

- Подготовка прошивки
- Запуск отладчика
- Настройка IDE
- Утилита ECAM для отправки произвольных запросов JSON-RPC и ONVIF
- Удалённая отладка Go через VSCode
  - Установка VSCode
  - Установка расширений VSCode
    - Обязательные расширения
      - Расширение Go Nightly
      - Расширение Command Variable
    - Рекомендуемые расширения
      - Расширение Go Critic (customizable)
      - Расширение Tab-Indent Space-Align
  - Предварительные требования
    - SSHPASS
    - RSYNC
    - Buildroot
  - Подготовка проекта
  - Удалённая отладка через Delve
  - Запуск приложения без Delve
  - Статический анализ staticcheck
  - Дополнительные функции
    - Статические анализаторы
    - CLEAN\_GOCACHE
    - COPY\_FILES и COPY\_CACHE
    - Загрузка файлов через Overlay
    - DIRECTORIES\_CREATE
    - DELETE\_FILES
    - EXECUTE\_COMMANDS
    - SERVICES\_STOP и SERVICES\_START
    - PROCESSES\_STOP и PROCESSES\_START
    - CAMERA\_FEATURES\_ON и CAMERA\_FEATURES\_OFF
    - TARGET\_IPADDR
    - TARGET\_ARCH и TARGET\_GOCXX
    - TARGET\_SUPPRESS\_MESSAGES
    - GOPROXY
    - Дополнительные параметры
  - Известные проблемы
  - TODO

# 1 Подготовка прошивки

⚠ В Buildroot отладка с использованием delve поддерживается только на aarch64/x86/x86\_64

Установить опции в конфиге Buildroot:

```
BR2_PACKAGE_DELVE=y  
BR2_ENABLE_DEBUG=y  
BR2_STRIP_strip=n
```

Для onvifd добавить флаг сборки `-gcflags "all=-N -l"` и пересобрать onvifd.

Примечание: Достаточно пересобрать onvifd и собрать образ Buildroot (с нуля пересобирать Buildroot не нужно).

## 2 Запуск отладчика

```
dlv attach `pidof -s onvifd` onvifd --listen=:2345 --headless=true --log=true --log-output=debugger,debuglineerr,gdbwire,lldbout,rpc --accept-multiclient --api-version=2
```

### 3 Настройка IDE

См. <https://golangforall.com/en/post/go-docker-delve-remote-debug.html#visual-studio-code>.

## 4 Утилита ECAM для отправки произвольных запросов JSON-RPC и ONVIF

См. [ecam](#): Утилита для отправки произвольных запросов JSON-RPC и ONVIF.

---

## 5 Удалённая отладка Go через VSCode

Содержимое данный репозитория предназначено для обеспечения по возможности прозрачной удалённой отладки Go приложений.

Удалённая отладка включает в себя:

- сборку приложения кросскомпилятором Go;
- загрузку приложения на целевую систему;
- запуск удалённой отладочной сессии при помощи отладчика Delve.

Все эти функции реализуются скриптами, написанными на простом `bash`. Кроме самой удалённой отладки скрипты позволяют автоматизировать некоторые второстепенные, но необходимые задачи, как то:

- загрузка вспомогательных файлов на целевую платформу (конфигурации, скриптов, программ и тд.);
- запуск и останов сервисов `systemd`, произвольных процессов;
- выполнение любых вспомогательных команд (например `wget` для включения специальных функций через HTTP);
- кеширование, сжатие, таким образом ускорение процесса загрузки данных по `ssh`.

Все скрипты и конфигурации находятся в стадии разработки (доработки) и могут быть кастомизированы/доработаны под конкретные условия и задачи.

---

### 5.1 Установка VSCode

Существуют 2 варианта установки VSCode под RHEL, Fedora и CentOS. Первый - это установка соответствующего `flatpak` пакета. Второй - добавление `rpm` репозитория и установка из него. Предполагается, что VSCode должен обновляться и если в вашу систему не интегрированы инструменты для автоматического обновления flatpak, например, Discover, то в этом случае будет целесообразно устанавливать VSCode из `rpm`.

Итак, для установки VSCode из Flathub достаточно выполнить:

```
flatpak install flathub com.visualstudio.code
```

Установка VSCode из `rpm` пакета описана на странице [Установка VSCode](#).

Добавление ключа и регистрация репозитория VSCode:

```
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
sudo sh -c 'echo -e "[code]\nname=Visual Studio Code\nbaseurl=https://
packages.microsoft.com/yumrepos/vscode\nenabled=1\nngpgcheck=1\nngpgkey=https://
packages.microsoft.com/keys/microsoft.asc" > /etc/yum.repos.d/vscode.repo'
```

Обновление списка пакетов и установка VSCode через `dnf`:

```
dnf check-update
sudo dnf install code
```

Так же можно использовать `yum`:

```
yum check-update  
sudo yum install code
```

После установки в меню **Пуск** должен появиться пункт **Visual Studio Code**. Так же VSCode можно запустить из командной строки:

```
$ code --version  
1.78.2  
b3e4e68a0bc097f0ae7907b217c1119af9e03435  
x64
```

## 5.2 Установка расширений VSCode

Для корректной работы VSCode с Go, а так же, чтобы упростить себе жизнь, необходимо установить несколько расширений.

Процесс установки расширений единообразен. Для установки необходимо запустить VS Code Quick Open (Ctrl+P), вставить определённую команду `ext install` и нажать Enter.

### 5.2.1 Обязательные расширения

#### Расширение Go Nightly

Расширение поддержки языка Go для VSCode. Целесообразно устанавливать именно основанной на `master` ночную сборку, т. к. в ней раньше всего появляются новые фичи. ([Link](#))

```
ext install golang.go-nightly
```

После установки расширение предложит доустановить некоторый инструментарий Go. Со всеми его просьбами желательно согласиться.

