ModemManager Evaluation

Motivation

An initial support for cellular connectivity was added to EVE OS already back in the version 4.3.0. Instead of using Hayes AT command set, which differs between modem manufactures, the decision was to use (proprietary) **QMI** and later also (standardized) **MBIM** protocols, providing a unified interface for controlling and managing various aspects of cellular modems, including functions like data transmission, voice calls, SMS messaging, and network configuration. These protocols can be used from open-source client libraries **libqmi** and **libmbim**, developed under the **freedesktop.org** project. However, there are no Golang bindings, hence we decided to use CLI tools provided alongside these libraries: **qmicli** and **mbimcli**. PoC for cellular connectivity in EVE simply consisted of a new container "wwan" with installed libqmi and libmbim libraries and their CLI binaries, plus a very simple **shell script** implementing a "management agent". This script periodically (every 5 minutes) checked if modem (only one was supported at the time) is connected (using qmicli/mbimcli) and if not, it would read APN from a certain file under **/run/wwan** (where it was written down by NIM) and start connection with this APN string as the only parameter. The script then obtained IP settings and applied them to the wwan0 interface. List of DNS servers was published to NIM via another file stored under /run/wwan. No status data or metrics were published from this script.

Later, we would keep receiving customer feature requests for enhancing mobile connectivity. We were asked to publish modem information and status updates (model, revision, IMEI, connection state, etc.), SIM info (ICCID, IMSI, state, etc.), network provider info (PLMN code, roaming status, etc.), signal metrics (RSSI, RSRQ, etc.), packet/byte counters (from modem, not from Linux kernel) and more. Next big feature request was Radio Silence mode, where we were asked to enable control of the modem power state from Local Profile Server. Recently, a requirement came to support multiple modems and user credentials. All of these requirements were implemented on top of that poor shell script (now split into multiple .sh files stored here).

There are multiple issues with our current implementation:

1. The "management agent" for modems is **way too complex for a shell script**. What we have today abuses the power of shell for something that should be implemented using a proper programming language. We are using nesting, cycles, running commands and functions in the background, using pipes for exchanging data between processes and other features to the degree far beyond what the shell is meant to be used for. The outcome is a very inefficient implementation that starts and stops some processes all the time (e.g. to poll the modem state; for string manipulations; to produce json output; etc.).
2. The CLI tools qmicli and mbimcli are also not very efficient. Every call requires to start a new process, establish connection with the modem, request a so-called client ID, use that ID in the request, receive the response and print it to stdout, release client ID, close connection, stop process (and then we have to parse stdout). Furthermore, these tools **often get stuck** (maybe allocation and release of client IDs has some limits) and we have to run them inside "timeout" to avoid triggering watchdog. It is better to keep a single connection open and use the same client ID, but that's only possible by directly using libqmi and libmbim libraries. And lastly, not all QMI/MBIM API methods have CLI commands provided.
3. CLI tools do not allow watching for modem notifications. This means that our shell script has to **periodically poll** for status updates (e.g. to react to connectivity loss). This adds delay and results in poor reaction time. It also makes the shell script even more inefficient because it keeps re-running the same set of commands quite often. With multiple modems, some command (i.e. process) is being run almost all the time.
4. As much as the shell script is complicated, it is **not that sophisticated** after all. For example, on any config change, everything is re-created (i.e. reconnected) for every modem. Handling every possible config change individually and effectively in this shell script is beyond imagination.
5. Because we use shell as opposed to Golang, input/output is done **using files instead of pubsub**. So wwan is rather disconnected from other EVE microservices.
6. Our solution is apparently **not very robust** based on the number of customer issues received over time. This is partially because the QMI protocol is proprietary and specification is not publicly available. All we have is some very limited documentation provided for qmicli (more like comments). It is not always clear what is the right sequence of commands that we should call and what the parameters should be (some are not even documented). To a large degree we are therefore maintaining our script using a trial and error method.
7. Every modem model has its "**nuances**", basically a "is-it-a-bug-or-is-it-a-feature" behavior added accidentally by manufacturers. It is therefore necessary to handle different models slightly differently, but that's not realistic in our script.
8. At this point, I cannot imagine evolving this shell script further. For some advanced use-cases like 5G and beyond we will have to find a better way (such as what is being presented and evaluated in this document).

