CHECKSUM CALCULATION FOR VERIFYING IMAGE INTEGRITY

This page contains information about checksum calculation:

For more information, see also *The IAR ELF Tool—ielftool*.

Briefly About Checksum Calculation

You can use a checksum to verify that the image is the same at runtime as when the image's original checksum was generated. In other words, to verify that image has not been corrupted.

This works as follows:

- You need an initial checksum.

  You can either use the IAR ELF Tool—`ielftool`—to generate an initial checksum or you might have a third-party checksum available.
- You must generate a second checksum during runtime.

  You can either add specific code to your application source code for calculating a checksum during runtime or you can use some dedicated hardware on your device for calculating a checksum during runtime.
- You must add specific code to your application source code for comparing the two checksums and take an appropriate action.

  If the two checksums have been calculated in the same way, and if there are no errors in the image, the checksums should be identical. If not, you should first suspect that the two checksums were not generated in the same way.

No matter which solutions you use for generating the two checksum, you must make sure that both checksums are calculated *in the exact same way*. If you use `ielftool` for the initial checksum and use a software-based calculation during runtime, you have full control of the generation for both checksums. However, if you are using a third-party checksum for the initial checksum or some hardware support for the checksum calculation during runtime, there might be additional requirements that you must consider.

For the two checksums, there are some choices that you must always consider and there are some choices to make only if there are additional requirements. Still, all of the details must be the same for both checksums.

Details always to consider:

- *Checksum range*

  The memory range (or ranges) that you want to verify by means of checksums. Typically, you might want to calculate a checksum for all memory. However, you might want to calculate a checksum only for specific ranges, for example, if you have some parts that are certified or if you have some very critical parts. Remember that:
  - It is OK to have several ranges for one checksum.

- The checksum must be calculated from the lowest to the highest address for every memory range.
- Each memory range must be verified in the same order as defined (for example, `0x100-0x1FF,0x400-0x4FF` is not the same as `0x400-0x4FF,0x100-0x1FF`).
- If several checksums are used, you should place them in sections with unique names and use unique symbol names.
- A checksum should never be calculated on a memory range that contains a checksum or a software breakpoint.

- *Algorithm and size of checksum*

  You should consider which algorithm is most suitable in your case. There are two basic choices, Sum (a simple arithmetic algorithm) or CRC (which is the most commonly used algorithm). For CRC there are different sizes to choose for the checksum, 2, 4, or 8 bytes where the predefined polynomials are wide enough to suit the size, for more error detecting power. The predefined polynomials work well for most, but possibly not for all data sets. If not, you can specify your own size and polynomial. However, if you want a decent error detecting mechanism, use the predefined CRC algorithm for your checksum size, typically CRC16 or CRC32.

  For more information about selecting an appropriate polynomial for data sets with non-uniform distribution, see for example section 3.5.3 in *Tannenbaum, A.S., "Computer Networks", Prentice Hall 1981, ISBN: 0131646990*.

- *Fill*

  Every byte in the checksum range must have a well-defined value before the checksum can be calculated. Typically, bytes with unknown values are any *pad bytes* that have been added for alignment. This means that you must specify which fill pattern to be used during calculation.

- *Initial value*

  The checksum must always have an explicit initial value.

In addition to these mandatory details, there might be other details to consider. Typically, this might happen when you have a third-party checksum, you want the checksum be compliant with the Rocksoft™ checksum model, or when you use hardware support for generating a checksum during runtime. `ielftool` provides support for also controlling alignment, complement, bit order, byte order within words, and checksum unit size.

Calculating And Verifying A Checksum

In this example procedure, a checksum is calculated for ROM memory from `0x8002` up to `0x8FFF` and the 2-byte calculated checksum is placed at `0x8000`.

1. If you are using `ielftool` from the command line, you must first allocate a place for the calculated checksum. You can do this in two ways; by creating a global C/C++ or assembler constant symbol with a proper size, residing in a specific section (in this example `.checksum`), or by using the linker option `--place_holder`.

For example, to allocate a 2-byte space for the symbol `__checksum` in the section `.checksum`, with alignment 4, specify:

```
--place_holder __checksum,2,.checksum,4
```

Note:

The `.checksum` section will only be included in your application if the section appears to be needed. If the checksum is not needed by the application itself, you can use the linker option `--keep=__checksum` or the linker directive `keep` to force the section to be included.

Note:

If you are using the IDE, the `__checksum`, `__checksum_begin`, and `__checksum_end` symbols, and the `.checksum` section are automatically allocated when you calculate the checksum.

2. To control the placement of the `.checksum` section, you must modify the linker configuration file. For

example, it can look like this (note the handling of the block `CHECKSUM`):

```
define block CHECKSUM    { ro section .checksum };
place in ROM_region  { ro, first block CHECKSUM };
```

Note:

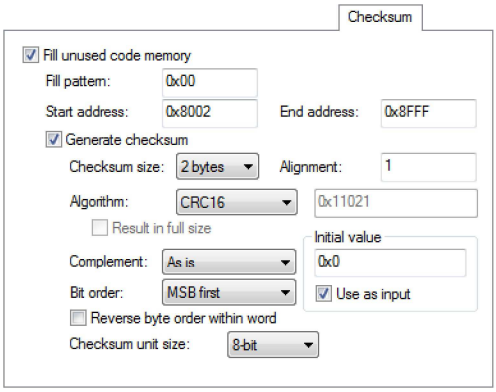
It is possible to omit this step, but in that case the `.checksum` section will automatically be placed with other read-only data.

