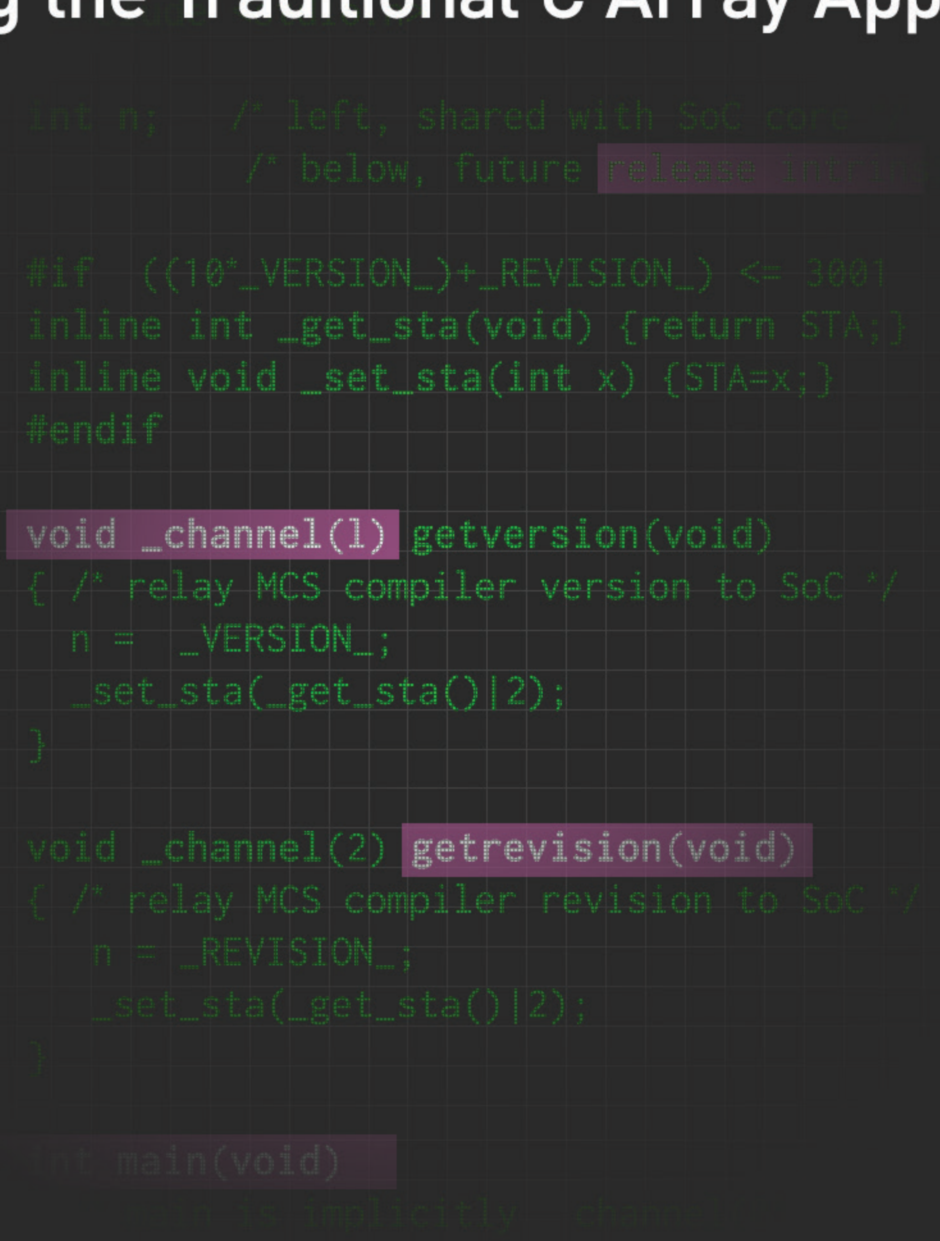
TASKING®

Programming the Bosch® GTM Using the Traditional C Array Approach



```
int n;    /* left, shared with SoC core
          /* below, future release intrin

#if  ((10*_VERSION_)+_REVISION_) <= 3001
inline int _get_sta(void) {return STA;}
inline void _set_sta(int x) {STA=x;}
#endif

void _channel(1) getversion(void)
{ /* relay MCS compiler version to SoC */
  n =  _VERSION_;
  _set_sta(_get_sta()|2);
}

void _channel(2) getrevision(void)
{ /* relay MCS compiler revision to SoC */
   n = _REVISION_;
   _set_sta(_get_sta()|2);
}

int main(void)
       is implicitly  chan
```

Henk-Piet Glas

Technical Product Specialist

PROGRAMMING THE BOSCH® GTM USING THE TRADITIONAL C ARRAY APPROACH

INTRODUCTION

Today's multi-core technology expects toolset vendors to extend their compiler toolset with add-on compilers for specialized 'guest' cores alongside the main CPU. An AURIX™ core, for example, is equipped with multiple TriCore® cores, an HSM core, an SCR core and a GTM core, each requiring their own specific compiler while being tightly connected in terms of actual silicon. The **TASKING tool suite** provides fully integrated compilers allowing you to target code for the primary CPU and any of its guest cores.

