

CMT216xA Development Hands-on

Overview

This document discusses the development environment establishment and debugging of the CMT216xA series transmitter chips for users to quickly get familiar with the tool kits usage and debugging of the series chips.

The product models covered in the document are listed in the table below.

Table 1. Product Models Covered in This Document

Product Model	Single-end PA	Differential PA	12-Bit ADC	Operational Amplifier	Low-frequency Wakeup	External 32.768 kHz	Packaging
CMT2160A		•	4-ch				SOP14
CMT2162A	•		8-ch		•		SSOP20
CMT2163A	•	•	9-ch		•	•	TSSOP28
CMT2165A	•		12-ch	•		•	TSSOP28
CMT2168A	•		12-ch	•	•	•	QFN32

Notes:

1. The performance and parameter details as well as the package size, silk screen and ordering information of each chip model are NOT covered in this document, please refer to the datasheet document of each chip model for details. For specific function details of the CMT216xA series, please refer to *CMT216xA User Guide*.

Reading guidelines:

1. This document is one of the preliminary documents for CMT216xA development, thus recommend that users read this one first.
2. For beginners of the CMT216xA system, it is recommended for users to read Section 1 Development Environment Establishment and Section 2 1-wire Online Debugging first to get the basic knowledge of the CMT216xA online debugging mode, and then to get familiar with the tools kits usage. Meanwhile by debugging the functions users can familiarize with the CMT216xA series products gradually.
3. For users that are familiar with the CMT216xA, like having read CMT216xA User Guide, and need offline verification and testing of the target system/product, it is recommended that users perform offline debugging using Bootloader based on the information in Section 3 Bootloader Offline Debugging. Users can establish the offline test environment for the target system/product through the Bootloader tool.
4. For users that have completed the first round target system/product design, a complete product evaluation is required (like sample production trial in a small batch), thereby the OTP burning is required. Recommend such users read Section 4 to get relevant guidance.

Table of Contents

Overview	1
1 Development Environment Establishment	3
1.1 Introduction	3
1.2 Create Project	3
2 1-wire Online Debugging.....	8
2.1 1-wire Hardware Establishment.....	8
2.2 1-wire Debugging Mode	9
2.3 1-wire Simulation Settings.....	10
3 Bootloader Offline Debugging.....	14
3.1 Basic Introduction.....	14
3.2 Bootloader Debugging.....	15
3.3 Bootloader Program Flow.....	16
3.4 User Program Interface.....	17
3.5 Bootloader Examples	18
3.5.1 Information Area.....	18
3.5.2 Startup Code.....	20
3.6 User Program Interrupt Service.....	22
3.7 Considerations for Bootloader Offline Debugging	26
4 OTP Programming	27
4.1 Programming Hardware Establishment.....	27
4.2 Programming UI Screen.....	28
5 Revise History	29
6 Contacts.....	30

1 Development Environment Establishment

1.1 Introduction

Embedded with enhanced 8051 core, the CMT216xA series chips employ 1-wire simulation debug interface. Since the dynamic link driver of CMT216xA's 1-wire debug interface is developed based on Keil C51 platform, this document will discuss the Keil C51 integrated development environment, a well-known IDE of 51 system, as the main development platform. In addition, the Keil 4 version is recommended.

1.2 Create Project

- Step 1, open Keil 4 development platform as shown in the below figure.

