Универзитет у Београду
Електротехнички факултет

Дипломски рад

Тема: Интеграција Percepio алата са FreeRTOS оперативним системом на MSP430 развојној платформи

Предмет: Системи у реалном времену

Студент: Стефан Кангрга-Микулић 0526/2017,
Ментор: проф. др Иван Поповић,

Београд, септембар 2023.

Садржај

1. Увод

Платформе микроконтролера су у последње време нашле широку примену у широком спектру уграђених система. Ови системи обухватају широк спектар домена, укључујући, али не ограничавајући се на Интернет ствари (*IoT*), системе контроле аутомобила, медицинске уређаје и индустријску аутоматизацију. Микроконтролери су основа ових уграђених система, служећи као рачунарско срце уређаја који напајају наш савремени свет. Да би се омогућила функционалност ових уграђених система, примењују се сложена софтверска решења, која се често ослањају на оперативне системе за рад у реалном времену (*RTOS - Real-Time operating system*) како би се обезбедило прецизно време и одзив. Како ови системи расту у софистицираности, програмери се сусрећу са огромним изазовом обезбеђивањем исправности софтверских операција уз придржавање строгих временских ограничења. Отклањање грешака и оптимизација таквог софтвера може бити захтеван подухват, с обзиром на замршену интеракцију таскова, прекида и догађаја у реалном времену. Овај изазов додатно отежава слојевита апстракција коју уводе *RTOS*-ови које отежавају увид у понашање система.

Овде долазе у игру алати за праћење прилагођени системима заснованим на *RTOS*-у. Ови алати обезбеђују програмерима кључну тачку увида у унутрашњи рад њиховог софтвера, нудећи средства за дијагностиковање проблема, побољшање перформанси и проверу да су захтеви у реалном времену доследно испуњени. *Percepio Tracealyzer* се истиче као пример међу овим алатима, нудећи моћне могућности за праћење програма који раде на популарним *RTOS*-овима.

Циљ рада је интеграција *Percepio Tracealyzer* са *FreeRTOS*-ом коју *Tracealyzer* нуди и укључује подршку за статистику за анализу времена извршавања *FreeRTOS*-а и удубљивање у његове могућности коришћења. *Tracealyzer* омогућава не само интеграцију са *FreeRTOS*-ом већ се може прилагодити за рад са осталим *RTOS*-овима, па чак и апликације које нису *RTOS*. Биће потребно да се прилагоди код пројекта за праћење догађаја и могу постојати проблеми везано за директну подршку за специфичну хардверску платформу, иако *Tracealyzer* подржава широк спектар микроконтролера и развојних окружења.

2. Преглед алата за надгледање за уграђене системе

Алати за надгледање су суштински инструменти у раду са уграђеним системима, који служе као помоћ у развоју, отклањању грешака и оптимизацији апликација у реалном времену и ограниченим ресурсима. Ови алати пружају увид у понашање њиховог софтвера током извршавања, олакшавајући идентификацију уских грла, оптимизацију кода и уверавање да се захтеви у реалном времену доследно испуњавају. Међу њима најпознатијима су *Percepio Tracealyzer*, *SEGGER SystemView* и *VisualGDB LiveWatch*. Алати за праћење нуде мноштво предности које значајно доприносе развоју и валидацији уграђених система као што су:

- видљивост у реалном времену - ови алати обезбеђују увид у извршавање кода у реалном времену, омогућавајући програмерима да надгледају заказивање таскова, прекидају интеракције и коришћење ресурса,

- ефикасно отклањање грешака - олакшавајући идентификацију проблема као што су застоји и кршења времена, алати за праћење убрзавају процес отклањања грешака,

- оптимизација перформанси - програмери могу да користе ове алате за прикупљање метрике о времену извршења, коришћењу меморије и интеракцијама таскова, што доводи до оптимизације кода за побољшану ефикасност и управљање ресурсима,

- валидација и усклађеност - алати за праћење помажу у валидацији софтверских система, осигуравајући да они раде у оквиру унапред дефинисаних ограничења у реалном времену и да су у складу са захтевима специфичним за апликацију,

- документација - аутоматско генерисање дневника и креирање извештаја служе као вредна документација за понашање кода и анализу перформанси.

Упркос бројним предностима, алати за праћење могу представљати одређене изазове:

- извршни трошкови - увођење надгледања може довести до одређеног степена оптерећености током извршавања, што потенцијално утиче на понашање система у реалном времену,

- сложеност - постављање и конфигурисање алата за праћење може бити сложено,

- потрошња ресурса - неки алати за праћење троше додатну меморију и процесорску снагу, што може бити проблем за уграђене системе са ограниченим ресурсима,

- трошкови - у зависности од алата, трошкови лиценцирања могу бити фактор за пројекте са буџетским ограничењима.

SEGGER SystemView је једна од апликација која омогућава анализу и профилисање понашања уграђеног система. Може се користити на било ком систему који нуди приступ за отклањање грешака. Посебно је дизајниран са јаким фокусом на апликације засноване на *RTOS*-у и истиче се у визуелизацији распореда таскова, руковању прекидима и

догађајима специфичним за *RTOS*. Обезбеђује графичку визуелизацију у реалном времену, наглашавајући непосредно разумевање понашања система. За разлику од осталих алата нуди бесплатну верзију *SystemView*-а са ограниченим функцијама, а додатне функције и комерцијалне лиценце су доступне за куповину. *SEGGER* има сопствени сет развојних алата, јаку заједницу и пружа одличну техничку подршку, што је драгоцено за кориснике који траже помоћ. Са *SEGGER J-Link*-ом и његовом технологијом преноса у реалном времену (*RTT*), *SystemView* може континуирано да снима податке и анализира их и визуализује у реалном времену. Функционише тако што на циљној страни треба да буде укључен мали софтверски модул, који садржи *SystemView* и *RTT*. *SystemView* модул прикупља и форматира податке монитора и прослеђује их *RTT*-у. *RTT* модул чува податке у циљном баферу временском ознаком високе прецизности, што омогућава континуирано снимање помоћу *J-Link*-а на подржаним системима. Од ресурса користе се комбиновано око 600 бајтова RAM-а је довољно за континуирано снимање помоћу *J-Link*-а (за разлику од *Percepio Tracealyzer* до неколико kB). Може се користити на било ком *CPU*-у. Континуирано снимање у реалном времену може се обавити на било ком систему који подржава *J-Link RTT* технологију.

VisualGDB LiveWatch је проширење за надгледање и отклањање грешака за *Visual Studio*, првенствено дизајнирано да побољша искуство отклањања грешака за уграђене системе и за разлику од осталих, интегрисан је са *Microsoft Visual Studio* што га чини погодним за програмере који су већ упознати са овим популарним интегрисаним развојним окружењем. Омогућава испитивање стања глобалних променљивих у уграђеном програму без заустављања извршавања програма. *LiveWatch* је такође дизајниран за апликације засноване на *FreeRTOS*-у.

Percepio Tracealyzer омогућава визуелну дијагностику таскова и објеката кернела за програмере наменских система заснованим на *RTOS*-у или Linux-у. Није ограничен на *RTOS*, може се користити са системима ван *RTOS*-а и са различитим развојним окружењима, то је самостални алат са сопственим корисничким интерфејсом. Нуди више од 30 прегледа који даје увид у понашање неког система у реалном времену (заказивање таскова, редослед догађаја, коришћење ресурса и тако даље) и олакшава оптимизацију програма. Алат помаже у анализирању проблема са временом тако што се визуализује време извршавања таскова, промене контекста и прекида. Подржава и праћење засновано на догађајима омогућујући снимање корисничких догађаја и података, то јест дијагностиковање специфичних понашања. Интерфејс је прилагођен кориснику који поједностављује анализу снимања као што су зумирање, филтрирање, напомене за лакшу навигацију и разумевање података о праћењу. Захтева посебну комерцијалну лиценцу, са различитим опцијама лиценцирања, укључујући различита издања и додатке.

3. Percepio Tracealyzer

Tracealyzer се ослања на библиотеку снимача који треба бити укључен у циљни (*target*) систем и та библиотека позива *RTOS* кернел како би се сачували важни догађаји као што су промена контекста, *RTOS API* позиви као и памћење „корисничких догађаја“. Главни кораци су: **конфигурисање *build* подешавања да би се укључили неопходне компоненте и библиотеку за праћење, позиви функција за иницијализацију библиотеке снимача и конфигурисање параметара у библиотеци како би одговарало циљном процесору микроконтролера и вези ка циљном (*target*) систему**.

Постоје два режима снимања: „*streaming*“ и „*snapshot*“. У „*streaming*“ режиму подаци се прикупљају у *host* развојном систему и на тај начин омогућава праћење на дужи период, док у „*snapshot*“ режиму подаци се снимају и складиште директно на циљном систему у RAM меморији. Могуће је конфигурисати величину кружног бафера и понашање када се напуни, „прегазити“ старе податке или зауставити праћење. У режиму „*snapshot*“ после снимања могуће је траг у циљном RAM баферу сачувати у датотеци на страни домаћина подразумевајући да постоји начин за преузимање података. У циљном систему се чувају као догађаји у виду четворобајтних записа. Подаци о праћењу могу да се отпреме на рачунар (*Host system*) на неколико начина:

- директно из *Tracealyzer* уз помоћ подржане *debug probes* као што су *SEGGER J-Link* и *ST-LINK*,
- коришћење додатка у апликацији *Percepio* које постоје у *IDE* заснованим на *Eclipse*-у,
- програмски у циљном систему прилагодити пренос на *host* систему,
- ручно чување RAM у развојном окружењу (*IDE*) под опцијом „Сачувај меморију“
- коришћење debugger скрипте, у *Keil µVision* и *IAR Embedded Workbench*.

Tracealyzer подржава датотеке у бинарном формату (.bin), у *Intel Hex* формату(.hex) и у *Macromedia Xtra Cache files* MCH(.mch) формату. Подаци снимача се чувају у једном блоку података, на кога указује показивач *RecorderDataPtr*.

4. Оперативни систем за рад у реалном времену (RTOS)

Оперативни системи за рад у реалном времену су специјализовани системи дизајнирани за апликације где су одговор у реалном времену и детерминистичко понашање критични. Кључне карактеристике и концепти *RTOS*-а укључују:

- детерминистичко време - систем гарантује да ће таск или нит извршити у оквирима одређених временских ограничења,

- управљање таскова - програмери имају могућност да креирају и управљају тасковима или нитима на основу њихових приоритета и алгоритми распоређивања,

- управљање прекидима – *RTOS*-ови ефикасно управљају хардверским и софтверским прекидима,

- управљање ресурсима – обезбеђени су механизми за управљање дељеним ресурсима коришћењем семафора, мутекси и редови порука,

- комуникација између таскова - обезбеђени су механизми комуникације и синхронизације за таскове за размену података и координацију својих активности,

- услуге тајмера – корисне за планирање таскова и догађаја у прецизним временским интервалима или одређеним временским тачкама,

- планирање засновано на приоритетима - тасковима се додељују приоритети, а планер обезбеђује да таскови вишег приоритета имају предност над тасковима нижег приоритета (заказивање засновано на приоритетима је кључно за испуњавање рокова у реалном времену),

- ниски трошкови - дизајнирани су тако да имају мале трошкове рада, што их чини погодним за уграђене системе са ограниченим ресурсима,

- *RTOS* кернел - језгро *RTOS*-а се често назива „*RTOS* кернел“ пружа основне услуге за управљање тасковима, заказивање и синхронизацију.

Неки примери оперативних системи за рад у реалном времену које се користе су *RTOS-32, VxWorks, QNX, µC/OS-II, µC/OS-III, ThreadX, embOS, Nucleus RTOS, RTEMS, ChibiOS/RT, Zephyr, CMSIS-RTOS, NuttX, AliOS Things, TI-RTOS (*раније *SYS/BIOS), RIOT, Salvo RTOS, LwIP, OpenRTOS, Phoenix-RTOS, T-Kernel, AliRTOS, Mbed OS, TinyOS* и многи други. Како микроконтролери и остали системи расту у софистицираности а технички захтеви су огромни, компаније су креирале окружења за подршку *RTOS*-а, да подржавају што већи број хардверских уређаја и тако су настале претходно наведени оперативни системи. Неке ћемо издвојити и навести предности и мане.

Zephyr подржава преко 170 хардверских уређаја (укључујући *Arm Cortex-M, Intel x86, and RISC-V*), дизајниран да буде модуларан, подржава низ комуникационих протокола (на пример *TCP/IP, Bluetooth Low Energy,* and *USB*), али са друге стране може бити сложенији од неких других *RTOS*-а (што би могло да га учини изазовнијим за учење и коришћење) и ослања се на неколико библиотека трећих страна за одређене функције.

Wind River VxWorks је познат по свом детерминизму и поузданости у реалном времену, што га чини погодним за критичне апликације и апликације критичне за безбедност, нуди низ компоненти међуверског софтвера као што су умрежавање и систем датотека, има дугу историју употребе у критичним системима али је комерцијални *RTOS* што може бити значајан фактор трошкова за пројекте.

Weston Cesium RTOS се може користити у безбедносно критичним апликацијама, може бити у складу са релевантним безбедносним стандардима када се користи са одговарајућим развојним процесима и може понудити бесплатну или јефтину

верзију(исплатив избор). Са друге стране може понудити ограничене компоненте и услуге средњег софтвера у поређењу са новијим *RTOS*-овима и програмери би морали да имплементирају одређене функције од нуле, доступност интегрисаних развојних окружења (*IDE*) може бити ограничена у поређењу са популарним *RTOS*-овима и има мању активну заједницу програмера који могу пружити подршку и помоћ.

Micrium µC/OS-III је познат по свом малом меморијском отиску али је дизајниран као *RTOS* са једним језгром, није отвореног кода (програмери немају приступ комплетном изворном коду) и *Micrium* нуди неколико варијанти *µC/OS*, укључујући *µC/OS-II* и *µC/OS-IIIИ*, што може довести до проблема са компатибилношћу приликом преласка са једне верзије на другу.