Not every toolset vendor stays up to par with the latest developments and there has been an increasing demand to offer a commercially available standalone MCS C compiler that can be used in conjunction with third-party compilers. In such a symbiosis, the partnership supports both the primary CPU, using a third-party compiler, and the add-on MCS core using TASKING. This demand heralded the advent of the standalone **TASKING VX-toolset for MCS** - a compiler capable of building unexpected relationships. More accurately, this compiler allows for programming the MCS on **RH850**, Power Architecture and AURIX™ architectures.

The standalone **TASKING VX-toolset for MCS** was developed in collaboration with Bosch®. A key feature to establish integration is the locator's ability to generate a C array output containing the MCS program image, a feature inherited from the original Bosch® assembler. Using a minimalistic sample case, this paper discusses its format and shows how to correlate it to your map file. Also we explain how to integrate the C array code into a project of any third-party compiler; running the sample code as a proof of concept.

But like with any good story you need to start at the beginning. We will be exploring what spawned all of this into existence and how it will impact the future.

THE GTM ARCHITECTURE

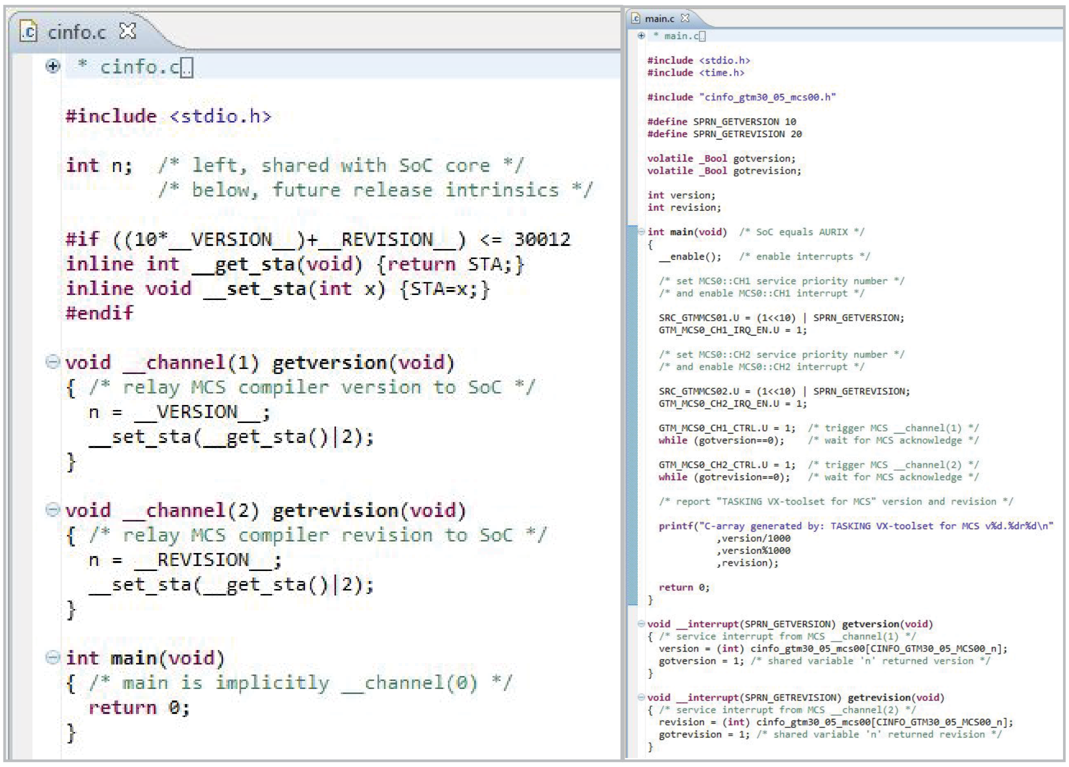
The Bosch® GTM core implements a generic timer platform for complex applications in the automotive industry like power train, power steering, chassis and transmission control. To serve these different application domains, the GTM provides a wide range of timer functions like counters, multi-action capture/compare, PWM functions, duty-cycle measurements and more.

In addition, the GTM includes internal RISC-like programmable cores for data processing and complex output sequence generation [1]. These cores are called multi-channel sequencer cores, or MCS cores for short. Its IP, now capable of being programmed through the TASKING standalone MCS C compiler, is designed to run with minimal CPU interaction while unloading the CPU from handling interrupt service requests as much as possible.

The GTM design is owned by Robert Bosch® GmbH. An SoC design can incorporate its IP alongside the main CPU using one or more GTM instances. Each GTM can have one or more MCS cores, and each MCS core implements 8 channels.

