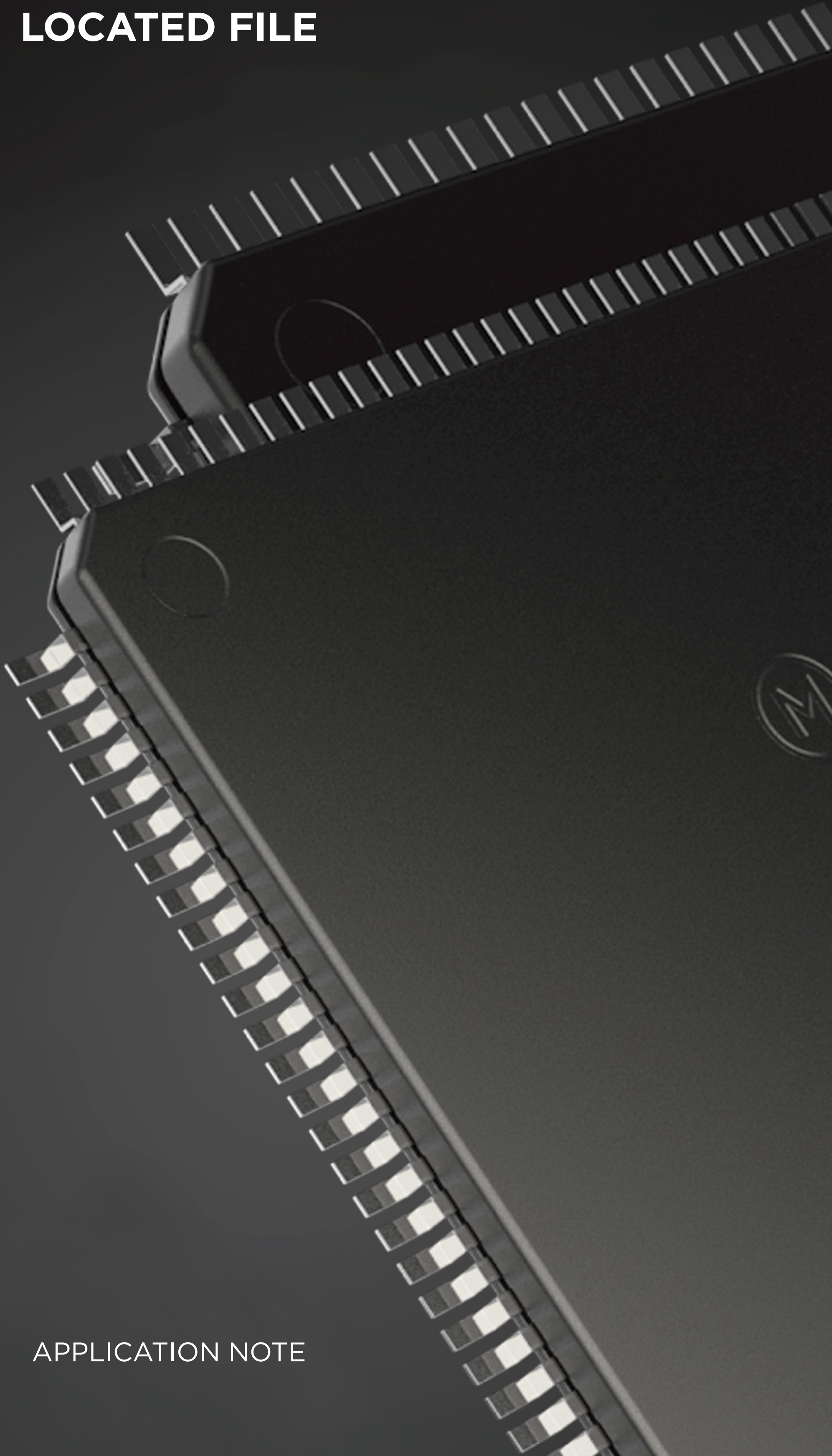
TASKING®

CROSS-LINKING SMARTCODE AND TRICORE VX V6.3R1 BINARIES INTO A SINGLE ABSOLUTE LOCATED FILE



APPLICATION NOTE

THIS APPLICATION NOTE WILL FOCUS ON CROSS-LINKING SMARTCODE BINARIES AND TRICORE VX V6.3R1 BINARIES INTO A SINGLE ABSOLUTE LOCATED FILE.

INTRODUCTION

Cross-linking is an important tool in the embedded software engineer's toolbox. Today's embedded safety systems include a myriad of performance, safety, and security requirements. These systems frequently rely on legacy binaries that have a proven track record when it comes to safety and reliability.

The TASKING TriCore development tools provide the capability to easily integrate relocatable object files (suffix .o) from legacy code built with older versions of the toolset.

The recently released SmartCode toolset which was designed to support the third generation of the AURIX™ microcontroller (TC4x) accepts and links object files built with v6.3r1 of the TriCore VX-Toolset.

The following application note will show two simple examples of how to do this. Please refer to the following illustration for a high-level view of the process.

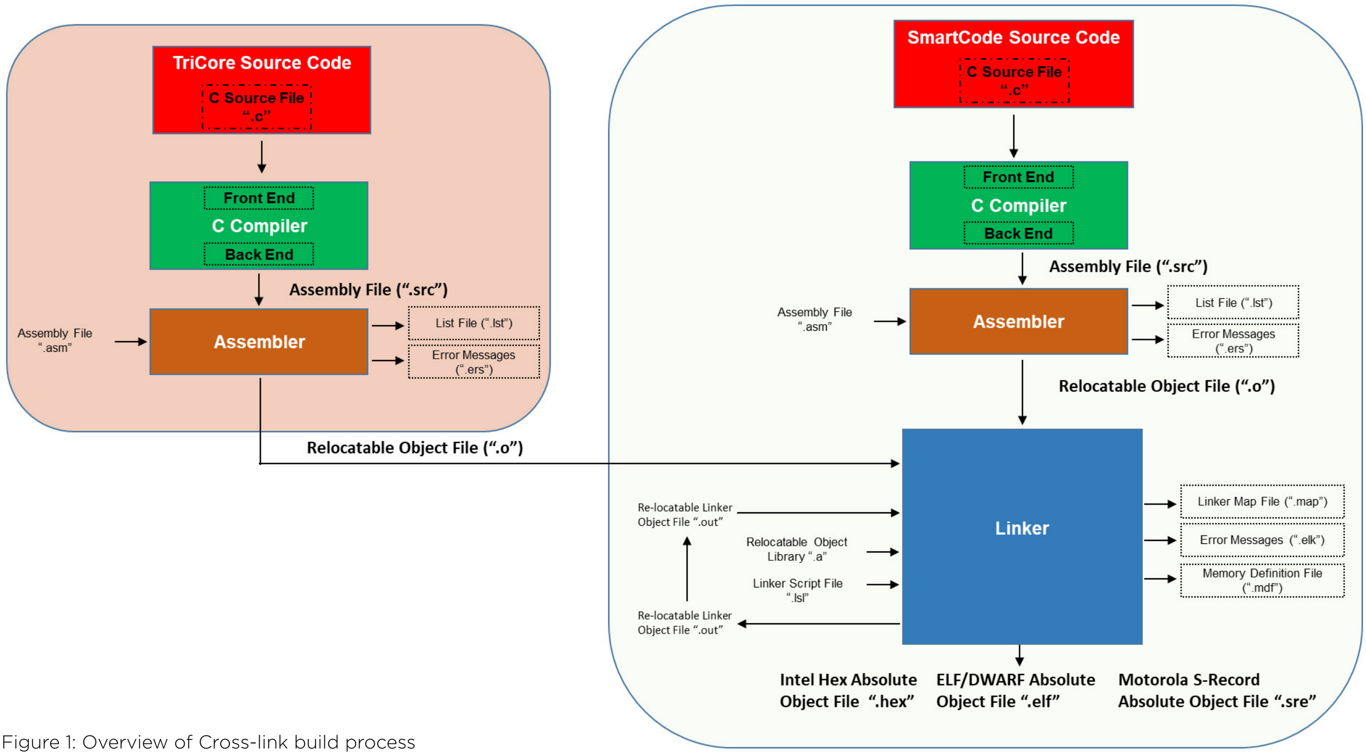


Figure 1: Overview of Cross-link build process

WHAT IS CROSS-LINKING?

Cross-Linking is a feature of the TASKING TriCore toolset that allows you to re-use object files built with previous versions of the toolset and link them into application code that is being developed with a newer version of the TriCore toolset. For this application note, SmartCode v10.1r1 is newer than TriCore VX v6.3r1.

WHY IS SOFTWARE RE-USE IMPORTANT?

Critical software components with a proven safety record are often re-used due their history of operating in the field without incident. The re-use of these legacy object files and binaries can save time and money since they have already undergone a costly validation and verification procedure.

Other notable reasons for using legacy software components:

- SW architects with detailed information on the algorithm IP may no longer be with the company
- Source code may not be available for re-compilation
- OEM safety case may require usage of specific SW components
- Third-party validated safety compliant SW components like MCAL and/or AUTOSAR specify compiler version and toolset options in their release notes.

WHY DO WE WANT TO USE THE NEWEST VERSION OF THE TOOLSET FOR OUR APPLICATION DEVELOPMENT?

The evolution of compiler toolsets like TASKING's SmartCode are continually introducing new features to increase the performance, safety, and efficiency of your embedded applications.

Examples of how new toolset features can improve software development:

- New application features can benefit from performance gains and improvements available with the latest compiler technology
- Latest compiler optimizations can improve performance and reduce footprint size allowing your application to fit into a more cost-effective device
- The improved performance and reduced application footprint size can help achieve any OEM specified overhead requirements.

WHY DO WE WANT TO CROSS-LINK?

Let us assume that the option to cross-link is not available. The developer of a new software component would have the following options:

1. The version of the compiler used for building the software component would have to be used for the rest of their software
2. All future software development with this component would be restricted to this version of the toolset
3. Any other applications wanting to use the software component would have to be ported to this version of the toolset.

Clearly this is undesirable. Thankfully, this is not the case. Software developers using the TASKING TriCore toolsets who need to use legacy object code built with previous versions of the TriCore toolset can cross-link with a newer version of the toolset. This provides the developer the following advantages:

- Seamlessly adding legacy object file(s) to their current build
- Being able to use certified third-party software (I.e., MCAL) that was developed with a previous toolset version.
- Commercial advantages -> avoid re-writing, porting, testing, or validating legacy object files

I think it is clear, why we want to cross-link, but should we? Is it safe? Why shouldn't we fear linking object files from different compiler versions? After all, calling conventions may have changed, data layouts may be different and who knows what other surprises lurk in the dark recesses of a micro-controller's memory map?

HOW DOES TASKING ENSURE THAT CROSS-LINKING IS SAFE?

All TriCore compiler versions adhere to the Infineon Embedded Application Binary Interface (EABI) specification. The EABI is a set of interface standards that writers of compilers, assemblers and linker/locators must use when creating compliant tools for the TriCore architecture. For example, characters must be stored on byte boundaries, short integers must be two-byte aligned and any data types with a size of four bytes or larger must be four-byte aligned.

Prior to specifying that two compiler versions are suitable for cross-linking, TASKING methodically compares the relevant aspects of the two compiler versions, such as data sizes and alignments, and tests the compatibility of the calling conventions. An additional set of tests has been generated to cover all possible cases; passing these tests means that the two compiler versions are cross-link compatible.

Note: The TriCore architecture has requirements related to the alignment of certain data types, structs and struct members. The following points should be considered:

- Alignment restrictions of the TriCore architecture
- Alignment requirements of the Infineon EABI
- Alignment options provided by the C compiler
- Alignment options provided by the LSL linker script language

Note: For additional details on the Infineon EABI in relation to the TASKING toolset, please refer to the application note "Alignment Requirements – Restrictions for the TriCore Architecture" located on the 'resources' tab of the TASKING website.

