MPLAB® XC8 C Compiler Version 2.39 Release Notes for AVR® MCU

This document contains important information relating to the MPLAB XC8 C compiler when targeting Microchip AVR devices. Please read it before running this software.

See the MPLAB XC8 C compiler release notes for PIC document if you are using the compiler for 8-bit PIC devices.

1. Overview

1.1. Introduction

This release of the Microchip MPLAB® XC8 C compiler is a functional safety compiler, based on the v2.36 release of this compiler and which now supports the Network Server License.

1.2. Release Date

The official release date of this compiler version is the 27 January 2022.

1.3. Previous Version

The previous MPLAB XC8 C compiler version was 2.36, released 25 January 2022.

1.4. Functional Safety Manual

A Functional Safety Manual for the MPLAB XC compilers is available in the documentation package when you purchase a functional safety license.

1.5. Component Licenses and Versions

The MPLAB® XC8 C Compiler for AVR MCUs tools are written and distributed under the GNU General Public License (GPL) which means that its source code is freely distributed and available to the public.

The source code for tools under the GNU GPL may be downloaded separately from Microchip's website. You may read the GNU GPL in the file named `license.txt` located the `avr/doc` subdirectory of your install directory. A general discussion of principles underlying the GPL may be found here.

Support code provided for the header files, linker scripts, and runtime libraries are proprietary code and not covered under the GPL.

This compiler is an implementation of GCC version 5.4.0, binutils version 2.26, and uses avr-libc version 2.0.0.

1.6. System Requirements

The MPLAB XC8 C compiler and the licensing software it utilizes are available for a variety of operating systems, including 64-bit versions of the following: Professional editions of Microsoft Windows 10; Ubuntu 18.04; and macOS 10.15.5. Binaries for Windows have been code-signed. Binaries for macOS have been code-signed and notarized.

If you are running a network license server, only computers with operating systems supported by the compilers may be used to host the license server. As of xclm version 2.0, the network license server can be installed on a Microsoft Windows Server platform, but the license server does not need to run on a server version of the operating system.

1.7. Devices Supported

This compiler supports all 8-bit AVR MCU devices known at the time of release. See `avr_chipinfo.html` (in the compiler's `doc` directory) for a list of all supported devices. These files also list configuration bit settings for each device.

1.8. Editions and License Upgrades

The MPLAB XC8 compiler can be activated as a licensed (PRO) or unlicensed (Free) product. You need to purchase an activation key to license your compiler. A license allows for a higher level of optimization compared to the Free product. An unlicensed compiler can be operated indefinitely without a license.

An MPLAB XC8 Functional Safety compiler must be activated with a functional safety license purchased from Microchip. The compiler will not operate without this license. Once activated, you can select any optimization level and use all the compiler features. This release of the MPLAB XC Functional Safety

Compiler supports the Network Server License.

See the *Installing and Licensing MPLAB XC C Compilers* (DS50002059) document for information on license types and installation of the compiler with a license.

1.9. Installation and Activation

See also the Migration Issues and Limitations sections for important information about the latest license manager included with this compiler.

If using MPLAB IDE, be sure to install the latest MPLAB X IDE version 5.0 or later before installing this tool. Quit the IDE before installing the compiler. Run the `.exe` (Windows), `.run` (Linux) or `.app` (macOS) compiler installer application, e.g. `XC8-1.00.11403-windows.exe` and follow the directions on the screen. The default installation directory is recommended. If you are using Linux, you must install the compiler using a terminal and from a root account. Install using a macOS account with administrator privileges.

Activation is now carried out separately to installation. See the document *License Manager for MPLAB® XC C Compilers* (DS52059) for more information.

If you choose to run the compiler under the evaluation license, you will now get a warning during compilation when you are within 14 days of the end of your evaluation period. The same warning is issued if you are within 14 days of the end of your HPA subscription.

The XC Network License Server is a separate installer and is not included in the single-user compiler installer.

The XC License Manager now supports roaming of floating network licenses. Aimed at mobile users, this feature allows a floating license to go off network for a short period of time. Using this feature, you can disconnect from the network and still use your MPLAB XC compiler. See the doc folder of the XCLM install for more on this feature.
MPLAB X IDE includes a Licenses window (Tools > Licenses) to visually manage roaming.

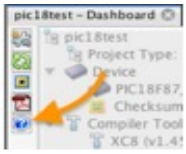
1.9.1. Resolving Installation Issues

If you experience difficulties installing the compiler under any of the Windows operating systems, try the following suggestions.

- Run the install as an administrator.
- Set the permissions of the installer application to 'Full control'. (Right-click the file, select Properties, Security tab, select user, edit.)
- Set permissions of the temp folder to ‘Full Control’.