MCS AND CPU SAMPLE PROGRAMS

Listed below is the sample program that will be used throughout the course of this paper. The sample program consist of two parts. The first part shown below in *Figure 1* (left) is the code that will run on an MCS core. The second part in *Figure 1* (right) is the code that will run on the hosting architecture (SoC) and interact with the generic timer module (GTM). The sample code is minimalistic in nature, since the aim is not to go deep in terms of functionality, but instead demonstrate what you need to do to hook up your application code into the project of your compiler.



cinfo.c

```
* cinfo.c

#include <stdio.h>

int n;   /* left, shared with SoC core */
         /* below, future release intrinsics */

#if ((10*__VERSION__)+__REVISION__) <= 30012
inline int __get_sta(void) {return STA;}
inline void __set_sta(int x) {STA=x;}
#endif

void __channel(1) getversion(void)
{ /* relay MCS compiler version to SoC */
  n = __VERSION__;
  __set_sta(__get_sta()|2);
}

void __channel(2) getrevision(void)
{ /* relay MCS compiler revision to SoC */
  n = __REVISION__;
  __set_sta(__get_sta()|2);
}

int main(void)
{ /* main is implicitly __channel(0) */
  return 0;
}
```

main.c

```
* main.c

#include <stdio.h>
#include <time.h>

#include "cinfo_gtm30_05_mcs00.h"

#define SPRN_GETVERSION 10
#define SPRN_GETREVISION 20

volatile _Bool gotversion;
volatile _Bool gotrevision;

int version;
int revision;

int main(void)  /* SoC equals AURIX */
{
  __enable();   /* enable interrupts */

  /* set MCS0::CH1 service priority number */
  /* and enable MCS0::CH1 interrupt */

  SRC_GTMMCS01.U = (1<<10) | SPRN_GETVERSION;
  GTM_MCS0_CH1_IRQ_EN.U = 1;

  /* set MCS0::CH2 service priority number */
  /* and enable MCS0::CH2 interrupt */

  SRC_GTMMCS02.U = (1<<10) | SPRN_GETREVISION;
  GTM_MCS0_CH2_IRQ_EN.U = 1;

  GTM_MCS0_CH1_CTRL.U = 1;  /* trigger MCS __channel(1) */
  while (gotversion==0);    /* wait for MCS acknowledge */

  GTM_MCS0_CH2_CTRL.U = 1;  /* trigger MCS __channel(2) */
  while (gotrevision==0);   /* wait for MCS acknowledge */

  /* report "TASKING VX-toolset for MCS" version and revision */

  printf("C-array generated by: TASKING VX-toolset for MCS v%d.%dr%d\n"
          ,version/1000
          ,version%1000
          ,revision);

  return 0;
}

void __interrupt(SPRN_GETVERSION) getversion(void)
{ /* service interrupt from MCS __channel(1) */
  version = (int) cinfo_gtm30_05_mcs00[CINFO_GTM30_05_MCS00_n];
  gotversion = 1; /* shared variable 'n' returned version */
}

void __interrupt(SPRN_GETREVISION) getrevision(void)
{ /* service interrupt from MCS __channel(2) */
  revision = (int) cinfo_gtm30_05_mcs00[CINFO_GTM30_05_MCS00_n];
  gotrevision = 1; /* shared variable 'n' returned revision */
}
```

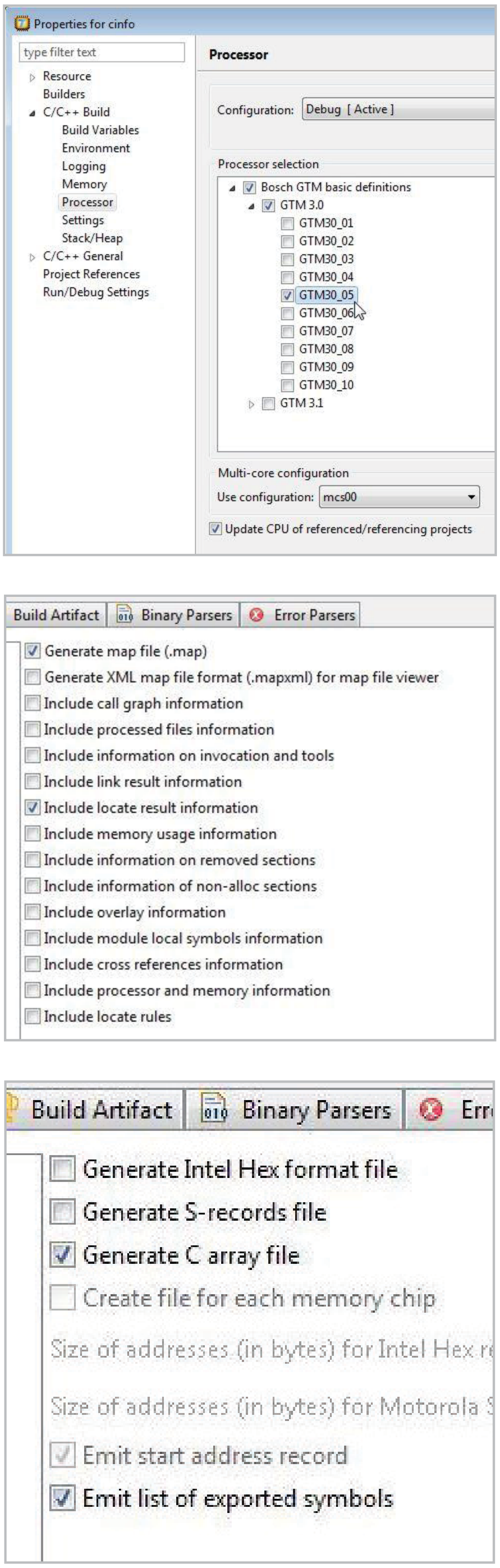
Figure 1 - Sample program for an MCS core