WHAT ISSUES CAN CAUSE CROSS LINKING ERRORS?

With the EABI, Infineon defines the calling conventions that are used; however, compiler vendors may offer alternative settings to achieve a more compact data placement. For example, the TASKING TriCore toolset allows the user to change the section/data alignment to increase data density and reduce unnecessary alignment gaps. In addition, TASKING has introduced some compiler flags to enable or disable specific deviations from the standard EABI.

IMPORTANT: Please note, that some of the following cross-linking points no longer apply to SmartCode since it was designed to always be EABI compliant. (I.e., SmartCode does not support the --eabi option to disable EABI compliance.)
However, these points are applicable, when trying to cross-link between two previous versions of the TriCore VX-toolset (e.g, v5.0r2 and v6.2r2).

To ensure cross-linking is successful:

- Ensure all object files are compiled with the same settings for these flags.
- Ensure all objects use the same size for the double type
- Ensure alignment options also match.

Potential issues with cross linking:

- EABI Compliance deviations within TriCore VX Toolset
  - Main two EABI settings affecting cross-linking are half-word-align AND treat double as float
  - Base type alignment may be an issue with bit-field alignment.
    Bit-field alignment issues can lead to errors when trying to access a register bit.
  - ISO C supports `'0'` size bit-field. This can lead to errors with Base type alignment.
    This topic is beyond the scope of this application note.
  - The –no-clear option violates the EABI. The specification declares that non-initialized global data must be zero'ed in startup code (I.e.,data needs to be cleared before used). However, Applications using battery backed RAM, need to know the content of RAM after a re-start. The –no-clear option ensures that data in battery backed RAM is retained and not overwritten by the initialization.
  - Word aligned structures and unions larger than or equal to 64-bit can cause compatibility issues with SmartCode. TASKING included for double word copy instructions (Not EABI compliant).
- Potential issues with cross-linking may arise when using the compiler options `'--integer-enumeration'` (always uses 32-bit integers for enumeration) and `'--signed-bitfields'`.
  Refer to Section 7.5 of the TASKING VX-toolset for TriCore v6.3r1 for more information.

CROSS-LINK A SIMPLE PROJECT BUILT IN TRICORE VX AND SMARTCODE

Create a simple TriCore project with a library cross-linked to SmartCode.

1. First step is to create a `'hello world'` project in TriCore VX.

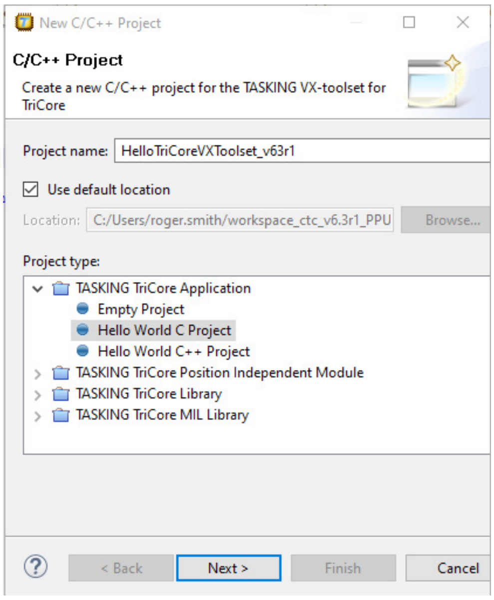


Figure 2: VX-Toolset for TriCore, New C/C++ Project dialog box

2. Choose an AURIX variant for the project. In this example, we use the TC39xB

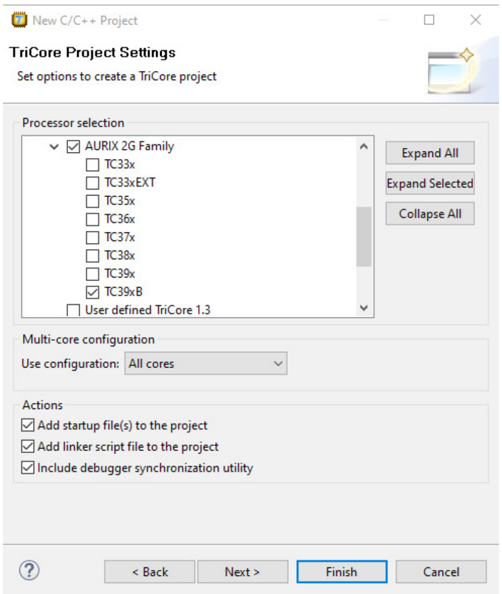


Figure 3: VX-Toolset for TriCore, New C/C++ Project Micro-controller selection dialog box

3. Adjust the project properties to include the following:

- Adjust the Allocation >> Threshold for putting data in `_near to 0`
- Set generate symbolic information to `'default'`
- Generate ascii .map file

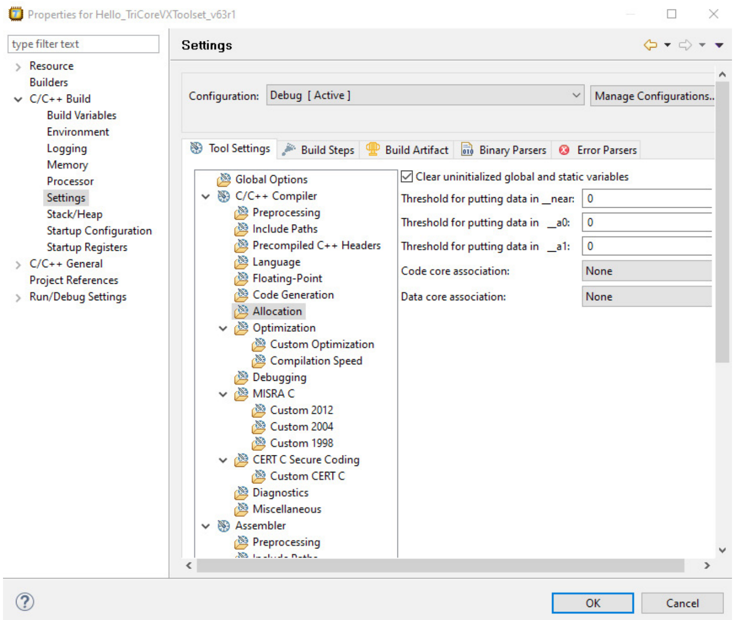


Figure 4: VX-Toolset for TriCore, New C/C++ Project Settings - memory allocation

4. Modify the Hello_TriCoreVXToolset_v63r1 as shown:

```
#include <stdio.h>

extern int libfunc1(void);
extern int libfunc2(void);
extern int libfunc3(void);

extern int lib1, lib2,lib3;
int sumext;

int main_6_3(void)
{
    printf( "Hello world\n" );
    printf("This was built with TriCore VX Toolset v63r1\n");
    sumext=lib1+lib2+lib3;
    printf("TriCore lib sumext= %d\n",sumext);
    return(1);
}

#if 1
void main(void){
    main_6_3();
}
#endif
```

Figure 5: VX-Toolset for TriCore, Hello_TriCoreVXToolset_v63r1 code modification

5. Create a new TASKING TriCore Library and select the AURIX family variant

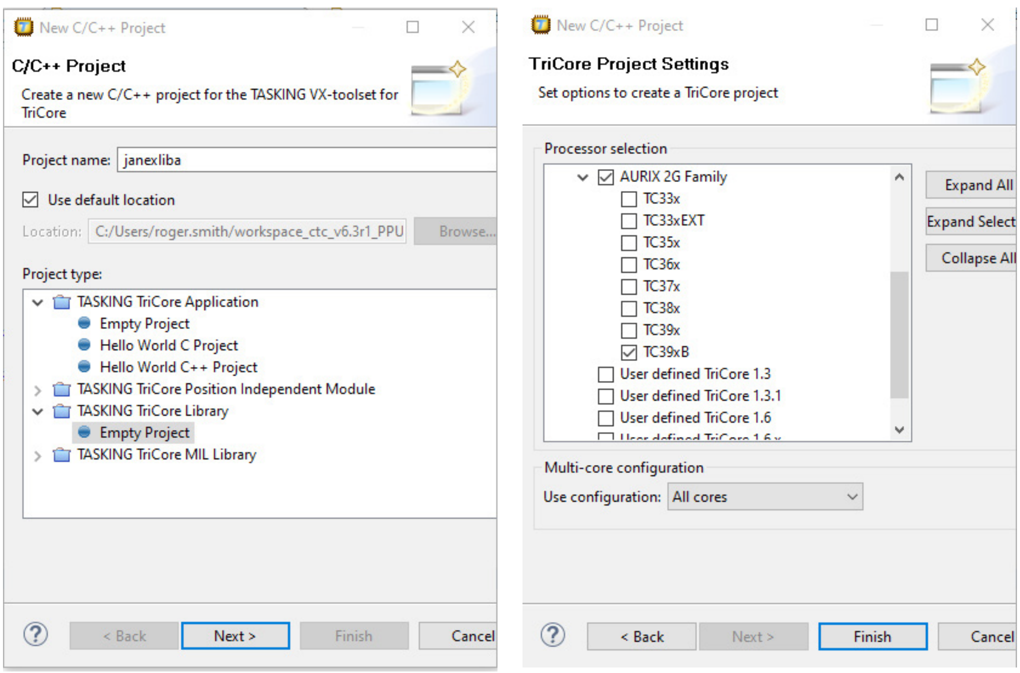


Figure 6: VX-Toolset for TriCore, New Library and micro-controller dialog boxes

6. Create '3' new source files for the library. Right click on the library name in your workspace, select `new`, select `source file`. The three new source files should be named `libfunc1.c`, `libfunc2.c` and `libfunc3.c`

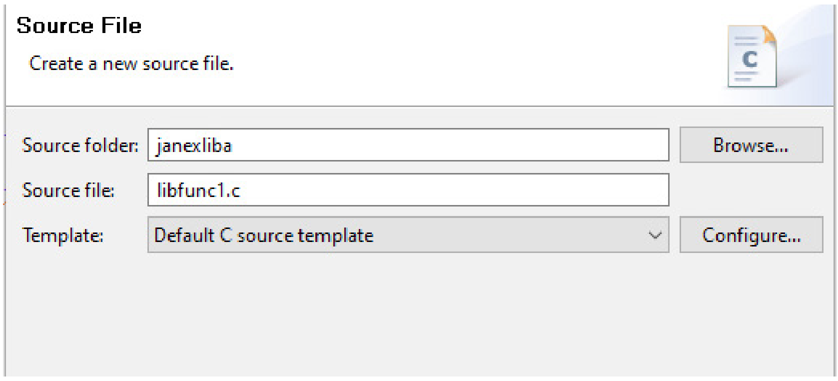


Figure 7: VX-Toolset for TriCore, New Library Source File dialog box

7. Define the contents of the library source files as shown:

```
 * libfunc1.c
#include <stdio.h>
int lib1=1;

int libfunc1(void){
    printf("libfunc1 tricore\n");
    return(lib1);
}
```

```
 * libfunc2.c
#include <stdio.h>
int lib2=2;

int libfunc2(void){
    printf("libfunc2 tricore\n");
    return(lib2);
}
```

```
 * libfunc1.c
#include <stdio.h>
int lib1=1;

int libfunc1(void){
    printf("libfunc1 tricore\n");
    return(lib1);
}
```