To determine the location of the temp folder, type `%temp%` into the Run command (Windows logo key + R). This will open a file explorer dialog showing that directory and will allow you to determine the path of that folder.

1.10. Compiler Documentation



The compiler's user's guides can be opened from the HTML page that opens in your browser when clicking the blue help button in MPLAB X IDE dashboard, as indicated in the screenshot.

If you are building for 8-bit AVR targets, the MPLAB® XC8 C Compiler User's Guide for AVR® MCU contains information on those compiler options and features that are applicable to this architecture.

1.11. Customer Support

Microchip welcomes bug reports, suggestions or comments regarding this compiler version. Please direct any bug reports or feature requests via the Support System.

2. Documentation Updates

For on-line and up-to-date versions of MPLAB XC8 documentation, please visit Microchip's Online Technical Documentation website.

New or updated AVR documentation in this release:

- Installing and Licensing MPLAB® XC C Compilers (DS50002059) revision L

The *Microchip Unified Standard Library Reference Guide* describes the behavior of and interface to the functions defined by the Microchip Unified Standard Library, as well as the intended use of the library types and macros. Some of this information was formerly contained in the *MPLAB® XC8 C Compiler User's Guide for AVR® MCU*. Device-specific library information is still contained in this compiler guide.

The *Hexmate User's Guide* has been included in the `docs` directory in this release. This guide is intended for those running Hexmate as a stand-alone application.

The following sections provide corrections and additional information to that found in the user's guides shipped with the compiler.

2.1. smart-io-format Option

The `-msmart-io-format=`*`fmt`* option, where *`fmt`* is a string containing formatted IO conversion specifications, notifies the compiler that the listed specifications are used by smart IO functions.

To reduce code size, the compiler customizes library code associated with the print and scan families of smart IO functions, based on the conversion specifications present in the format strings collated across all calls to these functions. This feature is fully automatic and cannot be disabled.

In some situations, the compiler is unable to determine usage information from the formatted IO function call. If the `-msmart-io-format=`*`fmt`* option has been used, the required conversion specifications for these functions are obtained from the *`fmt`* string; otherwise, the compiler makes no assumptions about how the functions are used and ensures that fully functional formatted IO functions are linked into the final program image.

For example, consider the following calls to smart IO functions.

```
vscanf("%d:%li", va_list1);
vprintf("%-s%d", va_list2);
vprintf(fmt1, va_list3);        // ambiguous usage
vscanf(fmt2, va_list4);         // ambiguous usage
```

When processing the last two calls, the compiler cannot deduce any usage information from either the format strings, nor the arguments. In these instances, the `-msmart-io-format` option can be used and will potentially allow more optimal formatted IO functions to be generated, thus reducing the program's code size. For example, if the format strings pointed to by `fmt1` and `fmt2` collectively use only the `"%d"`, `"%i"` and `"%s"` conversion specifiers, the `-msmart-io-format=fmt="%d%i%s"` option should be issued.

The *fmt* string may contain any valid conversion specification, including flags and modifiers (for example "%-13.9ls"), and should reflect exactly those used by the functions whose usage is ambiguous. Failure to include a specification in the *fmt* argument where it has been used by the formatted IO functions might result in code failure.

If *fmt* is an empty string or or contain no discernible conversion specifications, a warning shall be issued and fully functional formatted IO functions are linked into the final program image.

This option may be used multiple times on the command line. The conversion specifications used with each option are accumulated.

2.2. omit-frame-pointer Option

The -fomit-frame-pointer option instructs the compiler to directly use the stack pointer to access objects on the stack and, if possible, omit code that saves, initializes, and restores the frame register. It is enabled automatically at all non-zero optimization levels.

Negating the option, using -fno-omit-frame-pointer, might assist debugging optimized code; however, this option does not guarantee that the frame pointer will always be used.

2.3. unroll-loops Options

The -funroll-loops and -funroll-all-loops options control speed-orientated optimizations that attempt to remove branching delays in loops. Unrolled loops typically increase the execution speed of the generated code, at the expense of larger code size.

The -funroll-loops option unrolls loops where the number of iterations can be determined at compile time or when code enters the loop. The -funroll-all-loops option is more aggressive, unrolling all loops, even when the number of iterations is unknown. It is typically less effective at improving execution speed than the -funroll-loops option.

2.4. fat-lto-objects Option

The -ffat-lto-objects option requests that the compiler generate fat object files, which contain both object code and GIMPLE (one of GCC's internal representations), written to unique ELF sections. Such objects files are useful for library code that could be linked with projects that do and do not use the standard link-time optimizer, controlled by the -flto option.