#### Расширение Command Variable

Расширение, которое позволяет получать параметры конфигурации VSCode из внешних файлов. Расширение необходимо для автоматизации некоторых связанных с удалённой отладкой моментов. ([Link](#))

```
ext install rioj7.command-variable
```

### 5.2.2 Рекомендуемые расширения

#### Расширение Go Critic (customizable)

Еще один, пожалуй, *наиболее авторитетный* линтер для Go. ([Link](#))

```
ext install imgg.go-critic-imgg
```

## Расширение Tab-Indent Space-Align

VSCode из коробки *не умеет* автоматическую indentation текста (путает пробелы с табуляцией, неверно заполняет indentation). Это расширение исправляет данное недоразумение. ([Link](#))

```
ext install j-zeppenfeld.tab-indent-space-align
```

Можно сказать, что это минимально базовый набор расширений, который позволяет разрабатывать приложения на Go. Многие другие полезные расширения можно найти на [Visual Studio Marketplace](#).

## 5.3 Предварительные требования

### 5.3.1 SSHPASS

Удалённая отладка активно использует SSH соединение. Для того чтобы запускать SSH в неинтерактивном режиме, используется утилита `sshpass`. Если не установлена в системе, следует установить:

```
sudo dnf install sshpass
```

### 5.3.2 RSYNC

Для увеличения скорости загрузки желательна установка `rsync`.

```
sudo dnf install rsync
```

Копирование файлов через `rsync` можно отключить через `USE_RSYNC_METHOD=false`. В этом случае копирование файлов будет производится по SSH.

Сборка `rsync` для целевой платформы (`aarch64`, `armv7l`) не требуется. Готовые бинарные файлы поставляются вместе со скриптами отладки и при необходимости будут загружены на устройство автоматически. Установка `rsync` производится по пути `"/usr/bin/rsync"`.

### 5.3.3 Buildroot

Удалённая отладка предполагает кроссплатформенную сборку приложения Go и запуск полученного исполняемого файла на целевой платформе с архитектурой, как правило, отличной от архитектуры хоста (`x86_64`).

Итак, для сборки и отладки Go приложения понадобятся:

- кросскомпилятор `go` ;
- отладчик `delve` .



Все эти инструменты будут получены из `buildroot`. Если компилятор `go` включен в процесс сборки безусловно, то для сборки дополнительного пакета с отладчиком `delve` необходимо внести изменения в конфигурацию пакетов `buildroot`.

Для этого в каталоге `"$BUILDROOT/external-ipcam/ecam03-fragments"` необходимо создать файл с именем `"dbg.fragment"` и следующим содержимым:

```
BR2_PACKAGE_DELVE=y
BR2_PACKAGE_PPROF=y
```

В этот файл можно добавить любые необходимые настройки `buildroot`.

Необходимо включить новый фрагмент в конфигурацию `buildroot` и, если не собран, собрать `buildroot` полностью, либо дособрать пакет `delve`. Пример скрипта минимальной сборки:

```
#!/bin/sh

BUILD_DIRNAME=ecam03_toolchain
BUILD_TARGETS=(host-go delve libxml2 onvifd)
BUILD_POSTFIX=--reconfigure

export https_proxy=http://proxy.elvees.com:3128
export http_proxy=http://proxy.elvees.com:3128
export ftp_proxy=http://proxy.elvees.com:3128
export no_proxy=127.0.0.1,localhost,elvees.com

export BR2_PRIMARY_SITE=http://callisto.elvees.com/mirror/buildroot
export PYTHONUSERBASE=$HOME/.python
export PATH=$PYTHONUSERBASE/bin:$PATH

export GOPROXY=http://athens.elvees.com,https://proxy.golang.org,direct

SCRIPT_ARG="$1"
if [[ "$SCRIPT_ARG" == "delete" ]] && [[ -d "$BUILD_DIRNAME" ]]; then
    rm -rf "$BUILD_DIRNAME"
    SCRIPT_ARG=""
fi

if [[ ! -d "$BUILD_DIRNAME" ]]; then
    git clone "ssh://$USER@gerrit.elvees.com:29418/ecam03/buildroot" "$BUILD_DIRNAME"
    scp -p -P 29418 "$USER@gerrit.elvees.com:hooks/commit-msg" "$BUILD_DIRNAME/.git/hooks/"
fi

cd "$BUILD_DIRNAME"
git submodule init
git submodule update --recursive
git pull
git pull --recurse-submodules

git reset --hard
git submodule foreach --recursive git reset --hard
git submodule update --init --recursive
```

```

# Копируем local.mk в buildroot
if [[ -f "../local.mk" ]]; then
    cp "../local.mk" "../buildroot/"
fi

# Создание dbg.fragment
cat <<EOF >> "../external-ipcam/ecam03-fragments/dbg.fragment"
BR2_PACKAGE_DELVE=y
BR2_PACKAGE_PPROF=y
BR2_PACKAGE_DSP_THERMO_TESTS=n
EOF

# Применение конфигурации
#make distclean
make ecam03_defconfig FRAGMENTS=dev:dbg

# Сборка Buildroot
export BR2_JLEVEL="$(nproc)"
for target in "${BUILD_TARGETS[@]}; do
    make -j$(nproc) "$target$BUILD_POSTFIX"
done

```

В результате сборки должны быть созданы исполняемые файлы:

```

GO="$BUILDROOT/buildroot/output/host/bin/go"
DLV="$BUILDROOT/buildroot/output/target/usr/bin/dlv"

```

Следует заметить, что компилятор `go` собирается под архитектуру хоста (x86\_64), а отладчик `dlv` (delve) - под архитектуру целевой платформы.

Для сборки приложения кросс компилятор задействует менеджер пакетов Go. Менеджер пакетов автоматически скачивает все необходимые зависимости и устанавливает их, распределяя компоненты по каталогам. Менеджер пакетов Go ограничен границами `buildroot` (фактически, Go живет внутри `buildroot`). С этой точки зрения нам могут быть следующие переменные окружения Go:

```

BUILDROOT_HOSTDIR="$BUILDROOT/buildroot/output/host"
GOROOT="$BUILDROOT_HOSTDIR/lib/go"
GOPATH="$BUILDROOT_HOSTDIR/usr/share/go-path"
GOMODCACHE="$BUILDROOT_HOSTDIR/usr/share/go-path/pkg/mod"
GOTOOLDIR="$BUILDROOT_HOSTDIR/lib/go/pkg/tool/linux_arm64"
GOCACHE="$BUILDROOT_HOSTDIR/usr/share/go-cache"

```

Из-за того, что Go использует каталоги `buildroot` для хранения зависимостей, пакетов и кеша сборки, в процессе сборки в `buildroot` будет появляться некоторое количество неизвестных для `buildroot` файлов (например, имеющих неправильного с точки зрения `buildroot` владельца). Это не сломает сборку `buildroot`, но может привести к непредвиденным результатам. Поэтому:

*Настоятельно не рекомендуется использовать один и тот же `buildroot` для удалённой отладки и сборки образа либо обновлений для последующей загрузки на устройство.*

Для удалённой отладки желательно использовать отдельно стоящий `buildroot`, который никак не участвует в процессе разработки.