Figure 8: VX-Toolset for TriCore, New Library source code

8. Set your library to the active project and build. The console.log and screenshot of project workspace is shown.

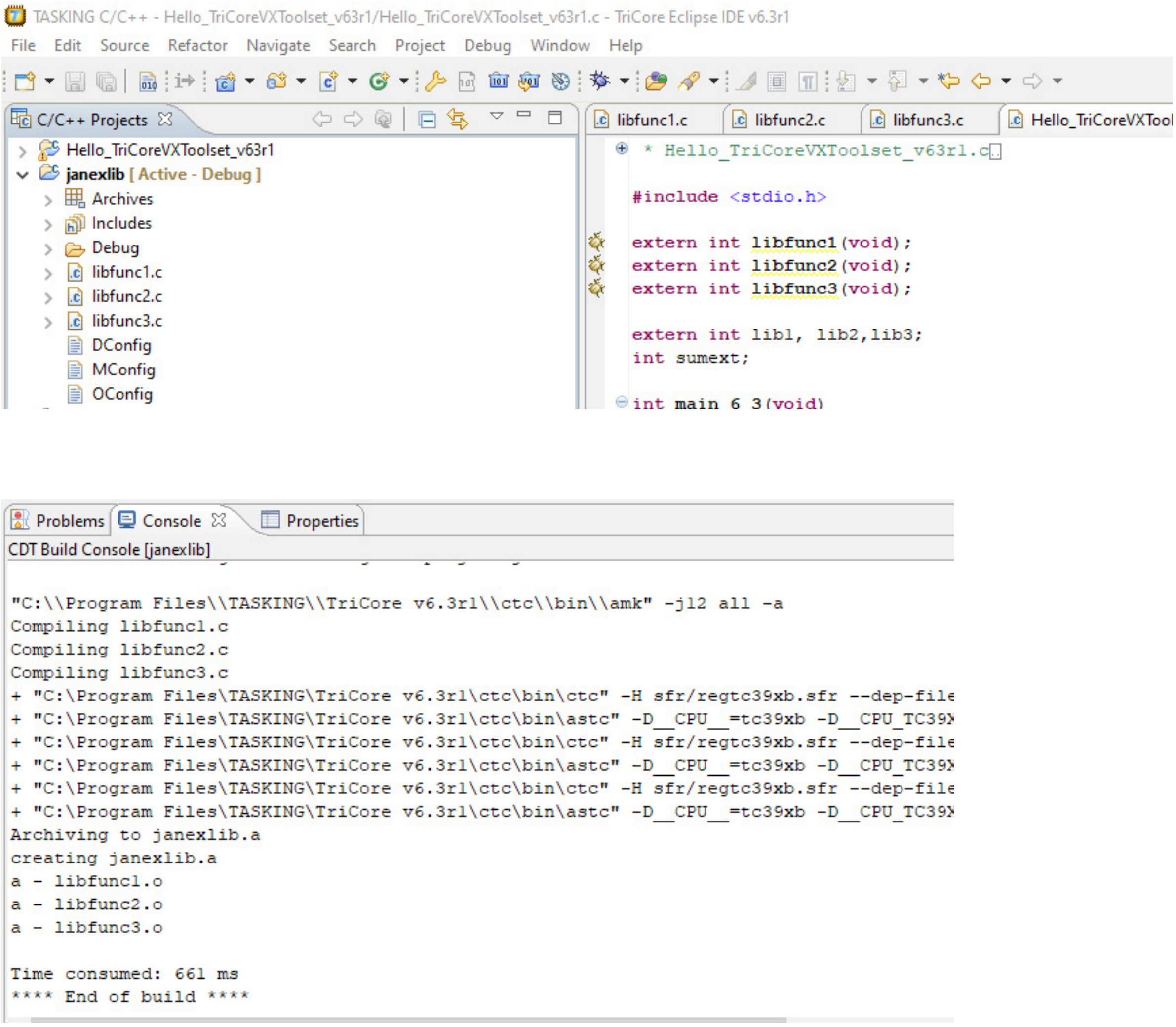


Figure 9: VX-Toolset for TriCore, New Library workspace view and build results

9. Copy the **janexlib.a**, from the "`janexlib >> debug folder`" -> denoted by green to the "`Hello_TriCoreVXToolset_v63r1 main project folder`" denoted by light blue.

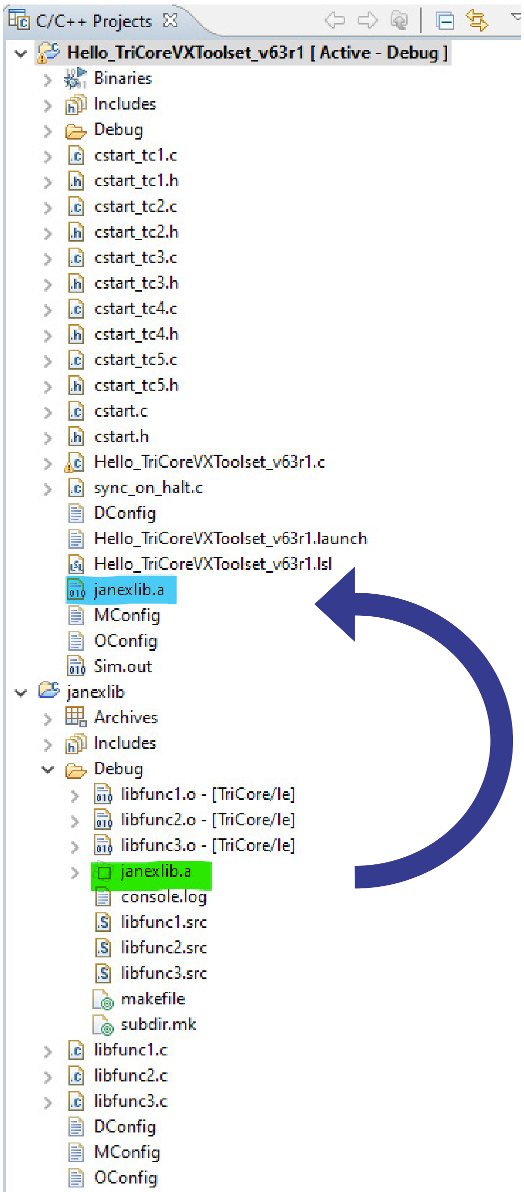


Figure 10: VX-Toolset for TriCore, New C/C++ Project and Library workspace view

10. In the project property settings, add the library, `janexlib.a` as shown:

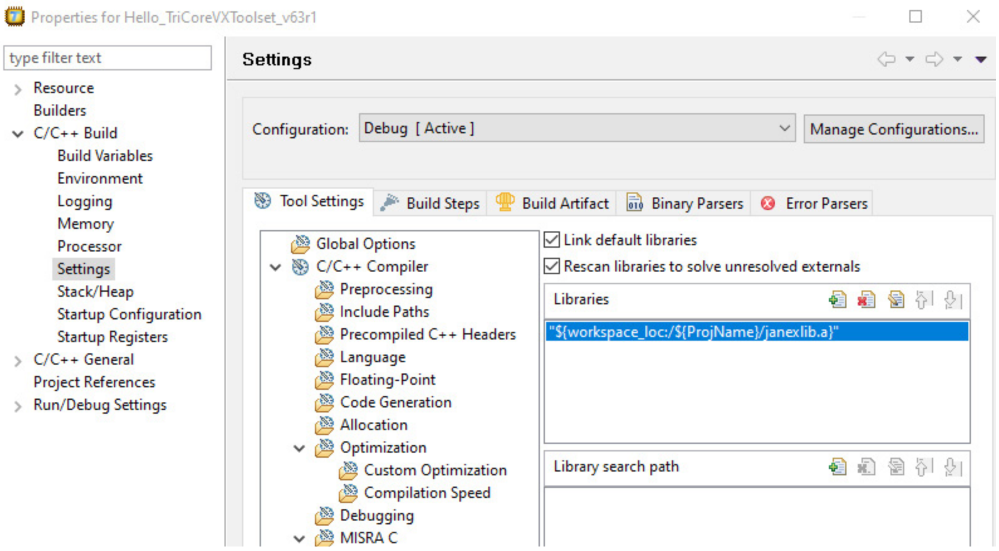


Figure 11: VX-Toolset for TriCore, Add Library to Project dialog box

11. Set `Hello_TriCoreVXToolset_v63r1 project` active, build and debug using the TriCore Simulator. From the debugger, hit `'restart'` and `'resume'`

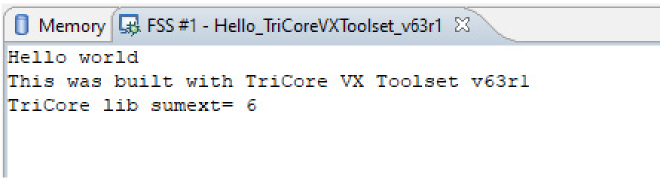


Figure 12: VX-Toolset for TriCore, Program output

12. Create a SmartCode project. Follow the same procedure as above except name the project `'SmartCode_helloworld'` and choose the `TC49x` (This is currently the only TC4x option available).

13. Modify the `SmartCode_helloworld.c` as shown:

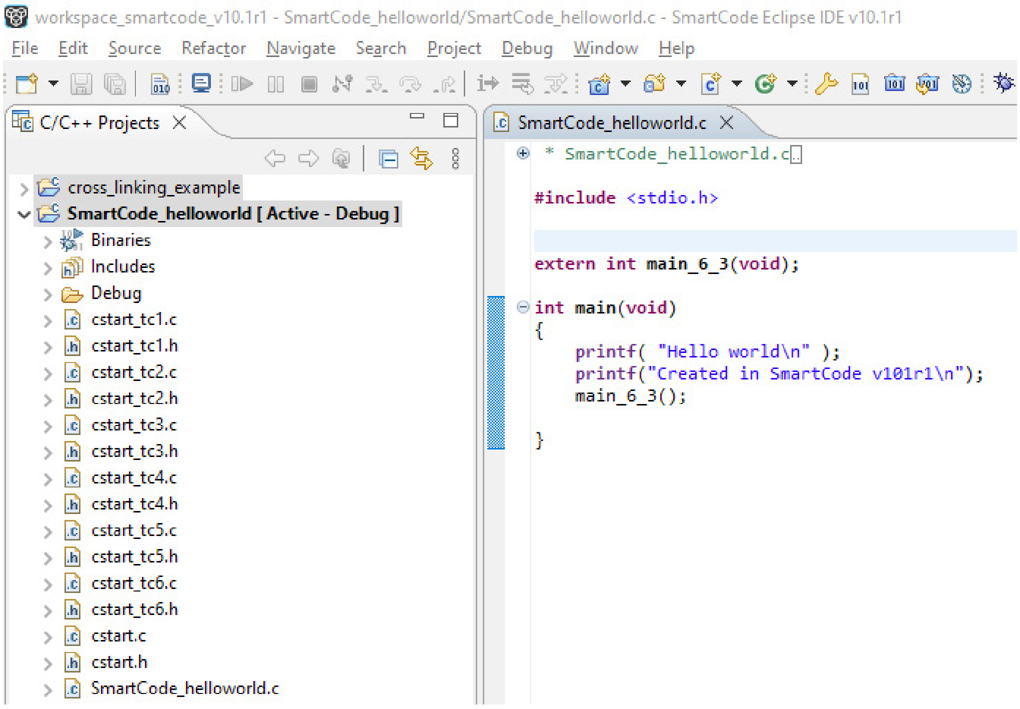


Figure 13: SmartCode, New Project workspace view

14. One option is to modify the TriCore Project since the TASKING Linker will not be able to create the SmartCode .elf file with '2' main functions defined. Modify the TriCore source file as shown.

```
 * Hello_TriCoreVXToolset_v63r1.c

#include <stdio.h>

extern int libfunc1(void);
extern int libfunc2(void);
extern int libfunc3(void);

extern int lib1, lib2,lib3;
int sumext;

int main_6_3(void)
{
    printf( "Hello world\n" );
    printf("This was built with TriCore VX Toolset v63r1\n");
    sumext=lib1+lib2+lib3;
    printf("TriCore lib sumext= %d\n",sumext);
    return(1);
}

#if 0
void main(void){
    main_6_3();
}
#endif
```

Figure 14: VX-Toolset for TriCore code modification

15. Re-build the TriCore Project. The linker error is due to the lack of a defined main().
    For cross-linking, copy `'Hello_TriCoreVXToolset_v63r1.o'` and the `janexlib.a` to the SmartCode Project folder.
    Don't forget to add the **janexlib.a library reference** as shown in **step 10**.