The -fno-fat-lto-objects form of this option, which is the default if no option is specified, suppresses the inclusion of the object code into object files, resulting in faster builds. However, such object files must always be linked using the standard link-time optimizer.

2.5. lto-partition Option

The -flto-partition=*algorithm* option controls the algorithm used to partition object files when running the link-time optimizer. The argument none disables partitioning entirely and executes the link-time optimization step directly from the whole program analysis (WPA) phase. This mode of operation will produce the most optimal results, at the expense of larger compiler memory requirements and longer build times, although this is unlikely to be an issue with small programs. Partitioning the object files can improve build performance. The argument one specifies that exactly one partition should be used, and the argument 1to1 specifies partitioning that mirrors that dictated by the original source files. The default argument is balanced, which specifies partitioning into equally sized chunks, when possible.

2.6. Addition to Section 3.6.11 Mapped Linker Options

The `-Wl,--section-start=`*`section=addr`* is missing from the table of commonly used linker options, accessible using the `-Wl` compiler driver option. This option allows placement of custom-named sections at the specified address. It cannot be used to place standard sections, like (`.data`, `.bss`, `.text`), which must be placed using a `-Wl,-T` option.

2.7. Amendment to Section 4.14.2 Changing and Linking the Allocated Section

Note that contrary to information contained in this section of the User's guide, changes made to the compiler in this release now mean that custom sections can be linked using the `-Wl,--section-start=`*`section=addr`* option and without having to modify the linker script.

3. What's New

The following are new AVR-target features the compiler now supports. The version number in the subheadings indicates the first compiler version to support the features that follow.

3.1. Version 2.39 (Functional Safety Release)

Network Server License This release of the MPLAB XC8 Functional Safety Compiler supports the Network Server License.

3.2. Version 2.36

None.

3.3. Version 2.35

New device support Support is available for the following AVR parts: ATTINY3224, ATTINY3226, ATTINY3227, AVR64DD14, AVR64DD20, AVR64DD28, and AVR64DD32.

Improved context switching The new `-mcall-isr-prologues` option changes how interrupt functions save registers on entry and how those registers are restored when the interrupt routine terminates. It works in a similar way to the `-mcall-prologues` option, but only affects interrupt functions (ISRs).

Even more improved context switching The new `-mgas-isr-prologues` option controls the context switch code generated for small interrupt service routines. When enabled, this feature will have the assembler scan the ISR for register usage and only save these used registers if required.

Configurable flash mapping Some devices in the AVR DA and AVR DB family have an SFR (e.g. FLMAP) that specifies which 32k section of program memory will be mapped into the data memory. The new `-mconst-data-in-config-mapped-progmem` option can be used to have the linker place all `const`-qualified data in one 32k section and automatically initialize the relevant SFR register to ensure that this data is mapped into the data memory space, where it will be accessed more effectively.

Microchip Unified Standard Libraries All MPLAB XC compilers will share a Microchip Unified Standard Library, which is now available with this release of MPLAB XC8. The *MPLAB® XC8 C Compiler User's Guide for AVR® MCU* no longer includes the documentation for these standard functions. This information can now be found in the *Microchip Unified Standard Library Reference Guide*. Note that some functionality previously defined by avr-libc is no longer available. (See Library functionality.)

Smart IO As part of the new unified libraries, IO functions in the printf and scanf families are now custom-

generated on each build, based on how these functions are used in the program. This can substantially reduce the resources used by a program.

Smart IO assistance option When analyzing calls to smart IO functions (such as `printf()` or `scanf()`), the compiler cannot always determine from the format string or infer from the arguments those conversion specifiers required by the call. Previously, the compiler would always make no assumptions and ensure that fully functional IO functions were linked into the final program image. A new `-msmart-io-format=`*`fmt`* option has been added so that the compiler can instead be informed by the user of the conversion specifiers used by smart IO functions whose usage is ambiguous, preventing excessively long IO routines from being linked. (See smart-io-format Option for more details.)

Placing custom sections Previously, the `-Wl,--section-start` option only placed the specified section at the requested address when the linker script defined an output section with the same name. When that was not the case, the section was placed at an address chosen by the linker and the option was essentially ignored. Now the option will be honoured for all custom sections, even if the linker script does not define the section. Note, however, that for standard sections, such `.text`, `.bss` or `.data`, the best fit allocator will still have complete control over their placement, and the option will have no effect. Use the `-Wl,-T`*`section=addr`* option, as described in the user's guide.

3.4. Version 2.32

Stack Guidance Available with a PRO compiler license, the compiler's stack guidance feature can be used to estimate the maximum depth of any stack used by a program. It constructs and analyzes the call graph of a program, determines the stack usage of each function, and produces a report, from which the depth of stacks used by the program can be inferred.
This feature is enabled through the `-mchp-stack-usage` command-line option. A summary of stack usage is printed after execution. A detailed stack report is available in the map file, which can be requested in the usual way.