MCS PROJECT SETTINGS

The MCS core project settings primarily consist of default settings. Those noteworthy to building the application are:

1. Processor Settings
2. Linker Map File
3. Linker Output Format

Of each of these you'll find snapshots listed below. Note that the version of the MCS core is 3.0 and that the target code is configured for core `mcs00`. Further note that the project uses the classic map file generation listing only the locate result. Finally note that the linker output format has been set to C array.


Properties for cinfo

type filter text

- Resource
- Builders
- C/C++ Build
  - Build Variables
  - Environment
  - Logging
  - Memory
  - Processor
  - Settings
  - Stack/Heap
- C/C++ General
- Project References
- Run/Debug Settings

Processor

Configuration: Debug [Active]

Processor selection

- ☑ Bosch GTM basic definitions
  - ☑ GTM 3.0
    - ☐ GTM30_01
    - ☐ GTM30_02
    - ☐ GTM30_03
    - ☐ GTM30_04
    - ☑ GTM30_05
    - ☐ GTM30_06
    - ☐ GTM30_07
    - ☐ GTM30_08
    - ☐ GTM30_09
    - ☐ GTM30_10
  - ☐ GTM 3.1

Multi-core configuration

Use configuration: mcs00

☑ Update CPU of referenced/referencing projects

Build Artifact | Binary Parsers | Error Parsers

- ☑ Generate map file (.map)
- ☐ Generate XML map file format (.mapxml) for map file viewer
- ☐ Include call graph information
- ☐ Include processed files information
- ☐ Include information on invocation and tools
- ☐ Include link result information
- ☑ Include locate result information
- ☐ Include memory usage information
- ☐ Include information on removed sections
- ☐ Include information of non-alloc sections
- ☐ Include overlay information
- ☐ Include module local symbols information
- ☐ Include cross references information
- ☐ Include processor and memory information
- ☐ Include locate rules

Build Artifact | Binary Parsers | Err

- ☐ Generate Intel Hex format file
- ☐ Generate S-records file
- ☑ Generate C array file
- ☐ Create file for each memory chip

Size of addresses (in bytes) for Intel Hex re

Size of addresses (in bytes) for Motorola S

- ☑ Emit start address record
- ☑ Emit list of exported symbols

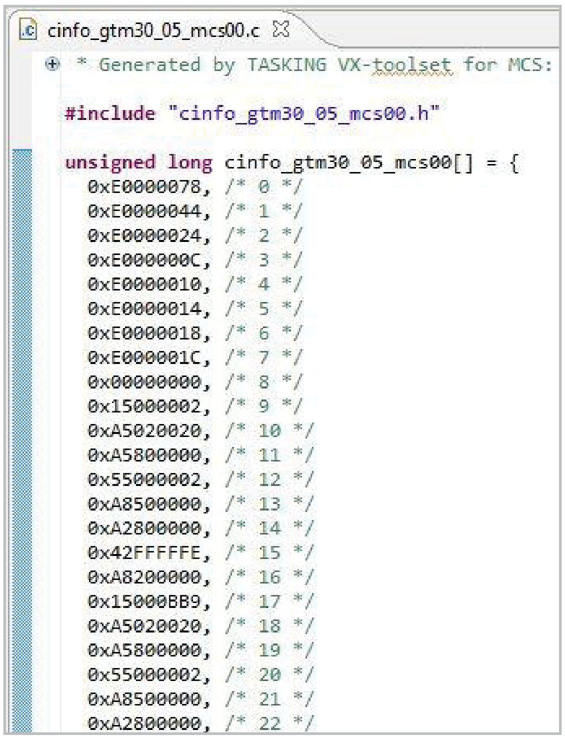
Figure 2 - MCS project settings - processor settings (top), linker map file (middle) linker output format (bottom)

CORRELATING AND LOCATING RESULTS OF A C ARRAY FOOTPRINT

Now that the MCS project has been set up, you can proceed to build. A few files will be generated as a result:

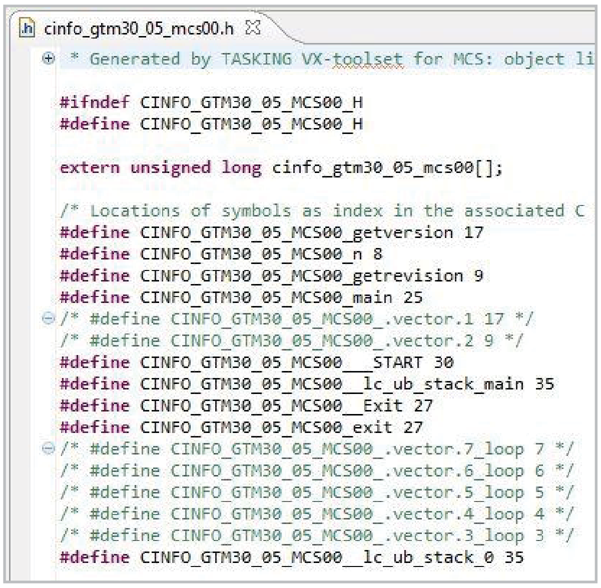
- `cinfo_gtm30_05_mcs00.c` (C file below)
- `cinfo_gtm30_05_mcs00.h` (H file below)
- `cinfo.map` (map file below)

The *C file* contains the MCS program image footprint poured into an array. This includes both code and data. The *H file* contains offsets within the array pointing to functions and global variables. This essentially creates a symbol table. Since the MCS core is memory mapped within the hosting architecture, the combination of these two files allows the third-party compiler to (a) locate the program image at its designated base address and (b) peek or poke MCS data or code. Listed below are snapshots of both the *C file* and *H file*.



cinfo_gtm30_05_mcs00.c

```
 * Generated by TASKING VX-toolset for MCS:

#include "cinfo_gtm30_05_mcs00.h"

unsigned long cinfo_gtm30_05_mcs00[] = {
  0xE0000078, /* 0 */
  0xE0000044, /* 1 */
  0xE0000024, /* 2 */
  0xE000000C, /* 3 */
  0xE0000010, /* 4 */
  0xE0000014, /* 5 */
  0xE0000018, /* 6 */
  0xE000001C, /* 7 */
  0x00000000, /* 8 */
  0x15000002, /* 9 */
  0xA5020020, /* 10 */
  0xA5800000, /* 11 */
  0x55000002, /* 12 */
  0xA8500000, /* 13 */
  0xA2800000, /* 14 */
  0x42FFFFFE, /* 15 */
  0xA8200000, /* 16 */
  0x15000BB9, /* 17 */
  0xA5020020, /* 18 */
  0xA5800000, /* 19 */
  0x55000002, /* 20 */
  0xA8500000, /* 21 */
  0xA2800000, /* 22 */
```