4.1 FreeRTOS

FreeRTOS је софтвер отвореног кода који се дистрибуира под *MIT* лиценцом. Ова природа отвореног кода чини га доступним широкој заједници програмера и омогућава прилагођавање различитим хардверским платформама. *FreeRTOS* није потпуни *RTOS*, већ планер у реалном времену са укљученим *TCP/IP* стеком. Често се користи као *RTOS* кернел за планирање у реалном времену, управљање међупроцесном комуникацијом (*IPC*) и синхронизовање таскова са *IPC*-овима. Дизајниран је да буде веома преносив, омогућавајући му да ради на широком спектру архитектура микроконтролера и развојних платформи. Ова преносивост је значајна предност за програмере наменских система. Често је интегрисан са развојним окружењима, програмима за отклањање грешака и компонентама средњег софтвера треће стране. Ова интеграција поједностављује процес развоја и отклањања грешака. Његова комбинација карактеристика, преносивости и природе отвореног кода учинила га је популарним *RTOS*-ом за програмере који раде на свему, од малих микроконтролера са ограниченим ресурсима до сложених наменских система високих перформанси.

5. Интеграција Percepio Tracealyzer алата са FreeRTOS оперативним системом

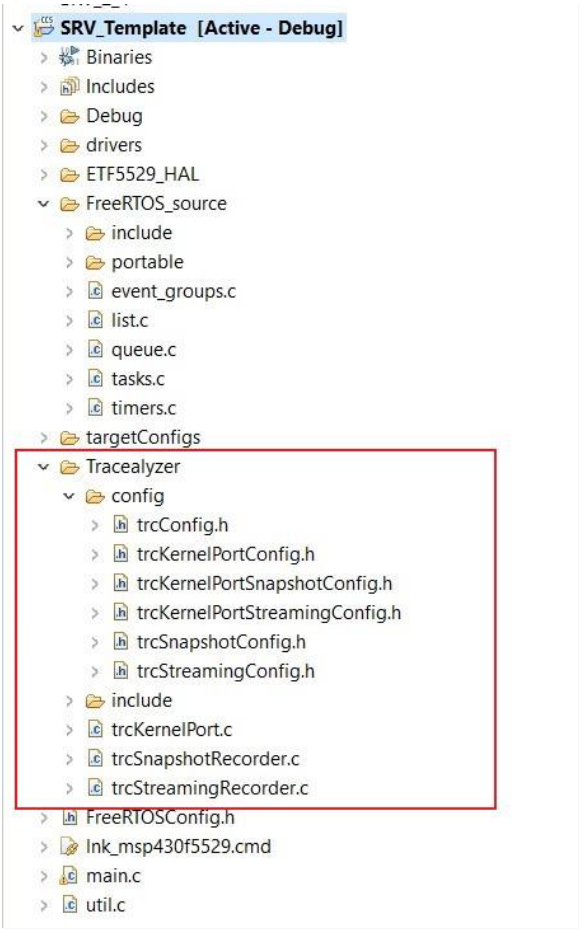
У овом одељку улазимо у основне аспекте интеграције софтвера *Percepio Tracealyzer* пролазећи кроз важне компоненте и функционалности како би се унео *Percepio* и прилагодио специфичним потребама. Основне ставке интеграције чине *TraceRecorder* библиотека, *Recorder API*, структура података снимача (*RecordData*) и прикази апликације *Tracealyzer*.

5.1 TraceRecorder библиотека

Библиотека снимача која је потребна да се дода у *FreeRTOS* пројекат може се наћи у *Tracealyzer* инсталационом директоријуму:

<Program Files>\Percepio\Tracealyzer 4\FreeRTOS\TraceRecorder.

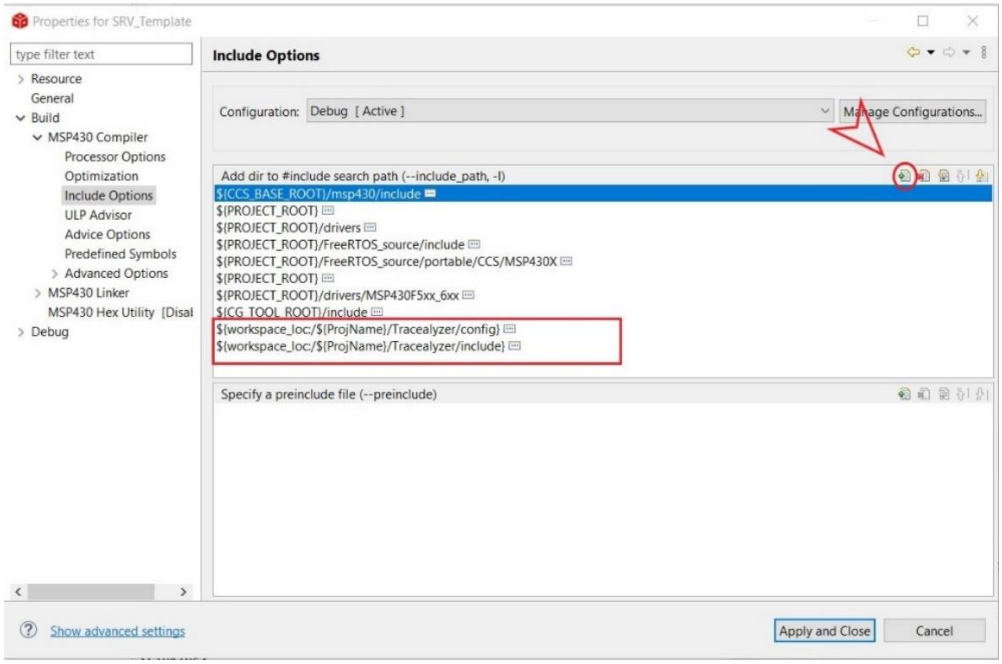
Са ове путање потребно је копирати *trcKernelPort.c*, *trcSnapshotRecorder.c*, *trcStreamingRecorder.c*, све header фајлове из датотеке „*config*“ и из датотеке „*include*“ као што је приказано на слици 1.



Слика 1.- Приказ копираних фајлова из библиотеке Tracealyzer који ће бити интегрисани у FreeRTOS пројекат

Увести све претходно копиране фајлове и датотеке како би се пројекат успешно компајлирао преко (Слика 2):

Properties→Build→Include Options.



Слика 2.- Додавање фајлова ради успешног комплајлирања и линковања (Properties→Build→Include Options)

Међу додатим фајловима, за конфигурисање снимача пројекта најбитнији су:

- */config* ***trcConfig.h*** - главна конфигурација библиотеке снимача, омогућава/онемогућава праћење одређених догађаја таскова, прекидних рутина и корисничких догађаја, конфигурише величину и понашање бафера снимача, конфигурише режим снимања - „*streaming*“ или „*snapshot*“, конфигурише величину табеле симбола која се користи за мапирање имена објеката за праћење података, омогућава/онемогућава прикупљање статистике током извршавања, анализа коришћења стека,

- */config* ***trcSnapshotConfig.h*** - конфигурација за „*snapshot*“ режим,

- */config* ***trcStreamingConfig.h*** - конфигурација за „*streaming*“ режим,

- */include* ***trcRecorder.h*** - садржи јавни програмски интерфејс апликације *API* снимача које је потребно укључити у *FreeRTOSConfig.h*,

- */include* ***trcHardwarePort.h*** - садржи неколико унапред дефинисаних хардверских портова као што су *ARM Cortex-M*, *PIC32*, *Renesas RX* и тако даље,

- */include* ***trcKernelPort.h*** - дефиниције специфичне за *FreeRTOS*, служи на интеграцију *Tracealyzer* са одређеним *RTOS* кернелом,

- ***trcKernelPort.c***,

- ***trcSnapshotRecorder.c***,

- ***trcStreamingRecorder.c*** .

У фајлу *FreeRTOSConfig.h* је потребно наместити ‘configUSE_TRACE_FACILITY’ константу на вредност „1“ који представља као „главни прекидач“ за снимање програма иначе цела библиотека снимача би била искључена из пројекта и потребно је додати у исти фајл (Слика 3).

```
/* Integrates the Tracealyzer recorder with FreeRTOS */
#if(configUSE_TRACE_FACILITY==1)
    #include "trcRecorder.h"
#endif
```

Слика 3.- Укључивање фајла trcRecorder.h у фајлу FreeRTOSConfig.h

Пре покретања снимања програма потребно је још конфигурисати следеће константе:

- */trcKernelPortConfig.h* TRC_CFG_RECORDER_MODE – помоћу вредности TRC_RECORDER_MODE_SNAPSHOT и TRC_RECORDER_MODE_STREAMING конфигурише се режим снимања,

- */trcConfig.h* TRC_CFG_HARDWARE_PORT – бира се хардверски порт чије дефиниције специфичне за хардвер се налазе у *trcHardwarePort.h* (у случају коришћења хардвера MSP430F5529 бира се TRC_HARDWARE_PORT_TEXAS_INSTRUMENTS_MSP430),

- */trcKernelPortConfig.h* TRC_CFG_FREERTOS_VERSION – мора да се поклапа са верзијом *FreeRTOS*-а,

- */trcConfig.h* TRC_CFG_RECORDER_BUFFER_ALLOCATION – селектује начин складиштења структуре података снимача:
  - TRC_RECORDER_BUFFER_ALLOCATION_STATIC,
  - TRC_RECORDER_BUFFER_ALLOCATION_DYNAMIC,
  - TRC_RECORDER_BUFFER_ALLOCATION_CUSTOM,

- */trcSnapshotConfig.h* TRC_CFG_SNAPSHOT_MODE – бира се између:
  - TRC_SNAPSHOT_MODE_RING_BUFFER (када се попуни бафер за смештање података, податак везан за нови догађај се преписује преко старих података),
  - TRC_SNAPSHOT_MODE_STOP_WHEN_FULL (снимање програма се зауставља када се попуни бафер),

- */trcSnapshotConfig.h* TRC_CFG_EVENT_BUFFER_SIZE - величина „*event*“ бафера изражена као број четворобајтних записа,

- */trcKernelPortSnapshotConfig.h* TRC_CFG_NTASK, TRC_CFG_NQUEUE, TRC_CFG_NISR, TRC_CFG_NSEMAPHORE, TRC_CFG_NTIMER, TRC_CFG_NEVENTGROUP, TRC_CFG_NSTREAMBUFFER и TRC_CFG_NMESSAGEBUFFER – максималан број истовремено активних објеката за сваку класу,

- */trcSnapshotConfig.h* TRC_CFG_SYMBOL_TABLE_SIZE – величина табеле симбола (*Symbol Table)* за складиштење стрингова на ефикасан начин,

- */trcSnapshotConfig.h* USE_SEPARATE_USER_EVENT_BUFFER – чување корисничких догађаја у засебном баферу корисничких догађаја (*User Event Buffer -UB*).

Позивом *vTraceEnable(TRC_START)* у *main* функцији за иницијализацију и почетак снимања мора бити покренут након почетног подешавања хардвера (Слика 4).

```
/**
 * @brief main function
 */
void main( void )
{
    /* Configure peripherals */
    prvSetupHardware();

    vTraceEnable(TRC_START);

    /* Create tasks */
    xTaskCreate( prvButtonTaskFunction,
                 "ButtonT",
                 configMINIMAL_STACK_SIZE,
                 NULL,
                 mainBUTTON_TASK_PRIO,
                 NULL
               );
    xTaskCreate( prvCountingTaskFunction,
                 "CountingT",
                 configMINIMAL_STACK_SIZE,
                 NULL,
                 mainCOUNTING_TASK_PRIO,
                 NULL
               );
```

Слика 4.- Позив функције vTraceEnable и почетак снимања после иницијализације хардвера

Како би се величина *Event* бафера поставила на што већу вредност (TRC_CFG_EVENT_BUFFER_SIZE) мора се сачувати што већи простор у RAM меморији. Више опција постоје како би се ослободила RAM меморија, једна од њих је смањење очекиваног броја за сваки тип објекта кернела (таскови, семафори, редови порука, тајмера итд. - TRC_CFG_NTASK, TRC_CFG_NQUEUE, TRC_CFG_NISR, TRC_CFG_NSEMAPHORE, TRC_CFG_NTIMER, TRC_CFG_NEVENTGROUP, TRC_CFG_NSTREAMBUFFER и TRC_CFG_NMESSAGEBUFFER) јер се резервише између осталог место у меморији за *Object Property Table* који се налази у оквиру *RecordDataPtr* који се чувају у RAM меморији циљног система. Такође, постављањем константе TRC_CFG_INCLUDE_OSTICK_EVENTS на вредност „0“ *OS tick events* се искључују из записа и тиме не заузимају додатно место у меморији. Ако се премаши количина доступне RAM меморије циљног система, линкер неће креирати извршни код (Слика 5), тако да прилагођавање вредности разних константи се може искористити цела меморија.

```
"../lnk_msp430f5529.cmd", line 140: error #10099-D: program will not fit into available memory, or the section contains a call site that requires a trampoline that
can't be generated for this section. run placement with alignment fails for section ".data" size 0x6f.  Available memory ranges:
   RAM          size: 0x2000       unused: 0x6c         max hole: 0x6c
"../lnk_msp430f5529.cmd", line 143: error #10099-D: program will not fit into available memory, or the section contains a call site that requires a trampoline that
can't be generated for this section. run placement with alignment fails for section ".stack" size 0xa0.  Available memory ranges:
   RAM          size: 0x2000       unused: 0x6c         max hole: 0x6c
error #10010: errors encountered during linking; "SRV_Template.out" not built

>> Compilation failure
makefile:167: recipe for target 'SRV_Template.out' failed
gmake[1]: *** [SRV_Template.out] Error 1
makefile:163: recipe for target 'all' failed
gmake: *** [all] Error 2

**** Build Finished ****
```

Слика 5.- Пример када у процесу компајлирања се јавља грешка да програм захтева већу меморију него што нуди циљни систем

Још једна хардверска зависност библиотеке за снимање јесте одабир порта за хардверски тајмер како би биле тачне временске ознаке снимљеним догађајима. За неке хардверске архитектуре постоји већ дефинисана имплементација у библиотеци а за неке портове имплементација је још у развоју. Ако порт није изабран онда се укључује хардверски независна резервна опција која обезбеђује *timestamping* (обично 1 ms). Развој и измена везано за хардверски порт је углавном везано за постављање *timestamping*. Састоји се од сета макроа (TRC_HWTC_TYPE, TRC_HWTC_COUNT, TRC_HWTC_FREQ_HZ итд.) која се дефинишу у складу са хардвером.