3 When configuring `ielftool` to calculate a checksum, there are some choices to make:

- Choose which checksum algorithm you want to use. In this example, the CRC16 algorithm is used.
- Specify the memory range (or ranges) for which the checksum should be calculated. If you instead want to calculate a checksum for several ranges, you must specify the ranges using the `--checksum` option. If you are using the IDE, you specify this option on the **Extra Options** page.
- Specify a fill pattern—typically `0xFF` or `0x00`—for bytes with unknown values. The fill pattern will be used in all checksum ranges.

For more information, see *Briefly about checksum calculation*.

To run `ielftool` from the IDE, choose **Project>Options>Linker>Checksum** and make your settings, for example:



In the simplest case, you can ignore (or leave with default settings) these options: **Complement**, **Bit order**, **Reverse byte order within word**, and **Checksum unit size**.

To run `ielftool` from the command line, specify the command, for example, like this:

```
ielftool --fill=0x00;0x8002-0x8FFF
--checksum=__checksum:2,crc16;0x8002-0x8FFF sourceFile.out destinationFile.out
```

Note that `ielftool` needs an unstripped input ELF image. If you use the linker option `--strip`, remove it and use the `ielftool`option `--strip` instead.

The checksum will be created when you build your project and will be automatically placed in the symbol `__checksum` in the section `.checksum`.

4 If you are using the IDE and want to specify several ranges, choose **Project>Options>Linker>Extra options**. Use the `--checksum` option and specify the ranges, for example like this:

```
ielftool --fill;0x0-0x3FF,0x8002-0x8FFF
--checksum=__checksum:2,crc16;0x0-0x3FF,0x8002-0x8FFF
```

5 Add a function for checksum calculation to your source code. Make sure that the function uses the same algorithm and settings as for the checksum calculated by `ielftool`. For example, a slow variant of the

crc16 algorithm but with small memory footprint (in contrast to the fast variant that uses more memory):

```
unsigned short SmallCrc16(unsigned short sum,

      unsigned char *p,
p,


    unsigned int le
ed int len)
--)
  {
unsigned char byte = *(p++);
ned char byte = *(p++);
    for (i = 0; i <
= 0; i < 8; ++i)
    {
  unsigned long oSum = sum;
 sum;
 <<= 1;
  if (byte &
(byte & 0x80)
 & 0x80)


sum |= 1;
 sum;
```

6 Make sure that your application also contains a call to the function that calculates the checksum, compares the two checksums, and takes appropriate action if the checksum values do not match. This code gives an example of how the checksum can be calculated for your application and to be compared with the `ielftool` generated checksum:

ou can find the source code for this checksum algorithm in the arm\src\linker directory of your product

of how the checksum can be calculated for your
ated for your
for your application and to be compared with th
pared with the ielftool generated checksum:
um:

```
*/
ls */
extern unsigned short const __checksum;
xtern unsigned short const __checksum;
nst __checksum;
_checksum;
st __checksum_start;
st __checksum_end;
checksum_end;
Checksum()
signed short calc = 0;
r  zeros[2] = {0, 0};
 the checksum algorithm */
rc16(0,
0,
tart,
har *) &__checksum_end -
) &__checksum_end -
            ((unsigned char *) &__checksum_s
 *) &__checksum_start)+1));
__checksum_start)+1));
Fill the end of the byte sequence with zeros. */
equence with zeros. */
ce with zeros. */
th zeros. */
Crc16(calc, zeros,
```

7 You can now build your application project and then download it.

```
(calc, zeros, 2);
  {
```

Troubleshooting checksum calculation:

Troubleshooting Checksum Calculation

If the two checksums do not match, there are several possible causes. These are some troubleshooting hints:

- If possible, start with a small example when trying to get the checksums to match.
- Verify that the exact memory range or ranges are used in both checksum calculations.

  To help you do this, `ielftool` produces useful information on `stdout` about the exact addresses that were used and the order in which they were accessed.
- Make sure that the checksum symbol is excluded from the checksum calculation.

  Compare the checksum placement with the checksum range and make sure they do not overlap. You can find information in the **Build** message window after `ielftool` has generated a checksum.
- Verify that the checksum calculations use the same polynomial.

  Note that for an *n*-bit polynomial, the *n*:th bit is always considered to be set. For a 16-bit polynomial (for example, CRC16) this means that `0x11021` is the same as `0x1021`.
- Verify that the bits in the bytes are processed in the same order in both checksum calculations, from the least to the most significant bit or the other way around. You control this with the **Bit order** option (or from the command line, the `-m` parameter of the `--checksum` option).
- If you are using the small variant of CRC, check whether you need to feed additional bytes into the algorithm.

  The number of zeros to add at the end of the byte sequence must match the size of the checksum, in other words, two zeros for a 1-byte checksum, four zeros for a 2-byte checksum, and eight bytes for a 4-byte checksum.
- By default, a symbol that you have allocated in memory by using the linker option `--place_holder` is considered by C-SPY to be of the type `int`. If the size of the checksum is different than the size of an `int`, you can change the display format of the checksum symbol to match its size.

  In the C-SPY **Watch** window, select the symbol and choose **Show As** from the context menu. Choose the display format that matches the size of the checksum symbol.