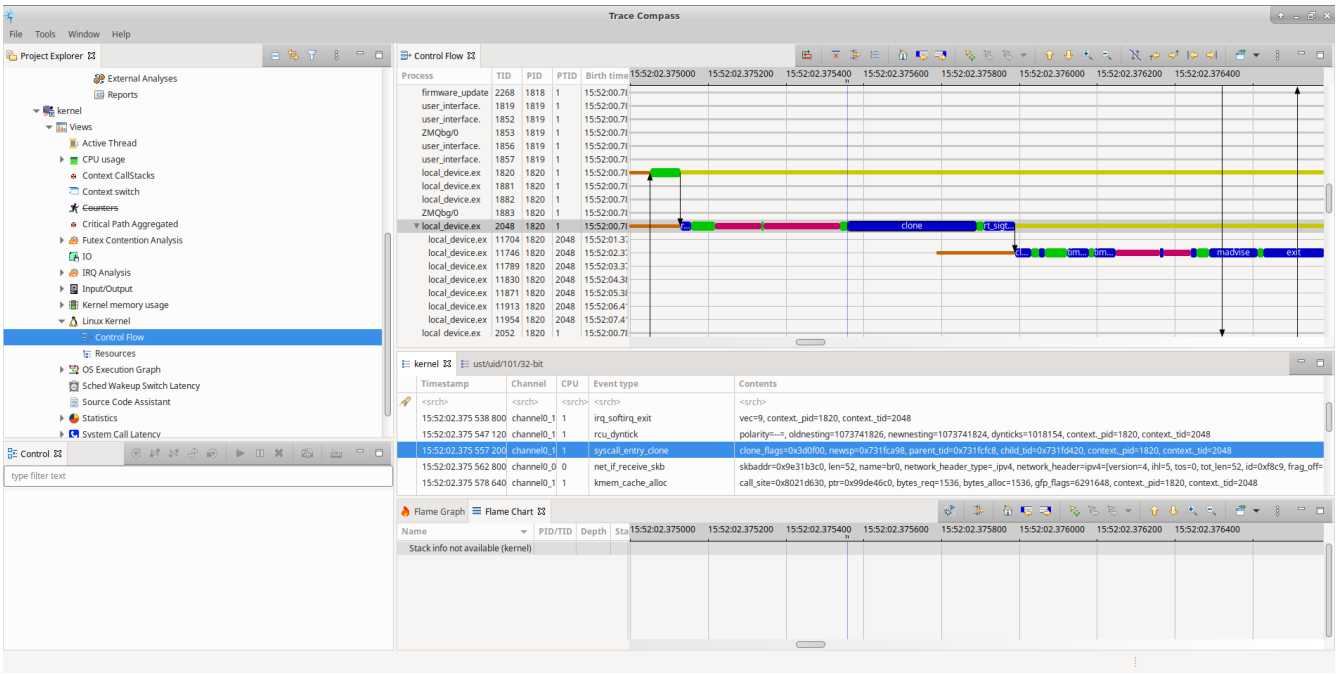
also in the "GNU nm" tab, add all the required libraries debug files, from the sysroot/usr/lib/.debug folder ...

If symbols are properly set, the flamecharts and flamegraph views are available

ALTERNATIVE: use **arm-galaxy-linux-gnueabi-nm local_modbus_server.extension --demangle › ~/mapping.txt**