New device support Support is available for the following AVR parts: ATTINY427, ATTINY424, ATTINY426, ATTINY827, ATTINY824, ATTINY826, AVR32DB32, AVR64DB48, AVR64DB64, AVR64DB28, AVR32DB28, AVR64DB32, and AVR32DB48.

Retracted device support Support is no longer available for the following AVR parts: AVR16DA28, AVR16DA32 and, AVR16DA48.

3.5. Version 2.31

None.

3.6. Version 2.30

New option to prevent data initialisation A new `-mno-data-init` driver option prevents the initialisation of data and the clearing of bss sections. It works by suppressing the output of the `do_copy_data` and `do_clear_bss` symbols in assembly files, which will in turn prevent the inclusion of those routines by the linker.

Enhanced optimizations A number of optimization improvements have been made, including the removal of redundant return instructions, the removal of some jumps following a skip-if-bit-is instruction, and improved procedural abstraction and the ability to iterate this process.
Additional options are now available to control some of these optimizations, specifically `-fsection-anchors`, which allows access of `static` objects to be performed relative to one symbol; `-mpa-`

`iterations=n`, which allows the number of procedural abstraction iterations to be changed from the default of 2; and, `-mpa-callcost-shortcall`, which performs more aggressive procedural abstraction, in the hope that the linker can relax long calls. This last option can increase code size if the underlying assumptions are not realized.

New device support Support is available for the following AVR parts: AVR16DA28, AVR16DA32, AVR16DA48, AVR32DA28, AVR32DA32, AVR32DA48, AVR64DA28, AVR64DA32, AVR64DA48, AVR64DA64, AVR128DB28, AVR128DB32, AVR128DB48, and AVR128DB64.

Retracted device Support Support is no longer available for the following AVR parts: ATA5272, ATA5790, ATA5790N, ATA5791, ATA5795, ATA6285, ATA6286, ATA6612C, ATA6613C, ATA6614Q, ATA6616C, ATA6617C, and ATA664251.

3.7. Version 2.29 (Functional Safety Release)

Header file for compiler built-ins To ensure that the compiler can conform to language specifications such as MISRA, the `<builtins.h>` header file, which is automatically included by `<xc.h>`, has been updated. This header contains the prototypes for all in-built functions, such as `__builtin_avr_nop()` and `__builtin_avr_delay_cycles()`. Some built-ins may not be MISRA compliant; these can be omitted by adding the define `__XC_STRICT_MISRA` to the compiler command line. The built-ins and their declarations have been updated to use fixed-width types.

3.8. Version 2.20

New device support Support is available for the following AVR parts: ATTINY1624, ATTINY1626, and ATTINY1627.

Better best fit allocation The best fit allocator (BFA) in the compiler has been improved so that sections are allocated in an order permitting better optimization. The BFA now supports named address spaces and better handles data initialization.

Improved procedural abstraction The procedural abstraction optimizations are now performed on more code sequences. Previous situations where this optimization might have increased code size have been addressed by making the optimization code aware of the linker's garbage collection process.

Absence of AVR Assembler The AVR Assembler is no longer included with this distribution.

3.9. Version 2.19 (Functional Safety Release)

None.

3.10. Version 2.10

Code Coverage This release includes a code coverage feature that facilitates analyzis of the extent to which a project's source code has been executed. Use the option `-mcodecov=ram` to enable it. After execution of the program on your hardware, code coverage information will be collated in the device, and this can be transferred to and displayed by the MPLAB X IDE via a code coverage plugin. See the IDE documentation for information on this plugin can be obtained.
The `#pragma nocodecov` may be used to exclude subsequent functions from the coverage analyzis. Ideally the pragma should be added at the beginning of the file to exclude that entire file from the coverage analyzis. Alternatively, the `__attribute__((nocodecov))` may be used to exclude a specific function from the coverage analyzis.

Device description files A new device file called `avr_chipinfo.html` is located in the `docs` directory of

the compiler distribution. This file lists all devices supported by the compiler. Click on a device name, and it will open a page showing all the allowable configuration bit setting/value pairs for that device, with examples.

