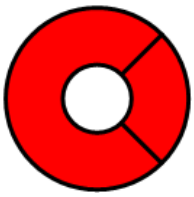
General Embedded C Library Manual

General Embedded C Libraries

An open source software library



Document status: ***Draft*** | *Preliminary* | *Release*

Github link	https://github.com/GeneralEmbeddedCLibraries
Developers	Žiga Miklošič
E-mail	ziga.miklosic@gmail.com

Revision History

Revision	Author	Date	Changes/Remarks
V1.0	*Miklošič Ž.*	*16.10.2021*	*Initial document*

Contents

1 About

Scope of that document is to describe usage and internal architecture/concept used in more complex modules that are part of larger General Embedded C Library package. Additional document will give general guidance how the modules shall be used.

2 Workflow

2.1 Module structure

Each module has identical directory structure as shown from picture bellow:

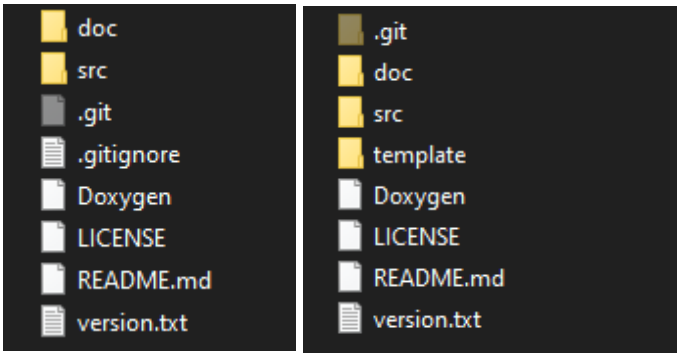
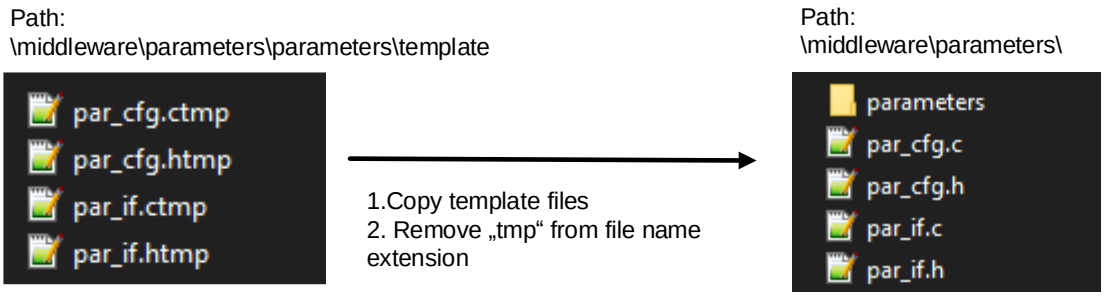


Figure 1 Top directory of each module. Simple module (left) and complex module (right)

More complex modules has also **template** folder where source code template files are prepared for user. When module has template folder its content must be copied and paste to one directory above the module directory. After coping it is mandatory to remove “tmp” string from file name extension.

E.g.:



Template files are used for module configuration for user application needs or to adapt module to specific platform needs.

2.2 Module usage

Each module usage is described in its repository main page.

2.3 Module dependencies

Each module dependencies are described in its repository main page.

2.4 Module API

Each module has Doxygen generated API description inside module **doc** directory.

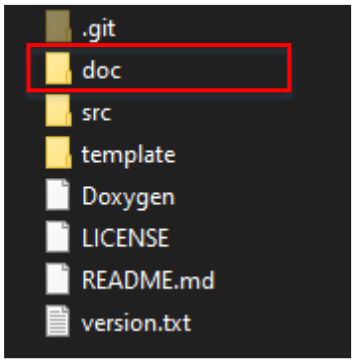


Figure 2 Module Doxygen documentation location

2.5 Git submodule

Each module shall be added to main project as git submodule.