## 5.4 Подготовка проекта

Конфигурация проекта для среды VSCode находится в корне проекта, в каталоге с именем `".vscode"`. Конфигурация проекта сводится к замене либо созданию данного каталога.

Итак, прежде всего необходимо распаковать приложенный к данному документу архив ([vscode-goflame-20230519-120748.tar.xz](https://github.com/rabramov/vscode-goflame/releases/download/v20230519-120748/vscode-goflame-20230519-120748.tar.xz)) либо клонировать репозиторий проекта [Gitlab](https://gitlab.com/rabramov/vscode-goflame) либо [Bitbucket](https://bitbucket.org/proton-workspace/vscode-goflame):

```
# Основной репозиторий:
git clone git@gitlab912.elvees.com:rabramov/vscode-goflame.git
# Зеркало:
git clone git@bitbucket.org:proton-workspace/vscode-goflame.git
```

*Т.к. репозиторий периодически обновляется (исправляются ошибки, добавляются новые функции), предлагаю подписаться на изменения на странице Gitlab: <https://gitlab912.elvees.com/rabramov/vscode-goflame/>*

В итоге в `"vscode-goflame"` получаем копию проекта со всеми необходимыми для удалённой отладки настройками VSCode и скриптами. Один и тот-же экземпляр `"vscode-goflame"` может быть использован в нескольких проектах одновременно. Для этого в каждом из проектов нужно создать символическую ссылку с именем `".vscode"`:

```
rm -rf "$PROJECT_DIR/.vscode" # удаление существующего симлинка
ln -s "$PWD/vscode-goflame/vscode" "$PROJECT_DIR/.vscode"
```

Далее, в файле `[vscode-goflame/vscode/config.ini]` в переменную `TARGET_IPADDR` необходимо записать IP адрес отлаживаемой камеры либо серийный номер камеры если она зарегистрирована в корпоративной сети (например, `ecam03-2364031`), а также установить переменную `BUILDDROOT_DIR` таким образом, чтобы она указывала на `buildroot` с собранными ранее инструментами Go (`go`, `dlv`).

```
TARGET_IPADDR=10.113.11.65
BUILDDROOT_DIR="$ECAM03_SUPERPROJECT/buildroot"

# IP адрес будет получен через DNS lookup относительно домена из переменной TARGET_DOMAIN
TARGET_IPADDR=ecam03-2364031
```

Где `ECAM03_SUPERPROJECT` - путь к суперпроекту, который содержит `buildroot`. Переменная `BUILDDROOT_DIR` необходима для получения путей к инструментам Go.

Назначение некоторых параметров в `[vscode-goflame/vscode/config.ini]`:

- `BUILDDROOT_DIR` - переменная необходима для получения путей к инструментам Go.
- `TARGET_USER` - имя пользователя SSH. `TARGET_PASS` - пароль доступа по SSH.
- `TARGET_IPPORT` - порт Delve, на котором запускается отладка.
- `GO_VET_ENABLE` - запуск линтера `go vet` перед сборкой проекта.
- `STATICCHECK_ENABLE` - запуск линтера `staticcheck` перед сборкой проекта.
- `STATICCHECK_CHECKS` - набор проверок реализуемых `staticcheck`.
- Переменные окружения Go (см. [Go Environment variables](#))

На этом конфигурирование проекта можно считать завершённым. Все необходимые файлы конфигурации и скрипты находятся в каталоге проекта, в подкаталоге `".vscode"`.

## 5.5 Удалённая отладка через Delve

В VSCode через меню «File/Open Folder» (Ctrl+K Ctrl+O) открываем проект (который содержит `".vscode"`) . Если все установлено правильно, VSCode подхватит настройки проекта и попытается запустить инструменты Go из окружения `buildroot` . Т.к. в ранее собранном `buildroot` отсутствуют такие инструменты как линтеры, приложение автоматического форматирования кода и некоторые другие, VSCode (уже второй раз) предложит их загрузить и установить. Все загруженное будет установлено в `buildroot` . После установки можно приступить к, собственно, сборке проекта и его отладке.

*Подробное описание функций отладки можно найти на странице [документации к VSCode](#).* Переходим к вкладке «Run and Debug» (Ctrl+Shift+D) и в выпадающем списке «RUN AND DEBUG» выбираем «**GO Deploy & Debug**». Это одна из пользовательских конфигураций запуска, объявленная в файле `".vscode/launch.json"` .

Перед первым запуском необходимо выполнить полную сборку проекта. Открываем «VS Code Quick Run» (Ctrl+Shift+P) и выполняем команду <Go: Build Workspace> (Ctrl+B). Этот шаг необходимо выполнять каждый раз, когда начинаем работу с новым устройством и кроме самой сборки команда <Go: Build Workspace> выполняет:

- загрузку вспомогательных скриптов и отладчика delve;
- загрузку собранного исполняемого файла;
- загрузку/обновление файлов конфигурации (если нужно);
- включение features на отлаживаемом устройстве;
- принудительная остановка некоторых служб (onvifd).

*Для отладки не обязательно, чтобы на целевое устройство была установлена прошивка содержащая Delve - отладчик будет загружен на устройство автоматически.*

Т.е. таким образом полная пересборка проекта так-же настраивает целевое устройство. Теоретически, все эти шаги можно выполнять непосредственно перед запуском отлаживаемого файла, но это несколько увеличивает и без того немалое время запуска.