Могуће је конфигурисати да се дода још један таск који ће периодично да извештава о неискоришћеном простору стека за сваки таск у апликацији. Подешавањем TRC_CFG_ENABLE_STACK_MONITOR на вредност „1“ се омогућава рад таска TzCtrl (*Tracealyzer Control task*) који покреће надгледање стека. Поред TRC_CFG_ENABLE_STACK_MONITOR подешавају се и:

- TRC_CFG_STACK_MONITOR_MAX_TASKS - контролише колико таскова ће надгледати стек, ако је број мањи од броја таскова у апликацији неки таскови неће бити укључени у надгледање стека,

- TRC_CFG_STACK_MONITOR_MAX_REPORTS - дефинише за колико таскова ће бити предмет анализе коришћења стека за свако извршавање таска TzCtrl али не укључује на који таск ће се пратити већ колико ће се анализирати у комбинацији са TRC_CFG_CTRL_TASK_DELAY за подешавање учестаности,

- TRC_CFG_CTRL_TASK_PRIORITY - дефинише приоритет таска TzCtrl, који довољно висок како би се обезбедило поуздано периодично извршавање и пренос података а истовремено довољно низак како не би изазвало било какво ометање задатих временских осетљивих таскова,

- TRC_CFG_CTRL_TASK_DELAY - дефинише кашњење између петљи таска TzCtrl, што утиче на учестаност праћења стека, у режиму „*streaming*“ ово утиче на пренос података ако се користи *stream* порт који је интерни бафер - *TCP/IP* и може да повећа оптерећење *CPU* за краће одлагање, изражена је различитим јединицама (за *FreeRTOS* користи се тикови, а за *Zephyr* ms),

- TRC_CFG_CTRL_TASK_STACK_SIZE – дефинише величину стека TzCtrl таска.

Овај таск се увек креира у „*streaming*“ режиму, док у „*snapshot*“ режиму TzCtrl таск се само користи за надгледање таска и не креира се док није омогућен. Краткотрајни таскови могу бити пропуштени за анализу стека и не показује тачно кретање повећања/смањење стека одређеног таска већ само где је направљен „узорак“.

Подразумевано, сви догађаји се чувају у истом баферу али кориснички догађаји, које генерише апликација, могу се ускладиштити у засебном баферу корисничких догађаја (*User Event Buffer - UB*) одвојено од *RTOS* догађаја. Да би се користио *User Event* бафер потребно је поставити TRC_CFG_USE_SEPARATE_USER_EVENT_BUFFER на „1“ и могао би да садржи дужу историју догађаја коју памти јер не деле бафер са чешћим догађајима. Користи се типично у „*snapshot ring-buffer*“ режиму када се снимање таскова наставља. Након што се главни бафер попуни, тада нови догађаји „прегазе“ старе док се у *Tracealyzer* приказују само „*Unknown Actor*“ уместо таскова које контролише распоређивач (*Scheduler*) и додатни подаци са *User Event* бафера. У овом моду,

кориснички догађаји су структуирани као UB канали који се састоји из имена канала и подразумеваног формата стрингова. UB канали се региструју позивом функција *xTraceRegisterUBChannel*. Догађаји и подаци се у *User Event* бафер уписују позивом функција *vTraceUBEvent, vTraceUBData* и *xTracePrintF* (која може помоћу аргумената креирати нови канал са називом и форматом стрингова и запамтити податак истовремено). Поред TRC_CFG_USE_SEPARATE_USER_EVENT_BUFFER, константе које дефинишу коришћење *User Event* бафера су још и:

- TRC_CFG_SEPARATE_USER_EVENT_BUFFER_SIZE – дефинишу величину *User Event* бафера као број слотова,

- TRC_CFG_UB_CHANNELS – дефинишу број *User Event* бафер канала који се користе да структуирају догађаје.

5.2 Recorder API

Јавне *API* функције *RTOS*-а које су део библиотеке снимача су дефинисани у фајлу *trcRecorder.h* и деле се у две групе: оне које су заједничке за два режима снимања и проширени списак *API* функција дефинисана искључиво за „*snapshot*“ режим. Најбитније међу њима су:

- *void vTraceEnable(int startOption)* – омогућава иницијализацију и почетак снимања и постоје два начина позива: одмах након иницијализације хардвера и у *startup* коду пре било каквог *RTOS* позива (укључујући и креирање таскова) или из корисничког таска након што је стартован кернел. У сваком случају снимач мора да буде иницијализован пре него што *RTOS* буде стартован. Као аргумент *startOption* постоје 3 опције: TRC_INIT, TRC_START и TRC_START_AWAIT_HOST. Ако је потребно започети снимање таскова у неком моменту након стартовања *RTOS*-а, онда је у *startup* коду потребно иницијализовати снимач са *vTraceEnable(TRC_INIT)* и TRC_CFG_RECORDER_DATA_INIT мора бити постављен на „0“ (*trcConfig.h*), а почетак снимања је могуће из *Tracealyzer* апликације ("Start Recording" дугме - „*streaming*“ режим) или касније из неког корисничког таска/прекидне рутине позивом *vTraceEnable (TRC_START).* Позивањем *vTraceEnable(TRC_START_AWAIT_HOST)* почетак снимања је могућ само из *Tracealyzer* апликације и подржава само „*streaming*“ режим, док позив *vTraceEnable (TRC_START)* у *startup* коду одмах иницијализује и започиње снимање.

```
myBoardInit();
...
/* From startup */
vTraceEnable(TRC_START);
...
/* RTOS scheduler starts */
vTaskStartScheduler();
```

```
myBoardInit();
...
/* From startup - blocks until start from host */
vTraceEnable(TRC_START_AWAIT_HOST);
...
/* RTOS scheduler starts */
vTaskStartScheduler();
```

```
myBoardInit();
...
/* Init only, trace starts later...*/
vTraceEnable(TRC_INIT);
...
/* RTOS scheduler starts */
vTaskStartScheduler();
...
/* In a task or ISR */
vTraceEnable(TRC_START);
```

```
myBoardInit();
...
/* Init only, trace starts later...*/
vTraceEnable(TRC_INIT);
...
/* RTOS scheduler starts */
vTaskStartScheduler();
...
/* "Trace Start" command from Tracealyzer
is received by TzCtrl task. */
```

Слика 6.- Примери позива функције vTraceEnable

- *void vTraceStop(void)* – зауставља снимање таскова, поставља *RecorderDataPrt→recordActive* на „0“,

- *void vTracePrintF(traceString chn, const char* fmt, ...)* – генерише корисничке догађаје које ће се приказати у *Tracealyzer* у виду форматираног текста и испису података које се преносе као аргументи функције, сви типови података су подржани и могу се нагласити 8-битни и 16-битни аргументи да би смањили употребу RAM-а, у режиму „*snapshot*“ број аргумената је ограничен на 15 који не смеју да пређу преко 32 бајта (у „*streaming*“ режиму 52 бајта),

- *void vTraceVPrintF(traceString chn, const char* fmt, va_list vl)*

- *void vTracePrint(traceString chn, const char* str)* - верзија *vTraceVPrintF* за брже евидентирање стрингова без аргумената података,

- *traceString xTraceRegisterString(const char* name) void* – креира *handle* која је потребна функцијама које складиште стрингове, омогућава памћење прилагођеног формата догађаја са подацима помоћу функција *vTracePrintF*, *vTraceVPrintF* и *vTracePrint*,

- *vTraceSet...Name(void* object, const char* name)* – придружује назив објекту кернела која је указана прослеђеним показивачем *object* на дати објекат (редови порука, семафори, mutex семафори, групе догађаја, стрим бафер и бафери порука),

- *traceHandle xTraceSetISRProperties(const char* name, uint8_t priority)* – чува име и приоритет за дату прекидну рутину,

- *void vTraceStoreISRBegin(traceHandle handle)* – памти почетак прекидне рутине користећи *traceHandle* добијен из функције *xTraceSetISRProperties* или *prvTraceGetObjectHandle*,

- *void vTraceStoreISREnd(int isTaskSwitchRequired)* - памти крај прекидне рутине, аргумент *isTaskSwitchRequired* показује да ли је прекид изазвао *task-switch* (1), на пример, сигнализацијом семафора или се прекид враћа у претходни контекст (0),

- *void vTraceInstanceFinishedNow(void)* – креира догађај који наглашава крај тренутне инстанце таска у том моменту тако што започиње нову инстанцу актера чак иако није дошло до промене контекста таскова,

- *void vTraceInstanceFinishedNext(void)* – креира догађај који наглашава крај инстанце таска на следећем позиву кернела,

- *char* xTraceGetLastError(void)* – враћа (ако постоји) прву регистровану грешку као стринг који се у чува у структури података *RecordData* под *systemInfo,*

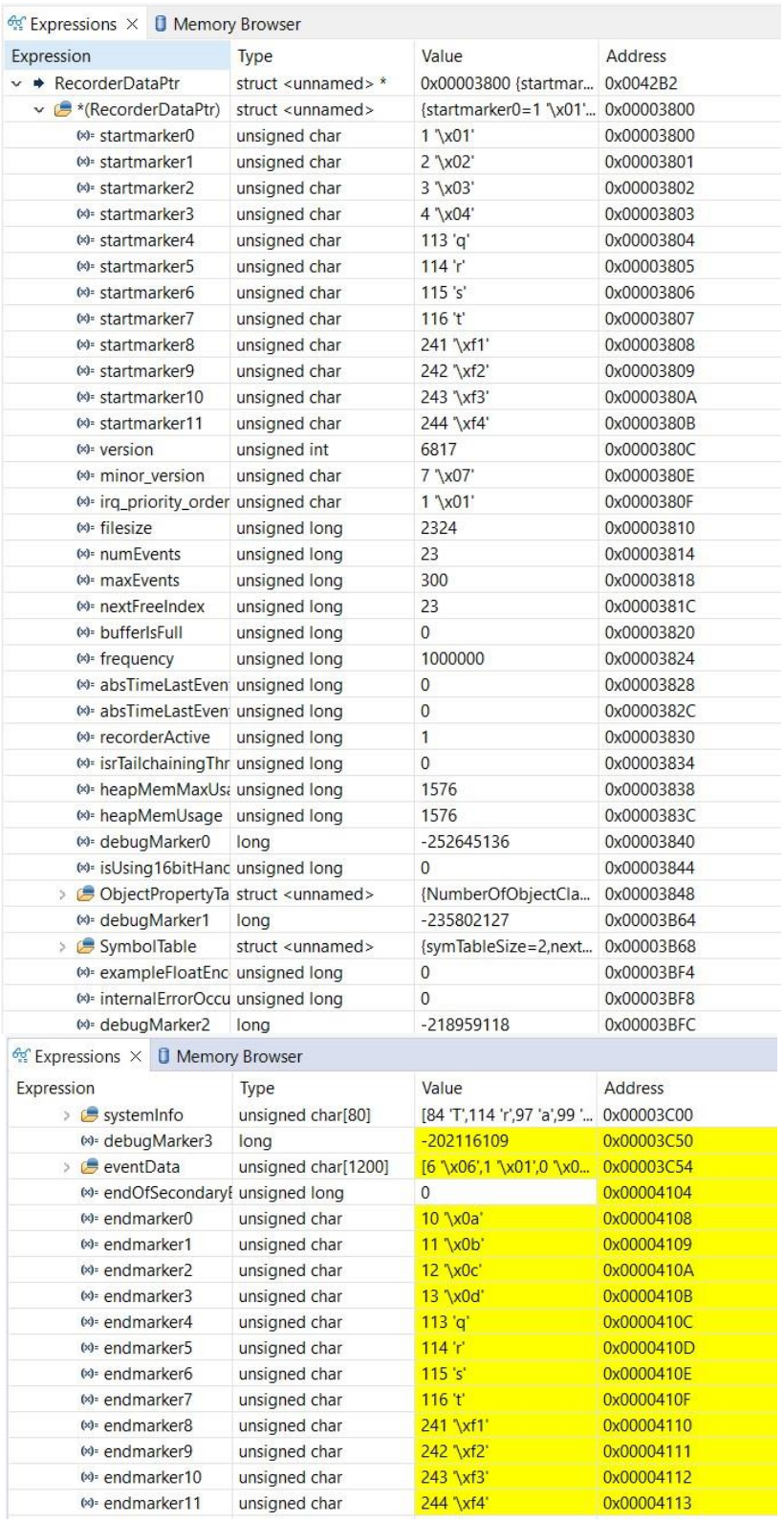
- *void vTraceSetFrequency(uint32_t frequency)* – поставља време одабира извора такта и мења подразумевану вредност означеном константом TRC_HWTC_FREQ_HZ и која се дели са вредношћу константе TRC_HWTC_DIVISOR у „*snapshot*“ режиму.

- *int xTraceIsRecordingEnabled(void*) – враћа вредност „1“ или „0“ у зависности од тога да ли је снимач омогућен или онемогућен.

- *uint32_t uiTraceGetTraceBufferSize(void)* - враћа величину бафера праћења снимака у бајтовима.