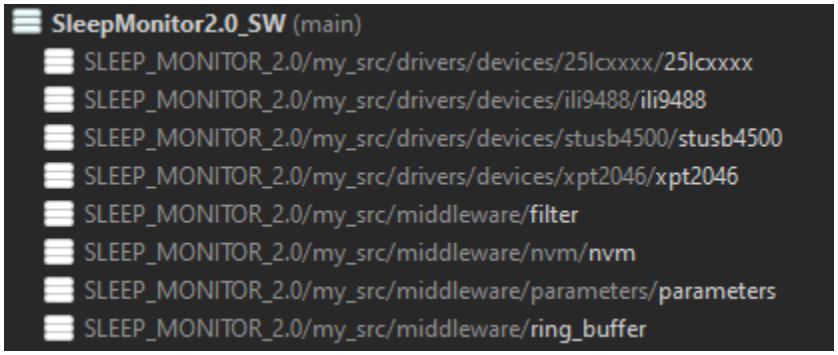


Figure 3 Using submodules with SmartGit git client tool

3 Parameters

3.1 Features

- Parameter configuration via single table
- Supported up to 8 data types: u8, i8, u16, i16, u32, i32, f32, string
- Each parameter has unique ID
- Access type configurable: Read only or Write-Read
- Storing & loading parameters from/to NVM on the fly
- Support of single parameter store (if NVM layer supports it)
- Back-compatibility between devices that have different parameter tables

Link to github: https://github.com/GeneralEmbeddedCLibraries/parameters

3.2 Exceptions

3.2.1 Loading parameters from NVM

In case parameter NVM header is corrupted all parameters will be set to default and old parameters stored inside NVM will be rewritten with new parameters. If that scenario occurs parameters stored inside NVM will not be compatible with old software.

In case single parameter NVM object is corrupted then complete parameters are cleared and completely rewritten with current definition of parameter table. If that scenario occurs parameters stored inside NVM will not be compatible with old software.