На закладке OUTPUT VSCode появится текст с ходом выполнения сборки:

```
Starting building the current workspace at $PROJECTDIR
$PROJECTDIR>Finished running tool: $PROJECTDIR/.vscode/scripts/go.sh build
17/05/2023 20:01:28 [go] Building `cmd/onvifd/onvifd.go'
17/05/2023 20:01:28 [go] Installing to remote host `root@10.113.11.65'
17/05/2023 20:01:29 [go] Camera feature "videoanalytics" is set to "true".
17/05/2023 20:01:29 [go] Stopping 2 services: onvifd, onvifd-debug
17/05/2023 20:01:29 [go] Terminating 3 processes: dlv, onvifd, onvifd_debug
17/05/2023 20:01:29 [go] Removing 3 files: onvifd_debug, onvifd_debug.log, dlv.log
17/05/2023 20:01:29 [go] Uploading 7 files: dl, ds, onvifd-debug.service,
onvifd_debug, dlv, onvifd.conf, users.toml
17/05/2023 20:01:33 [go] Total runtime: 4.9802s
```

*Если закладка OUTPUT остаётся пустой, следует убедиться, что в выпадающем списке выбора вывода выбрано «Go».*

PROBLEMS 31 **OUTPUT** DEBUG CONSOLE TERMINAL COMMENTS Go

Теперь необходимо открыть любой терминал (GNOME Terminal, Konsole). Так же можно воспользоваться терминалом встроенным в VSCode (но, как показала практика, это менее удобно). В терминале необходимо войти на устройство по SSH и запустить загруженный ранее скрипт отладки `dl` (Delve Loop):

```
$ ssh root@IP_ADDRESS
```

```
# dl
```

```
Starting Delve headless server loop in DAP mode. To stop use: $ ds
DAP server listening at: [::]:2345
```

В этот момент Delve ожидает входящего соединения на порту 2345. Запускаем отладку в VSCode через «Run/Start Debugging» (F5). VSCode попытается собрать проект и одновременно загрузит собранный файл на целевое устройство. На вкладке «TERMINAL» появятся следующие сообщения:

```
17/05/2023 20:04:43 [launch-deploy-debug] Building & deploying `onvifd_debug' to
remote host http://10.113.11.65
17/05/2023 20:04:43 [launch-deploy-debug] Total runtime: 0.4136s
* Terminal will be reused by tasks, press any key to close it.
```

Текст `http://10.113.11.65` распознается VSCode как гиперссылка и при его помощи можно быстро открыть браузер по IP адресу устройства.

После этого VSCode запустит удалённую отладку и переключится на вкладку «DEBUG CONSOLE». Но эта консоль останется пустой - Delve не умеет пробрасывать STDOUT/STDERR отлаживаемой программы на хостовую систему. Таким образом в открытом терминале (GNOME Terminal, Konsole) можно наблюдать, что приложение запустилось и даже что-то пишет в консоль:

```
|| Starting Delve headless server loop in DAP mode.
DAP server listening at: [::]:2345
2023-05-18T08:09:39Z info layer=debugger launching process with args: [/usr/bin/
onvifd_debug -settings /root/onvifd.settings]
2023/05/18 08:09:43 server.go:376: Starting server at 127.0.0.1:8899 ...
2023/05/18 08:09:43 operations.go:126: Starting discovery service
2023/05/18 08:09:44 server.go:779: Operation: Device.GetSystemDateAndTime
2023/05/18 08:09:44 server.go:779: Operation: Device.GetServiceCapabilities
2023/05/18 08:09:44 server.go:779: Operation: Device.GetSystemDateAndTime
2023/05/18 08:09:44 server.go:779: Operation: Device.GetServiceCapabilities
2023/05/18 08:09:44 server.go:779: Operation: Device.GetSystemDateAndTime
2023/05/18 08:09:44 server.go:775: Operation: Device.GetDeviceInformation:
Unauthorized
```

На данном этапе с отладчиком можно работать так же, как будто приложение отлаживается локально: ставить точки останова, останавливать, выполнять пошагово, перезапускать, просматривать содержимое переменных.

## 5.6 Запуск приложения без Delve

Некоторые платформы не поддерживают отладку при помощи Delve. С полным списком неподдерживаемых платформ можно ознакомиться [по ссылке](#). Тем не менее, отладочные скрипты дают возможность настроить аппаратную платформу для сборки проекта, а также позволяют прозрачно загрузить исполняемый файл на отлаживаемую систему и произвести его автоматический запуск.

Как пример можно рассмотреть камеру esam02, работающую на 32-х разрядном ARM которую Delve не поддерживает.

Для архитектур "arm" и "arm64" достаточно направить переменную `BUILDR00T_DIR` на нужный buildroot. Скрипты сборки автоматически определяют архитектуру и используемый компилятор. В нашем случае с esam02 - `BUILDR00T_DIR` должен указывать на buildroot собранный для 32-х разрядного ARM.

Для других платформ архитектура процессора и название компилятора задаются в файле `[config.ini]` переменными `TARGET_ARCH` и `TARGET_GOCXX`, например:

```
# Сборка под MIPS32
TARGET_ARCH="mips"
TARGET_GOCXX="mipsel-buildroot-linux-gnu"
```

Полный список поддерживаемых *Golang* платформ и операционных систем находится на [странице документации](#).

Далее все так же как и в случае отладки необходимо пересобрать проект через `<Go: Build Workspace>`, процесс сборки задеплоит все необходимые скрипты на устройство с адресом `TARGET_IPADDR`. Для обеспечения прозрачного запуска необходимо открыть терминал, подключиться к устройству по SSH и вместо команды `dl` набрать `de`. Пересборка проекта через `<Go: Build Workspace>` также автоматически запускает приложение.

```
$ de
|| Beginning /usr/bin/onvifd_debug execution loop...
|| Waiting for application to be started (Run/Start Debugging)...
|| Starting /usr/bin/onvifd_debug...
2023/06/23 10:55:47 server.go:425: Starting server at 127.0.0.1:8899 ...
2023/06/23 10:55:47 operations.go:126: Starting discovery service
2023/06/23 10:55:52 server.go:828: Operation: Device.GetSystemDateAndTime
2023/06/23 10:55:52 server.go:828: Operation: Device.GetServiceCapabilities
2023/06/23 10:55:52 server.go:828: Operation: Device.GetSystemDateAndTime
2023/06/23 10:55:52 server.go:824: Operation: Device.GetDeviceInformation:
Unauthorized
```

**Важно!** В VSCode переходим к вкладке «RUN AND DEBUG» (Ctrl+Shift+D) и в выпадающем списке выбираем режим запуска «**GO Deploy & Execute**».