Introducing ModemManager

ModemManager (sometimes abbreviated to **MM** in this document) is a system daemon which controls WWAN (2G/3G/4G/5G) devices and connections. Alongside NetworkManager, ModemManager is the default mobile broadband management system in most standard GNU/Linux distributions (Debian, Fedora, Ubuntu, Arch Linux…), and is also available in custom systems built with e.g. buildroot, yocto/openembedded or ptxdist. ModemManager may also be used in routers running openwrt.

MM provides a **standardized and consistent interface** for interacting with different types of modems, such as USB dongles, embedded cellular modules or RS232 modems, enabling seamless integration of these devices into the Linux ecosystem. MM uses **plugin architecture**, where each plugin is a dynamically loaded library implementing a *MMPlugin* interface for a class of modems. For example, there is `libmm-plugin-sierra.so` implementing support for modems from Sierra Wireless. As of this writing, there are 48 plugins in total, covering all relevant modem manufacturers out there.

ModemManager is an actively developed project with contributions from a diverse group of developers and maintainers, the main one being Aleksander Morgado. A wide range of companies from different sectors are using and frequently contributing to this project, including: Google (Chromium), Ericsson, T-Mobile US (used here), Deutsche telekom, RedHat, Canonical, Samsung, Quectel, Huawei and many less known companies. The project is hosted on **freedesktop.org** and licensed under **GNU LGPLv2.1**. It is written in **C**, using glib and gio.

Here are some useful links related to ModemManager:

1. **Official Website**: You can find detailed information about ModemManager, its features, and its usage on its official website
2. **Repository**: The source code and issue tracking for ModemManager can be found on GitLab repository
3. **Documentation**: Comprehensive documentation on how to use ModemManager is all that was needed to integrate MM with EVE (see next section)
4. **Mailing List**: To stay updated with the latest developments and discussions around ModemManager, you can join the mailing list

ModemManager interfaces

Important for us, is also to understand how ModemManager interacts with the operating system. MM and its dependencies are typically started by systemd. However, this is optional and MM can be built without systemd support. MM uses **libqmi**, **libmbim** and sometimes **AT commands** to control modems. These libraries are developed under the same project umbrella by the same contributors (led by Aleksander). In EVE, we are already using libqmi and libmbim through the provided CLI tools qmicli and mbimcli. ModemManager also depends on **udev** to discover modems and to detect hardware changes (modem (dis)connected from USB port etc.).

ModemManager is controlled using APIs exposed via **DBus**. These APIs are imperative in nature, meaning that for a declaratively defined config, there must be an agent translating the config into the corresponding sequence of MM API calls with the right arguments. These APIs allow getting state data, metrics, calling methods and watching for notifications (aka signals in DBus terminology). There is a detailed documentation for MM application interface, describing all DBus interfaces, objects, signals, properties and methods provided by MM. **DBus daemon** is therefore a mandatory dependency of MM. Also used is Polkit to define and handle policy restricting access to these DBus APIs, but this is optional. Access can be allowed for any DBus client (that is able to access DBus UNIX socket) and Polkit does not have to be installed.

MM on its own does not do much, just discovers modems and allows to print some details about them using a CLI tool **mmcli**.
In standard Linux distributions, it is up to **NetworkManager daemon** to:

- tell ModemManager what it should do, e.g. connect modem at <DBus-path> with parameters <apn,user,password,...>
- obtain IP settings (IP address, gateway IP, DNS servers) and apply them in the network stack (i.e. this is done by NetworkManager, not MM, which as a result does not depend on the Linux network stack)
- obtain modem state data to then display via UI and nmcli (NetworkManager CLI tool)
- trigger reconnect (if enabled by user) when MM sends notification about a modem losing connection

EVE microservice controlling ModemManager

In order to evaluate the feasibility of using ModemManager inside EVE to control cellular modems, I ended up preparing pretty much a **complete integration** inside my EVE fork. All features of mobile connectivity currently offered by EVE are covered by this integration and now implemented using ModemManager.