3.4 Persistent parameters

3.4.1 NVM structure

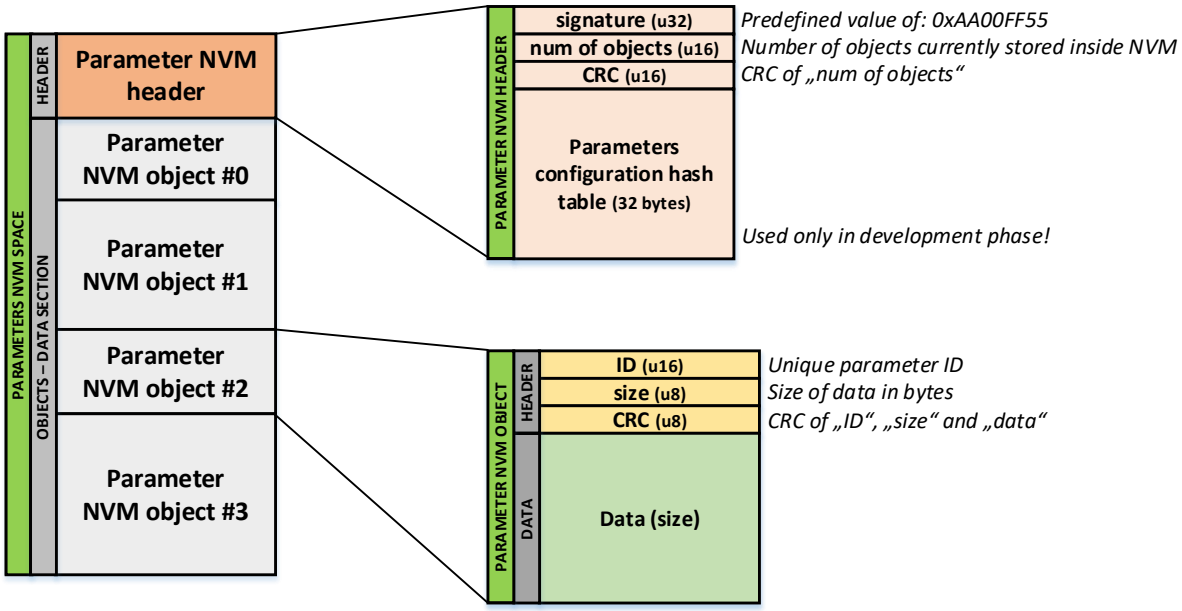


Figure 4 Parameter NVM memory layout

Parameter NVM header “num of objects” value is only change during initialization and it only increases during changes of parameters between different software.

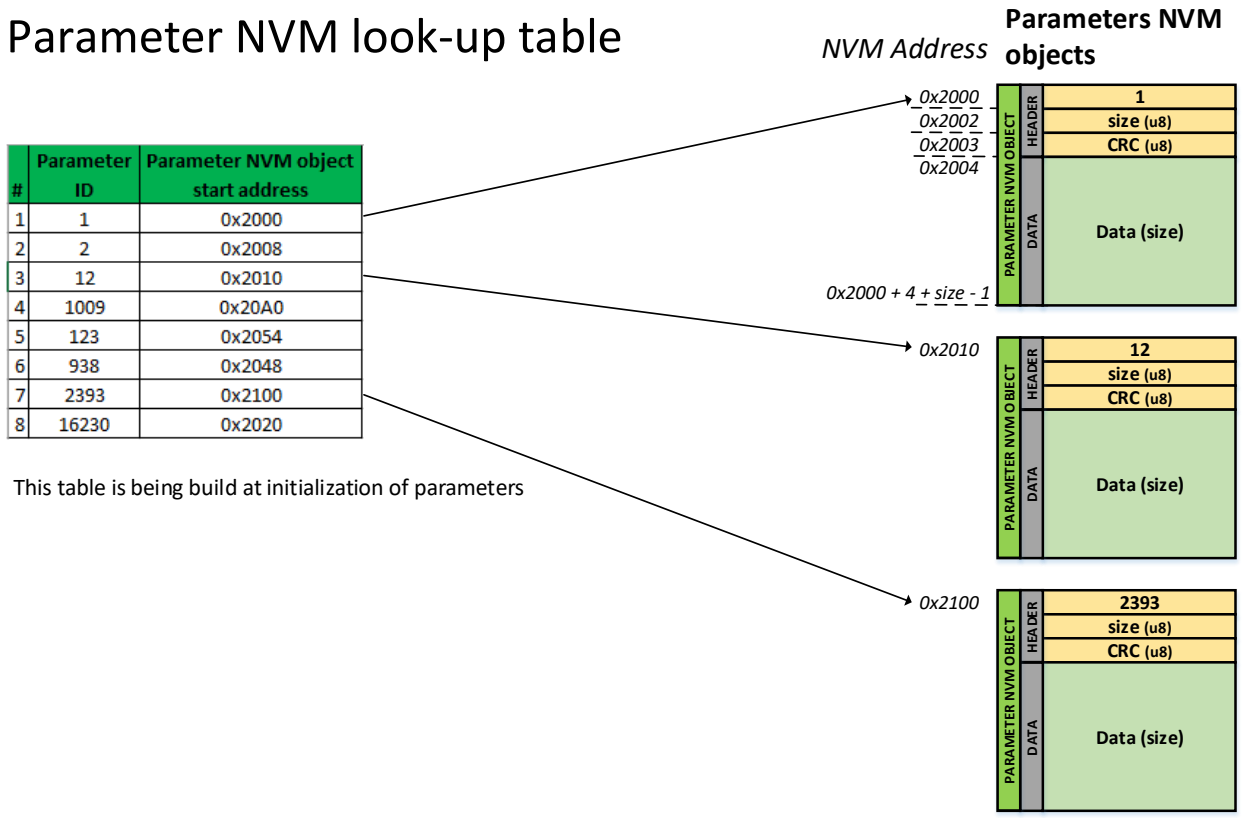
In case new software has new parameter (new ID) then “num of objects” will be increment for number of new persistent parameters. Nevertheless if some parameters are being removed and one parameter is being added the value “num of objects” will be incremented by one (as one newly created parameter).