5.3 Структура података снимача (RecordData)

Сви подаци везани за снимање *RTOS* апликације се прикупљају у RAM меморији у оквиру структуре *RecordData* на коју показује показивач *RecordDataPtr* и то ће представљати главну структуру података коју *Tracealyzer* ће користити за визуелну дијагностику таскова (Слика 7). У оквиру *RecordData* налазе се *Object property table*, *Symbol Table*, *Event Data* бафер, низ у којој се памте поруке о грешкама (*systemInfo*), променљиве које означавају разна стања снимача и процес снимања (minor_version, irq_priority_order, filesize, numEvents, maxEvents, nextFreeIndex, bufferIsFull, frequency, absTimeLastEvent, absTimeLastEventSecond, recorderActive, isrTailchainingThreshold, heapMemMaxUsage, heapMemUsage, debugMarker0, isUsing16bitHandles, exampleFloatEncoding, internalErrorOccured, endOfSecondaryBlocks), startmarker-a и endmarker-a.



Expressions | Memory Browser

Expression	Type	Value	Address
RecorderDataPtr	struct <unnamed> *	0x00003800 {startmar...	0x0042B2
*(RecorderDataPtr)	struct <unnamed>	{startmarker0=1 '\x01'...	0x00003800
startmarker0	unsigned char	1 '\x01'	0x00003800
startmarker1	unsigned char	2 '\x02'	0x00003801
startmarker2	unsigned char	3 '\x03'	0x00003802
startmarker3	unsigned char	4 '\x04'	0x00003803
startmarker4	unsigned char	113 'q'	0x00003804
startmarker5	unsigned char	114 'r'	0x00003805
startmarker6	unsigned char	115 's'	0x00003806
startmarker7	unsigned char	116 't'	0x00003807
startmarker8	unsigned char	241 '\xf1'	0x00003808
startmarker9	unsigned char	242 '\xf2'	0x00003809
startmarker10	unsigned char	243 '\xf3'	0x0000380A
startmarker11	unsigned char	244 '\xf4'	0x0000380B
version	unsigned int	6817	0x0000380C
minor_version	unsigned char	7 '\x07'	0x0000380E
irq_priority_order	unsigned char	1 '\x01'	0x0000380F
filesize	unsigned long	2324	0x00003810
numEvents	unsigned long	23	0x00003814
maxEvents	unsigned long	300	0x00003818
nextFreeIndex	unsigned long	23	0x0000381C
bufferIsFull	unsigned long	0	0x00003820
frequency	unsigned long	1000000	0x00003824
absTimeLastEven	unsigned long	0	0x00003828
absTimeLastEven	unsigned long	0	0x0000382C
recorderActive	unsigned long	1	0x00003830
isrTailchainingThr	unsigned long	0	0x00003834
heapMemMaxUsa	unsigned long	1576	0x00003838
heapMemUsage	unsigned long	1576	0x0000383C
debugMarker0	long	-252645136	0x00003840
isUsing16bitHanc	unsigned long	0	0x00003844
ObjectPropertyTa	struct <unnamed>	{NumberOfObjectCla...	0x00003848
debugMarker1	long	-235802127	0x00003B64
SymbolTable	struct <unnamed>	{symTableSize=2,next...	0x00003B68
exampleFloatEnc	unsigned long	0	0x00003BF4
internalErrorOccu	unsigned long	0	0x00003BF8
debugMarker2	long	-218959118	0x00003BFC

Expressions | Memory Browser

Expression	Type	Value	Address
systemInfo	unsigned char[80]	[84 'T',114 'r',97 'a',99 '...	0x00003C00
debugMarker3	long	-202116109	0x00003C50
eventData	unsigned char[1200]	[6 '\x06',1 '\x01',0 '\x0...	0x00003C54
endOfSecondaryE	unsigned long	0	0x00004104
endmarker0	unsigned char	10 '\x0a'	0x00004108
endmarker1	unsigned char	11 '\x0b'	0x00004109
endmarker2	unsigned char	12 '\x0c'	0x0000410A
endmarker3	unsigned char	13 '\x0d'	0x0000410B
endmarker4	unsigned char	113 'q'	0x0000410C
endmarker5	unsigned char	114 'r'	0x0000410D
endmarker6	unsigned char	115 's'	0x0000410E
endmarker7	unsigned char	116 't'	0x0000410F
endmarker8	unsigned char	241 '\xf1'	0x00004110
endmarker9	unsigned char	242 '\xf2'	0x00004111
endmarker10	unsigned char	243 '\xf3'	0x00004112
endmarker11	unsigned char	244 '\xf4'	0x00004113

Слика 7.- Приказ израза и променљивих са вредностима уоквиру структуре података RecordDataPtr током debug сесије

У RAM меморији циљног система *Object handle stack* структура није део структуре *RecordData* и представља структуру података која обезбеђује механизам за једнобајтни запис који представља *handle* за одређени објекат кернела. Запис од једног бајта је довољан уместо записа од 4 бајта (показивач) када се памти референца на објекат и

омогућава до 255 објеката сваке класе да буде активна у било ком тренутку. Ова структура прати слободне *handle* које се не користе и може бити додељена првом следећем објекту одређене класе која се креира у програму и памти се у меморији, тј. у *Object property table*. Приликом брисања датог објекта, враћа се *handle* назад на стек (ова структура података садржи један стек по класи објеката). *Object handle stack* чине:

- indexOfNextAvailableHandle (Слика 8) – за сваку класу објеката чувају се индекси следећих *handle* за алокацију, број класа објеката је дефинисана константом TRACE_NCLASSES (*trcKernelPort.h*) и свака класа има свој јединствени индекс у низу indexOfNextAvailableHandle редом редови порука [0], семафори [1], mutex семафори [2], таскови [3], прекидне рутине [4], тајмери [5], групе догађаја [6], стрим бафери [7] и бафери са порукама [8],

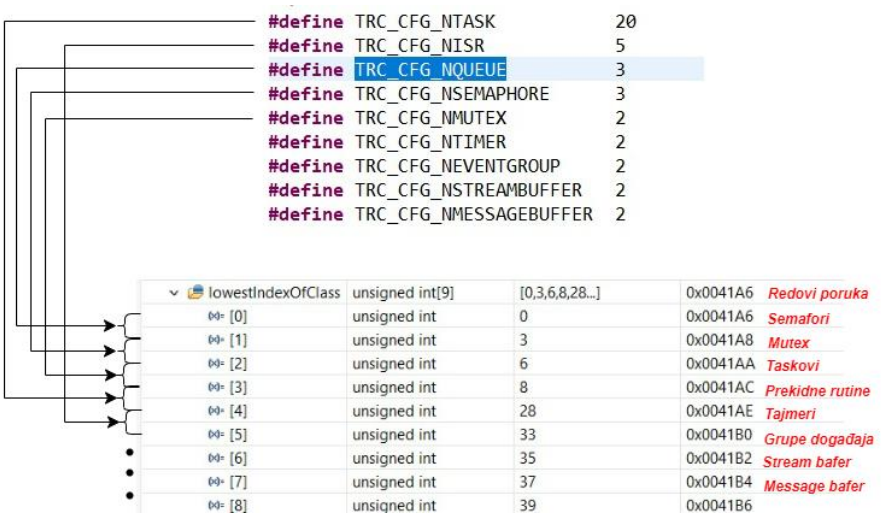
objectHandleStacks	struct <unnamed>	{indexOfNextAvailabl...	0x004194
indexOfNextAvailab	unsigned int[9]	[0,5,6,12,28...]	0x004194
[0]	unsigned int	0	0x004194
[1]	unsigned int	5	0x004196
[2]	unsigned int	6	0x004198
[3]	unsigned int	12	0x00419A
[4]	unsigned int	28	0x00419C
[5]	unsigned int	33	0x00419E
[6]	unsigned int	35	0x0041A0
[7]	unsigned int	37	0x0041A2
[8]	unsigned int	39	0x0041A4

Слика 8.- Приказ ObjectHandleStacks→ indexOfNextAvailableHandle

```
/* The object classes */
#define TRACE_NCLASSES 9
#define TRACE_CLASS_QUEUE ((traceObjectClass)0)
#define TRACE_CLASS_SEMAPHORE ((traceObjectClass)1)
#define TRACE_CLASS_MUTEX ((traceObjectClass)2)
#define TRACE_CLASS_TASK ((traceObjectClass)3)
#define TRACE_CLASS_ISR ((traceObjectClass)4)
#define TRACE_CLASS_TIMER ((traceObjectClass)5)
#define TRACE_CLASS_EVENTGROUP ((traceObjectClass)6)
#define TRACE_CLASS_STREAMBUFFER ((traceObjectClass)7)
#define TRACE_CLASS_MESSAGEBUFFER ((traceObjectClass)8)
```

Слика 9.- /trcKernelPort.h

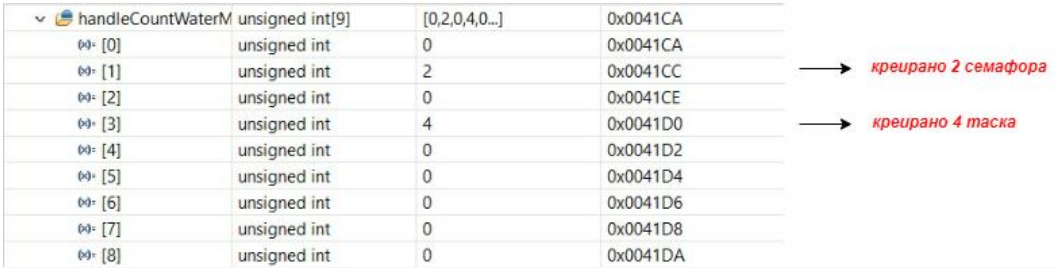
- lowestIndexOfClass (Слика 10) – дефинише најмањи индекс за дату класу објекта одакле се памте *handle* у *objectHandles* и зависе од дефинисаног укупног броја активних објеката за сваку класу TRC_CFG_NTASK, TRC_CFG_NISR, TRC_CFG_NQUEUE, TRC_CFG_NSEMAPHORE, TRC_CFG_NMUTEX, TRC_CFG_NTIMER, TRC_CFG_NEVENTGROUP , TRC_CFG_NSTREAMBUFFER, TRC_CFG_NMESSAGEBUFFER (*trcKernelPortSnapshotConfig.h*),



Слика 10.- Приказ ObjectHandleStacks→lowestIndexOfClass заједно са вредностима које потичу од дефинисаних константи у trcKernelPortSnapshotConfig.h

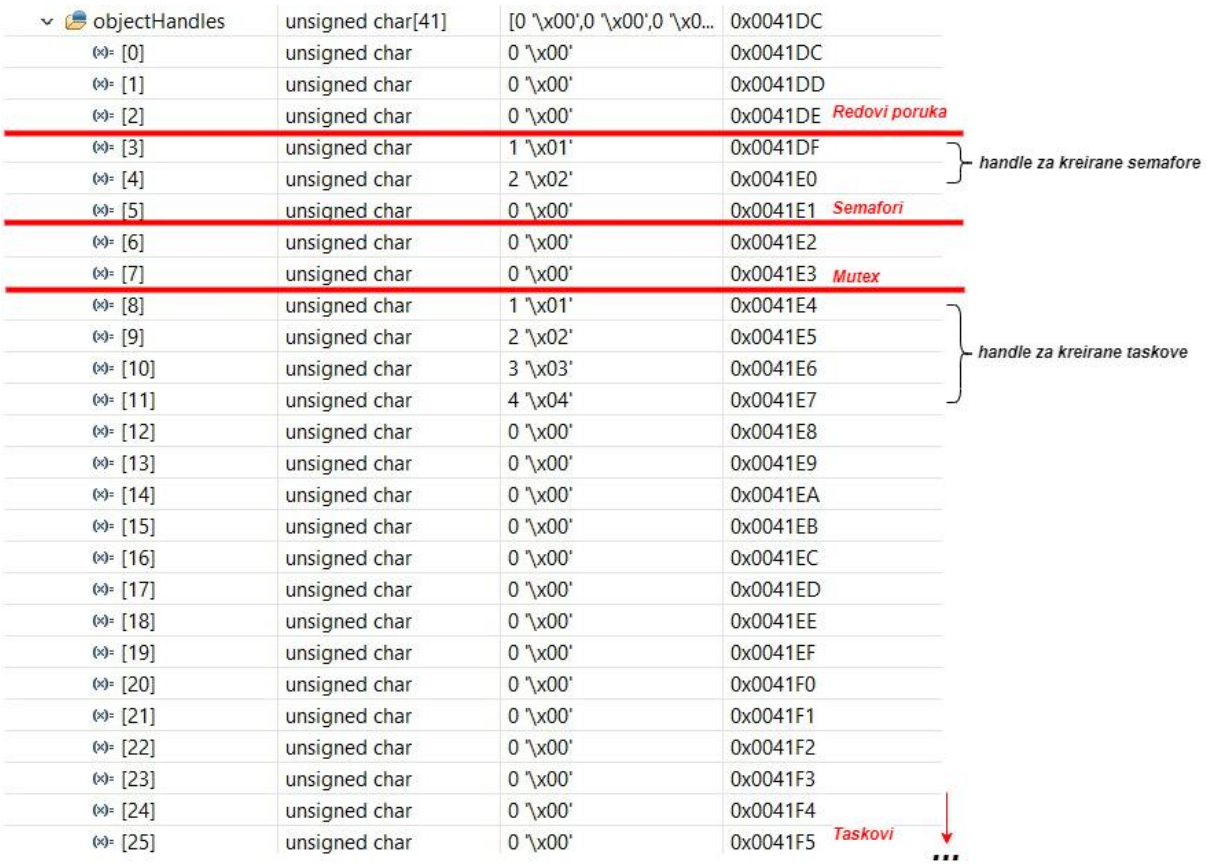
- highestIndexOfClass - Дефинише највећи индекс за дату класу објекта одакле се памте *handle* у *objectHandles*,

- handleCountWaterMarksOfClass (Слика 11) – Број креираних објеката сваке класе у апликацији у датом тренутку,



Слика 11.- Приказ ObjectHandleStacks→handleCountWaterMarksOfClass

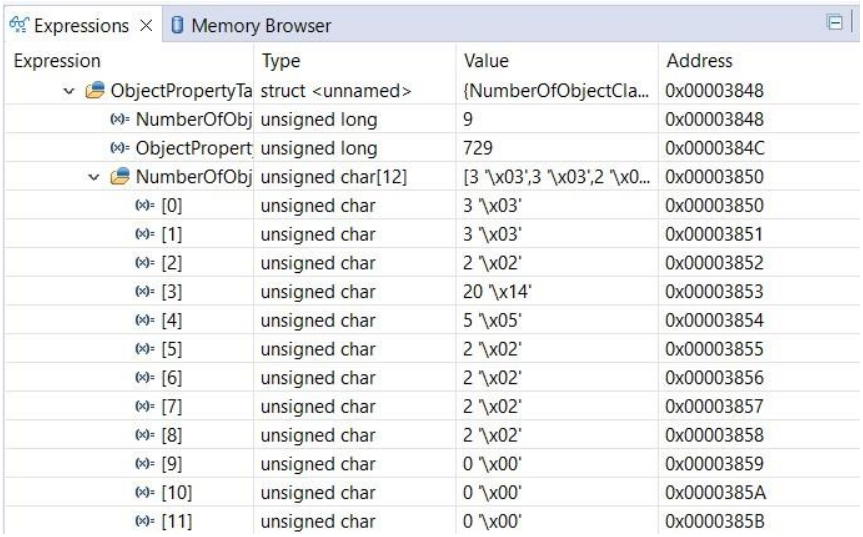
- objectHandles (Слика 12) – садржи слободне *handles*, тј. сет стекова уоквиру низа. Сваки стек креће од позиције која је дата у lowestIndexOfClass.