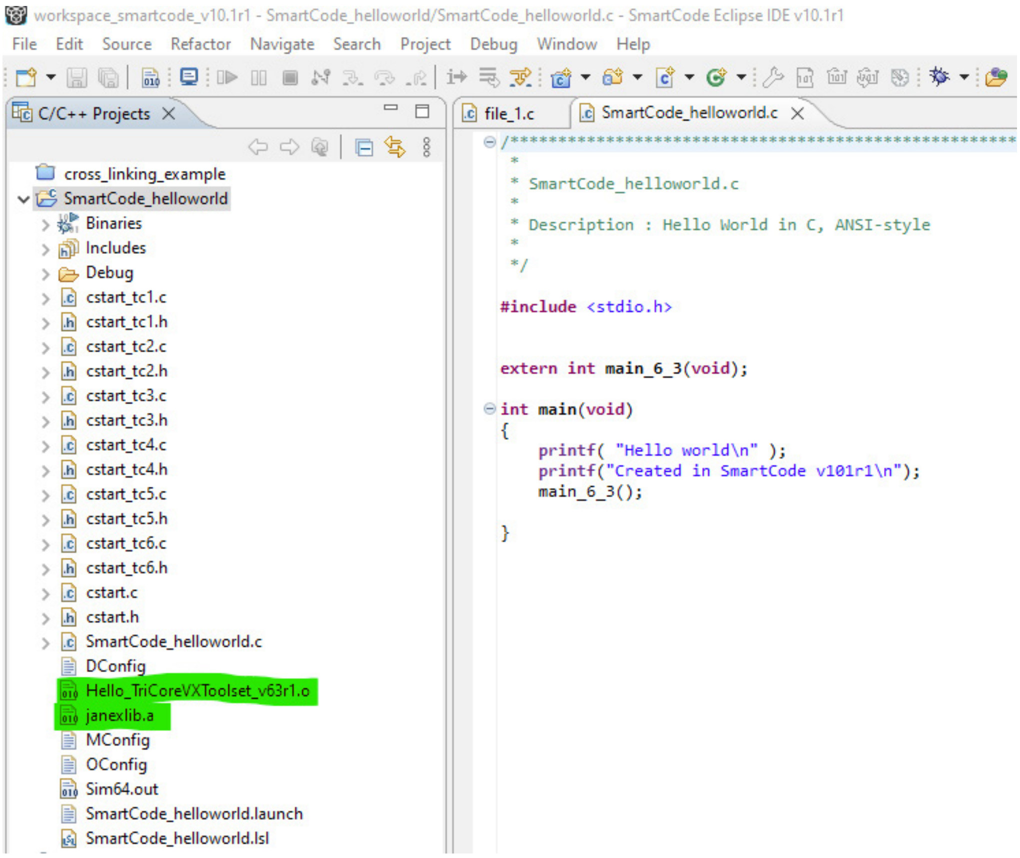


Figure 15: SmartCode project workspace view after VX-Toolset for TriCore „.o" and Library „.a" files are added

16. Build the SmartCode Project. You will receive the following warnings since the TriCore project is based on the TC39xB(TC1V1.6.2) and the SmartCode project is for the TC49x(TC1V1.8)

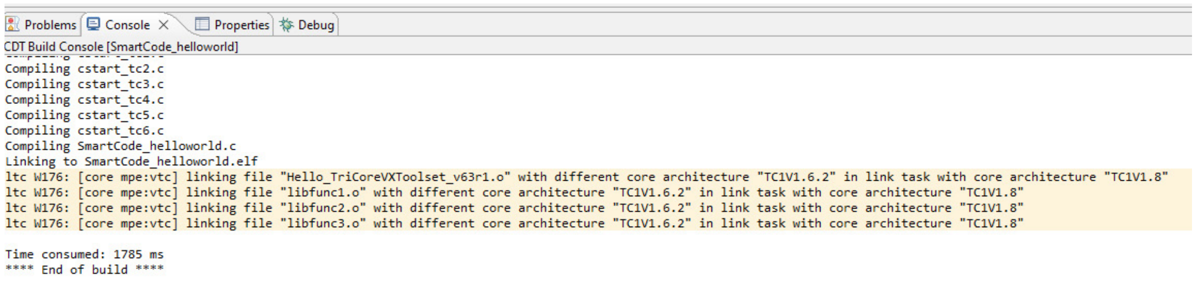


Figure 16: SmartCode Project Build Results

17. Run the SmartCode project with the simulator. (Hardware is not available)

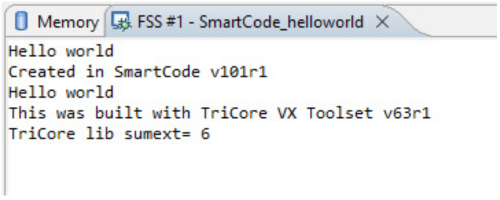


Figure 17: SmartCode and VX-Toolset for TriCore cross-linked program output

You have successfully cross-linked a TriCore VX Toolset project with a library -AND- a SmartCode Project.

This project was very simple and didn't introduce any of the potential stumbling blocks that will be shown in the next example.

DEMONSTRATE CHALLENGES WHEN CROSS-LINKING TRICORE VX AND SMARTCODE

When cross-linking TriCore VX v6.3r1 and SmartCode binaries, special attention needs to be paid with the following.

- Float Point Model: The TriCore TC1.6.x has an integrated Floating-Point Unit (FPU) that supports single precision floating point operations. The TriCore VX toolset offers the compiler option 'treat double as float' which significantly improves performance if true double precision is not required. The SmartCode toolset was developed to support the TC4x which has a double precision FPU. In this use-case, there is the possibility to have a floating-point mismatch. (Double Precision -> 8bytes, Singe Precision -> 4bytes)
- Memory Alignment: The TriCore VX Toolset can specify the alignment of data types in memory. For example, with the --eabi option, halfword memory alignment is possible. The SmartCode toolset had this feature removed and certain data types are always EABI compliant (I.e, int's are always full word aligned, no exceptions).

Note: With `'Global Type Checking'` enabled, the toolset will give an error with any memory misalignments.

Please reference ***project 220223_cross_linking_example***.

1. Copy the project file to a folder and unzip.

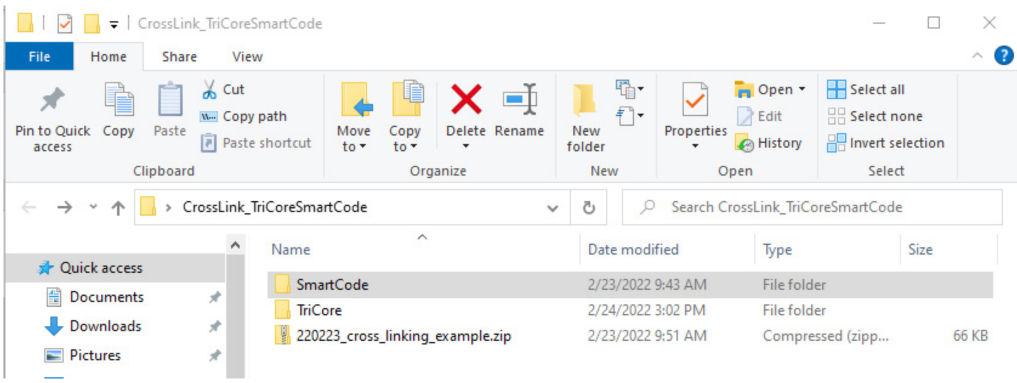


Figure 18: Cross-link example project 220223

2. Import the SmartCode Project. Select File >> Import.
   (Enabling `'copy projects into workspace'` will leave original project files untouched.)

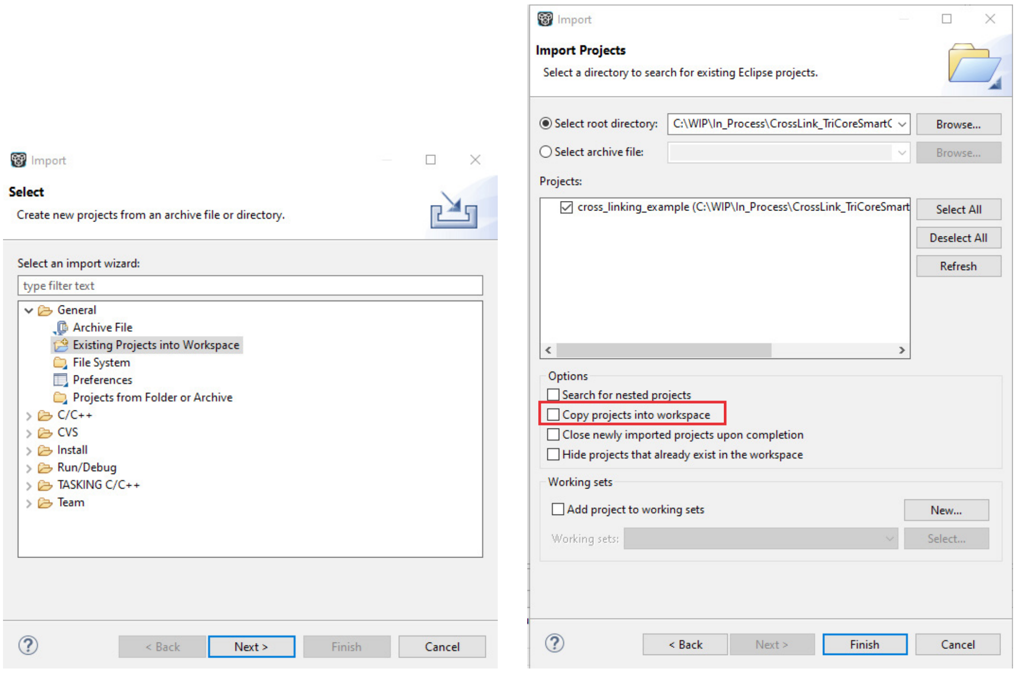


Figure 19: SmartCode, Import Project dialog boxes

3. In the TriCore Folder, rename `gen.txt` -> `gen.bat`

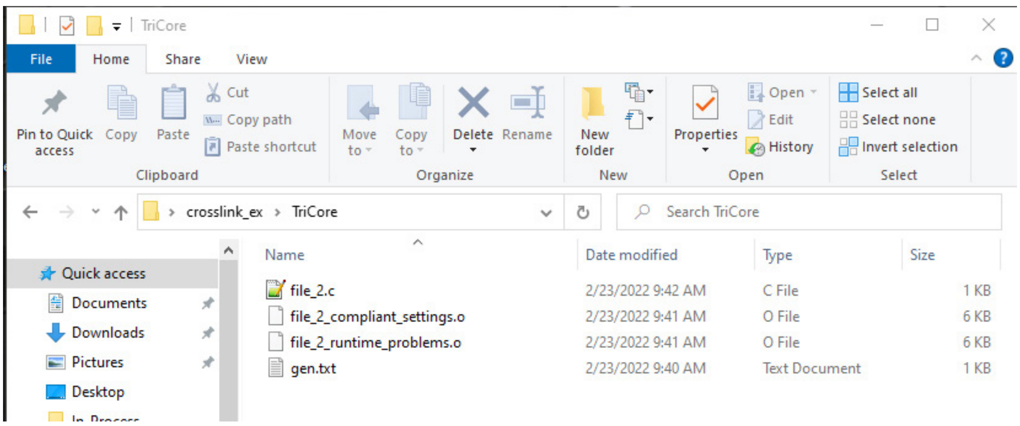


Figure 20: VX-Toolset for TriCore, „o' file directory view

The **gen.bat file** contains the command line invocation calls for the TriCore VX Toolset. This batch file will generate two separate object files. The `'file_2_compliant_settings.o'` is built with the correct EABI and floating-point double settings. The `'file_2_runtime_problems.o'` is built with `'treat double as float'` and `'half-word'` alignment as supported by the TriCore Toolset.

```
C:\WIP\In_Process\CrossLink_TriCoreSmartCode\TriCore\gen.bat - Notepad++
cctc file_2.c -t --core=tc1.6.2 -t -c -g -v --global-type-checking -o file_2_runtime_problems.o
cctc file_2.c -t --core=tc1.6.2 -t -c -g -v --eabi=H --fp-model=F --global-type-checking -o file_2_compliant_settings.o
```