Parameter NVM header is only being erased/resets on CRC failure, corrupted signature or any NVM interface error. Additional if table hash check functionality is being enabled, then NVM header can also be erased on mismatch of hashes.

3.5.1 NVM look-up table

Purpose of NVM look-up table is linkage between parameter ID and its start location in NVM space. When storing single parameter to NVM this information is mandatory to know.

Parameter NVM look-up table contains only data for “live” persistent parameters.



#	Parameter ID	Parameter NVM object start address
1	1	0x2000
2	2	0x2008
3	12	0x2010
4	1009	0x20A0
5	123	0x2054
6	938	0x2048
7	2393	0x2100
8	16230	0x2020

This table is being build at initialization of parameters

3.6.1 Loading persistent parameters from NVM at start-up

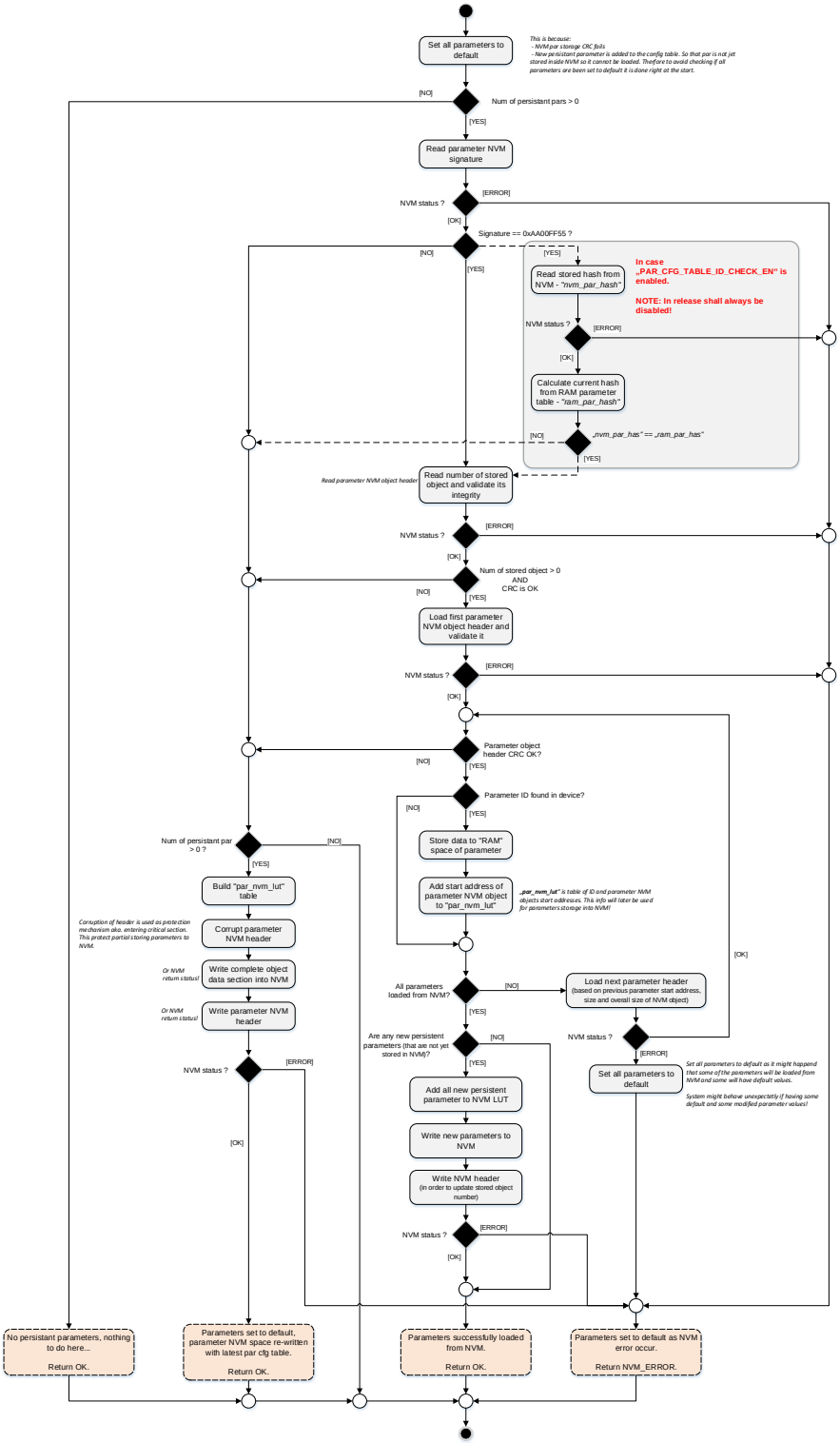


Figure 5 Loading parameter at start-up

3.6.2 Storing all persistent parameter to NVM

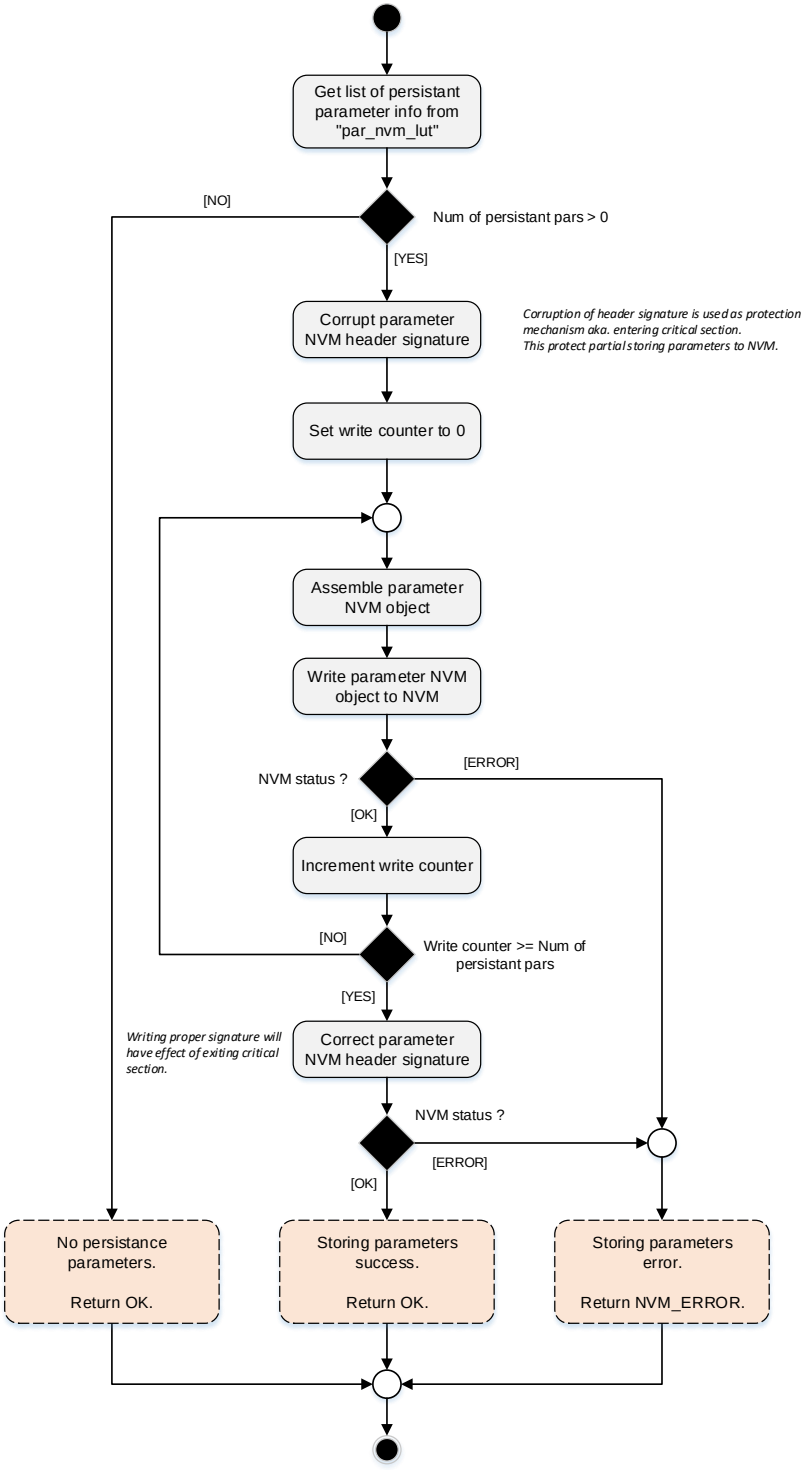