Слика 12.- Приказ ObjectHandleStacks→objectHandles

Object property table или табела својства објеката садрже имена и друге параметре објеката кернела. Вредности у табели се непрекидно преписују и увек представљају тренутно стање. За сваки креирани објекат неке класе у програму, објекат има дефинисан скуп својстава која конфигуришу понашање тог објекта. *Object property table* чине:

- NumberOfObjectsPerClass (Слика 13) – садржи вредности очекиваног укупног броја објеката кернела за сваку класу истим редоследом класа као што је био случај у *Object handle stack* структури (TRC_CFG_NTASK, TRC_CFG_NISR, TRC_CFG_NQUEUE, TRC_CFG_NSEMAPHORE, TRC_CFG_NMUTEX, TRC_CFG_NTIMER, TRC_CFG_NEVENTGROUP , TRC_CFG_NSTREAMBUFFER, TRC_CFG_NMESSAGEBUFFE). Дужина низа се рачуна по формули 4*((TRACE_NCLASSES+3)/4),



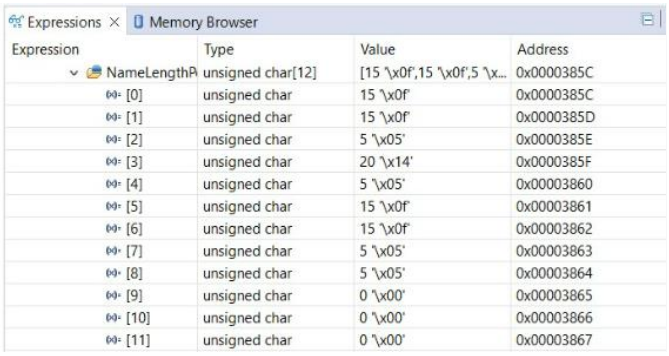
Expressions | Memory Browser

Expression	Type	Value	Address
ObjectPropertyTa	struct <unnamed>	{NumberOfObjectCla...	0x00003848
NumberOfObj	unsigned long	9	0x00003848
ObjectPropert	unsigned long	729	0x0000384C
NumberOfObj	unsigned char[12]	[3 '\x03',3 '\x03',2 '\x0...	0x00003850
[0]	unsigned char	3 '\x03'	0x00003850
[1]	unsigned char	3 '\x03'	0x00003851
[2]	unsigned char	2 '\x02'	0x00003852
[3]	unsigned char	20 '\x14'	0x00003853
[4]	unsigned char	5 '\x05'	0x00003854
[5]	unsigned char	2 '\x02'	0x00003855
[6]	unsigned char	2 '\x02'	0x00003856
[7]	unsigned char	2 '\x02'	0x00003857
[8]	unsigned char	2 '\x02'	0x00003858
[9]	unsigned char	0 '\x00'	0x00003859
[10]	unsigned char	0 '\x00'	0x0000385A
[11]	unsigned char	0 '\x00'	0x0000385B

Слика 13.- Приказ ObjectPropertyTable→ NumberOfObjectsPerClass

- NameLengthPerClass (Слика 14) – садржи вредности максималне дужине стрингова резервисаних за имена сваког објеката било које класе и дефинисана су константама TRC_CFG_NAME_LEN_TASK, TRC_CFG_NAME_LEN_ISR, TRC_CFG_NAME_LEN_QUEUE, TRC_CFG_NAME_LEN_SEMAPHORE, TRC_CFG_NAME_LEN_MUTEX, TRC_CFG_NAME_LEN_TIMER, TRC_CFG_NAME_LEN_EVENTGROUP, TRC_CFG_NAME_LEN_STREAMBUFFER и TRC_CFG_NAME_LEN_MESSAGEBUFFER (*trcKernelPortSnapshotConfig.h*),

```
#define TRC_CFG_NAME_LEN_TASK           20
#define TRC_CFG_NAME_LEN_ISR            5
#define TRC_CFG_NAME_LEN_QUEUE          15
#define TRC_CFG_NAME_LEN_SEMAPHORE      15
#define TRC_CFG_NAME_LEN_MUTEX          5
#define TRC_CFG_NAME_LEN_TIMER          15
#define TRC_CFG_NAME_LEN_EVENTGROUP     15
#define TRC_CFG_NAME_LEN_STREAMBUFFER   5
#define TRC_CFG_NAME_LEN_MESSAGEBUFFER  5
```



Expressions | Memory Browser

Expression	Type	Value	Address
NameLengthPe	unsigned char[12]	[15 '\x0f',15 '\x0f',5 '\x...	0x0000385C
[0]	unsigned char	15 '\x0f'	0x0000385C
[1]	unsigned char	15 '\x0f'	0x0000385D
[2]	unsigned char	5 '\x05'	0x0000385E
[3]	unsigned char	20 '\x14'	0x0000385F
[4]	unsigned char	5 '\x05'	0x00003860
[5]	unsigned char	15 '\x0f'	0x00003861
[6]	unsigned char	15 '\x0f'	0x00003862
[7]	unsigned char	5 '\x05'	0x00003863
[8]	unsigned char	5 '\x05'	0x00003864
[9]	unsigned char	0 '\x00'	0x00003865
[10]	unsigned char	0 '\x00'	0x00003866
[11]	unsigned char	0 '\x00'	0x00003867

Слика 14.-Приказ ObjectPropertyTable→ NameLengthPerClass са вредностима одређене константама из trcKernelPortSnapshotConfig.h

- TotalPropertyBytesPerClass (Слика 15) – дефинише број бајтова у низу *objbytes* резервисан за објекат класе, за сваку класу се резервише различита величина у меморији. На пример, за таскове величина за памћење једне инстанце у *objbytes* се рачуна као TRC_CFG_NAME_LEN_TASK + 4 јер од додатних 4 бајта бајт 0 и бајт 1 које се додају носиће информацију о тренутном приоритету и стање таска респективно, а за семафоре се рачуна као TRC_CFG_NAME_LEN_SEMAPHORE + 1 (додатни бајт за стање – доступан или недоступан) итд.

Expressions × Memory Browser

Expression	Type	Value	Address
TotalPropertyE	unsigned char[12]	[16 '\x10',16 '\x10',6 '\...	0x00003868
[0]	unsigned char	16 '\x10'	0x00003868
[1]	unsigned char	16 '\x10'	0x00003869
[2]	unsigned char	6 '\x06'	0x0000386A
[3]	unsigned char	24 '\x18'	0x0000386B
[4]	unsigned char	7 '\x07'	0x0000386C
[5]	unsigned char	16 '\x10'	0x0000386D
[6]	unsigned char	19 '\x13'	0x0000386E
[7]	unsigned char	9 '\x09'	0x0000386F
[8]	unsigned char	9 '\x09'	0x00003870
[9]	unsigned char	0 '\x00'	0x00003871
[10]	unsigned char	0 '\x00'	0x00003872
[11]	unsigned char	0 '\x00'	0x00003873

Слика 15.- Приказ ObjectPropertyTable→ TotalPropertyBytesPerClass

- StartIndexOfClass (Слика 16) – за сваку класу садржи вредности индекса у низу одакле се памте информације за сваку инстанцу објекта одређене класе. Примери за семафоре и таскове како се рачунају:

$$StartIndexSemaphore = StartIndexQueue + TRC\_CFG\_NQUEUE * PropertyTableSizeQueue,$$
$$StartIndexTask = StartIndexMutex + TRC\_CFG\_NMUTEX * PropertyTableSizeMutex.$$

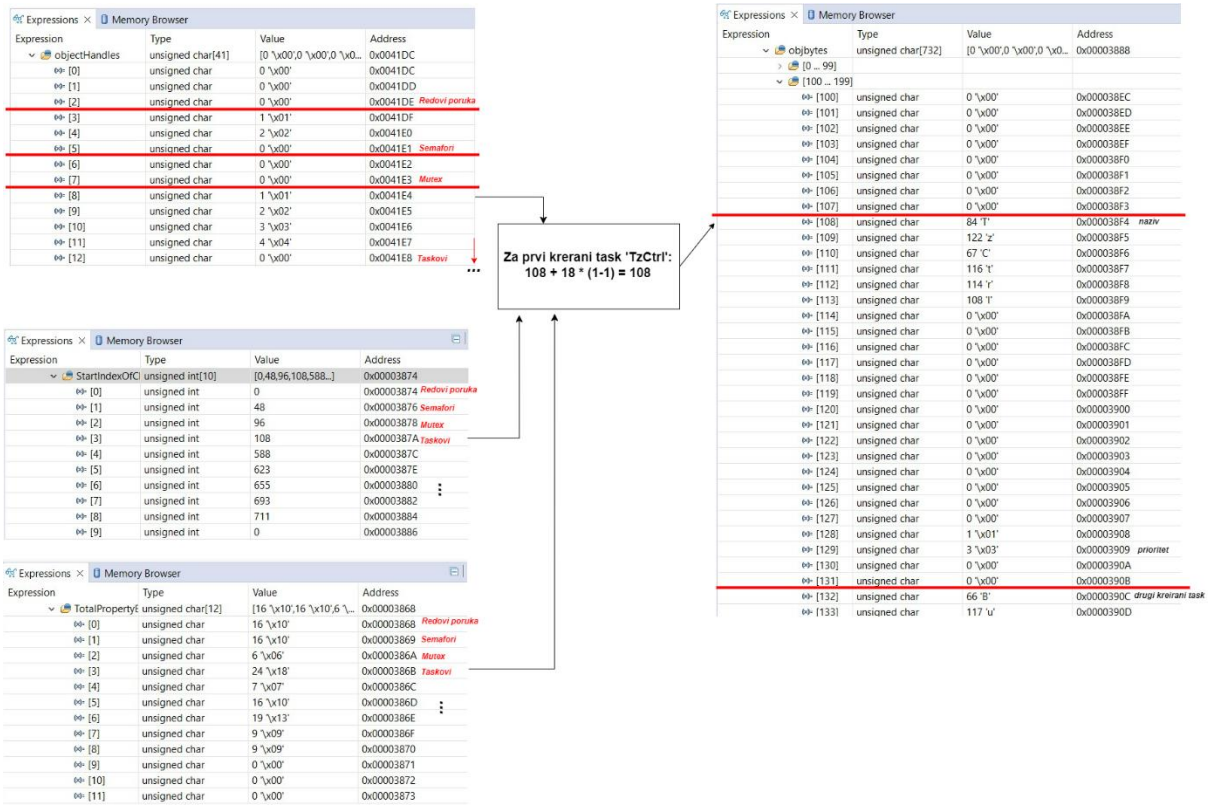
Expressions × Memory Browser

Expression	Type	Value	Address
StartIndexOfCl	unsigned int[10]	[0,48,96,108,588...]	0x00003874
[0]	unsigned int	0	0x00003874
[1]	unsigned int	48	0x00003876
[2]	unsigned int	96	0x00003878
[3]	unsigned int	108	0x0000387A
[4]	unsigned int	588	0x0000387C
[5]	unsigned int	623	0x0000387E
[6]	unsigned int	655	0x00003880
[7]	unsigned int	693	0x00003882
[8]	unsigned int	711	0x00003884
[9]	unsigned int	0	0x00003886

Слика 16.- Приказ ObjectPropertyTable→ StartIndexOfClass

- objbytes – низ у којем се памте све потребне информације о објектима кернела а информације се уносе приликом креирања инстанце и позивом функција *prvTraceSetObjectName* и *prvTraceSetPriorityProperty*. На слици 17 је показано како индекс одакле почиње запис о инстанци објекта одређене класе се одређује помоћу *handle* дате инстанце која се узима из *ObjectHandleStacks→objectHandles*, вредности индекса из *RecordDataPtr→ObjectPropertyTable→ StartIndexOfClass* за дату класу објекта и вредности величине простора у меморији за запис инстанце из *RecordDataPtr→ObjectPropertyTable→ TotalPropertyBytesPerClass*:

index = RecorderDataPtr→ObjectPropertyTable.StartIndexOfClass[objectclass] + (RecorderDataPtr→ObjectPropertyTable.TotalPropertyBytesPerClass[objectclass] * (objecthandle - 1)),