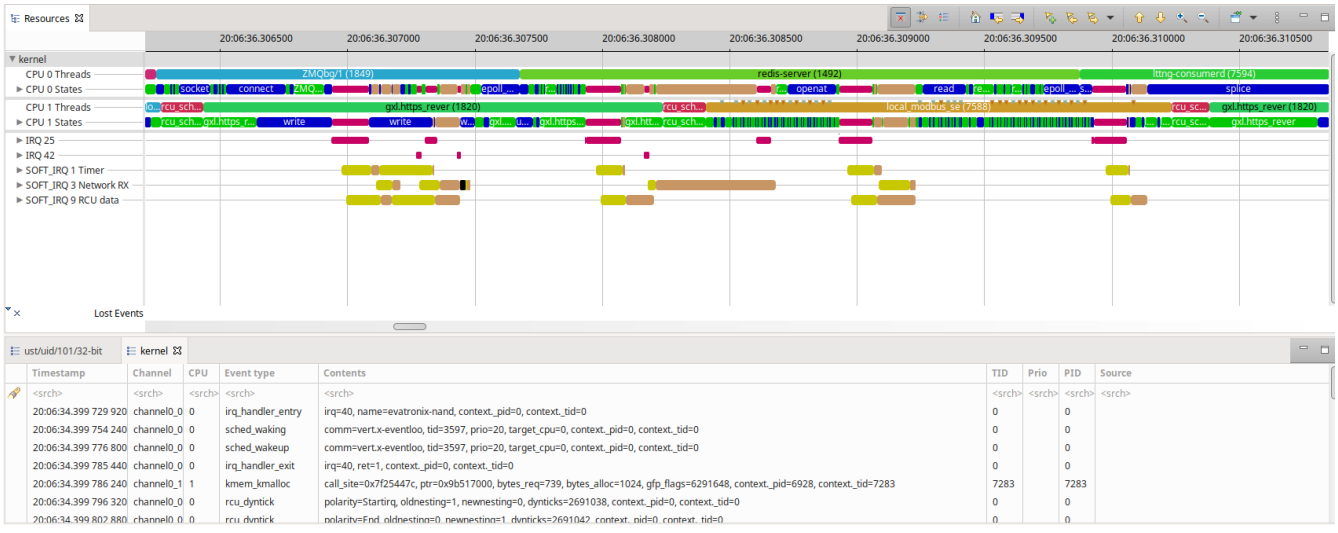
3.2. Kernel traces : Control-Flow view

The control-flow view is crucial, to analyse the system dynamics, for instance here, we wee how ever 25ms a short-lived thread is created in LMBS:



3.3. Kernel traces : Resource View

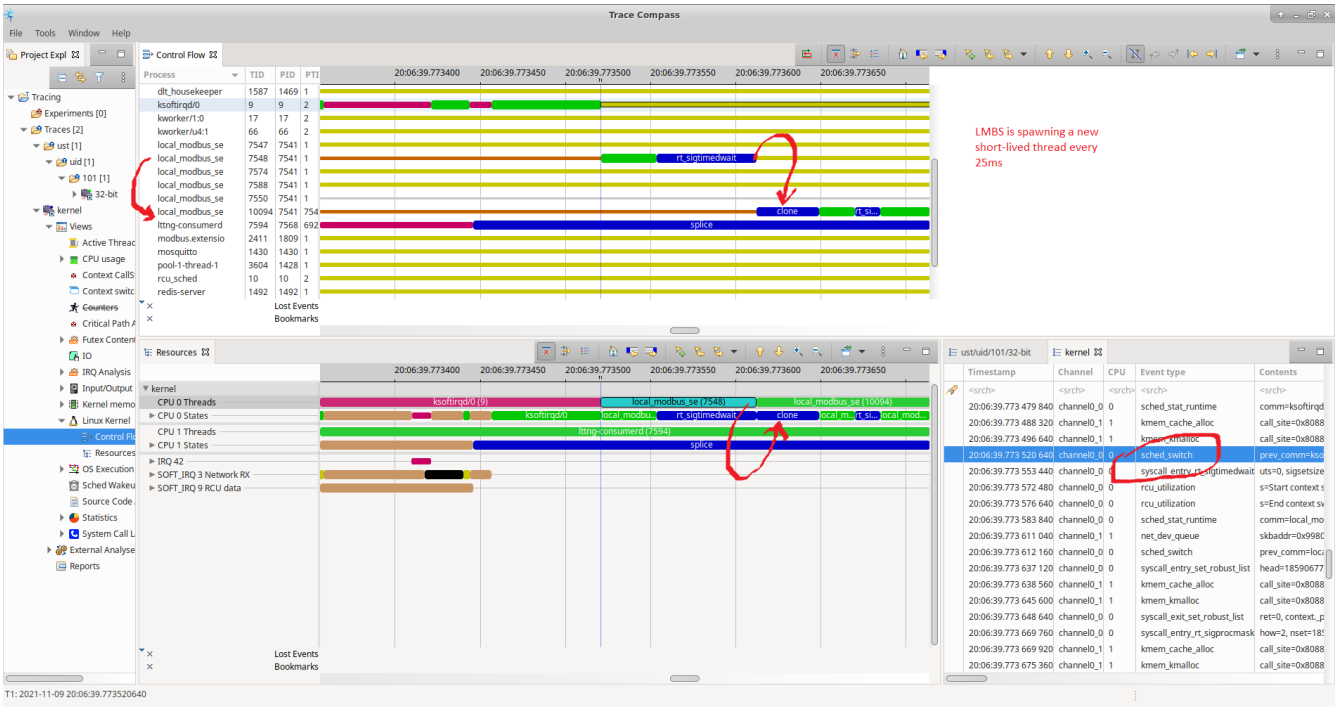
The resource view shows per resource (CPU, IRQ context) what process is scheduled, and its currently executed tracepoint (syscalls etc...)