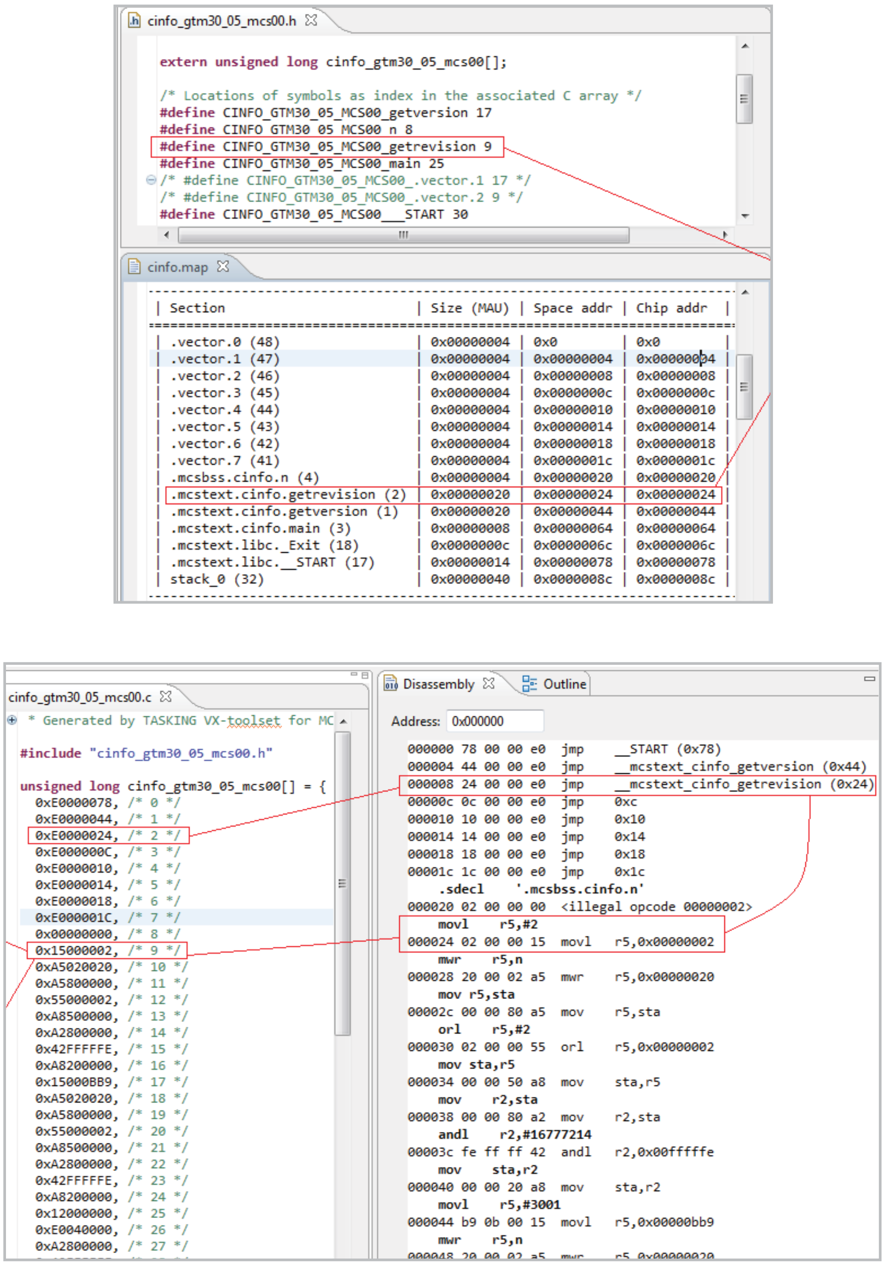


cinfo_gtm30_05_mcs00.h

```
 * Generated by TASKING VX-toolset for MCS: object li

#ifndef CINFO_GTM30_05_MCS00_H
#define CINFO_GTM30_05_MCS00_H

extern unsigned long cinfo_gtm30_05_mcs00[];

/* Locations of symbols as index in the associated C
#define CINFO_GTM30_05_MCS00_getversion 17
#define CINFO_GTM30_05_MCS00_n 8
#define CINFO_GTM30_05_MCS00_getrevision 9
#define CINFO_GTM30_05_MCS00_main 25
/* #define CINFO_GTM30_05_MCS00_.vector.1 17 */
/* #define CINFO_GTM30_05_MCS00_.vector.2 9 */
#define CINFO_GTM30_05_MCS00___START 30
#define CINFO_GTM30_05_MCS00__lc_ub_stack_main 35
#define CINFO_GTM30_05_MCS00__Exit 27
#define CINFO_GTM30_05_MCS00_exit 27
/* #define CINFO_GTM30_05_MCS00_.vector.7_loop 7 */
/* #define CINFO_GTM30_05_MCS00_.vector.6_loop 6 */
/* #define CINFO_GTM30_05_MCS00_.vector.5_loop 5 */
/* #define CINFO_GTM30_05_MCS00_.vector.4_loop 4 */
/* #define CINFO_GTM30_05_MCS00_.vector.3_loop 3 */
#define CINFO_GTM30_05_MCS00__lc_ub_stack_0 35
```