Figure 21: Command Line invocation for VX-Toolset for TriCore ‚.o' files

- Open a command prompt (`cmd/k`) and set the path to include the TriCore VX Toolset v6.3r1 ctc\bin folder. A simple batch file is shown below.

```
set temppath=%path%
set path=C:\program files\tasking\TriCore v6.3r1\ctc\bin;%temppath%
```

Figure 22: VX-Toolset for Tricore, batch file for setting path environment variable

- From the command prompt, type `cctc -help`, to show the control program options.

```
C:\Windows\System32\cmd.exe
                                  diagnostic messages using the specified
                                  format: html, rtf or text (default)
      --eabi=<flag>,...           control EABI compliancy (default: bcfhnsw)
                                  lowercase/+ enables, uppercase/- disables:
        b/B +/-bitfield-align       base type alignment for 0-size bit-field
        c/C +/-char-bitfield        access char bit-field as a byte
        f/F +/-float                allow treating 'double' as 'float'
        h/H +/-half-word-align      allow half-word alignment
        n/N +/-no-clear             allow the use of option --no-clear
        s/S +/-structure-return     allow structure-return optimization
        w/W +/-word-struct-align    word align structures and unions larger
                                    than or equal to 64-bit
      --eabi-compliant            the code generated will adher to the EABI
                                  whereever possible
      --error-file                redirect all diagnostic messages
      --error-limit=<number>      set maximum number of errors (default: 42)
      --exceptions                enable C++ exception handling
      --force-c                   treat C++ files as C files
      --force-c++                 force C++ compilation and linking
      --force-munch               force invocation of C++ muncher
      --format=<type>             set linker output format [ihex|srec]
      --fp-model[=<flag>,...]     floating-point model (default: cflnrSTz)
        0                           alias for --fp-model=CFLNRStZ (strict)
        1                           alias for --fp-model=cFLNRSTZ (precise)
        2                           alias for --fp-model=cFlnrSTz (fast-dp)
        3                           alias for --fp-model=cflnrSTz (fast-sp)
                                  lowercase/+ enables, uppercase/- disables:
        c/C +/-contract             allow expression contraction
        f/F +/-float                treat 'double' as 'float'
        l/L +/-fastlib              allow less precise library functions
        n/N +/-nonan                allow optimizations to ignore NaN/Inf
        r/R +/-rewrite              allow expression rewriting
        s/S +/-soft                 use software floating point library
        t/T +/-trap                 support trapping on exceptions
        z/Z +/-negzero              ignore sign of -0.0
      --global-type-checking      enable global type checking
      --ignore-default-library-path ignore the default search path for
```

Figure 23: VX-Toolset for Tricore, build program ‚EABI' related options

--eabi = H -> Removes half-word align option.

--fp-model = F -> Removes treat double as a float.

Note: The TriCore EABI default is: `bcfhnsw` (The `'f'` treats `'double as a float'` -and- the `'h'` allows half-word alignment.)

- Run the batch file to generate updated object files

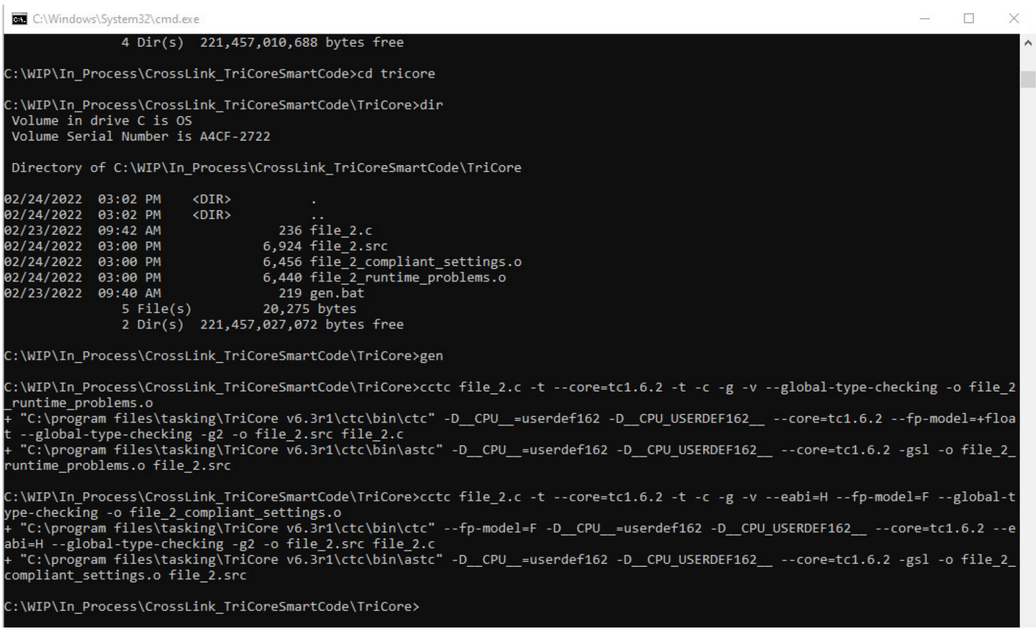


Figure 24: VX-Toolset for TriCore, build results generating new ‚.o' files

- Copy the `'file_2_runtime_problems.o'` to the SmartCode project, build the project and launch the debugger. Note the linker warning message regarding the different TriCore Versions in the console window.

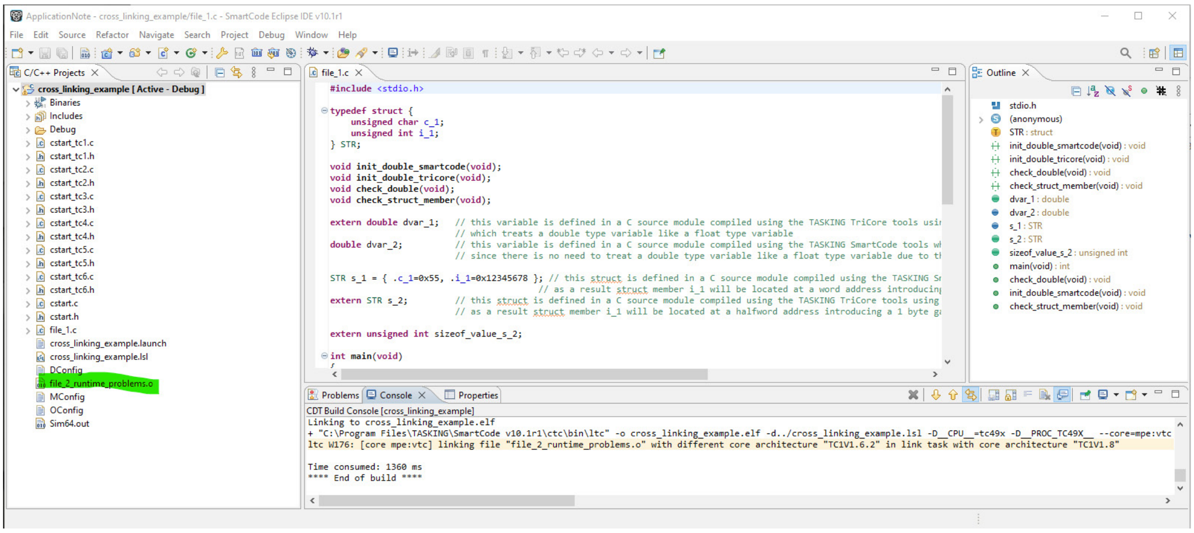


Figure 25: VX-Toolset for TriCore, project workspace view

Note: When you add an object file to an eclipse project folder, it is treated as an input file, not an intermediate file. Eclipse adds to linker invocation. No need to do a refresh.

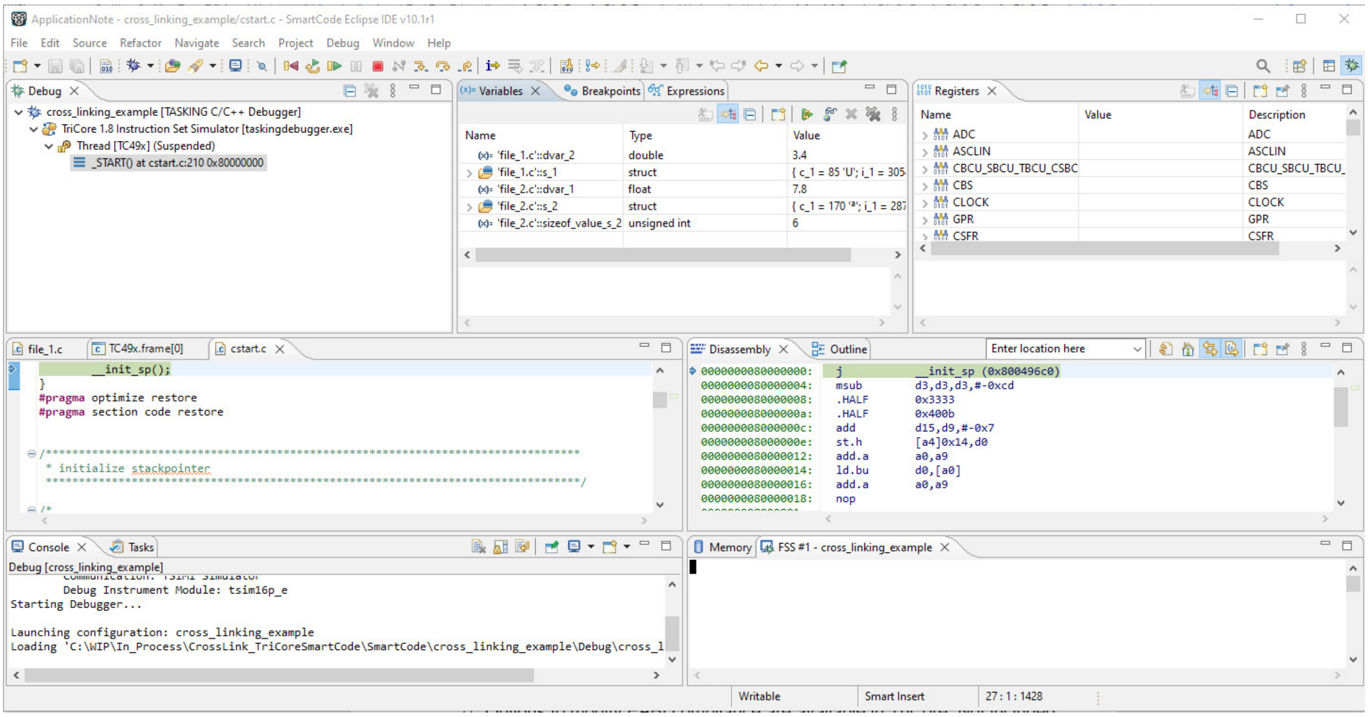


Figure 26: Cross-linked SmartCode and TriCore project shown in SmartCode debug perspective

NON-COMPLIANT PORTION OF THE EXAMPLE

The TriCore source file, **file_2.c** is shown below:

```
typedef struct {
    unsigned char c_1;
    unsigned int i_1;
} STR;

STR s_2 = { .c_1=0xAA, .i_1=0x1122334

unsigned int sizeof_value_s_2 = sizec
double dvar_1;

void init_double_tricore(void)
```

During the build process the TriCore toolset ignores the `'double'` definition of dvar_1 in the c source and creates `'float'`. (Based on command line settings)