The only notable limitation is that between Static, DHCP and PPP bearer IP methods, only **Static** is supported. But this is also the case with our current solution based on the shell script. With modern modems implementing QMI/MBIM protocols, the PPP method, which merely emulates a legacy analog modem, is no longer recommended and rarely used these days due to its performance and other limitations. It is therefore up to our consideration if supporting PPP for cellular connectivity in EVE is worth the extra effort and additional dependencies (pppd or some implementation in Golang). And as for DHCP, we haven't yet received any request from EVE users to support DHCP client running on the wwan interface. It seems that at least the modems supported and verified on EVE always pass full IP configuration to the host, thus it is not necessary to run a DHCP client on the host. However, since we already support DHCP for ethernet interfaces, this could be implemented with little effort and no extra dependencies (NIM would be told by the wwan microservice to start dhcpcd for wwan* interface as well).

I decided to maintain the **container separation** between pillar and wwan. Inside Dockerfile for the wwan container (see here), we continue building libmbim and libqmi libraries as before. Additionally, we build ModemManager (without systemd and polkit) and our Go agent, called **mmagent**, to control MM. We also install DBus daemon and udev using apk.

Entry point of the wwan container is still a shell script, but so much simpler in this case (see here). First it loads all kernel modules used for control and data-plane between modems and Linux, then starts DBus daemon, Udev daemon, ModemManager and lastly our mmagent.

MMAgent

mmagent is an EVE microservice, leveraging agentbase, logging system, pubsub, types and other common packages from pillar. It has similar set of responsibilities as NetworkManager in standard Linux distributions:

- Translate declarative configuration to imperative DBus calls of MM API
- Configure wwanX interfaces in the Linux network stack with IP settings obtained from MM (which receives them from the network provider)
- Try to reconnect modem when it loses connection
- Publish state data, metrics

Of course, there are significant differences between NetworkManager and mmagent that stems from EVE not being a standard Linux distribution. The main difference being is that mmagent receives configuration from a remote controller. It comes in the form of **WwanConfig** pubsub publication from **NIM** microservice, which builds it from DevicePortConfig that zedagent created based on a portion of EdgeDevConfig.

Output of mmagent are publications:

- **WwanStatus**: modem information/status, SIM info/status, network info, etc.
- **WwanMetrics**: packet and byte counters (actually, only byte counters are exposed by MM)
- **WwanLocationInfo**: location information obtained from GNSS receiver that is part of the modem

Subscribers of these publications are:

- **zedagent**: to forward status, location and metrics to LPS, LOC and controller
- **NIM**: to build DeviceNetworkStatus
- **zedrouter**: to publish status and location to apps via metadata HTTP server

The source code of mmagent written in Go is available here.

All mentioned pubsub types that mmagent interacts with are defined here.

Additionally to these, mmagent reads global config (ConfigItemValueMap) and controller + node certificates to decrypt user password encrypted using EVE's object-level encryption method.

Full difference between the current EVE master and this MM integration can be seen here.

Evaluation

After testing ModemManager integrated with EVE on a device with two modems (Sierra Wireless EM7565 and QUECTEL EC21), I can now present the evaluation results, list the pros and cons of this solution, and share my thoughts on the way forward.

First, I must point out that the **documentation of ModemManager was very helpful and clear**.
I was able to integrate the product with EVE in just around one month, with no major problems and without having to ask MM contributors for any help. APIs are clearly described and using native Go Dbus bindings it was fairly easy to start controlling MM from my mmagent.

During my testing/development, I saw **only one crash of MM** and the problem disappeared after I got the startup sequence of MM and its dependencies right.