Figure 3 - MCS data code snapshots for C and H files

The best exercise to get an understanding of the C array program image footprint is to try and correlate some of its *H file* symbols to actual addresses in the map file or the simulator's disassembly window. As an example let's try to cross-reference function `getrevision` in *Figure 4* below (note that the snapshot is folded to increase resolution).



```
extern unsigned long cinfo_gtm30_05_mcs00[];

/* Locations of symbols as index in the associated C array */
#define CINFO_GTM30_05_MCS00_getversion 17
#define CINFO_GTM30_05_MCS00_n 8
#define CINFO_GTM30_05_MCS00_getrevision 9
#define CINFO_GTM30_05_MCS00_main 25
/* #define CINFO_GTM30_05_MCS00_.vector.1 17 */
/* #define CINFO_GTM30_05_MCS00_.vector.2 9 */
#define CINFO_GTM30_05_MCS00___START 30
```

```
-------------------------------------------------------------------------
| Section                           | Size (MAU) | Space addr | Chip addr  |
=========================================================================
| .vector.0 (48)                    | 0x00000004 | 0x0        | 0x0        |
| .vector.1 (47)                    | 0x00000004 | 0x00000004 | 0x00000004 |
| .vector.2 (46)                    | 0x00000004 | 0x00000008 | 0x00000008 |
| .vector.3 (45)                    | 0x00000004 | 0x0000000c | 0x0000000c |
| .vector.4 (44)                    | 0x00000004 | 0x00000010 | 0x00000010 |
| .vector.5 (43)                    | 0x00000004 | 0x00000014 | 0x00000014 |
| .vector.6 (42)                    | 0x00000004 | 0x00000018 | 0x00000018 |
| .vector.7 (41)                    | 0x00000004 | 0x0000001c | 0x0000001c |
| .mcsbss.cinfo.n (4)               | 0x00000004 | 0x00000020 | 0x00000020 |
| .mcstext.cinfo.getrevision (2)    | 0x00000020 | 0x00000024 | 0x00000024 |
| .mcstext.cinfo.getversion (1)     | 0x00000020 | 0x00000044 | 0x00000044 |
| .mcstext.cinfo.main (3)           | 0x00000008 | 0x00000064 | 0x00000064 |
| .mcstext.libc._Exit (18)          | 0x0000000c | 0x0000006c | 0x0000006c |
| .mcstext.libc.__START (17)        | 0x00000014 | 0x00000078 | 0x00000078 |
| stack_0 (32)                      | 0x00000040 | 0x0000008c | 0x0000008c |
-------------------------------------------------------------------------
```

```
 * Generated by TASKING VX-toolset for MC

#include "cinfo_gtm30_05_mcs00.h"

unsigned long cinfo_gtm30_05_mcs00[] = {
  0xE0000078, /* 0 */
  0xE0000044, /* 1 */
  0xE0000024, /* 2 */
  0xE000000C, /* 3 */
  0xE0000010, /* 4 */
  0xE0000014, /* 5 */
  0xE0000018, /* 6 */
  0xE000001C, /* 7 */
  0x00000000, /* 8 */
  0x15000002, /* 9 */
  0xA5020020, /* 10 */
  0xA5800000, /* 11 */
  0x55000002, /* 12 */
  0xA8500000, /* 13 */
  0xA2800000, /* 14 */
  0x42FFFFFE, /* 15 */
  0xA8200000, /* 16 */
  0x15000BB9, /* 17 */
  0xA5020020, /* 18 */
  0xA5800000, /* 19 */
  0x55000002, /* 20 */
  0xA8500000, /* 21 */
  0xA2800000, /* 22 */
  0x42FFFFFE, /* 23 */
  0xA8200000, /* 24 */
  0x12000000, /* 25 */
  0xE0040000, /* 26 */
  0xA2800000, /* 27 */
```

```
000000 78 00 00 e0  jmp    __START (0x78)
000004 44 00 00 e0  jmp    __mcstext_cinfo_getversion (0x44)
000008 24 00 00 e0  jmp    __mcstext_cinfo_getrevision (0x24)
00000c 0c 00 00 e0  jmp    0xc
000010 10 00 00 e0  jmp    0x10
000014 14 00 00 e0  jmp    0x14
000018 18 00 00 e0  jmp    0x18
00001c 1c 00 00 e0  jmp    0x1c
    .sdecl    '.mcsbss.cinfo.n'
000020 02 00 00 00  <illegal opcode 00000002>
    movl    r5,#2
000024 02 00 00 15  movl   r5,0x00000002
    mwr    r5,n
000028 20 00 02 a5  mwr    r5,0x00000020
    mov r5,sta
00002c 00 00 80 a5  mov    r5,sta
    orl    r5,#2
000030 02 00 00 55  orl    r5,0x00000002
    mov sta,r5
000034 00 00 50 a8  mov    sta,r5
    mov    r2,sta
000038 00 00 80 a2  mov    r2,sta
    andl    r2,#16777214
00003c fe ff ff 42  andl   r2,0x00fffffe
    mov    sta,r2
000040 00 00 20 a8  mov    sta,r2
    movl    r5,#3001
000044 b9 0b 00 15  movl   r5,0x00000bb9
    mwr    r5,n
```

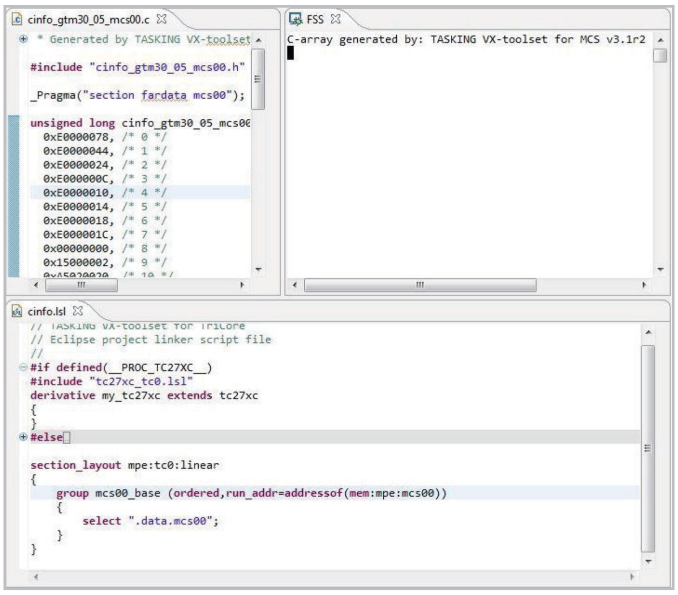
Figure 4 - Cross-referencing functions in a C array program