Figure 6 Storing persistent parameter to NVM during runtime procedure

3.6.3 Storing single parameter

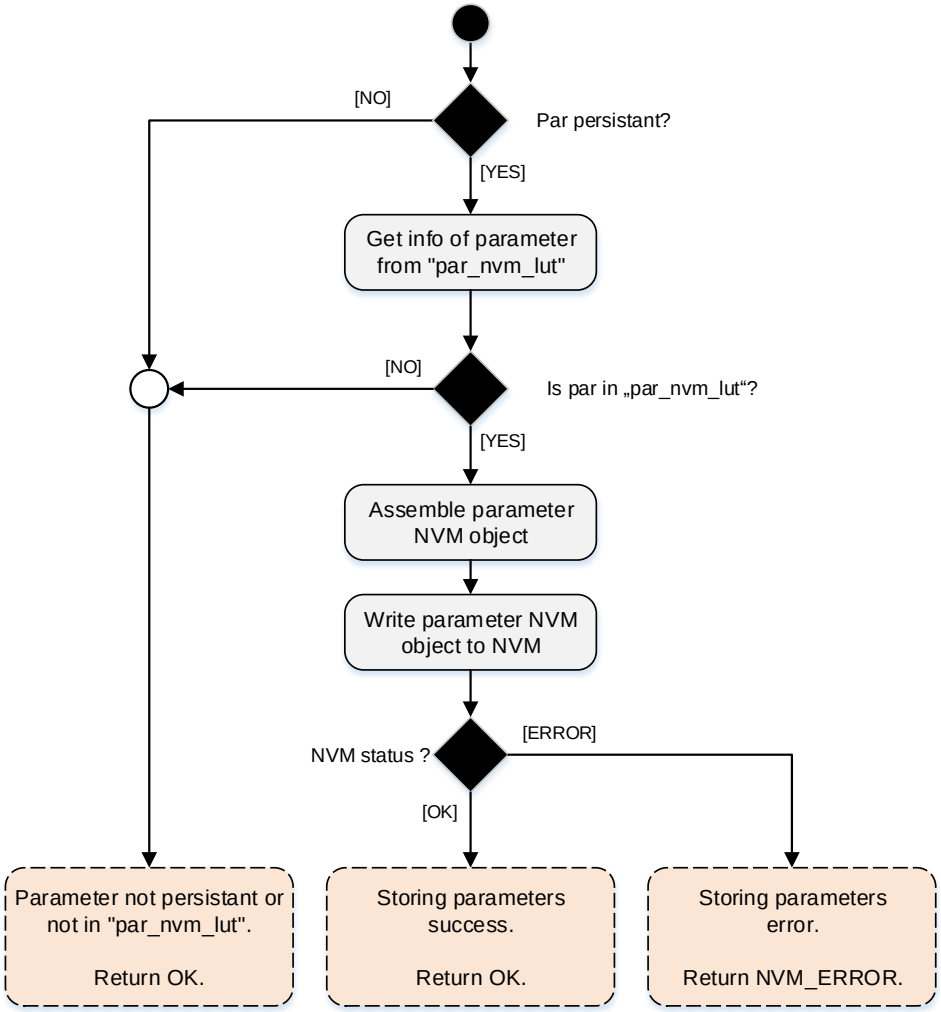


Figure 7 Storing single parameter to NVM

3.6.5 Usage

Parameter can be removed or added to configuration table as long as new parameters doesn't have equal ID as old parameters. This is only constrain for persistent parameters. In case you create a new parameter with same ID as old parameters has had then when loading value from NVM type error can occur and loaded value into new parameter will be junk.

Good practice is to comment out old/not needed parameters just to make a track of used ID numbers. As shown in picture bellow from parameter configuration table:

```
// NOT USED ANYMORE
// Meaning that IDs 1234, 11111 and 65000 are taken and shall not be used anymore!
//{ .id = 1234,     .name = "Test F32",     .min.f32 = -5.0f,       .max.f32 = 5.0f,        .def.f32 = 0.0f,        .unit = "V",        .type = ePAR_TYPE_F32,  .access = ePAR_ACCESS_RW,
