COSMIC Compiler and Debugger for ST Microelectronics

STLUX & STNRG

Family

COSMIC Development Tools for STLUX and STNRG include Cosmic's market-standard cxstm8 compiler for the STLUX & STNRG family and a special version of Cosmic's ZAP debugger specifically developped for STLUX and STNRG devices.

The I.D.E.A. Windows editor provides a complete Integrated Development Environment under Windows. It offers instant access to a fully-integrated editor, project manager, compiler, linker, utilities, and the highly intuitive source-level debugger ZAP.

The CXSTM8 compiler is field tested, reliable, and incorporates many features that help ensure your embedded STLUX & STNRG design meets and exceeds performance specifications both for code size and speed.

The ZAP Debugger is a full featured C and Assembly language source-level debugger for embedded applications. In its STLUX & STNRG version, ZAP has been adapted to provide useful specific features such as trasparent option bytes programming.

Key Features

Supports All STLUX & STNRG chips
SMED configurator support
Option bytes in the source code support
Global and Processor-Specific Optimizations
C support for Zero Page Data
C support for Bit Variables
C support for Interrupt Handlers
In-Line Assembly

Microcontroller Specific Design

cxstm8 is the market standard compiler for the STM8 architecture that is the core of the STLUX & STNRG family of microcontrollers; all STLUX & STNRG family processors are supported. A special code generator and optimizer targeted for the STLUX & STNRG family eliminates the overhead and complexity of a more generic compiler. You also get header file support for many of the popular STLUX & STNRG peripherals, so you can access their memory mapped objects by name either at the C or assembly language levels. C level support is provided for Short Addressing and Bit Variables. Code generated by the SMED configurator is compiled automatically and transparently.

ANSI / ISO Standard C

This implementation conforms with the ANSI and ISO Standard C specifications which helps you protect your software investment by aiding code portability and reliability.

Memory Models for different applications

The Compiler provides 2 different memory models depending on the size of the application. For applications smaller that 64k, the “section 0” memory model provides the best code density by defaulting function calls and pointers to 2 bytes. For applications bigger than 64k, the standard memory model provides the best flexibility for using easily the linear addressing space. Each model comes with its own set of libraries.

Optimizations

The Compiler includes global and microcontroller specific optimizations to give your application maximum chance of meeting and exceeding its performance specifications. You retain control over optimizations via compile-time options and keyword extensions to ANSI C, so you can fine tune your application code to match your design specification.

Example of STLUX & STNRG specific optimizations include:

- Function arguments can be passed char-sized without widening to *int*.
- Commonly used static data can be selectively, using the @tiny keyword, or globally, using a compile-time option, placed into zero page memory (the first 256 bytes of memory) to decrease access time.
- The Code Factorization optimization replaces duplicated chunks of code by grouping them into subroutines.
- Assembler instructions rearrangement allows to avoid pipeline stalls for the best performance.
- Strict single-precision (32-bit) floating point arithmetic and math functions. Floating point numbers are represented as in the IEEE754 Floating Point Standard.
- Other optimizations include: branch shortening logic, jump-to-jump elimination, constant folding, elimination of unreachable code, removal of redundant loads/stores, and switch statement optimizations.

Controller Specific Extensions to ANSI C

The Compiler includes several extensions to the ANSI C standard which have been designed specifically to give you maximum control of your application at the C level and to simplify the job of writing C code for your embedded STLUX & STNRG design :

- You can define in-line assembly using _asm() to insert assembly instructions directly in your C code to avoid the overhead of calling assembly language subroutines.
- Also you can use #asm/#endasm to insert assembly instructions directly in your C code.
- You can declare bit variables (_Bool type) packed into bytes by the compiler for global and local variables. The bit instruction will be used wherever possible.
- You can define C functions as interrupt handlers using the @interrupt keyword. Compiler saves volatile registers for handling exceptions and interrupts.
- You can define a C object or C function to have an absolute address at the C-level, using the @<address> syntax appended to you data definition; this is useful for interrupt handlers written in C and for defining memory mapped I/O.