- Looking at the TriCore code in Debugger (`'treat double as float'` enabled)

```
void init_double_tricore(void)
{
    dvar_1 = 7.8;
}
```

```
                         dvar_1 = 7.8;
00000080049922:   ld.w        d15,.1.cnt (0x8000000
00000080049926:   st.w        dvar_1 (0x9000001c),d
                  }
```

The Disassembly view shows `ld.w` and `st.w` (load and store word)

- Referring to the .map file,

Link Result

[in] File	[in] Section	[in] Size (MAU)	[out] Offset	[out] Section	[out] Size (MAU)

Chip	Group	Section	Size (MAU)	Space addr	Chip addr	Alignment
mpe:pflash00		.zrodata.file_1..11.cnt (191)	0x00000008	0x80000004	0x00000004	0x00000004
mpe:pflash00		.zrodata.file_2..1.cnt (240)	0x00000004	0x8000000c	0x0000000c	0x00000002
mpe:dlmucpu0		.zdata.file_1.s_1 (197)	0x00000008	0x90000000	0x0	0x00000004
mpe:dlmucpu0		.zdata.file_2.s_2 (242)	0x00000006	0x90000008	0x00000008	0x00000002
mpe:dlmucpu0		.zdata.file_2.sizeof_value_s_2 (243)	0x00000004	0x9000000e	0x0000000e	0x00000002
mpe:dlmucpu0		.zbss.file_1.dvar_2 (196)	0x00000008	0x90000014	0x00000014	0x00000004
mpe:dlmucpu0		.zbss.file_2.dvar_1 (244)	0x00000004	0x9000001c	0x0000001c	0x00000002

dvar_1 is 4 bytes (Float), half-word aligned and located at address 0x9000 001c

The variable and memory windows are shown:

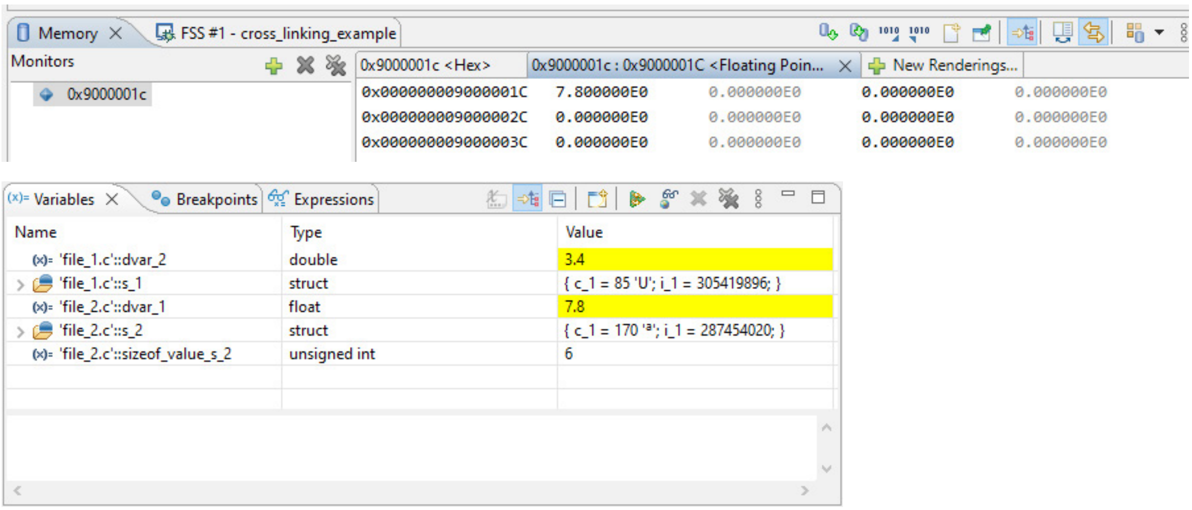


Figure 27: SmartCode, Debug Perspective showing global variables

The debugger correctly displays dvar_1 as type ***float*** with a value of ***7.8***. The problem will be highlighted below when we print out the variables.

file_1.c is built with SmartCode. (Treats `'double'` as a `'double'` and type int is word-aligned)

```c
#include <stdio.h>

typedef struct {
    unsigned char c_1;
    unsigned int i_1;
} STR;

void init_double_smartcode(void);
void init_double_tricore(void);
void check_double(void);
void check_struct_member(void);

extern double dvar_1;   // this variable is defined in a C source module compiled using the TASKING TriCore tools using option --fp-model=+float
                        // which treats a double type variable like a float type variable
double dvar_2;          // this variable is defined in a C source module compiled using the TASKING SmartCode tools which do not support the option --fp-model=+float
                        // since there is no need to treat a double type variable like a float type variable due to the existence of a hardware FPU unit which supports double precision too

STR s_1 = { .c_1=0x55, .i_1=0x12345678 }; // this struct is defined in a C source module compiled using the TASKING SmartCode tools which do not support option --eabi=+half-word-align
                                          // as a result struct member i_1 will be located at a word address introducing a 3 byte gap following struct member c_1
extern STR s_2;         // this struct is defined in a C source module compiled using the TASKING TriCore tools using option --eabi=+half-word-align
                        // as a result struct member i_1 will be located at a halfword address introducing a 1 byte gap following struct member c_1

extern unsigned int sizeof_value_s_2;

int main(void)
{
    check_double();
    check_struct_member();
    return 0;
}

void check_double(void)
{
    float *fp_1 = (float *)&dvar_1;

    init_double_smartcode();
    init_double_tricore();      // this variable is defined in a C source module compiled using the TASKING TriCore tools using option --fp-model=+float
                                // which treats a double type variable like a float type variable

    printf("The expected double value of dvar_1 is 7.8. The value read is  : %e\n\n", dvar_1);
    printf("Printing the float value of dvar_1 instead of the double value : %f\n\n", *fp_1);

    printf("The expected value of dvar_2 is 3.4. The value read is         : %e\n\n", dvar_2);
}

void init_double_smartcode(void)
{
    dvar_2 = 3.4;
}

void check_struct_member(void)
{
    printf("s_1.c_1 value is: 0x%x\n\n", s_1.c_1);
    printf("s_1.i_1 value is: 0x%x\n\n", s_1.i_1);
    printf("sizeof(s_1) is  : %d\n\n",sizeof(s_1));
    printf("s_2.c_1 value is: 0x%x\n\n", s_2.c_1);
    printf("s_2.i_1 value is: 0x%x\n\n", s_2.i_1);
    printf("sizeof(s_2) is from SmartCode view : 0x%d\n\n",sizeof(s_2));
    printf("sizeof(s_2) is from TriCore view   : 0x%d\n\n",sizeof_value_s_2);
}
```

- The Check_double() function uses a pointer to the variable &dvar_1 (defined in TriCore) allowing you to `'cast'` a double value to a float value.
  - The function init_double_smartcode(), defines a constant value of 3.4 which is shown as ld.d and st.d (load and store double) in the disassembly view. (8 bytes for the value 3.4)

```
void init_double_smartcode(void)
{
    dvar_2 = 3.4;
}
```

```
48                          dvar_2 = 3.4;
000000008004990c:   ld.d        d0/d1,0x80000004
0000000080049910:   st.d        dvar_2 (0x90000014),d0/d1
49                  }
```

  - The function init_double_tricore() was already discussed.
  - The printf() calls display: dvar_1, pointer to dvar_1 and dvar_2.

```
    printf("The expected double value of dvar_1 is 7.8. The value read is  : %e\n\n", dvar_1);
    printf("Printing the float value of dvar_1 instead of the double value : %f\n\n", *fp_1);

    printf("The expected value of dvar_2 is 3.4. The value read is         : %e\n\n", dvar_2);
}
```

```
The expected double value of dvar_1 is 7.8. The value read is  : 5.385808e-315

Printing the float value of dvar_1 instead of the double value : 7.800000

The expected value of dvar_2 is 3.4. The value read is         : 3.400000e+00
```

- The first printf call displays dvar1 with the `'%e'` conversion character. Per the TASKING SmartCode user manual, this represents `'type double'`.

Character	Printed as
d, i	int, signed decimal
o	int, unsigned octal
x, X	int, unsigned hexadecimal in lowercase or uppercase respectively
u	int, unsigned decimal
c	int, single character (converted to unsigned char)
s	char *, the characters from the string are printed until a NULL character is found. When the given precision is met before, printing will also stop
f, F	double
e, E	double
g, G	double
a, A	double
n	int *, the number of characters written so far is written into the argument. This should be a pointer to an integer in default memory. No value is printed.
p	pointer
r, lr	__fract, __lfract
R, lR	__accum, __laccum
%	No argument is converted, a '%' is printed.

printf conversion characters

Figure 28: SmartCode, printf data format options

From the disassembly view, notice printf() loads a double. [printf() gets 8bytes, but there are only 4 bytes of data!]

```
40                       printf("The expected double value of dvar_1 is 7.8. The value read is  : %e\n\n", dvar_1);
0000000080049850:    ld.d        d0/d1,dvar_1 (0x9000001c)
0000000080049854:    st.d        [sp]0x0,d0/d1
0000000080049858:    movh.a      a4,#0x8005
000000008004985c:    lea         a4,[a4]-0x648d
0000000080049860:    call        printf (0x80049a30)
```

Let's take a closer look at the math. As previously mentioned, the memory address of dvar_1 is 0x9000 001c.

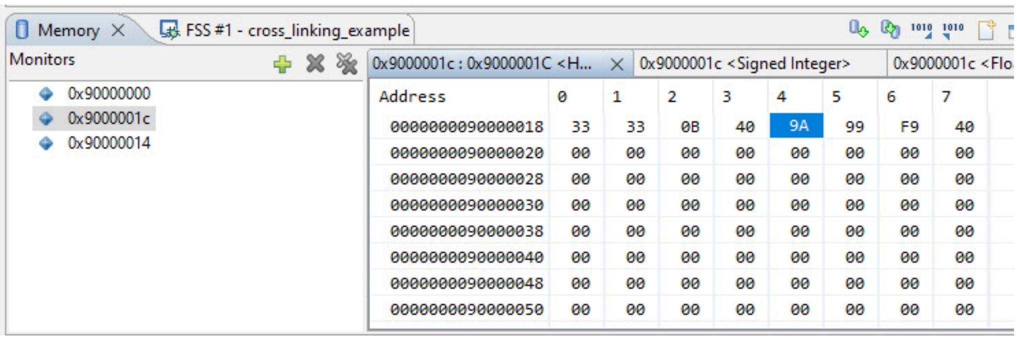


Figure 29: SmartCode, Debug Perspective memory watch

The hex integer value (32-bit) of 0x9000 001c = 40F9 999A

Converting this value to Float (32-bit) and Double (64-bit) ->

	HEX	4 0 F 9 9 9 9 A
Convert to Float	Binary	0 1 0 0 0 0 0 0 1 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0
	Sign	0
	Exponent	1 0 0 0 0 0 0 1
	Mantissa	1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0
	Sign	+
	Exponent	$1x2^7+1x2^0 = 129$
	Mantissa	$1x2^{-1}+1x2^{-2}+1x2^{-3}+1x2^{-6}+..+1x2^{-21}$= 1.95000004768
		$+1 * 2^{(129-127)}$ * 1.9500000476837158 = 7.8
Convert to Double	Binary	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0
	Sign	0
	Exponent	0 0 0 0 0 0 0 0 0 0 0
		0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0
	Sign	+
	Exponent	0 - 1023 = -1023
	Mantissa	0.0000004841014745
		$+1 * 2^{(-1023)}$ * 0.0000004841014745 = 5.3858076800000000e-315

Figure 30: Procedure for converting INT to FLOAT and DOUBLE

- The second printf() displays the contents of a pointer. With type casting we reference the address of a double but show it as a float. Thus, the correct result.
- The third printf() shows the value in SmartCode which is always a double. Per the .map file, the address of dvar_2 is 0x9000 0014.