ModemManager did not have any problems recognizing, initializing and connecting my modems. It could properly handle even if a modem was (dis) connected to/from device at runtime (by (un)plugging modem dongle).

Additionally, there is a firmware problem with my EM7565 modem it seems - it can get stuck sometimes shortly after boot. I'm able to reproduce this problem on the original upstream EVE (with shell script controlling the modem), on this evaluated EVE with ModemManager, but also on Ubuntu 22.04. The problem can be fixed by restarting the modem. Much to my satisfaction, ModemManager is able to **detect stuck modems and restart them automatically**. As a result, connectivity recovers fairly quickly in just a few minutes. Modems getting stuck is quite a frequent problem related to firmware bugs. Internet forums are full of Netdev watchdogs reporting stuck wwan transmit queues. Developers of MM have apparently decided to implement some recovery mechanism (I tried as well, see here and here, but it is not so reliable). I'm attaching a watchdog from my modem at the end of this document as an appendix.

But apart from the watchdog, which is not an EVE problem, I was able to test all features of modem connectivity that EVE should support and all common scenarios without issues. This included enabling/disabling a modem, changing configuration, fail-over from eth to wwan connectivity, radio silence mode triggered from LPS, publishing location info, scanning visible providers, setting user credentials and more. A major improvement is the **reaction time**. With MM, our agent is getting notification as soon as modem state changes. We are therefore able to react to connectivity loss, modem discovery and other events very quickly. With our currently used shell script, we run polling with some period and this period is an extra delay that limits the reaction time.

Image Size

An important aspect that needs evaluation is the image size, which was expected to increase because we only added stuff to the wwan container and nothing substantial in size was removed (libqmi and libmbim had to remain). The only real advantage of the shell script is its small size. Having microservice written in Go adds several MBs into the image size as we are already aware of, having put most of our microservices under one zedbox binary. For the time being, I decided to have mmagent as a separate binary for cleaner separation, but nothing really prevents me from putting it under zedbox in the pillar container. To communicate with MM over DBus, only `/run/dbus/system_bus_socket` needs to be shared between the containers, which it already is.

ModemManager itself, along with its dependencies dbus and udev daemon, doesn't significantly increase the image size.
I tried to build rootfs image for the latest master and then for this MM integration and the increase is **12MB**. I looked at the uncompressed content and mmagent indeed contributes the most:

```
ls -al *img

-rw-rw-r-- 1 mlenco mlenco 220753920 sep 20 10:09 0.0.0-master-b3bdbc6e-kvm-amd64.img
-rw-rw-r-- 1 mlenco mlenco 232804352 sep 20 10:47 0.0.0-modem-manager-e3186241-kvm-amd64.img

unsquashfs -d img_extracted 0.0.0-modem-manager-80e5cfae-kvm-amd64.img
find img_extracted/containers/services/wwan -type f -exec du -h {} + | sort -rh | head -n 25

17M     img_extracted/containers/services/wwan/lower/usr/bin/mmagent
4,1M    img_extracted/containers/services/wwan/lower/usr/lib/libqmi-glib.so.5
2,6M    img_extracted/containers/services/wwan/lower/usr/bin/ModemManager
2,5M    img_extracted/containers/services/wwan/lower/lib/libcrypto.so.1.1
1,7M    img_extracted/containers/services/wwan/lower/usr/lib/libgio-2.0.so.0.7200.4
1,5M    img_extracted/containers/services/wwan/lower/usr/lib/libmm-glib.so.0
1,1M    img_extracted/containers/services/wwan/lower/usr/lib/libglib-2.0.so.0.7200.4
824K    img_extracted/containers/services/wwan/lower/bin/busybox
764K    img_extracted/containers/services/wwan/lower/usr/lib/libmbim-glib.so.4
592K    img_extracted/containers/services/wwan/lower/lib/ld-musl-x86_64.so.1
588K    img_extracted/containers/services/wwan/lower/usr/bin/qmicli
512K    img_extracted/containers/services/wwan/lower/lib/libssl.so.1.1
500K    img_extracted/containers/services/wwan/lower/usr/lib/libzstd.so.1.5.2
364K    img_extracted/containers/services/wwan/lower/usr/lib/libpcre.so.1.2.13
332K    img_extracted/containers/services/wwan/lower/lib/libmount.so.1.1.0
320K    img_extracted/containers/services/wwan/lower/usr/lib/libgobject-2.0.so.0.7200.4
308K    img_extracted/containers/services/wwan/lower/lib/libblkid.so.1.1.0
304K    img_extracted/containers/services/wwan/lower/bin/udevadm
296K    img_extracted/containers/services/wwan/lower/usr/lib/libdbus-1.so.3.32.3
288K    img_extracted/containers/services/wwan/lower/sbin/udevd
272K    img_extracted/containers/services/wwan/lower/usr/bin/mmcli
212K    img_extracted/containers/services/wwan/lower/etc/ssl/certs/ca-certificates.crt
208K    img_extracted/containers/services/wwan/lower/usr/bin/mbimcli
200K    img_extracted/containers/services/wwan/lower/usr/bin/dbus-daemon
180K    img_extracted/containers/services/wwan/lower/lib/libapk.so.3.12.0
```