Удалённый запуск приложения происходит через эмуляцию отладки и перехват некоторых команд: VSCode запускает отладку приложения-заглушки и параллельно управляет запуском приложения на целевом устройстве.

Поддерживаются стандартные команды `<Run/Start Debugging>` и `<Run/Stop Debugging>`. Попытка останова приложения через `<Pause>` приведёт к останову приложения-заглушки. Точки останова тоже работать не будут.

Последовательность	Действие
<code>&lt;Run/Start Debugging&gt;</code>	Запуск приложения
<code>&lt;Run/Stop Debugging&gt;</code>	Останов приложения

В остальном удаленный запуск работает так же, как удаленная отладка. Изменения приводят к сборке исполняемого файла. Измененный исполняемый файл заливается на устройство. Скрипт `de` отслеживает изменения и в случае необходимости перезапускает приложение.

## 5.7 Статический анализ `staticcheck`

Для запуска статического анализатора `staticcheck` в VSCode переходим к вкладке «RUN AND DEBUG» (Ctrl+Shift+D) и в выпадающем списке выбираем режим запуска «**GO Run StaticCheck**». Теперь при запуске проекта будет производиться статический анализ исходного кода, а результаты проверки будут выведены в терминал сборки с возможностью быстрой навигации по выявленным замечаниям.

За конфигурацию staticcheck отвечает группа переменных `STATICCHECK_XXX` из `[config.ini]`.

## 5.8 Дополнительные функции

Часто в процессе отладки необходимо обновлять, настраивать целевую систему. Скрипты отладки поддерживают некоторые функции либо операции, которые могут упростить как разработку, так и ускорить процесс отладки.

Управление этими функциями происходит при помощи переменных shell скрипта. Далее рассмотрим некоторые из них.

*Все описанные в данном разделе переменные могут находиться в файле `[config.ini]`.*

### 5.8.1 Статические анализаторы

По умолчанию процесс сборки запускает статические анализаторы кода, как то `golangci-lint`, `staticcheck`, `go vet`, `line-length-limit` и `pre-commit`.

Анализируются только изменённые части исходного кода: те файлы и строки на которые указывает `go diff`. Диагностические сообщения которые не относятся к текущим изменениям игнорируются и не выводятся. Такой алгоритм позволяет включить все проверки `golangci-lint` (`all`), `staticcheck` (`checks=all`), а так же ограничение длины строк `line-length-limit` и не обращать внимания на сообщения относящиеся к legacy коду.

Статический анализ выполняется непосредственно перед сборкой проекта, но только если были обнаружены изменения в исходном коде (фактически, в любом файле из папки проекта). Это позволяет исключить многократный запуск статических анализаторов во время отладки.

Для управление статическим анализом используются следующие переменные `[config.ini]`:

Параметр	Значение	Описание
REBUILD_FORCE_LINTERS	true/false	Если установлено, команда Go: Build Workspace будет запускать все линтеры не зависимо от состояния флагов включения. Установка REBUILD_FORCE_LINTERS полезна для ускорения запуска отладки после внесения изменений в исходный код программы: во время отладки часть ресурсоемких линтеров (или даже все) может быть отключена, в то время как полная сборка будет принудительно запускать все линтеры.
GOLANGCI_LINT_ENABLE	true/false	Включения статического анализатора <code>golangci-lint</code> .
GOLANGCI_LINT_LINTERS	("all")	Список активных линтеров <code>golangci-lint</code> . Параметр "all" включает все линтеры. Если какой-либо линтер необходимо отключить, перед его именем необходимо подставить знак минус "-".



Параметр	Значение	Описание
GOLANGCI_LINT_FILTER	true/false	Если установлено, результаты golangci-lint будут отфильтрованы и отображены только замечания, которые относятся в текущему git diff. В противном случае будут выведены все результаты.
GOLANGCI_LINT_FAIL	true/false	Если установлено, сборка провалится если golangci-lint найдёт какие либо замечания. Замечания относятся только к текущим изменениям.
GOLANGCI_LINT_SUPPRESSED	("depguard" "gochecknoglobals" "tagliatelle" "tagalign")	Список линтеров, которые должны быть отключены. Golangci-lint позволяет запускать большое количество линтеров, часть из которых не соответствует выбранному стилю написания ПО. Такие линтеры должны быть отключены.
GOLANGCI_LINT_DEPRECATED	()	Список устаревших линтеров. См. документацию golangci-lint.
STATICCHECK_ENABLE	true/false	Включения анализатора staticcheck.
STATICCHECK_CHECKS	"all"	Список проверок для staticcheck. С полным списком можно ознакомиться в <a href="#">документации</a> .
STATICCHECK_FILTER	true/false	Если установлено, результаты staticcheck будут отфильтрованы и отображены только замечания, которые относятся в текущему git diff. В противном случае будут выведены все результаты.
STATICCHECK_SUPPRESSED	"(SA5008),(ST1000),(ST1003), (ST1016),(ST1020),(ST1021), (ST1023)"	Игнорировать сообщения которые содержат ключевые слова из заданного списка. Игнорируются только сообщения вне текущего git diff. Сообщения связанные с текущими изменениями выводятся всегда. Параметр предназначен для фильтрации сообщений относящихся к legacy коду.
STATICCHECK_FAIL	true/false	Если установлено, сборка провалится если staticcheck найдёт какие либо замечания. Замечания относятся только к текущим изменениям.
GO_VET_ENABLE	true/false	Включения анализатора go vet.