//{ .id = 11111,    .name = "Test I8",      .min.i8 = INT8_MIN,     .max.i8 = INT8_MAX,     .def.i8 = -1,           .unit = NULL,       .type = ePAR_TYPE_I8,   .access = ePAR_ACCESS_RW,
//{ .id = 65000,    .name = "Test U16",     .min.u16 = 0,           .max.u16 = 13000,       .def.u16 = 12,          .unit = NULL,       .type = ePAR_TYPE_U16,  .access = ePAR_ACCESS_RW,


{   .id = 50,       .name = "Test I16",     .min.i16 = -500,        .max.i16 = 3000,        .def.i16 = -12,         .unit = NULL,       .type = ePAR_TYPE_I16,  .access = ePAR_ACCESS_RW,
{   .id = 51,       .name = "Test I16",     .min.i16 = -500,        .max.i16 = 3000,        .def.i16 = -12,         .unit = NULL,       .type = ePAR_TYPE_I16,  .access = ePAR_ACCESS_RW,
{   .id = 52,       .name = "Test I16",     .min.i16 = -500,        .max.i16 = 3000,        .def.i16 = -12,         .unit = NULL,       .type = ePAR_TYPE_I16,  .access = ePAR_ACCESS_RW,
{   .id = 355,      .name = "Test I162",        .min.i16 = -500,        .max.i16 = 3000,        .def.i16 = -12,         .unit = NULL,       .type = ePAR_TYPE_I16,  .access = ePAR_ACCESS_R
{   .id = 356,      .name = "Test I163",        .min.i16 = -500,        .max.i16 = 3000,        .def.i16 = -12,         .unit = NULL,       .type = ePAR_TYPE_I16,  .access = ePAR_ACCESS_R
```

Figure 8 Making track of used parameter IDs

If doing so device will be back-compatible with older software.

3.7 Test procedure

- Generate test parameter configuration table
- Corrupt signature and make reset
- Store some non-default values to parameters and save all
- Reset device and check if values are loaded
- Add new parameters to configuration table
- Change values to new parameters to non-default and save all
- Reset device and check if all values are correctly loaded
- Comment out some of the existing parameters
- Check if loaded correctly, change values and save all
- Remove comment from old parameters
- Check if loaded correctly, change values and save all