Procedural abstraction Procedural abstraction optimizations, which replace common blocks of assembly code with calls to an extracted copy of that block, have been added to the compiler. These are performed by a seperate application, which is automatically invoked by the compiler when selecting level `2`, `3` or `s` optimizations. These optimizations reduce code size, but they may reduce execution speed and code debugability.
Procedural abstraction can be disabled at higher optimization levels using the option `-mno-pa`, or can be enabled at lower optimization levels (subject to your license) by using `-mpa`. It can be disabled for an object file using `-mno-pa-on-file=`*`filename`*, or disabled for a function by using `-mno-pa-on-function=`*`function`*.
Inside your source code, procedural abstraction can be disabled for a function by using `__attribute__((nopa))` with the function's definition, or by using `__nopa`, which expands to `__attribute__((nopa,noinline))` and thus prevents function inlining from taking place and there being abstraction of inlined code.

Lock bit support in pragma The `#pragma config` can now be used to specify the AVR lock bits as well as the other configuration bits. Check the `avr_chipinfo.html` file (mentioned above) for the setting/value pairs to use with this pragma.

New device support Support is available for the following parts: AVR28DA128, AVR64DA128, AVR32DA128, and AVR48DA128.

3.11. Version 2.05

More bits for your buck The macOS version of this compiler and license manager is now a 64-bit application. This will ensure that the compiler will install and run without warnings on recent versions of macOS.

Const objects in program memory The compiler can now place `const`-qualified objects in the program Flash memory, rather than having these located in RAM. The compiler has been modified so that `const`-qualified global data is stored in program flash memory and this data can be directly and indirectly accessed using the appropriate program-memory instructions. This new feature is enabled by default but can be disabled using the `-mno-const-data-in-progmem` option. For avrxmega3 and avrtiny architectures, this feature is not required and is always disabled, since program memory is mapped into the data address space for these devices.

Standard for free Unlicensed (Free) versions of this compiler now allow optimizations up to and including level 2. This will permit a similar, although not identical, output to what was previously possible using a Standard license.

Welcome AVRASM2 The AVRASM2 assembler for 8-bit devices is now included in the XC8 compiler installer. This assembler is not used by the XC8 compiler, but is available for projects based on hand-written assembly source.

New device support Support is available for the following parts: ATMEGA1608, ATMEGA1609, ATMEGA808, and ATMEGA809.

3.12. Version 2.00

Top-level Driver A new driver, called `xc8-cc`, now sits above the previous `avr-gcc` driver and the xc8

driver, and it can call the appropriate compiler based on the selection of the target device. This driver accepts GCC-style options, which are either translated for or passed through to the compiler being executed. This driver allows a similar set of options with similar semantics to be used with any AVR or PIC target and is thus the recommended way to invoke the compiler. If required, the old `avr-gcc` driver can be called directly using the old-style options it accepted in earlier compiler versions.

Common C Interface This compiler can now conform to the MPLAB Common C Interface, allowing source code to be more easily ported across all MPLAB XC compilers. The `-mext=cci` option requests this feature, enabling alternate syntax for many language extensions.

New librarian driver A new librarian driver is positioned above the previous PIC `libr` librarian and the AVR `avr-ar` librarian. This driver accepts GCC-archiver-style options, which are either translated for or passed through to the librarian being executed. The new driver allows a similar set of options with similar semantics to be used to create or manipulate any PIC or AVR library file and is thus the recommended way to invoke the librarian. If required for legacy projects, the previous librarian can be called directly using the old-style options it accepted in earlier compiler versions.

4. Migration Issues

The following are features that are now handled differently by the compiler. These changes may require modification to your source code if porting code to this compiler version. The version number in the subheadings indicates the first compiler version to support the changes that follow.

4.1. Version 2.39 (Functional Safety Release)

None.

4.2. Version 2.36

None.

4.3. Version 2.35

Handling of string-to bases (XC8-2420) To ensure consistency with other XC compilers, the XC8 string-to functions, like `strtol()` etc., will no longer attempt to convert an input string if the base specified is larger than 36 and will instead set `errno` to `EINVAL`. The C standard does not specify the behaviour of the functions when this base value is exceeded.

Inappropriate speed optimizations Procedural abstraction optimizations were being enabled when selecting level 3 optimizations (`-O3`). These optimizations reduce code size at the expense of code speed, so should not have been performed. Projects using this optimization level might see differences in code size and execution speed when built with this release.

Library functionality The code for many of the standard C library functions now come from Microchip's Unified Standard Library, which might exhibit different behaviour in some circumstances compared to that provided by the former avr-libc library. For example, it is no longer necessary to link in the lprintf_flt library (`-lprintf_flt` option) to turn on formatted IO support for float-format specifiers. The smart IO features of the Microchip Unified Standard Library makes this option redundant. Additionally, the use of _P suffixed routines for string and memory functions (e.g. `strcpy_P()` etc..) that operate on const strings in flash are no longer necessary. The standard C routines (e.g. `strcpy()`) will work correctly with such data when the const-data-in-program-memory feature is enabled.