  This is useful to add option bytes values directly in the source code.
- You can define *char*- and *int*-sized bitfields, and select bit numbering from right-to-left or left-to-right.

Additional Compiler Features

- Full C and assembly source-level debugging support.
- Absolute and relocatable listing file output, with interspersed C, assembly language and object code; optionally, you can include compiler errors and compiler optimization comments.
- Extensive and useful compile-time error diagnostics.
- Fast compile and assemble time.
- Full user control over include file path(s), and placement of output object, listing and error file(s).
- All objects are relocatable and host independent. (i.e. files can be compiled on a workstation and debugged on a PC).
- Function code and switch tables are generated into the code *(.text)* section. Constant data such as string constants and *const* data are generated into a separate program *(.const)* section.
- Initialized static data can be located separately from uninitialized data or data initialized to zero.
- All function code is (by default) never self-modifying, including structure assignment and function calls, so it can be shared and placed in ROM.
- Code is generated as a symbolic assembly language file so you can examine compiler output.
- *The compiler* creates all its tables dynamically on the heap, allowing very large source files to be compiled.
- Common string manipulation routines are implemented in assembly language for fast execution.

Assembler

The COSMIC STLUX & STNRG assembler, *caSTM8*, supports macros, conditional assembly, up to 255 named program sections, includes, branch optimizations, expression evaluation, relocatable or absolute output, relocatable arithmetic, listing files and cross references. The assembler also passes through line number information, so that COSMIC’s ZAP

debugger can perform full source-level debug at the assembly language level.

Linker

The COSMIC linker, *clnk*, combines relocatable object files created by the assembler, selectively loading from libraries of object files made with the librarian, *clib*, to create an executable format file.

clnk features:

- Flexible and extensive user-control over the linking process and selective placement of user application code and data program sections.
- Multi-segment image construction, with user control over the address for each code and data section. Specified addresses can cover the full logical address space of the target processor with up to 255 separate segments. This feature is useful for creating an image which resides in a target memory configuration consisting of scattered areas of ROM and RAM.
- Generation of memory map information to assist debugging, including the full call tree.
- All symbols and relocation items can be made absolute to prelocate code that will be linked in elsewhere.
- Symbols can be defined, or aliased**,** from the Linker command File.

Librarian

The COSMIC librarian, *clib*, is a development aid which allows you to collect related files into one named library file, for more convenient storage. *clib* provides the functions necessary to build and maintain object module libraries. The most obvious use for *clib* is to collect related object files into separate named library files, for scanning by the linker. The linker loads from a library only those modules needed to satisfy outstanding references.

Absolute Hex File Generator

The COSMIC hex file generator, *chex*, translates executable images produced by the linker to one of several hexadecimal interchange formatsStandard Intel hex format.

- Motorola S-record and S2 record format.
- Rebiasing of text and data section load addresses. Allows you to generate hex files to load anywhere and execute anywhere in the target system address space.

Absolute C and Assembly Language Listings

Paginated listings can be produced to assist program understanding. Listings can include original C source code with interspersed assembly code and absolute object code. Optionally, you can include compiler errors and optimization comments.

Debugging Utilities

The cross compiler package includes utility programs which provide listings for all debug and map file information. The *clst* utility creates listings showing the C source files that were compiled to obtain the relocatable or executable files. The *cprd* utility extracts and prints information on the name, type, storage class and address of program static data, function arguments and function automatic data.

ZAP Debugger

ZAP is a full featured source-level debugger available for Windows. ZAP's intuitive graphical interface is uniform for all targets and execution environments.
ZAP for STLUX and STNRG is available for the STLink hardware and supports all the existing devices.
In its STLUX & STNRG version, ZAP has been adapted to provide useful specific features such as trasparent option bytes programming.