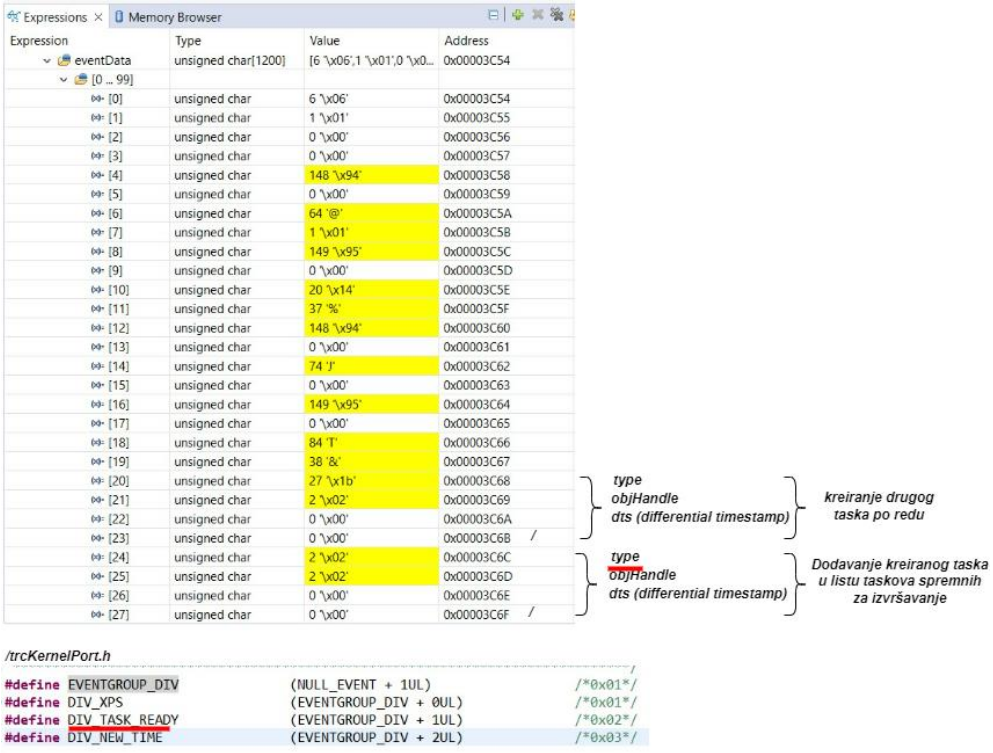


Слика 17.- Приказ ObjectPropertyTable→ objbytes и приказ одређивања индекса за инстанцу објекта неке класе (у овом случају таска)

- ObjectPropertyTableSizeInBytes – укупна величина у бајтовима низа *objbytes*.

Величина *Event Data* бафер је дата са TRC_CFG_EVENT_BUFFER_SIZE * 4 јер сваки догађај се записује у четворобајтном запису. Да би се омогућило снимање што већег броја података потребан је што већи бафер који може да стане у RAM меморију циљног система. У фајлу *trcRecord.h* дефинисане су структуре података који се прослеђују у *Event Data* бафер и који означавају догађаје (TSEvent, TREvent, LPEvent, KernelCall, KernelCallWithParamAndHandle, KernelCallWithParam16, ObjCloseNameEvent, ObjClosePropEvent, TaskInstanceStatusEvent, UserEvent, XTSEvent, XPSEvent, MemEventSize, MemEventAddr), а неки од њих су величине од 3 бајта чиме последњи бајт остаје неискоришћен. Ако променљива *dts* (временска разлика где се добија време догађаја) уоквиру претходно наведених структура података не може да стане у бафер онда се узима следећа четворобајтна реч за запис пре свега ове променљиве (од 32 бита искористи се 24 - XPSEvent). Функција *prvTraceNextFreeEventBufferSlot* враћа показивач на локацију која је прва слободна за чување новог догађаја у баферу. Структура података која се памти у баферу у случају да се памти у меморији у једном запису четворобајтне речи чине тип, временска ознака и *handle* за дату инстанцу објекта као на слици 18.

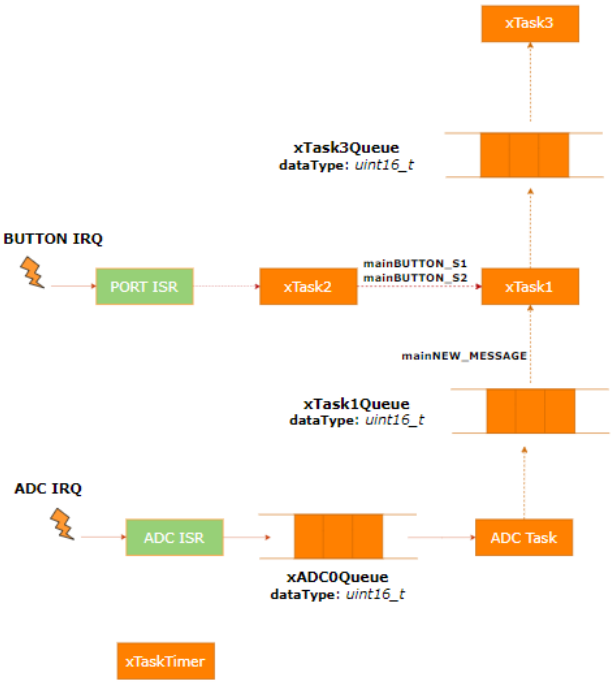
Handle-ови се дохватају из *Object handle* стека, док су типови догађаја дефинисани константама у фајлу *trcKernelPort.h*.



Слика 18.- Приказ Event Data бафера

5.4 Визуализација FreeRTOS пројекта у апликацији Tracealyzer

Примена *TraceRecorder* библиотеке ће бити демонстрирана на апликацији чији блок дијаграм је приказана на слици 19. Задатак апликације је да стартује аквизицију са канала A0, A1 на сваких 1000 ms помоћу таска xTaskTimer који користи *vTaskDelayUntil*. Имплементирана је одложена обрада прекида (*deffered interrupt processing*) АД конвертора, тако што се резултат конверзије у прекидној рутини уписује у ред са порукама (*xADC0Mailbox*) и обавештава се таск xTask1 о приспећу нове поруке путем директне нотификације таскова (*Direct-to-task notification*) у виду групе догађаја. Порука треба да садржи информацију о каналу који је очитан горњих 9 бита резултата АД конверзије. Таск xTask1 чува последњу очитану вредност за сваки канал. Таск xTask2 испитује тастере S1-S4 и на притисак одговарајућег тастера обавештава таск xTask1 путем директне нотификације таскова (*Direct-to-task notification*) у виду групе догађаја о каналу чије очитане вредности резултата конверзије треба да шаље таску xTask3. Сваки пут када стигне нова вредност са АД конвертора таск xTask1 смешта одговарајући податак у ред са порукама (*xTask3Mailbox*) на којем чека таск xTask3. Таск xTask3 рачуна разлику између узастопних вредности очитаног канала и приказује на *UART*-у.



Слика 19.- Блок дијаграм FreeRTOS пројекта у који се интегрише библиотека снимача

Пројекат од објеката кернела садржи осам таскова (укључујући још додатна три таска која ће се аутоматски генерисати при покретању *RTOS*-а), две прекидне рутине, три редова порука и то очекујемо да буде приказано на апликацији *Tracealyzer*. Хардвер који ће се користити је *Texas Instruments MSP430F5529LP* чија је величина RAM меморије 8 kB што је потребно имати на уму приликом дефиницања осталих константи у фајловима *FreeRTOSConfig.h*, *trcConfig.h*, *trcHardwarePort.h*, *trcKernelPortConfig.h*, *trcSnapshotConfig.h* и *trcKernelPortSnapshotConfig.h*.

Након интеграције библиотеке *TraceRecorder* у пројекат, потребно је покренути циљни систем у *debug* сесију и зауставити систем насумично или на одређеном месту у коду програма постављањем *breakpoint*-а. Снимање програма је трајало у зависности од конфигурације како је постављено, за „*snapshot*“ режим у моду TRC_SNAPSHOT_MODE_RING_BUFFER снимање трајало све време док је систем био покренут и трајаће и након паузирања процеса и наставка али ће се памтити последњи догађаји који су остали у баферу. У режиму TRC_SNAPSHOT_MODE_STOP_WHEN_FULL снимање се зауставља када се попуни бафер. Након снимања, могуће је учитати у *Tracealyzer* помоћу додатка *Percepio* које нуде интегрисано развојно окружења (*IDE*) базирана на *Eclipse*. Додатак *Percepio* се не подразумева и могуће је инсталирати у интегрисаном развојном окружењу у опцији:

Help → Install New Software,

унети URL „https://percepio.com/exporter“. Након инсталације уоквиру:

Percepio→Save Snapshot Trace,

отвара се *Tracealyzer* и учитава датотеке исписа RAM меморије креиране на било који начин. Испис може да садржи и друге податке пре или после бафера за праћење али *Tracealyzer* аутоматски проналази корисне податке.

У фајлу *FreeRTOSConfig.h* промењене су, између осталог што је наглашено у одељку *4.1 TraceRecorder библиотека*, константе configMAX_TASK_NAME_LEN на „10“ како би се назив таска xTaskTimer приказао у целости.

У фајлу *trcConfig.h* предефинисане константе су TRC_CFG_HARDWARE_PORT на TRC_HARDWARE_PORT_TEXAS_INSTRUMENTS_MSP430 с обзиром на избор микроконотролера, TRC_CFG_INCLUDE_OSTICK_EVENTS на „0“ , TRC_CFG_ENABLE_STACK_MONITOR на „1“ како се омогућило надгледање неискоришћеног простора стека, TRC_CFG_STACK_MONITOR_MAX_TASKS на „7“ како би се обухватили сви таскови за надгледање, TRC_CFG_CTRL_TASK_PRIORITY на „2“ како би се периодично извршавао без ометања извршавање осталих корисничких таскова.

У фајлу *trcKernelPortConfig.h* предефинисане константе су TRC_CFG_RECORDER_MODE на TRC_RECORDER_MODE_SNAPSHOT за одабир режим снимања, TRC_CFG_FREERTOS_VERSION на TRC_FREERTOS_VERSION_10_2_0 као верзија FreeRTOS-а који се користи, TRC_CFG_INCLUDE_EVENT_GROUP_EVENTS на „1“ за снимање "event group" догађаја креираних коришћењем групе догађаја, TRC_CFG_INCLUDE_TIMER_EVENTS на „1“ због позива функција коју користи таск xTaskTimer.

У фајлу *trcSnapshotConfig.h* предефинисане константе су TRC_CFG_SNAPSHOT_MODE на TRC_SNAPSHOT_MODE_RING_BUFFER или TRC_SNAPSHOT_MODE_STOP_WHEN_FULL (користиће се оба мода како бисмо приказали на разне аспекте визуализације), TRC_CFG_EVENT_BUFFER_SIZE на „300“ јер толико је преостало простора у меморији за чување догађаја (300 * 4 = 1200 B) а остатак меморије је резервисан за хип(*heap* – око 3400 бајтова) и за снимач (*recorder* – нешто мање од 3500 бајтова) и TRC_CFG_SYMBOL_TABLE_SIZE на „70“ јер је довољан како би се памтиле/исписале поруке креиране од стране корисника.

У фајлу *trcKernelPortSnapshotConfig.h* предефинисане константе су TRC_CFG_NTASK (15), TRC_CFG_NISR (4), TRC_CFG_NQUEUE (4) , TRC_CFG_NSEMAPHORE (2), TRC_CFG_NMUTEX (1), TRC_CFG_NTIMER (2), TRC_CFG_NEVENTGROUP (2), TRC_CFG_NSTREAMBUFFER (1), TRC_CFG_NMESSAGEBUFFER (1), TRC_CFG_NAME_LEN_TASK (15), TRC_CFG_NAME_LEN_ISR (8), TRC_CFG_NAME_LEN_QUEUE (15), TRC_CFG_NAME_LEN_SEMAPHORE (5), TRC_CFG_NAME_LEN_MUTEX (5), TRC_CFG_NAME_LEN_TIMER (15), TRC_CFG_NAME_LEN_EVENTGROUP (15), TRC_CFG_NAME_LEN_STREAMBUFFER (5) и TRC_CFG_NAME_LEN_MESSAGEBUFFER (5).

Такође и у коду програма су додате позиви одређених функција ради бољег прегледа догађаја у *Tracealyzer*, али уведене су и измене у библиотеци снимача (Слика 20 и 21).