4.4. Version 2.32

None.

4.5. Version 2.31

None.

4.6. Version 2.30

None.

4.1. Version 2.29 (Functional Safety Release)

None.

4.2. Version 2.20

Changed DFP layout The compiler now assumes a different layout used by DFPs (Device Family Packs). This will mean that an older DFP might no work with this release, and older compilers will not be able to use the latest DFPs.

4.3. Version 2.19 (Functional Safety Release)

None.

4.4. Version 2.10

None

4.5. Version 2.05

Const objects in program memory Note that the by default, `const`-qualfiied objects will be placed and accessed in program memory (as described here). This will affect the size and execution speed of your project, but should reduce RAM usage. This feature can be disabled, if required, using the `-mno-const-data-in-progmem` option.

4.6. Version 2.00

Configuration fuses The device configuration fuses can now programmed using a `config` pragma followed by setting-value pairs to specify the fuse state, e.g.

```
#pragma config WDTON = SET
#pragma config BODLEVEL = BODLEVEL_4V3
```

Absolute objects and functions Objects and functions can now be placed at specific address in memory using the CCI `__at(`*address*`)` specifier, for example:

```
#include <xc.h>
int foobar __at(0x800100);
char __at(0x250) getID(int offset) { ... }
```

The argument to this specifier must be a constant that represents the address at which the first byte or instruction will be placed. RAM addresses are indicated by using an offset of 0x800000. Enable the CCI to use this feature.

New interrupt function syntax The compiler now accepts the CCI `__interrupt(`*num*`)` specifier to indicate that C functions are interrupt handlers. The specifier takes an interrupt number, for example:

```
#include <xc.h>
void __interrupt(SPI_STC_vect_num) spi_Isr(void) { ... }
```

5. Fixed Issues

The following are corrections that have been made to the compiler. These might fix bugs in the generated code or alter the operation of the compiler to that which was intended or specified by the user's guide. The version number in the subheadings indicates the first compiler version to contain fixes for the issues that follow. The bracketed label(s) in the title are that issue's identification in the tracking database. These may be useful if you need to contact support.

Note that some device-specific issues are corrected in the Device Family Pack (DFP) associated with the device. See the MPLAB Pack Manager for information on changes made to DFPs and to download the latest packs.

5.1. Version 2.39 (Functional Safety Release)

None.

5.2. Version 2.36

Error when delaying (XC8-2774) Minor changes in the default Free mode optimizations prevented constant folding of operand expressions to the delay built-in functions, resulting in them being treated as non-constants and triggering the error: `__builtin_avr_delay_cycles expects a compile time integer constant`.

5.3. Version 2.35

Contiguous allocation using __at (XC8-2653) Contiguous allocation of multiple objects places in a section with the same name and using `__at()` did not work correctly. For example:

```
const char arr1[] __attribute__((section(".mysec"))) __at (0x500) = {0xAB, 0xCD};
const char arr2[] __attribute__((section(".mysec"))) = {0xEF, 0xFE};
```

should have placed `arr2` immediately after `arr1`.

Specifying section start addresses (XC8-2650) The `-Wl,--section-start` option was silently failing to place sections at the nominated start address. This issue has been fixed for any custom-named sections; however, it will not work for any standard sections, such as `.text` or `.bss`, which must be placed using a `-Wl,-T` option.

Linker crashes when relaxing (XC8-2647) When the `-mrelax` optimization was enabled and there were code or data sections that did not fit into the available memory, the linker crashed. Now, in such a circumstance, error messages are issued instead.

No no-falling-back (XC8-2646) The `--nofallback` option was not correctly implemented, nor documented. This can now be selected to ensure that the compiler will not fall back to a lower optimization setting if the compiler is unlicensed, and will instead issue an error.

Inappropriate speed optimizations (XC8-2637) Procedural abstraction optimizations were being enabled when selecting level 3 optimizations (`-O3`). These optimizations reduce code size at the expense of code speed, so should not have been performed.

Bad EEPROM access (XC8-2629) The `eeprom_read_block` routine did not work correctly on Xmega devices when the `-mconst-data-in-progmem` option was enabled (which is the default state),

resulting in EEPROM memory not being read correctly.

Invalid memory allocation (XC8-2593, XC8-2651) When the `-Ttext` or `-Tdata` linker option (for example passed through using a `-Wl` driver option) is specified, the corresponding text/data region origin was updated; however, the end address was not adjusted accordingly, which could have led to the region exceeding the target device's memory range.

Crash with over-attributed function (XC8-2580) The compiler crashed if a function was declared using more than one of the interrupt, signal or nmi attributes, e.g., `__attribute__((__signal__, __interrupt__))`.