Параметр	Значение	Описание
GO_VET_FLAGS	("composites=true")	Дополнительные флаги go vet. Больше информации находится на странице <a href="#">документации go vet</a> .
GO_VET_FAIL	true/false	Если установлено, сборка провалится если go vet найдёт замечания.
LLENCHECK_ENABLE	true/false	Включение проверки ограничения длины строк line-length-limit.
LLENCHECK_TABWIDTH	4	Ширина табуляции.
LLENCHECK_LIMIT	100	Максимально допустимая длина строки.
LLENCHECK_FAIL	true/false	Если установлено, сборка провалится если line-length-limit найдёт слишком длинные строки.
PRECOMMIT_ENABLE	true/false	Включение запуска pre-commit перед сборкой проекта. Pre-commit запускает тесты в соответствии с описанием из .pre-commit-config.yaml.
PRECOMMIT_FAIL	true/false	Если установлено, сборка провалится если pre-commit найдет проблемы.

В дефолтном `[config.ini]` все линтеры включены по умолчанию. При желании либо необходимости, например, чтобы уменьшить время запуска приложения, линтеры можно отключить.

Пример:

```
# Запуск всех линтеров при пересборке проекта
REBUILD_FORCE_LINTERS=true

# Включение и параметры линтера `golangci-lint`
GOLANGCI_LINT_ENABLE=true
GOLANGCI_LINT_LINTERS=("all")
GOLANGCI_LINT_FILTER=true
GOLANGCI_LINT_FAIL=false
GOLANGCI_LINT_SUPRESSED+=("depguard" "gochecknoglobals" "tagliatelle" "tagalign")
GOLANGCI_LINT_DEPRECATED+=()

# Включение и набор проверок для `staticcheck`
STATICCHECK_ENABLE=yes
STATICCHECK_CHECKS="all"
STATICCHECK_FILTER=yes
STATICCHECK_SUPPRESS=("SA5008", (ST1000), (ST1003), (ST1016), (ST1020), (ST1021), (ST1023))
STATICCHECK_FAIL=yes
```

```
# Включение и параметры запуска `go vet`
GO_VET_ENABLE=yes
GO_VET_FLAGS=(-composites=true)
GO_VET_FAIL=yes

# Параметры для `line-length-limit`
LLENCHECK_ENABLE=yes
LLENCHECK_TABWIDTH=4
LLENCHECK_LIMIT=100
LLENCHECK_FAIL=yes

# Включение запуска `pre-commit`
PRECOMMIT_ENABLE=true
PRECOMMIT_FAIL=true
```

### 5.8.2 CLEAN\_GOCACHE

Инструментарий Golang активно использует кеширование промежуточных результатов. Это одна из причин, по которой повторная сборка проекта происходит практически моментально. Тем не менее, в некоторых случаях кеширование может приводить к непредсказуемым эффектам. Например, после кеширования результатов golangci-lint перестает находить некоторые ошибки.

Чтобы избежать такого поведения, если флаг CLEAN\_GOCACHE установлен, скрипты сборки будут производить очистку Go кеша перед запуском линтеров и сборки.

*Удаление кеша приводит к значительному замедлению сборки. Эта функция по умолчанию отключена.*

### 5.8.3 COPY\_FILES и COPY\_CACHE

Переменная COPY\_FILES задает список файлов, которые должны быть загружены на отлаживаемую систему либо наоборот, скачаны с нее. Список COPY\_FILES реализован в виде массива строк. Каждая строка имеет следующий формат:

```
"[?][arch#][:]SOURCE_NAME|[:]TARGET_NAME"
```

SOURCE\_NAME задает имя файла источника, TARGET\_NAME - имя файла назначения. Префикс "?" говорит о том, что копируемый файл может отсутствовать и в этом случае копирование завершится без ошибок. Префикс ":" перед именем файла указывает на то, что путь относится к отлаживаемой, удаленной системе. Отсутствие префикса ":" означает, что будет использован локальный файл. Только у одного из SOURCE\_NAME либо TARGET\_NAME должен содержать префикс ":". Локальное имя файла может быть как полным, начинаться с "/", либо относительным. Относительные пути расцениваются как пути относительно корня проекта. SOURCE\_NAME и TARGET\_NAME должны быть разделены знаком "|". Дополнительный префикс arch# говорит о том, для какой архитектуры предназначен данный файл. Поддерживаются следующие значения: armv7l (ECAM02DM), aarch64 (ECAM03XX r1.0, r2.0). В случае несовпадения arch# с текущей архитектурой файл не будет загружен.

Пример, который позволяет загружать на устройство файлы onvifd.conf, опционально users.digest, а так же .bashrc из домашней директории:

```
# Копирование файлов
```

```
COPY_FILES+=(
    "init/onvifd.conf|:/etc/onvifd.conf"
    "?init/users.digest|:/var/lib/onvifd/users.digest"
    "$HOME/.bashrc|:/root/.bashrc_new"
    "aarch64#init/some_aarch64.file|:/usr/bin/some_aarch64.file"
)
```

Переменная `COPY_CACHE` включает либо отключает кеширование: если файл не изменился с последней загрузки, он не будет загружен повторно. По умолчанию кеширование файлов включено, но отключается для команды <Go: Build Workspace>.

```
# Отключение кеширования файлов
COPY_CACHE=no

# Включение кеширования файлов
COPY_CACHE=yes
```

### 5.8.4 Загрузка файлов через Overlay

Каталог `./.vscode/overlay` может содержать файлы, которые будут автоматически загружены на отлаживаемое устройство во время сборки проекта.

Путь	Назначение
<code>./.vscode/overlay/common</code>	Файлы общие для всех архитектур
<code>./.vscode/overlay/aarch64</code>	Файлы для архитектуры aarch64 (ECAM03)
<code>./.vscode/overlay/armv7l</code>	Файлы для архитектуры armv7l (ECAM02)

Структура каталогов, пути размещения файлов должны соответствовать структуре каталогов на целевой системе.

### 5.8.5 DIRECTORIES\_CREATE

Переменная `DIRECTORIES_CREATE` содержит список директорий, которые должны быть созданы на отлаживаемой системе. Директории создаются перед загрузкой файлов.

```
# Создание директорий
DIRECTORIES_CREATE+=(
    "/var/tmp/new_directory"
)
```

### 5.8.6 DELETE\_FILES

Переменная `DELETE_FILES` содержит список файлов, которые необходимо удалить на целевой системе.

```
# Удаление файлов
```

```
DELETE_FILES+=(  
    "/var/log/nginx/error.log"  
)
```

### 5.8.7 EXECUTE\_COMMANDS

Переменная `EXECUTE_COMMANDS` содержит список команд, которые должны быть выполнены на отлаживаемой системе.

```
"[@]COMMAND[ ...ARGS]"
```

Где, префикс `"@"` запрещает вывод информацию о команде в терминал, а `COMMAND` и `ARGS` - Shell команда и ее аргументы.