The symbol value for getrevision equals decimal 9. For an architecture with `sizeof(long)` equal to 4 this makes for an offset of decimal 36 or hexadecimal 24. Note that for this address the C array contains an instruction code of 0x15000002 which the simulator disassembly window translates to `movl r5,0x2` which is a preload of predefined MCS compiler macro `__REVISION__` (since we used MCS v3.1r2) into register `r5`. Note that function getversion was coded with `__channel(2)`, which means MCS address `0x000008` must contain a matching jump vector. This is achieved by MCS instruction code `0xE0000024` residing at decimal offset 9 within the C array.

ESTABLISHING THIRD-PARTY SYMBIOSES

Once you have the C array footprint, copy it to your third-party compiler project and make sure it is located at its designated MCS base address. Each third-party declares this their own way but generally you'll find language extensions similar to the TASKING `__at()` keyword. You need to add this to the C array declaration yourself. Alternatively, you can leave the C array unscathed and use a locator equivalent. That's all it takes! The startup code of your toolchain will initialize the C array in the normal fashion, effectively bootstrapping the MCS program image into memory. Once that is all done, it only needs a little push to kick it into motion. You'll see this in the second part of the sample code in *Figure 5*.

As proof of concept we used the **TASKING VX-toolset for TriCore v6.0r1** to act as a third-party compiler and proceeded to incorporate the C array footprint. The snapshot on the next page shows what that looks like. Note the usage of `_Pragma("section fardata mcs00")` to assure a unique section for the MCS C array. Also note that the linker script language (LSL) is subsequently used to anchor it to core `mcs00`. Finally, note what happens if you run it on an Infineon® TriBoard [2] using TASKING file system simulation (FSS). This will provide obvious proof that the MCS program was downloaded properly and behaves as intended.



```
 * Generated by TASKING VX-toolset

#include "cinfo_gtm30_05_mcs00.h"

_Pragma("section fardata mcs00");

unsigned long cinfo_gtm30_05_mcs00
  0xE0000078, /* 0 */
  0xE0000044, /* 1 */
  0xE0000024, /* 2 */
  0xE000000C, /* 3 */
  0xE0000010, /* 4 */
  0xE0000014, /* 5 */
  0xE0000018, /* 6 */
  0xE000001C, /* 7 */
  0x00000000, /* 8 */
  0x15000002, /* 9 */
  0xA5020020, /* 10 */
```

```
C-array generated by: TASKING VX-toolset for MCS v3.1r2
```

```
// TASKING VX-toolset for TriCore
// Eclipse project linker script file
//
#if defined(__PROC_TC27XC__)
#include "tc27xc_tc0.lsl"
derivative my_tc27xc extends tc27xc
{
}
#else

section_layout mpe:tc0:linear
{
    group mcs00_base (ordered,run_addr=addressof(mem:mpe:mcs00))
    {
        select ".data.mcs00";
    }
}
```

Figure 5 - Code snapshot incorporating a C array footprint

It should be noted that because we used a TASKING compiler for testing, the SFRs used in the sample code are specific to TriCore® and will most likely be different from your primary architecture. There's also a chance that your third-party toolchain uses different keywords for the TASKING `__interrupt` and `_Bool` qualifiers. Some extra tweaks may be required before you can actually build and run the code.

The TASKING startup code for TriCore® automatically enables the MCS memory, whereas your third-party toolchain might leave that up to you. Whereas the TASKING toolset can immediately 'bootstrap' the code into MCS memory (provided the C array program image has been located properly at its base address) your third-party project will need user code to do that same copy.

If your third-party toolchain/debugger does not support a similar mechanism to TASKING file system simulation then you may not receive output from your debugger. If this happens you can restrict debugging to placing watches on variables `version` and `revision`.

CONCLUSION

The **TASKING VX-toolset for MCS** is the ideal partner when your third-party toolset is limited to code generation for the CPU of the hosting architecture. With its traditional linker C array output, the MCS toolset generates a pure ANSI-C application footprint that can be woven into the project of your primary CPU with little effort, other than making sure that it's located properly to its designated MCS base address. With the help of the TASKING standalone compiler for the GTM, you'll be able to enhance your development environment for programming complete RH850, Power Architecture or AURIX™ based applications, regardless the C compiler used for the primary CPU.

Additional Resources:

[1] Bosch GTM IP Module. Web.

http://www.bosch-semiconductors.de/en/automotive_electronics/ip_modules/timer_ip_module/timer_platform_1.html

[2] Infineon TriBoard Development Boards.

http://www.infineon.com/cms/en/product/microcontroller/development-tools-software-and-kits/tricore-tm-development-tools-software-and-kits/aurix-starter-and-application-kits/channel.html?channel=5546d4614babddc8014bbc8126de00ad