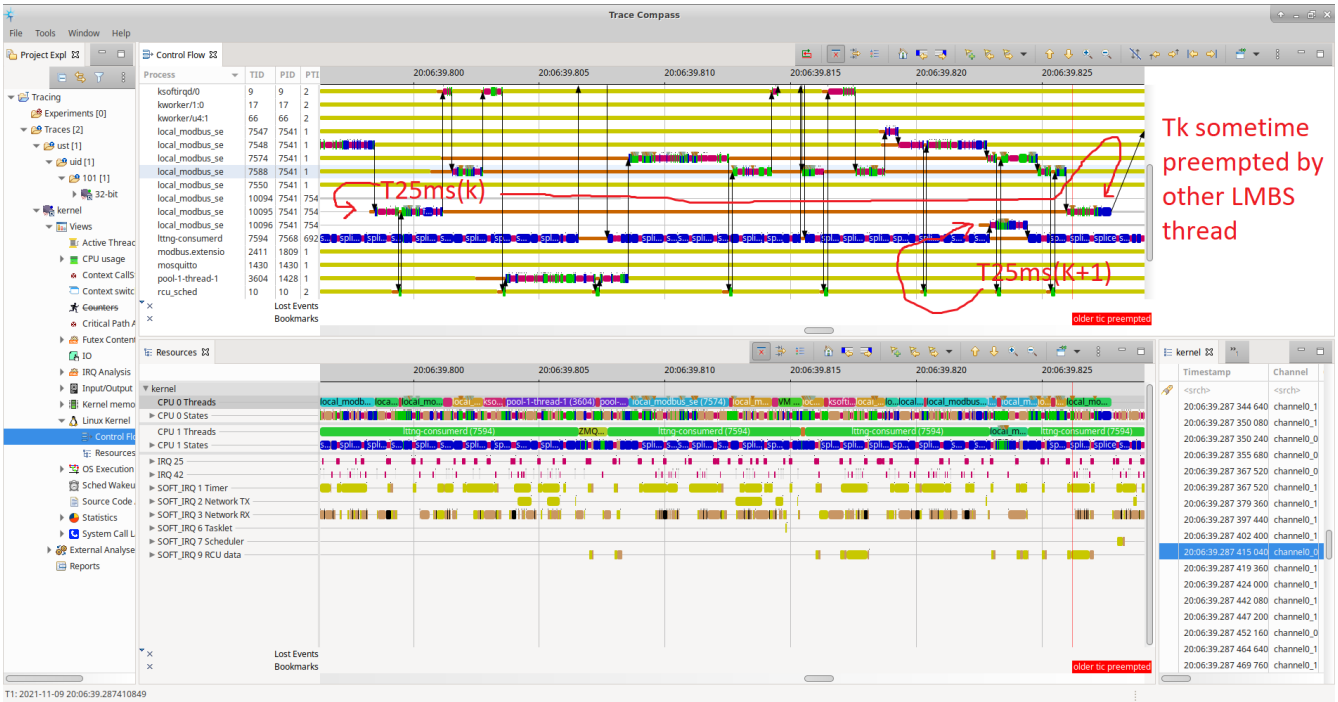
Chapter 4. Use-case, few findings with Local Modbus Server

4.1. Kernel traces

It appears, that the way how 25ms tics are handled (through a *lambda* timer function) leads to spawning of a dedicated short-lived thread:

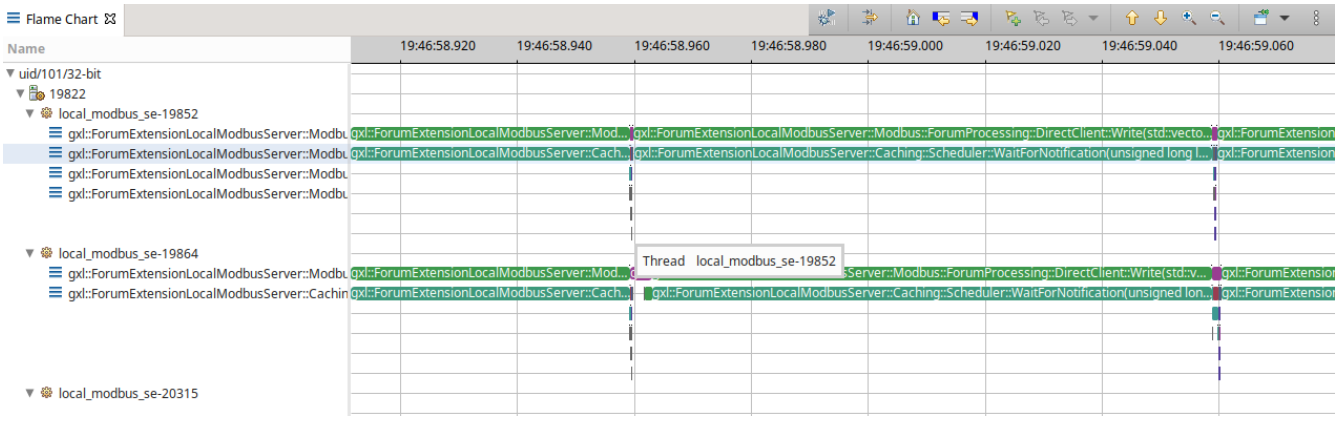


These threads can sometimes last longer than a 25ms period, as they can be prempted by other LMBS threads:

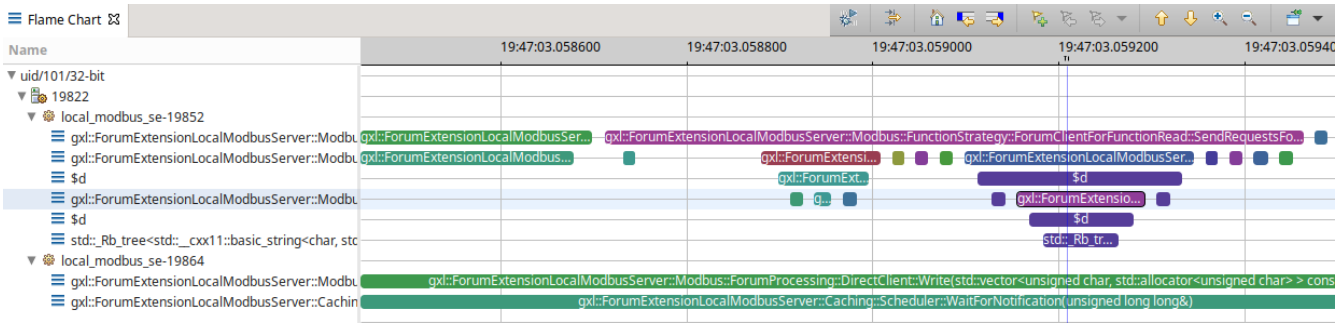


4.2. Usermode traces in LMBS

ust traces show that the SlaveID theads are scheduled every 100ms, in sequence as expected:



In the example; we can measure SendRequestsXXX, and see that it usually takes around 0.6 ms, as seen already with DLT:



4.3. Preliminary Conclusion

- UST traces don't show a code path in LMBS that takes an unexpected amount of CPU time
- However, the timing and scheduling scheme must be reviewed, as the API used to wait (notify) and run 25ms counter increment, and 100ms slave IDs lead to thread spawning (**sys_sched_fork/ and *clone+exit**) which doesn't look like a wanted behavior.
- Maybe a ipc or semapohre based approach, or a waitqueue would be more sensible.
- This is likely to not scale well with more devices.
- It is also likely that this pattern is used by other extensions, (modbus RS, zigbee?) and should be reworked.