Invalid ATtiny interrupt code (XC8-2465) When building for ATtiny devices and the optimizations were disabled (`-O0`), interrupt functions may have triggered `operand out of range` assembler messages.

Options not being passed through (XC8-2452) When using the `-Wl` option with multiple, comma-separated linker options, not all of the linker options were being passed to the linker.

Error indirectly reading program memory (XC8-2450) In some instances, the compiler produced an internal error (`unrecognizable insn`) when reading a two byte value from a pointer to program memory

5.4. Version 2.32

Second access of library fails (XC8-2381) Invoking the Windows version of the `xc8-ar.exe` library archiver a second time to access an existing library archive may have failed with an `unable to rename` error message.

5.5. Version 2.31

Unexplained compiler failures (XC8-2367) When running on Windows platforms that had the system temporary directory set to a path that included a dot '`.`' character, the compiler may have failed to execute.

5.6. Version 2.30

Global labels misplaced after outlining (XC8-2299) Hand-written assembly code that places global labels within assembly sequences that are factored out by procedural abstraction might not have been correctly repositioned.

A relaxing crash (XC8-2287) Using the `-mrelax` option might have caused the linker to crash when tail jump relaxation optimizations attempted to remove `ret` instruction that were not at the end of a section.

Crash when optimizing labels as values (XC8-2282) Code using the "Labels as values" GNU C language extension might have caused the procedural abstraction optimizations to crash, with an `Outlined VMA range spans fixup` error.

Not so const (XC8-2271) The prototypes for `strstr()` and other functions from `<string.h>` no longer specify the non-standard `const` qualifier on returned string pointers when the `-mconst-data-in-progmem` feature is disabled. Note that with avrxmega3 and avrtiny devices, this feature is permanently enabled.

Lost initializers (XC8-2269) When more than one variable in a translation unit was placed in a section (using `__section` or `__attribute__((section))`), and the first such variable was zero initialized or did not have an initializer, initializers for other variables in the same translation unit that were placed in the same section were lost.

5.1. Version 2.29 (Functional Safety Release)

None.

5.2. Version 2.20

Error with long commands (XC8-1983) When using an AVR target, the compiler may have stopped with a file not found error, if the command line was extremely large and contained special characters such a quotes, backslashes, etc.

Unassigned rodata section (XC8-1920) The AVR linker failed to assign memory for custom rodata sections when building for avrxmega3 and avrtiny architectures, potentially producing memory overlap errors

5.3. Version 2.19 (Functional Safety Release)

None.

5.4. Version 2.10

Relocation failures (XC8-1891) The best fit allocator was leaving memory 'holes' in between sections after linker relaxation. Aside from fragmenting memory, this increased the possibility of there being linker relocation failures relating to pc-relative jumps or calls becoming out of range.

Instructions not transformed by relaxation (XC8-1889) Linker relaxation did not occur for jump or call instructions whose targets become reachable if relaxed.

Missing <power.h> functionality (XC8E-388) Several definitions from `<power.h>`, such as `clock_div_t` and `clock_prescale_set()`, were not defined for devices, including the ATmega324PB, ATmega328PB, ATtiny441, and ATtiny841.

Missing macros The preprocessor macros `_XC8_MODE_`, `__XC8_VERSION`, `__XC`, and `__XC8` were not automatically defined by the compiler. These are now available.

5.5. Version 2.05

Internal compiler error (XC8-1822) When building under Windows, an internal compiler error might have been produced when optimizing code.

RAM overflow not detected (XC8-1800, XC8-1796) Programs that exceeded that available RAM were not detected by the compiler in some situations, resulting in a runtime code failure.

Omitted flash memory (XC8-1792) For avrxmega3 and avrtiny devices, parts of the flash memory might have been left un-programmed by the MPLAB X IDE.

Failure to execute main (XC8-1788) In some situations where the program had no global variables defined, the runtime startup code did not exit and the `main()` function was never reached.

Incorrect memory information (XC8-1787) For avrxmega3 and avrtiny devices, the avr-size program was reporting that read-only data was consuming RAM instead of program memory.

Incorrect program memory read (XC8-1783) Projects compiled for devices with program memory mapped into the data address space and that define objects using the PROGMEM macro/attribute might have read these objects from the wrong address.

Internal error with attributes (XC8-1773) An internal error occurred if you defined pointer objects with the `__at()` or attribute() tokens in between the pointer name and dereferenced type, for example, `char * __at(0x800150) cp;` A warning is now issued if such code is encountered.

Failure to execute main (XC8-1780, XC8-1767, XC8-1754) Using EEPROM variables or defining fuses using the config pragma might have caused incorrect data initialisation and/or locked up program execution in the runtime startup code, before reaching `main()`.