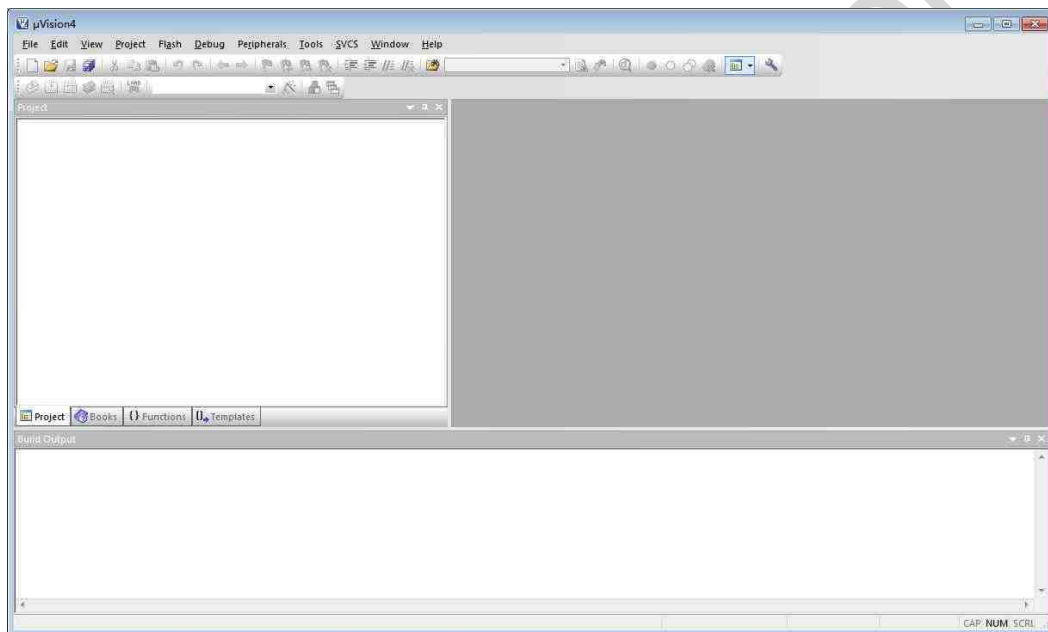


Figure 1. Keil 4 Development Platform UI Screen

- Step 2, click *Project* menu, then click *New μ Vision Project...* sub-menu, *Create New Project* window pops up as shown in the below Figure 2 and Figure 3.