Let's now summarize pros and cons of using ModemManager.

Pros

- **More efficient implementation**: C and Go code as opposed to shell; notifications used as opposed to polling; MM reusing QMI/MBIM connection with a modem; communication with MM is in binary format as opposed to parsing string output from CLI tools; config changes handled efficiently as opposed to recreating everything
- **Better stability**: This still needs more testing across a wider range of modems, devices and cellular networks to confirm, but my initial experience suggests that ModemManager will provide a more stable solution with less customer issues. After all, customers often complain that there is a problem with mobile connectivity on EVE while the same works on Ubuntu (where MM is running). During my testing I didn't experience any issues and MM was even able to restart and recover my modem when it got stuck and a kernel watchdog was triggered.
- **Better reaction time**: MM uses so called QMI/MBIM indications to get state updates from modems and mmagent uses DBus signals to watch for changes published by MM. This allows us to avoid polling and instead get notified when something changes. mmagent can therefore react almost immediately to lost connectivity and other events.
- **Better expertise:** MM community has deeper understanding and more experience working with cellular modems than we do. Furthermore, they have access to QMI spec and possibly other material from modem manufacturers that is not publicly available. Using MM we will be in position to post questions on the mailing lists, create tickets in GitLab, etc. We cannot expect them to help us with our current custom solution.
- **Wider modem support:** With MM, it should be easier to bring in new modems and use them with EVE. It already provides 48 plugins covering all relevant modem manufacturers. It also supports modems connected via serial ports (as opposed to USB) and also some that do not understand QMI/MBIM protocols (in that case AT commands are typically used instead).
- **Integration with pillar/pubsub**: mmagent uses pubsub and other common packages of pillar, such as logging and agentbase, for a native integration with pillar microservices. Meanwhile, the shell script requires the use of files inside the in-memory /run file-system to transfer config /status/metrics and fsnotify to get notified about changes. We can avoid this special solution and unify wwan with other EVE microservices.

Cons

- **Image size increase**: The size of EVE rootfs image will increase by 12MB. This can be reduced by merging mmagent with pillar, at the cost of making pillar and wwan containers tightly-coupled. In the current integration of EVE with MM, the interaction between pillar and wwan is defined only by pubsub channels and their messages, meaning that for pillar it is irrelevant how wwan is implemented.
- **Missing features**: There are few things not exposed by ModemManager API. For example, packet and drop counters are not available and only byte counters can be retrieved. Also, it is not possible to differentiate between inactive SIM slot with SIM card inserted and inactive SIM slot without SIM card. With qmicli we can tell these states apart. Lastly, for a visible network provider (which is scanned and published if `wwan.query.visible.providers` is enabled), we cannot tell if roaming is required (we publish that as a boolean flag).
  Neither of these are particularly important features and these gaps could be filled in with our future contributions to ModemManager.