Пример команд:

```
# Выполнение команды  
EXECUTE_COMMANDS+=(  
    "ls -l /var/log"  
    "@cat /dev/null"  
)
```

### 5.8.8 SERVICES\_STOP и SERVICES\_START

Переменные `SERVICES_STOP` и `SERVICES_START` содержат имена сервисов `systemd` которые необходимо остановить перед загрузкой файлов и запустить после окончания загрузки непосредственно перед запуском приложения и началом отладки.

```
# Останов сервисов  
SERVICES_STOP+=(  
    "onvifd"  
    "mediad"  
)  
  
# Запуск сервисов  
SERVICES_START+=(  
    "mediad"  
)
```

### 5.8.9 PROCESSES\_STOP и PROCESSES\_START

Переменные `PROCESSES_STOP` и `PROCESSES_START` позволяют останавливать и запускать процессы. При этом для останова процесса необходимо указать его имя, а для запуска — полный путь к исполняемому файлу вместе с аргументами запуска.

*Процессы завершаются через `kill`, что в некоторых случаях может быть небезопасно.*

```
# Останов процессов  
PROCESSES_STOP+=(
```

```
    "mediad"
)

# Запуск процессов
PROCESSES_START+=(
    "/usr/bin/mediad --syslog --fork --print-pid 4 --print-address 6 --session"
)
```

### 5.8.10 CAMERA\_FEATURES\_ON и CAMERA\_FEATURES\_OFF

Список `feature` (фич веб-интерфейса) IP камеры которые должны быть включены или отключены. См. [документацию](#).

Параметры применяются только при полной пересборке проекта через `<Go: Build Workspace>`.

```
# Включение фич камеры
CAMERA_FEATURES_ON+=(
    "audio"
    "videoanalytics"
)

# Выключение фич камеры
CAMERA_FEATURES_OFF+=(
    "somefeature"
)
```

### 5.8.11 TARGET\_IPADDR

Переменная `TARGET_IPADDR` задает IP-адрес или способ получения IP-адреса целевого устройства. Поддерживаются следующие значения:

Значение	Описание
<code>TARGET_IPADDR="X.X.X.X"</code>	Стандартный IPv4 адрес устройства.

Значение	Описание
<code>TARGET_IPADDR="tty"</code>	<p>Получить IP-адрес камеры через адаптер USB COM-порт. Удобно для отладки на камерах с динамическим IP-адресом. Для работы данной функции необходимо, чтобы в системе была установлена утилита <code>picocom</code> :</p> <pre># dnf install picocom</pre> <p>Адаптер USB-COM определяется автоматически. Если в системе несколько USB-адапторов, указать конкретный можно при помощи переменной <code>TTY_PORT</code> , например, <code>TTY_PORT="/dev/ttyUSB0"</code> . Для тонкой настройки параметров связи служит следующая группа переменных:</p> <pre>TTY_PORT="auto" # пустая строка или "auto" - автоматическое определение TTY_SPEED="115200" TTY_PICOCOM="picocom" TTY_DIRECT=false # Не использовать picocom, устанавливать связь через bash TTY_LOGIN="\$TARGET_USER" TTY_PASS="\$TARGET_PASS" TTY_DELAY="200" # milliseconds TTY_RETRY="5" # Количество повторов установки связи</pre> <p>Результат получения IP-адреса кешируется.</p> <p>Флаг <code>TTY_DIRECT</code> позволяет отказаться от использования <code>picocom</code>. Работа с COM-портом производится средствами <code>bash</code>. Не всегда стабильно работает с IP-камерами ECAM02.</p>
<code>TARGET_IPADDR="ecam03-1234567"</code>	<p>Получение IP-адреса по серийному номеру камеры. Поиск IP производится в домене заданном переменной <code>TARGET_DOMAIN</code> (по умолчанию: <code>elvees.com</code>).</p>
<code>TARGET_IPADDR="&lt;MAC-адрес&gt;"</code>	<p>Метод используется если известен MAC-адрес устройства. Устройство должно находиться в локальной сети. Для работы необходим <code>nmap</code> :</p> <pre># dnf install nmap</pre>

### 5.8.12 TARGET\_ARCH и TARGET\_GOCXX

Переменные позволяют задать целевую архитектуру сборки. Список поддерживаемых платформ приведен в описании переменной `GOARCH`. Следует учитывать, что Delve, а как следствие и отладка поддерживается на **небольшом количестве платформ**. Тем не менее, эти переменные позволяют простым способом собрать и задеплоить Go-проект даже на устройства, которые не поддерживают отладку.

Если переменная `TARGET_ARCH` не задана, архитектура целевой платформы будет определена на основании архитектуры текущего buildroot.

Установка имени компилятора для архитектур `"arm"` и `"arm64"` не обязательна: если значение `TARGET_GOCXX` не задано, скрипт автоматически подставит нужное значение.

```
# Сборка для ARM64 (ecam03)
TARGET_ARCH="arm64"
TARGET_GOCXX="aarch64-buildroot-linux-gnu"

# Сборка под ARM (ecam02)
TARGET_ARCH="arm"
TARGET_GOCXX="arm-buildroot-linux-gnueabihf"
```


### 5.8.13 TARGET\_SUPPRESS\_MSSGS

Список `TARGET_SUPPRESS_MSSGS` позволяет фильтровать неинформативные сообщения отлаживаемой программы и отладчика delve. Достаточно частичное совпадение строк. Если какое-либо сообщение не появляется в терминале отладки следует убедиться что текст сообщения не попал под данный фильтр.

```
TARGET_SUPPRESS_MSSGS+=(
    "layer=debugger launching process with args"
    "Unsupported action:"
    "Device.GetDeviceInformation: Unauthorized"
    "Device.GetDeviceInformation failed: Sender not authorized"
    "Device.GetSystemDateAndTime"
    "Device.GetServiceCapabilities"
    "Device.GetHostname"
    "Login.Ping"
    "Warning: Failed to get device serial number"
    "Failed to retrieve light sensor parameters"
)
```

### 5.8.14 GOPROXY

Устанавливает адрес Proxy сервера, который будет использован для загрузки пакетов Go. См.