Fuse error with tiny devices (XC8-1778, XC8-1742) The attiny4/5/9/10/20/40 devices had an incorrect fuse length specified in their header files that lead to linker errors when attempting to build code that defined fuses.

Segmentation fault (XC8-1777) An intermittent segmentation fault has been corrected.

Assembler crash (XC8-1761) The `avr-as` assembler might have crashed when the compiler was run under Ubuntu 18.

Objects not cleared (XC8-1752) Uninitialized static storage duration objects might not have been cleared by the runtime startup code.

Conflicting device specification ignored (XC8-1749) The compiler was not generating an error when multiple device specification options were used and indicated different devices.

Memory corruption by heap (XC8-1748) The `__heap_start` symbol was being incorrectly set, resulting in the possibility of ordinary variables being corrupted by the heap.

Linker relocation error (XC8-1739) A linker relocation error might have been emitted when code contained a `rjmp` or `rcall` with a target exactly 4k bytes away.

5.6. Version 2.00

None.

6. Known Issues

The following are limitations in the compiler's operation. These may be general coding restrictions, or deviations from information contained in the user's manual. The bracketed label(s) in the title are that issue's identification in the tracking database. This may be of benefit if you need to contact support. Those items which do not have labels are limitations that describe *modi operandi* and which are likely to remain in effect permanently.

6.1. MPLAB X IDE Integration

MPLAB IDE integration If Compiler is to be used from MPLAB IDE, then you must install MPLAB IDE prior to installing Compiler.

6.2. Code Generation

Incorrect initialization (XC8-2679) There is a discrepancy between where the initial values for some global/static byte-sized objects are placed in data memory and where the variables will be accessed at runtime.

strtod incorrectly sets endptr (XC8-2652) In instances where a subject string for conversion by `strtod()` contains what appears to be a floating-point number in exponential format and there is an unexpected character after an `e` character, then the `endptr` address, if provided, will point to the character after the `e` and not the `e` itself. For example:

```
strtod("100exy", endptr);
```

will result in `endptr` pointing to the `x` character.

Bad indirect function calls (XC8-2628) In some instances, function calls made via a function pointer stored as part of a structure might fail.

strtof returns zero for hexadecimal floats (XC8-2626) The library functions `strtof()` et al and `scanf()` et al, will always convert a hexadecimal floating-point number that does not specify an exponent to zero. For example:

```
strtof("0x1", &endptr);
```

will return the value 0, not 1.

Inaccurate stack advisor messaging (XC8-2542, XC8-2541) In some instances, the stack advisor warning regarding recursion or indeterminate stack used (possibly through the use of alloca()) is not emitted.

Failure with duplicate interrupt code (XC8-2421) Where more than one interrupt function has the same body, the compiler might have the output for one interrupt function call the other. This will result in all call-clobbered registers being saved unnecessarily, and the interrupts will be enabled even before the epilogue of the current interrupt handler has run, which could lead to code failure.

Const objects not in program memory (XC8-2408) For avrxmega3 and avrtiny projects unitialized `const` objects are placed into data memory, even though a warning suggests that they have been placed in program memory. This will not affect devices that do not have program memory mapped into the data memory space, nor will it affect any object that is initialized.

Bad output with invalid DFP path (XC8-2376) If the compiler is invoked with an invalid DFP path and a 'spec' file exists for the selected device, the compiler is not reporting the missing device family pack and instead selecting the 'spec' file, which might then lead to an invalid output. The 'spec' files might not be up to date with the distributed DFPs and were intended for use with internal compiler testing only.

Memory overlap undetected (XC8-1966) The compiler is not detecting the memory overlap of objects made absolute at an address (via `__at()`) and other objects using the `__section()` specifier and that are linked to the same address.

Failure with library functions and __memx (XC8-1763) Called libgcc float functions with an argument in the `__memx` address space might fail. Note that library routines are called from some C operators, so, for example, the following code is affected:

```
return regFloatVar > memxFloatVar;
```

Limited libgcc implementation (AVRTC-731) For the ATTiny4/5/9/10/20/40 products, the standard C / Math library implementation in libgcc is very limited or not present.

Program memory limitations (AVRTC-732) Program memory images beyond 128 kb are supported by the toolchain; however, there are known instances of linker aborts without relaxation and without a helpful error message rather than generating the required function stubs when the `-mrelax` option is used.

Name space limitations (AVRTC-733) Named address spaces are supported by the toolchain, subject to the limitations mentioned in the user's guide section Special Type Qualifiers.

Time zones The `<time.h>` library functions assume GMT and do not support local time zones, thus `localtime()` will return the same time as `gmtime()`, for example.