/trcRecorder.h

```
/* This macro will create a task in the object table */
#undef trcKERNEL_HOOKS_TASK_CREATE
#define trcKERNEL_HOOKS_TASK_CREATE(SERVICE, CLASS, pxTCB) \
    if ((pxTCB) != 0) \
    { \
        TRACE_SET_OBJECT_NUMBER(TASK, pxTCB); \
        TRACE_SET_OBJECT_FILTER(TASK, pxTCB, CurrentFilterGroup); \ izbrisani red koda
        prvTraceSetObjectName(TRACE_CLASS_TASK, TRACE_GET_TASK_NUMBER(pxTCB), TRACE_GET_TASK_NAME(pxTCB)); \
        prvTraceSetPriorityProperty(TRACE_CLASS_TASK, TRACE_GET_TASK_NUMBER(pxTCB), TRACE_GET_TASK_PRIORITY(pxTCB)); \
        if (TRACE_GET_OBJECT_FILTER(TASK, TRACE_GET_CURRENT_TASK()) & CurrentFilterMask) \
            if (TRACE_GET_OBJECT_FILTER(TASK, pxTCB) & CurrentFilterMask) \
                prvTraceStoreKernelCall(SERVICE, TRACE_CLASS_TASK, TRACE_GET_TASK_NUMBER(pxTCB)); \
    } \
    else \
    { \
        /* pxTCB is null */ \
        if (TRACE_GET_OBJECT_FILTER(TASK, TRACE_GET_CURRENT_TASK()) & CurrentFilterMask) \
        { \
            prvTraceStoreKernelCall(SERVICE, TRACE_CLASS_TASK, 0); \
        } \
    }

/* This macro will create a call to a kernel service with a certain result, with an object as parameter */
#undef trcKERNEL_HOOKS_KERNEL_SERVICE_WITH_PARAM
#define trcKERNEL_HOOKS_KERNEL_SERVICE_WITH_PARAM(SERVICE, CLASS, pxObject, param) \
    if (TRACE_GET_OBJECT_FILTER(TASK, TRACE_GET_CURRENT_TASK()) & CurrentFilterMask) \
        if (TRACE_GET_OBJECT_FILTER(CLASS, pxObject) & CurrentFilterMask) \ izbrisani red koda
            prvTraceStoreKernelCallWithParam(SERVICE, TRACE_GET_OBJECT_TRACE_CLASS(CLASS, pxObject), TRACE_GET_OBJECT_NUMBER(CLASS, pxObject), (uint32_t)param);
```

Слика 20.- Измене у фајлу trcRecorder.h

/trcKernelPort.h

```
/**
 * @brief Retrieve filter of task
 *
 * @param[in] pxTask Task handle
 *
 * @returns uint16_t Task filter
 */
#define TRACE_GET_TASK_FILTER(pxTask) prvTraceGetTaskNumberHigh16((void*)pxTask)

/**
 * @brief Set filter of task
 *
 * @param[in] pxTask Task handle
 * @param[in] group Group
 */
#define TRACE_SET_TASK_FILTER(pxTask, group) prvTraceSetTaskNumberHigh16((void*)pxTask, group)

/**
 * @brief Retrieve filter of queue
 *
 * @param[in] pxQueue Queue handle
 *
 * @returns uint16_t Queue filter
 */
#define TRACE_GET_QUEUE_FILTER(pxQueue) prvTraceGetQueueNumberHigh16((void*)pxQueue)
```

Low

Low

Low

Слика 21.- Измене у фајлу trcKernelPort.h

У *trcKernelPort.h* је била потребна измена да се ознаке објекта кернела дохватају и постављају кроз ниже бите иначе у супортном не би се позивале функције које памте догађаје у *Event Data* баферу. У програмском коду за сваку прекидну рутину су креирани и иницијализовани *handle*-ови пре почетка распоређивача (*Scheduler*) позивима:

PORT1Handle = prvTraceGetObjectHandle(TRACE_CLASS_ISR);

PORT1Handle = xTraceSetISRProperties("PORT1", PRIO_ISR_PORT1); (при чему се додају име и приоритет),

да би на почетку и на крају сваке прекидне рутине били уписани у бафер позивањем функције *vTraceStoreISRBegin* и *vTraceStoreISREnd* као на слици 22.

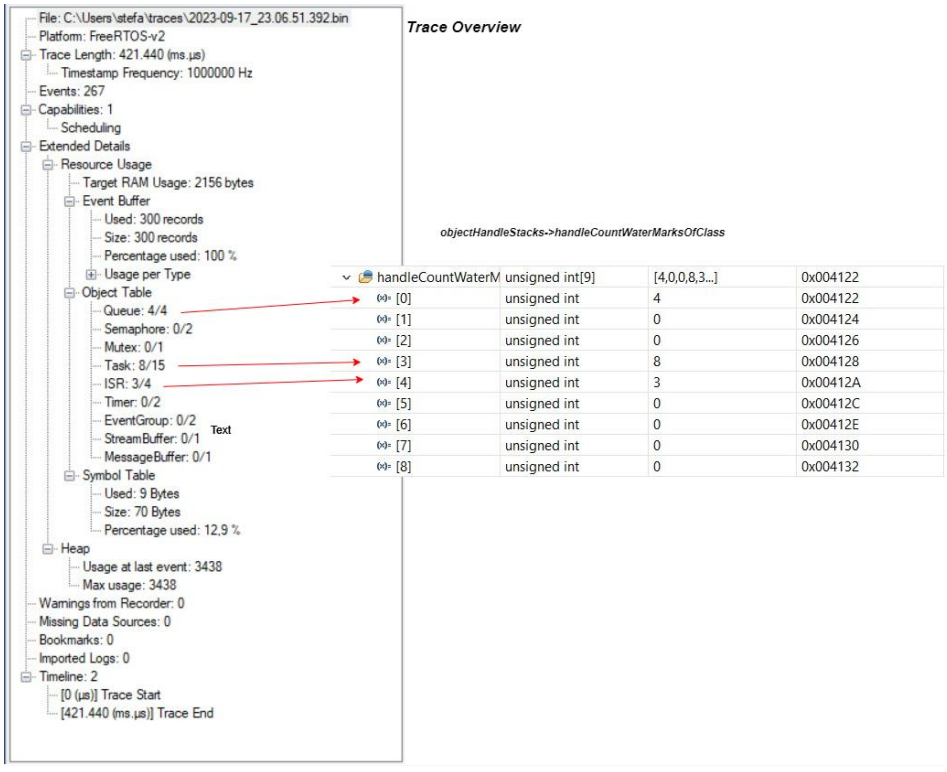
```
void __attribute__ ( ( interrupt( PORT1_VECTOR  ) ) ) vPORT1ISR( void )
{
    /*Clear IFG register on exit. Read more about it in offical MSP430F5529 documentation*/
    P1IFG &=~0x20;
    vTraceStoreISRBegin(PORT1Handle);
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    vTaskNotifyGiveFromISR(xTask2Handle, &xHigherPriorityTaskWoken);

    /* trigger scheduler if higher priority task is woken */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
    vTraceStoreISREnd(0);
}
```

Слика 22.-Пример прекидне рутине

Пре позива функције *vTraceEnable*, предефинисала се вредност TRC_HWTC_FREQ_HZ позивом *vTraceSetFrequency(1000000)* јер у фајлу *trcHardwarePort.h* није у потпуности дефинисан порт TRC_HARDWARE_PORT_TEXAS_INSTRUMENTS_MSP430. Након креирања редова порука, пре стартовања кернела дефинисана су и имена ради боље прегледности у *Tracealyzer* позивањем, на пример за ред поруке *xADC0Mailbox*, функције:

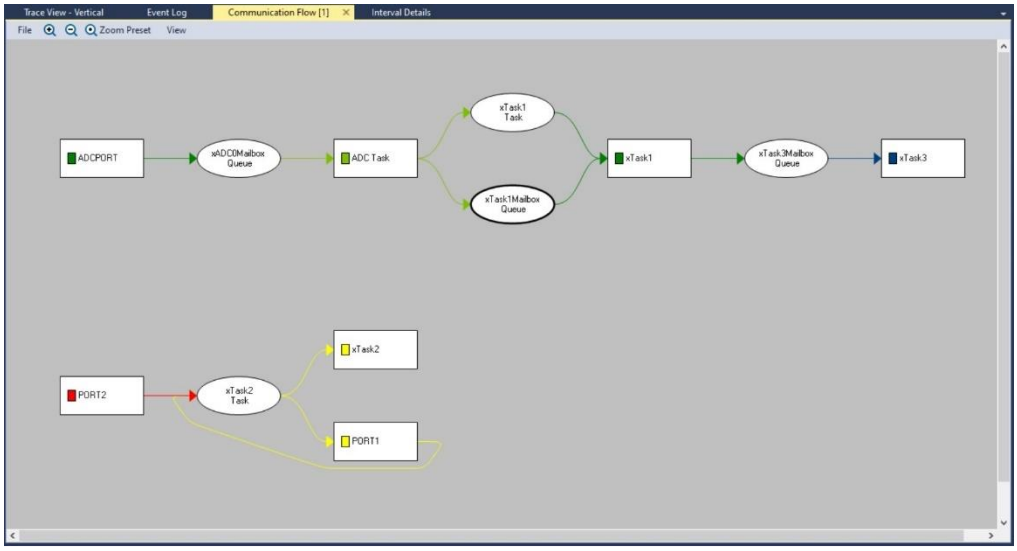
vTraceSetQueueName(xADC0Mailbox, "xADC0Mailbox").

Резултати интеграције се могу видети на сликама испод:



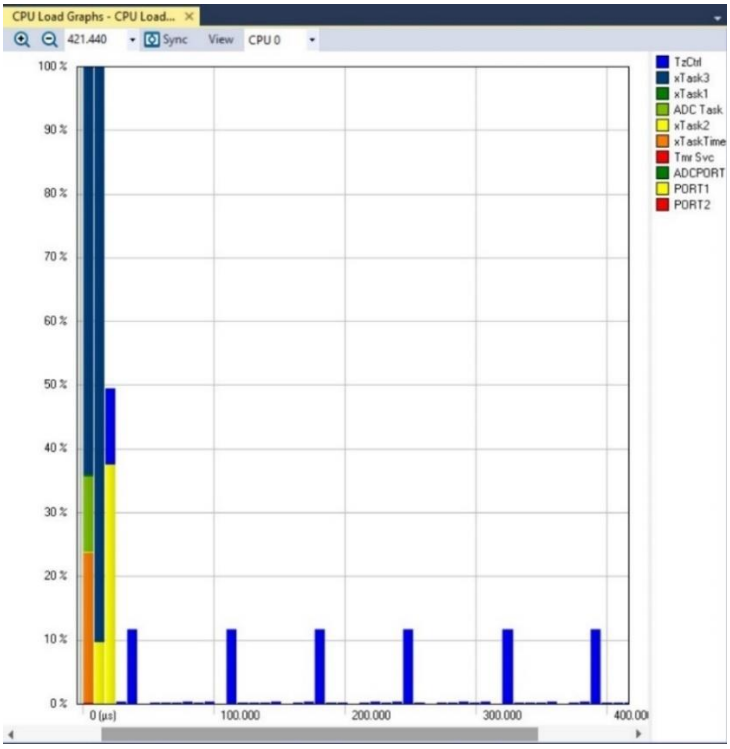
Слика 23.- Trace Overview - Tracealyzer

Преглед праћења (*Trace Overview*) приказује детаље снимљеног трага и сва упозорења која може произвести снимање (Слика 22). У овом прегледу се јасно види поклапање броја активних објеката за сваки класу и да је *Event Buffer* цео искоришћен пре чему се прекида снимање у моду TRC_SNAPSHOT_MODE_STOP_WHEN_FULL.



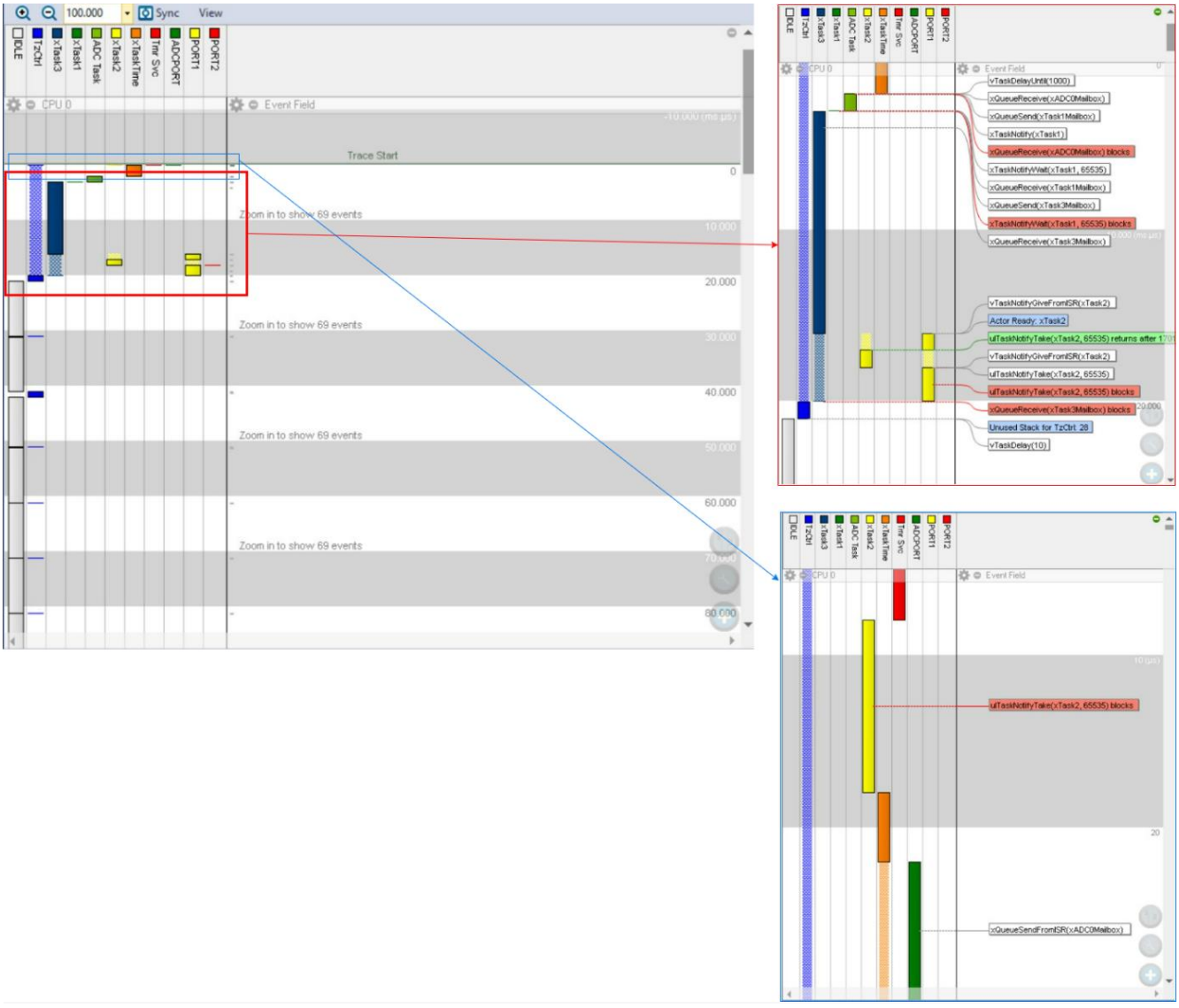
Слика 24.- Communication Flow - Tracealyzer

График тока комуникације (*Communication Flow*) нуди преглед комуникација и синхронизације између актера у програму кроз редове порука, семафора и мутексе (Слика 23). Може се генерисати на целом снимљеном интервалу или за одређени интервал. Главни актери (углавном таскови) су приказани као правоугаоници, док елипсе представљају објекте за синхронизацију и за усмерену комуникацију и хексагоне представљају двосмерне објекте. За усмерене и двосмерне објекте се одређују на основу позива сервиса које обављају. Тренутни график тока комуникације илуструје односе и везе међу објеката кернела само за време снимања, нови догађаји у наставку снимања могу да измене график тока комуникације.