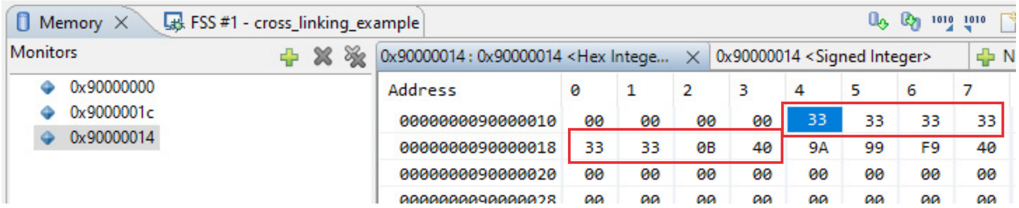


Figure 31: SmartCode, Debug Perspective memory watch

The hex integer value (64-bit) of 0x9000 0014 = 400B 3333 3333 3333

	HEX	4	0	0	B	3	3	3	3	3	3	3	3	3	3	3	3
Convert to Double	Binary	0100	0000	0000	1011	0011	0011	0011	0011	0011	0011	0011	0011	0011	0011	0011	0011
	Sign	0															
	Exponent	100	0000	0000													
					1011	0011	0011	0011	0011	0011	0011	0011	0011	0011	0011	0011	0011
	Sign	+															
	Exponent	1024 - 1023 = 1															
	Mantissa				1.1011001100110011001100110011001100110011001100110011												
		$+1 * 2^{(1)} * 1.7 = 3.4$															

Figure 32: Procedure for converting INT to DOUBLE

- The Check_struct_member() function shows how the struct member alignment affects the results:

```
void check_struct_member(void)
{
    printf("s_1.c_1 value is: 0x%x\n\n", s_1.c_1);
    printf("s_1.i_1 value is: 0x%x\n\n", s_1.i_1);
    printf("sizeof(s_1) is  : %d\n\n",sizeof(s_1));
    printf("s_2.c_1 value is: 0x%x\n\n", s_2.c_1);
    printf("s_2.i_1 value is: 0x%x\n\n", s_2.i_1);
    printf("sizeof(s_2) is from SmartCode view : 0x%d\n\n",sizeof(s_2));
    printf("sizeof(s_2) is from TriCore view   : 0x%d\n\n",sizeof_value_s_2);
}
```

Note: *s_1* is built with SmartCode and s_2 is built with TriCore. The printf() formatter %x -> hex display data (refer to table in user guide shown above)

The TriCore initialized struct members:

```
typedef struct {
    unsigned char c_1;
    unsigned int i_1;
} STR;

STR s_2 = { .c_1=0xAA,  .i_1=0x11223344 };

unsigned int sizeof_value_s_2 = sizeof(s_2);
double dvar_1;

void init_double_tricore(void)
{
    dvar_1 = 7.8;
}
```

The TriCore initialized struct members:

```
#include <stdio.h>

typedef struct {
    unsigned char c_1;
    unsigned int i_1;
} STR;

STR s_1 = { .c_1=0x55, .i_1=0x12345678 };
                                      //
```

Per the .map file, s_1 is located at 0x9000 0000 and s_2 is located at 0x9000 0008

Chip	Group	Section	Size (MAU)	Space addr	Chip addr	Alignment
mpe:pflash00		.zrodata.file_1..11.cnt (191)	0x00000008	0x80000004	0x00000004	0x00000004
mpe:pflash00		.zrodata.file_2..1.cnt (240)	0x00000004	0x8000000c	0x0000000c	0x00000002
mpe:dlmucpu0		.zdata.file_1.s_1 (197)	0x00000008	0x90000000	0x0	0x00000004
mpe:dlmucpu0		.zdata.file_2.s_2 (242)	0x00000006	0x90000008	0x00000008	0x00000002
mpe:dlmucpu0		.zdata.file_2.sizeof_value_s_2 (243)	0x00000004	0x9000000e	0x0000000e	0x00000002
mpe:dlmucpu0		.zbss.file_1.dvar_2 (196)	0x00000008	0x90000014	0x00000014	0x00000004
mpe:dlmucpu0		.zbss.file_2.dvar_1 (244)	0x00000004	0x9000001c	0x0000001c	0x00000002

Looking at the memory location 0x9000 0000 and 0x9000 0008:

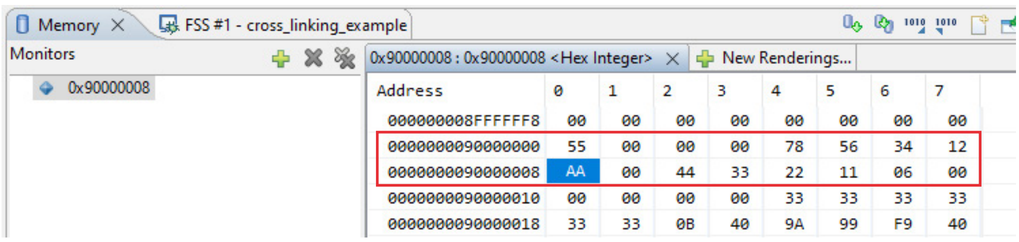


Figure 33: SmartCode, Debug Perspective memory watch

The output of check_struct_member():

```
s_1.c_1 value is: 0x55
s_1.i_1 value is: 0x12345678
sizeof(s_1) is  : 8
s_2.c_1 value is: 0xaa
s_2.i_1 value is: 0x61122
sizeof(s_2) is from SmartCode view : 0x8
sizeof(s_2) is from TriCore view   : 0x6
```

- SmartCode Results -> `'s_1'`:
  - SmartCode is word aligned. Therefore, the char 0x55 takes 4 bytes of memory. (1 byte for the 0x55 and 3 gap bytes).
  - The int takes an additional 4 bytes for the 78 56 34 12.
  - The size of operator yields a correct response of 8.

0x90000000 : 0x90000000 <Hex Integer> | New Renderings...

Address	0	1	2	3	4	5	6	7
0000000090000000	55	00	00	00	78	56	34	12

- SmartCode Results -> `'s_2'`:
  - Per the TriCore build, the struct s_2 is half-word aligned. (memory is saved by reducing the number of alignment gap bytes).
  - The char c_1 starts on a word address. Since it is half-word aligned, the character is followed with a 1-byte gap. The displayed 0xAA is still correct.

0x90000008 : 0x90000008 <Hex Integer> | New Renderings...

Address	0	1	2	3	4	5	6	7
0000000090000008	AA	00	44	33	22	11	06	00

  - For the TriCore *int*, the expected value is 44 33 22 11. However, the value printed is 00 06 11 22.
    · The int member can start on a half-word aligned address since this EABI violation has been enabled in the TriCore project.
    · The 11 22 are shown, but the 33 44 are missing. Why? SmartCode is accessing data not part of the TriCore struct.
    · Based on how the project was built, the TriCore struct is 6-bytes.
    2-bytes for the char (char + 1 gap) and 4-bytes for the int.
    · SmartCode is expecting 8-bytes, so it loads byte7 and byte8 as shown in the purple box. SmartCode includes the 44 and 33 as part of the gap as shown in the green box. Thus, 00 06 11 22

0x90000008 : 0x90000008 <Hex Integer> | New Renderings...

Address	0	1	2	3	4	5	6	7
0000000090000008	AA	00	44	33	22	11	06	00

    · The TriCore sizeof operator reports the length of 6-bytes.

USE COMPLIANT VERSION

1. The SmartCode project will be re-built with '`file_2_compliant_settings.o`', for reference, the following command line innovation was used:

   cctc file_2.c -t --core=tc1.6.2 -t -c -g -v --eabi=H --fp-model=F --global-type-checking -o file_2_compliant_settings.o

   --eabi = H -> Removes half-word align option
   --fp-model = F -> Removes treat double as a float.

   Please note in other use cases --EABI=W is needed to prevent word alignment of structs larger or equal than 64 bit to be SmartCode compliant. The --EABI=B is needed when 0-size bit fields are used and the struct in question is accessed by both TriCore v6.3r1 and SmartCode generated modules.