 [INFRA-1618](#) - Падает сборка Buildroot при клонировании пакетов Go **ЗАКРЫТО**

```
# Установка PROXY
GOPROXY="http://athens.elvees.com,https://proxy.golang.org,direct"
```

### 5.8.15 Дополнительные параметры

В таблице ниже представлены дополнительные конфигурационные флаги тонкой настройки среды сборки.

Флаг	Значение по умолчанию	Описание
USE_RSYNC_METHOD	true	<p>Использовать rsync для загрузки файлов на IP-камеру. Для работы в системе должен быть установлен rsync:</p> <pre>dnf install rsync</pre> <p>Rsync позволяет не загружать файлы целиком, а обновлять существующие на основе встроенных алгоритма diff/crc, что ускоряет запуск приложения после изменений. Если rsync отключен скрипт отладки использует стандартный scp со сжатием файлов gzip/pigz (медленней примерно в двое).</p> <p>Если на IP-камере отсутствует rsync, нужная версия будет установлена автоматически. Поддерживаются платформы: aarch64, armv7l.</p>
USE_RSYNC_BINARY	"rsync"	Команда rsync.
USE_PIGZ_COMPRESSION	true	<p>Использовать многопоточный архиватор pigz для сжатия данных. Pigz является прямой заменой gzip и сохраняя совместимость в формате сжатых данных позволяет добиться как большей скорости, так и лучшей степени сжатия данных.</p> <pre>dnf install pigz</pre> <p>Сайт проекта: <a href="https://zlib.net/pigz/">https://zlib.net/pigz/</a></p>
USE_PIGZ_BINARY	"pigz"	Имя исполняемого файла pigz.
USE_ASYNC_LINTERS	true	Включение параллельного запуска линтеров. USE_ASYNC_LINTERS позволяет запускать линтеры параллельно, уменьшая время запуска приложения. Из минусов такого решения: вывод результатов линтеров происходит в непредсказуемом порядке (можно доработать/исправить, но не критично).
USE_NO_COLORS	false	Отключение вывода в цвете и форматирования вывода. Устанавливать в true если в системный терминал не поддерживает вывод в цвете.
USE_SERVICE_MASKS	false	Дополняет команды запуска и останова сервисов командами mask и unmask.



Флаг	Значение по умолчанию	Описание
USE_OVERLAY_DIR	true	Если флаг USE_OVERLAY_DIR установлен, содержимое папки ".vscode/overlay" будет загружено на отлаживаемую IP-камеру. Структура каталогов в overlay должна повторять структуру каталогов устройства. Загрузка overlay-файлов производится только при полной пересборке проекта. Для загрузки/синхронизации изменяемых файлов следует использовать массив <code>COPY_FILES</code> .
USE_SHELL_TIMEOUT	10	Таймаут выполнения shell-команды, секунд.
USE_GOLANG_TIMEOUT	60	Таймаут запуска инструментов Golang, секунд.
INSTALL_SSH_KEYS	false	Создать пару ключей для SSH и загрузить публичный ключ на отлаживаемое устройство. На данный момент не используются. Связь с IP-камерой устанавливается через логин/пароль задаваемых в переменных TARGET_USER/TARGET_PASS.

## 5.9 Известные проблемы

Не всегда все работает как задумано, а изредка даже так, как не задумано.

1. Иногда, после обновления исходного кода внешней программой, например, через `git fetch` в VSCode перестают работать точки останова. Пересборка/перезапуск приложения проблему не решает. В таких случаях помогает исключительно перезапуск VSCode.
2. В отладчике не всегда видно содержимое локальных переменных. Проблема связана с оптимизацией приложений Go и на данный момент не имеет решения. Проект собирается с флагами, которые запрещают оптимизацию и добавляют отладочную информацию — все в соответствии с рекомендациями документации Go. Единственный вариант посмотреть состояние таких переменных: выводить их значение через `log.Println`. Как правило, в этом случае Go перестаёт оптимизировать такую переменную и она становится видна и в отладчике.
3. В некоторых случаях после обновления прошивки отладка становится невозможна из-за устаревания отпечатка в `$HOME/.ssh/known_hosts`. Лечится удалением соответствующей строки и повторным входом на устройство по SSH. Хотя скрипты отладки работают таким образом, что не добавляют отпечатки в `known_hosts`, но, как показала практика, уже добавленные в других SSH сессиях неправильные отпечатки могут приводить к невозможности установки соединения и запуска отладки.
4. Разночтения в `[config.ini]`. Хотя файл `[config.ini]` по сути является bash скриптом, этот же файл используется VSCode расширением Command Variable для получения IP адреса и номера порта. Проблема разночтения происходит если в `[config.ini]` несколько раз переопределяется `TARGET_IPADDR` либо `TARGET_IPPORT`. Bash использует последнее присвоенное значение, в то время как Command Variable - первое найденное.

## 5.10 TODO

Для поддержки разных проектов Go необходимо вынести некоторые переменные в `[config.ini]`, а сам файл `[config.ini]` перенести из `.vscode` в корень проекта, назвав его, например, `[vcsode_config.ini]` (этот файл можно сделать опциональным).

Список внутренних переменных, которые хотелось бы видеть в составе `[config.ini]`:

`TARGET_BUILD_LAUNCHER`, `TARGET_BUILD_GOFLAGS`, `TARGET_BUILD_LDFLAGS`,  
`TARGET_BIN_SOURCE`, `TARGET_BIN_DESTIN`, `TARGET_EXEC_ARGS`.

Так как для разработки на Go в основном используется VSCode, было бы неплохо внесение каталога `.vscode` и сопутствующих/промежуточных файлов в `.gitignore`.

Для упрощения разработки где-то в отдельном `.ini` файле можно вести список ASSET-ов и соответствующих им IP адресов, что должно упростить переключение между несколькими устройствами.