Слика 25.- Приказ The CPU Load Graph - Tracealyzer

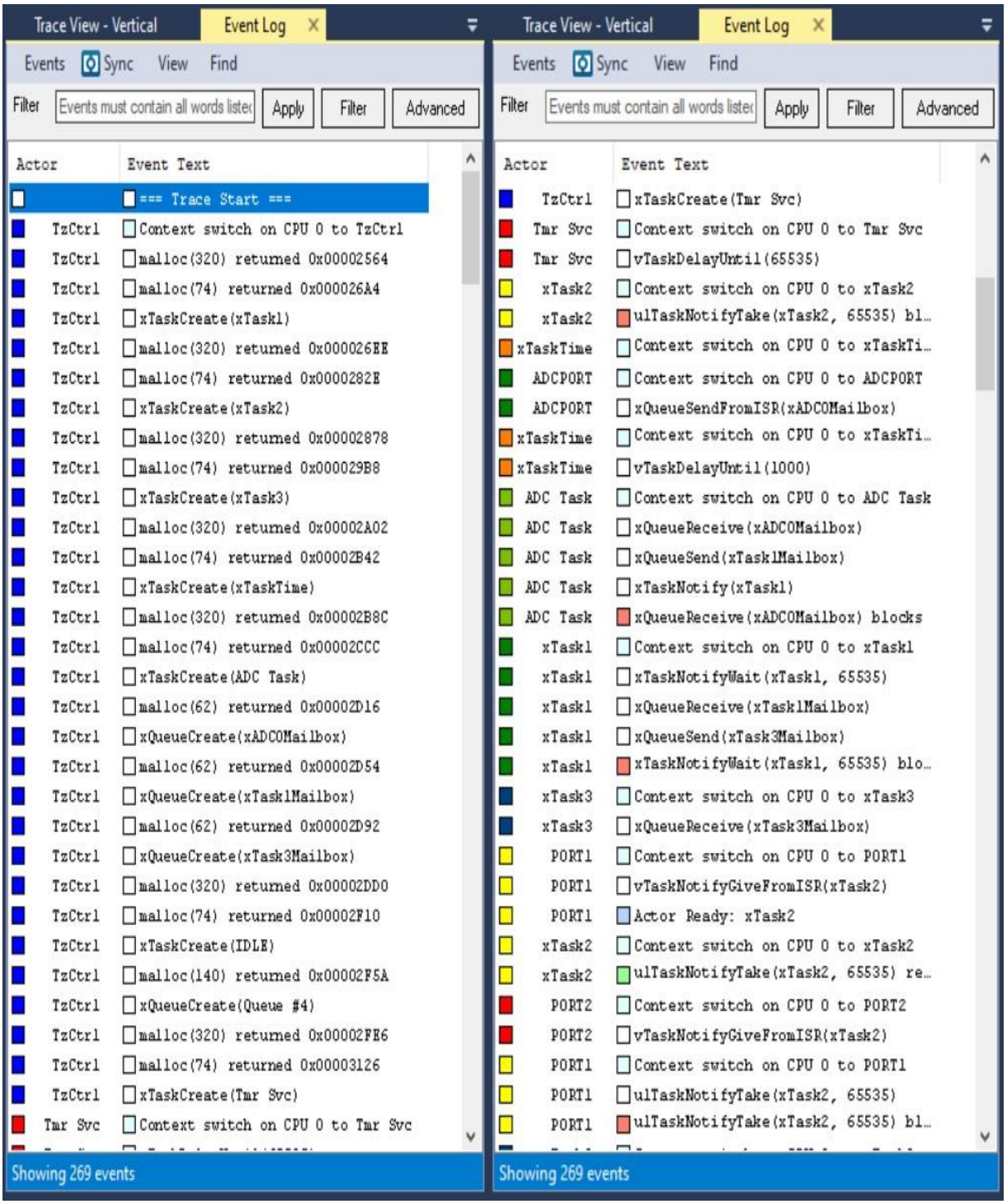
Графикон оптерећења *CPU* (*The CPU Load Graph*) приказује употребу процесора током времена за сваког актера током снимања (Слика 24). Сваки актер (таск или прекидна рутина) су приказани осим *IDLE* таска и анализа се дели на одређени број интервала (подразумевано 100). У једном интервалу за одређеног актера је приказано количина процесорског времена коју је актер искористио. Ако је више актера користио процесорско време у датом интералу, они ће се приказати наслагани један на другог и висина правоугаоника ће означавати ниво оптерећености процесора. Могуће је филтрирати које актере да се приказују и резолуција (огледа се као број приказаних интервала).



Слика 26.- Приказ Trace View - Tracealyzer

Главни приказ апликације *Tracealyzer* јесте *Trace View* (Слика 25) која приказује све забележене догађаје на вертикалној или хоризонталној временској линији. Сваки актер (таск или прекидна рутина) има свој обојени фрагмент што указује на извршавање које је почело и завршило се пребацивањем контекста, омогућава приказ време њиховог извршавања и како таскови, прекидне рутине и други догађаји међусобно делују хронолошки. Подразумевана шема боја иде од црвене до плаве боје по опадајућем

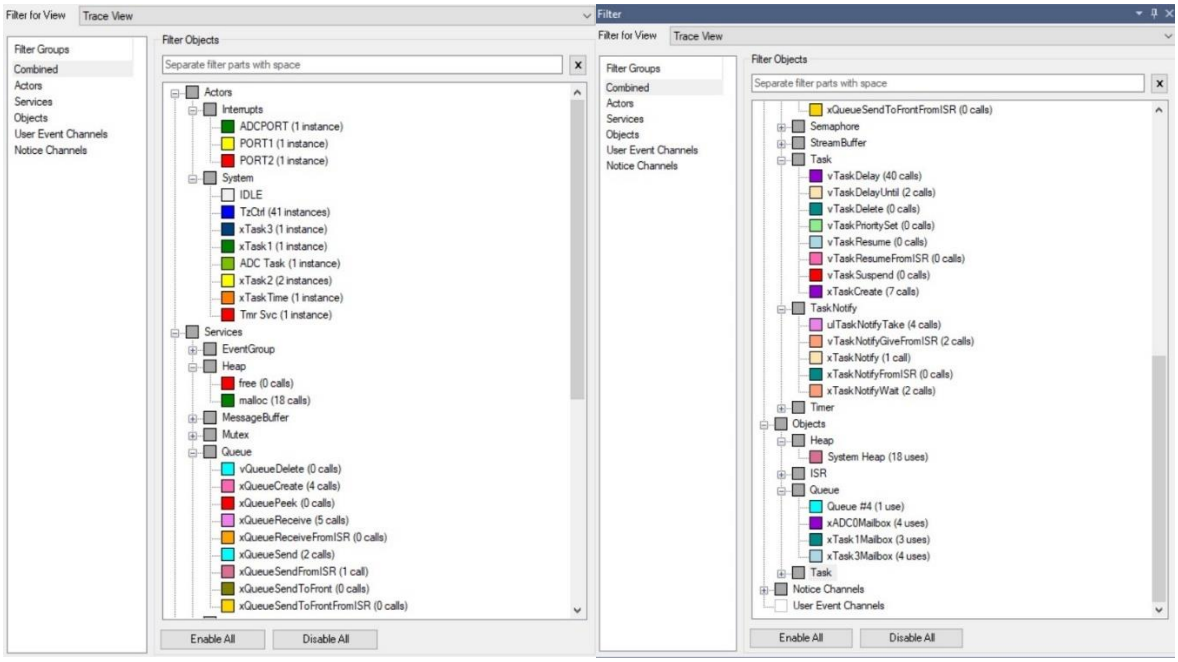
приоритету. Потребно је зумирати како би се појавили поља догађаја у датим временским тренуцима (*Events field*). Приказују се догађаји у зависности од приоритета типа догађаја која се дефинишу у подешавањима. Поред поља догађаја у овом прегледу се приказују и поља за распоређивање (*Scheduling field*), поља метрике (*Metrics field*) и поља временских интервала (*Intervals field*).



Слика 27.- Приказ Event Log - Tracealyzer

Дневник догађаја (*Event Log*) приказује прикупљене податке као текстуални списак догађаја са могућностима филтрирања и извоз списка као текстуалне датотеке

(Слика 26). Кликом на било који догађај, истовремено ће се истаћи у *Trace View* прегледу. Као филтер могу се у комбинацији приказати кориснички догађаји (*User Events*), услуге (*Service*), инфо (*Info Events*) и необрађени догађаји (*Raw Events*). Поред претходних начина филтрирања могуће је и по садржају и по временској ознаци. Из прегледа *Trace View* и *Event Log* се јасно види да *startup* таск (у овом случају TzCtrl) креиран за иницијализацију система и креирања корисничких таскова и приликом стартовања распоређивача, кернел креира сопствени сет системских таскова. Први таск ко је добио процесорско време јесте xTask2 на основу највећег приоритета који се блокирао чекајући нотификацију. Следећи таск који се извршава и има исти приоритет као xTask2 је xTaskTimer који стартује конверзију и иницира ADCPORT прекидну рутину која дохвата вредност и уписује на крај реда са порукама *xADC0Mailbox*, у таску ADC Task обавештава xTask1-у о приспећу нове вредности. xTask1 даље ставља вредност на крај реда са порукама *xTask3Mailbox* где xTask3 излази из блокираног стања у стање извршавања и приказује вредност на *UART*-у. Међутим, приликом извршавања таска xTask3, притиском на дугме S4 на микроконтролеру се иницира прекидна рутина PORT1 и прекида извршавање xTask3. Прекидна рутина PORT1 пребацује ток контроле на xTask2 који је већег приоритета од xTask3 и тек после xTask2 таск xTask3 наставља извршавање које је било прекинуто прекидном рутином. Када ниједан други таск није спреман за извршавање, онда се извршава таск Idle коју прекида таск TzCtrl за контролу неискоришћености стека.



Слика 28.- Приказ Filter - Tracealyzer

Филтрирање у *Percepio Tracealyzer* се врши у приказу Филтер (*Filter*) које приказује објекте апликације организоване у различите групе (Слика 27). Објекти су организовани хијерархијски што олакшава сакривање читаве групе објеката. Промена на филтеру ће се одразити на свим приказима у апликацији.

6. Закључак

Софтвер *Percepio Tracealyzer* је корисно средство које може учинити развој и процеси за отклањање грешака лакше и брже. Једноставан за коришћење уколико се зна унапред где је потребно изменити макрое кључне за промене тока контроле између таскова и прекидних рутина, записивање догађаја везане за објекте различитих класа, у том случају је једноставан и користан и на образовним курсевима везане за развој система за рад у реалном времену. Тренд коришћења је у порасту у односу на друге сличне софтвере(на пример *Trace Compass*). Међутим, постоје и друге апликације које програмери наменских система користе за отклањање грешака и верификацију попут *A trace enabled debugger, Micrium uC Probe, SEGGER System View, SEGGER Ozone, Express Logic TraceX*. Тренутне верзије су доступне за интеграцију пројекта које су засноване на *FreeRTOS, OpenRTOS, SafeRTOS, RTXC Quadros, Zephyr, ThreadX, µC/OS-III, BareMetal* и *On Time RTOS-32*. Приликом бирања између *Percepio Tracealyzer* и његових алтернатива, потребно је узети у обзир RTOS компатабилност, једноставност употребе, цена и прилагођавање програма за прикупљање и анализу података.

7. Литература

[1] Introduction to Tracealyzer, https://percepio.com/docs/FreeRTOS/manual/index.html#Creating_and_Loading_Traces___Percepio_Trace_Recorder___FreeRTOS_integration ,

[2] Richard Barry, „Mastering the FreeRTOS™ Real Time Kernel“, *Real Time Engineers Ltd.*, 2016. 161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf,

[3] https://www.pomad.fr/PoMAD/node/37 ,

[4] Trace Hook Macros, https://www.freertos.org/rtos-trace-macros.html ,

[5] Product details – MSP430F5529, https://www.ti.com/product/MSP430F5529 ,

[6] Др. Иван Поповић, мс. Харис Туркмановић „Наменски рачунарски системи за рад у реалном времену“

[7] The FreeRTOS™ Reference Manual, https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf

[8] Timer_A - from the MSP430x5xx and MSP430x6xx Family User's Guide, https://www.ti.com/lit/ug/slau400f/slau400f.pdf?ts=1694617023771&ref_url=https%253A%252F%252Fwww.bing.com%252F,

[9] https://www.segger.com/supported-devices/texas-instruments/tms320,

[10] https://app.diagrams.net/,

[11] Johan Kraft, “Benefits of Tracealyzer vs. Trace Compass”, https://percepio.com/partner-material/tracealyzer-vs-tracecompass.pdf,

8. Списак слика