2. Delete '`file_2_runtime_problems.o`' and replace with '`file_2_compliant_settings.o`'

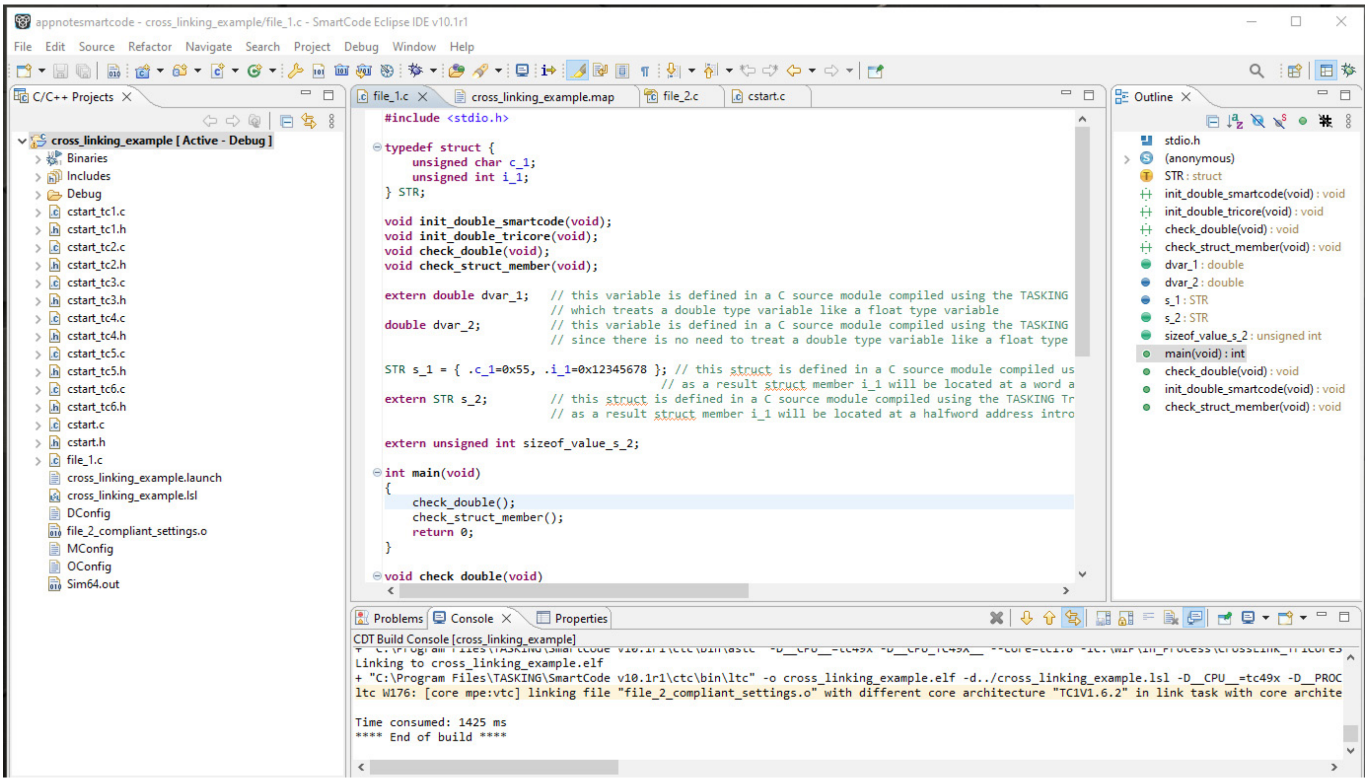


Figure 34: SmartCode, Project Workspace view with new TriCore ‚.o' file

3. Build the new SmartCode project and launch the debugger

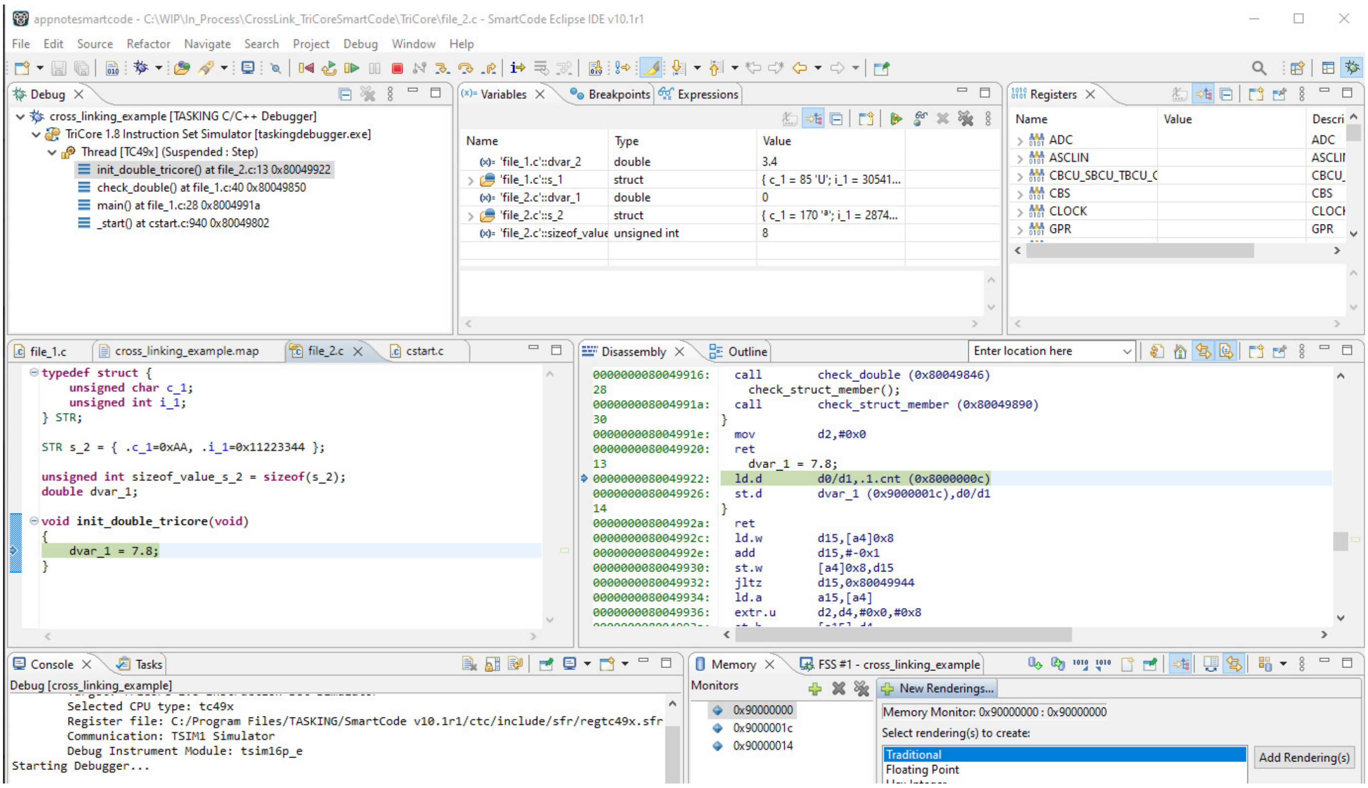


Figure 35: SmartCode, Debug Perspective with compliant version of Cross-linked program

Zooming in on the disassembly view, notice the load and store double with the TriCore defined variable: dvar_1.

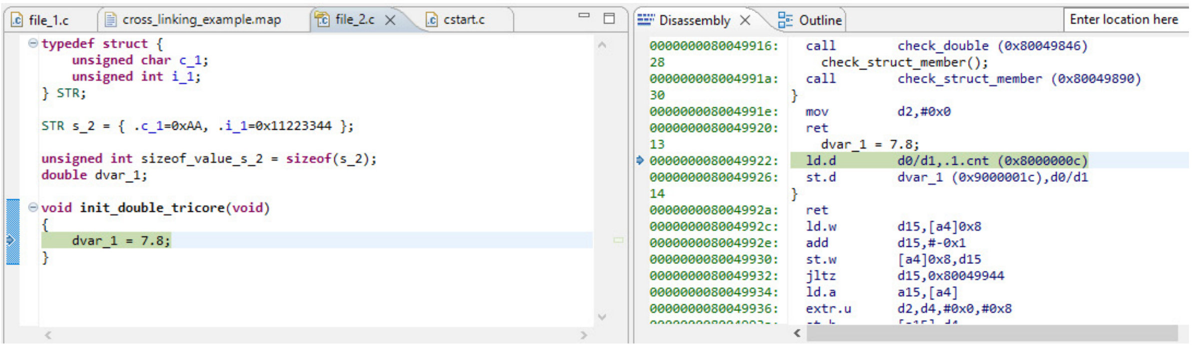
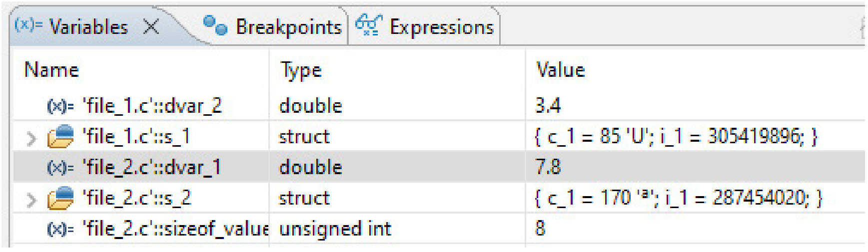


Figure 36: SmartCode, Debug Perspective with compliant version of Cross-linked program, Disassembly view

The following excerpt from the .map file shows the memory location for the key symbols

```
|-------------------------------------------------------|
| dvar_1                | 0x9000001c | mpe:vtc:abs18    |
| dvar_2                | 0x90000014 |                  |
| s_1                   | 0x90000000 |                  |
| s_2                   | 0x90000008 |                  |
| sizeof_value_s_2      | 0x90000010 |                  |
|-------------------------------------------------------|
```

The variable table shows the correct type and value for both dvar_1 and dvar_2.

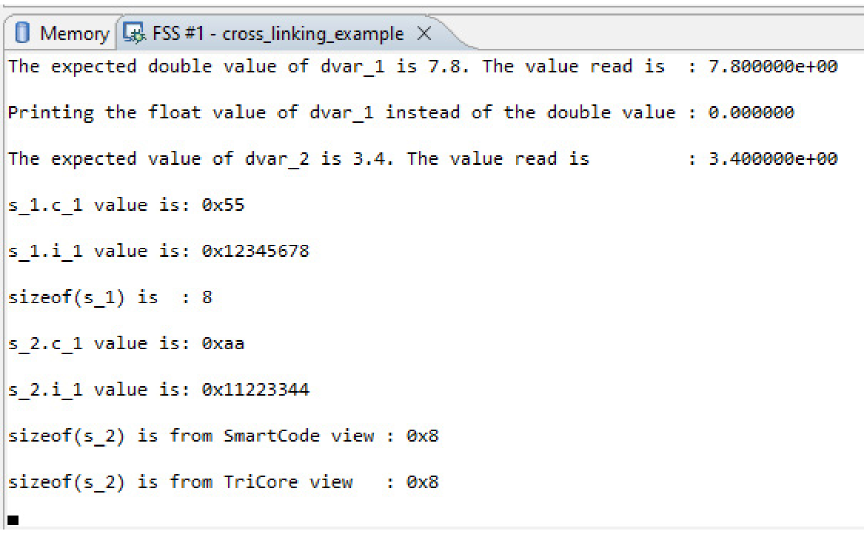


(x)= Variables | Breakpoints | Expressions

Name	Type	Value
'file_1.c'::dvar_2	double	3.4
'file_1.c'::s_1	struct	{ c_1 = 85 'U'; i_1 = 305419896; }
'file_2.c'::dvar_1	double	7.8
'file_2.c'::s_2	struct	{ c_1 = 170 'ª'; i_1 = 287454020; }
'file_2.c'::sizeof_value	unsigned int	8

Figure 37: SmartCode, Debug Perspective variable view

The printf() shows the expected results. Please note that the pointer to a float value will show 0 since the value isn't a float value anymore but a double value. The variable table shows the correct type and value for both dvar_1 and dvar_2.



Memory | FSS #1 - cross_linking_example

```
The expected double value of dvar_1 is 7.8. The value read is  : 7.800000e+00

Printing the float value of dvar_1 instead of the double value : 0.000000

The expected value of dvar_2 is 3.4. The value read is         : 3.400000e+00

s_1.c_1 value is: 0x55

s_1.i_1 value is: 0x12345678

sizeof(s_1) is  : 8

s_2.c_1 value is: 0xaa

s_2.i_1 value is: 0x11223344

sizeof(s_2) is from SmartCode view : 0x8

sizeof(s_2) is from TriCore view   : 0x8
```

The TriCore variables dvar_1 and s_2 are located at 0x9000 001c and 0x9000 0008 respectively.

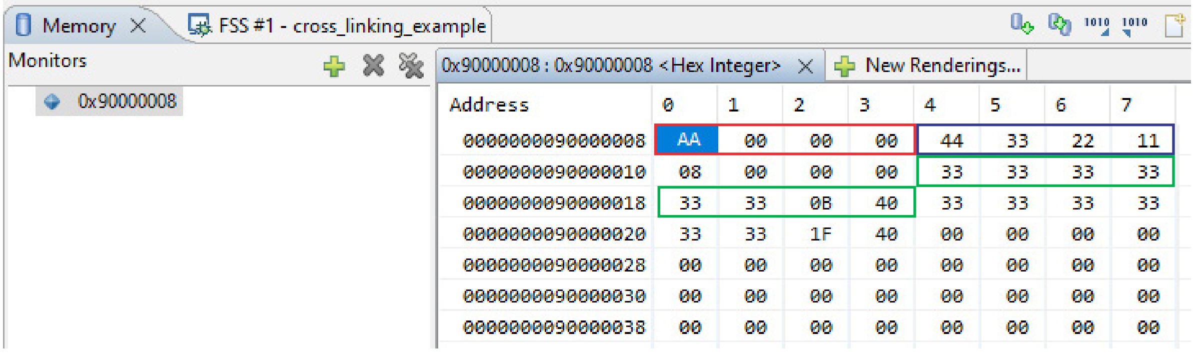


Figure 38: SmartCode, Debug Perspective memory view

- The red box shows the s_2.c_1 with correct size and number of gap bytes
- The violet box shows the correct 4 byte int
- The orange box shows the correct 8 bytes double 40 0B 33 33 33 33 33 33

SUMMARY

In summary, the TASKING TriCore toolset offers many competitive advantages including best-in-class code optimization performance based on their proprietary in-house Viper technology. Additional advantages include advanced multicore support, an integrated debugger, a highly configurable linker with versatile script language for optimal memory control, integrated MISRA C and CERT C static analysis tools, and integration into the popular Eclipse™ platform (IDE).

One advantage that is frequently over-looked is the ability for newer TriCore toolset versions to cross-link legacy object files built with older versions of the toolset. This is a powerful feature which allows the user to easily integrate older trusted software components without the hassles of porting or re-validation.

When semiconductor vendors release new microcontroller's (new family or new variant of an existing family) there is the potential for differences between toolset versions with respect to memory layout. It is imperative that TASKING provide a set of cross-link guidelines to ensure that these differences between toolset versions do not result in erroneous program behavior.

With the introduction of SmartCode, the '--eabi' option was removed, to guarantee that SmartCode generated code is always EABI compliant with the exception of known EABI violations, reported on the TASKING issues portal.

When cross-linking SmartCode with previous versions of the TriCore toolset, users must take special care to ensure that their legacy object files were not built with 'treat double as a float', half-word alignment, base type alignment for 0-size bit field or word-struct-align toolset options. If these options were used, please re-compile the legacy software with the compliant toolset options if data will be exchanged between the TriCore and SmartCode objects.