- **More dependencies**: Using ModemManager introduces new dependencies, notably the D-Bus daemon and udev. In standard Linux distributions, these daemons are utilized by multiple services, which justifies their presence. However, in EVE, they would serve the sole purpose of managing cellular connectivity (although there is some potential for more uses of udev, especially for handling of hot-plug devices). On the other hand, it appears that neither of these daemons consume significant resources, as confirmed by checking with 'top'.

```
PID   PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
28    1    root     S      706m  4%   0   0% mmagent
25    1    root     S      16576 0%   0   0% ModemManager --debug
38    1    root     S      9760  0%   0   0% /usr/libexec/qmi-proxy
12    1    root     S      5596  0%   0   0% udevd --debug --daemon
 1    0    root     S      1616  0%   0   0% {mm-init.sh} /bin/sh /usr/bin/mm-init.sh
10    1    messageb S      1472  0%   0   0% dbus-daemon --system
```

Conclusion

To wrap up this evaluation and give my personal opinion, **I'm in favor** of replacing our shell script with ModemManager going forward. I see many advantages, with better stability and reaction time being at forefront, and with slight increase of image size as the only major downside. However, this can be mitigated by merging mmagent into zedbox if deemed necessary.

But please note that I tested this only with my personal device and just two modems. I suggest that some of the EVE users that plan to rely on modems heavily could give it a try on their devices and **provide feedback**. Images for testing are published on dockerhub:

- Eve image for AMD64: milan4zededa/eve:0.0.0-modem-manager-4cc09d2a-kvm-amd64
- Eve image for ARM64: milan4zededa/eve:0.0.0-modem-manager-4cc09d2a-kvm-arm64

Should we decide to use ModemManager in EVE, there is a **follow-up story** to productize this integration. This mostly involves more testing, writing documentation, cleaning up the code a bit and getting it through reviews.

Appendix: watchdog from EM7565

```
[  159.218839] ------------[ cut here ]------------
[  159.218850] NETDEV WATCHDOG: wwan0 (cdc_mbim): transmit queue 0 timed out
[  159.218867] WARNING: CPU: 0 PID: 0 at net/sched/sch_generic.c:467 dev_watchdog+0x11e/0x18a
[  159.218871] Modules linked in: dummy usbmouse usbkbd usbhid cdc_acm leds_gpio gpio_pca953x regmap_i2c hpwdt
hwmon_vid zfs(PO) zunicode(PO) zzstd(O) zlua(O) zavl(PO) icp(PO) zcommon(PO) znvpair(PO) spl(O) qmi_wwan option
cdc_mbim cdc_ncm cdc_ether cdc_wdm usbnet mii qcserial usb_wwan usbserial btusb btrtl btbcm btintel bluetooth
ecdh_generic ecc iwlmvm led_class mac80211 e1000e i2c_i801 i2c_smbus iwlwifi cfg80211 tpm_crb
[  159.218915] CPU: 0 PID: 0 Comm: swapper/0 Kdump: loaded Tainted: P               O          5.10.186-
linuxkit #1
[  159.218918] Hardware name: GEEKOM Mini IT 8/Mini IT 8, BIOS U6G03 07/21/2022
[  159.218923] RIP: 0010:dev_watchdog+0x11e/0x18a
[  159.218926] Code: 20 01 01 00 75 36 48 89 ef c6 05 dd 20 01 01 01 e8 9e f9 fb ff 44 89 e1 48 89 ee 48 c7 c7
24 12 58 af 48 89 c2 e8 41 4c 1f 00 <0f> 0b eb 0e 41 ff c4 48 05 40 01 00 00 e9 5c ff ff ff 48 8b 83 a0
[  159.218933] RSP: 0018:ffffad0880003ed8 EFLAGS: 00010286
[  159.218936] RAX: 0000000000000000 RBX: ffff949205fa9440 RCX: 0000000000000027
[  159.218939] RDX: 0000000000000003 RSI: ffffad0880003d60 RDI: ffff94955dc1c3e0
[  159.218943] RBP: ffff949205fa9000 R08: ffffffffaf8dada8 R09: 0000000000000017
[  159.218946] R10: 3a474f4448435441 R11: 572056454454454e R12: 0000000000000000
[  159.218949] R13: 00000000ffffc8f8 R14: ffffad0880003f28 R15: ffffffffaea118b9
[  159.218953] FS:  0000000000000000(0000) GS:ffff94955dc00000(0000) knlGS:0000000000000000
[  159.218957] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  159.218960] CR2: 000000c000bd7000 CR3: 0000000112240003 CR4: 00000000003726b0
[  159.218963] Call Trace:
[  159.218966]  <IRQ>
[  159.218971]  ? __warn+0x98/0xda
[  159.218974]  ? dev_watchdog+0x11e/0x18a
[  159.218979]  ? report_bug+0x96/0xda
[  159.218983]  ? handle_bug+0x46/0x6e
[  159.218987]  ? exc_invalid_op+0x14/0x65
[  159.218990]  ? asm_exc_invalid_op+0x12/0x20
[  159.218993]  ? dev_deactivate_queue+0x25/0x25
[  159.218998]  ? dev_watchdog+0x11e/0x18a
[  159.219001]  ? dev_watchdog+0x11e/0x18a
[  159.219004]  ? dev_deactivate_queue+0x25/0x25
[  159.219008]  call_timer_fn+0x63/0xfb
[  159.219011]  __run_timers+0x146/0x188
[  159.219015]  ? timekeeping_get_ns+0x19/0x33
[  159.219018]  run_timer_softirq+0x19/0x2d
[  159.219021]  __do_softirq+0xf7/0x233
[  159.219025]  asm_call_irq_on_stack+0xf/0x20
[  159.219028]  </IRQ>
[  159.219031]  do_softirq_own_stack+0x31/0x42
[  159.219035]  __irq_exit_rcu+0x45/0x84
[  159.219038]  sysvec_apic_timer_interrupt+0x6c/0x7a
[  159.219041]  asm_sysvec_apic_timer_interrupt+0x12/0x20
[  159.219046] RIP: 0010:cpuidle_enter_state+0x12c/0x1f2
[  159.219049] Code: ff 45 84 ff 74 1d 9c 58 0f 1f 44 00 00 0f ba e0 09 73 09 0f 0b fa 66 0f 1f 44 00 00 31 ff
e8 b0 19 8a ff fb 66 0f 1f 44 00 00 <45> 85 f6 0f 88 99 00 00 00 49 63 c6 4c 2b 24 24 48 6b c8 68 48 6b
[  159.219055] RSP: 0018:ffffffffaf803e68 EFLAGS: 00000246
[  159.219058] RAX: ffff94955dc2ec80 RBX: ffffcd087fc2d100 RCX: 000000000000001f
[  159.219061] RDX: 0000000000000000 RSI: 0000000000000000 RDI: 0000000000000000
[  159.219065] RBP: 0000000000000001 R08: 00000000ffffffff R09: 071c71c71c71c71c
[  159.219068] R10: 0000000000000020 R11: 000000000000001b R12: 00000025122e9ed8
[  159.219071] R13: ffffffffaf991000 R14: 0000000000000001 R15: 0000000000000000
[  159.219077]  ? cpuidle_enter_state+0x103/0x1f2
[  159.219080]  cpuidle_enter+0x2a/0x3a
[  159.219084]  do_idle+0x17c/0x1ee
[  159.219087]  cpu_startup_entry+0x1d/0x1f
[  159.219091]  start_kernel+0x524/0x54b
[  159.219096]  secondary_startup_64_no_verify+0xb0/0xbb
[  159.219100] ---[ end trace 122b4cdcf5fdb33e ]---
```