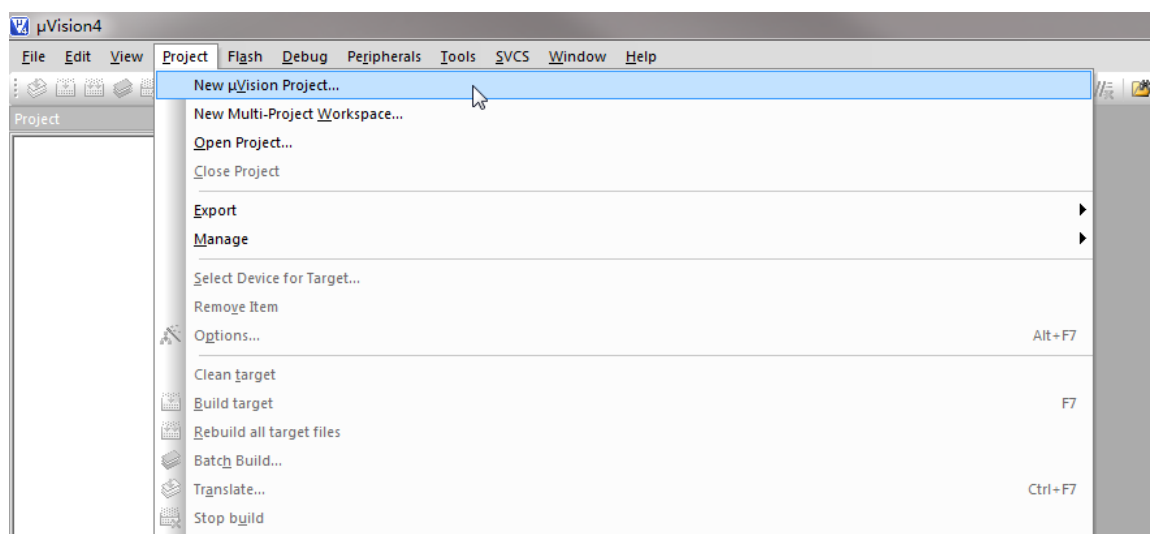


Figure 2. Project Menu

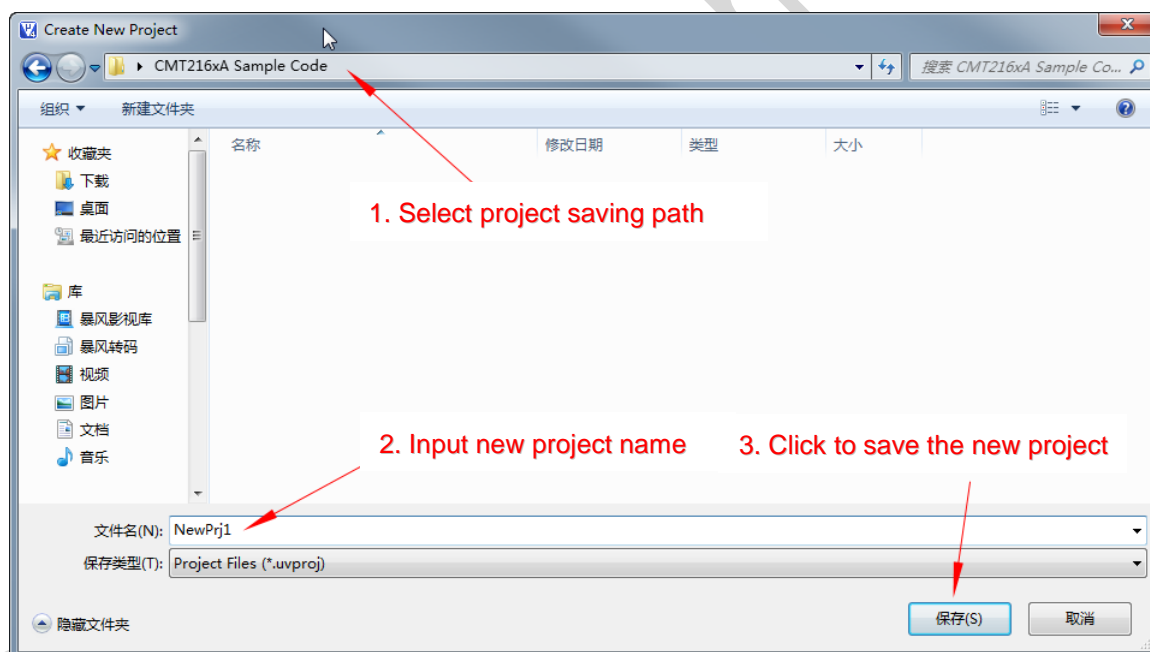


Figure 3. Create New Project

In the *New Project Window*,

1. Select the saved path of the the new project.
2. Input new project name.
3. Click Save.

- Step 3, after the project is saved in Step 2, the CPU model selection window pops up as shown in the below figure. In the below figure, select *T8051 core of CAST Inc.*, then click *OK*. The window pops up for users to confirm whether to generate *STARTUP.A51* file as shown in Figure 5.

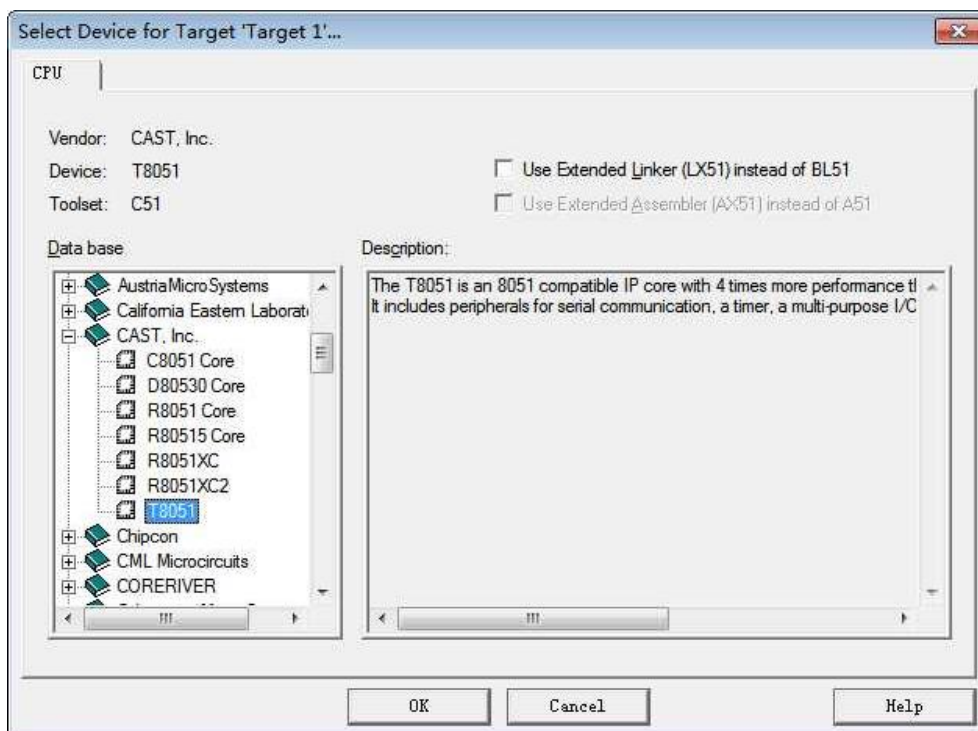


Figure 4. Select CPU Model

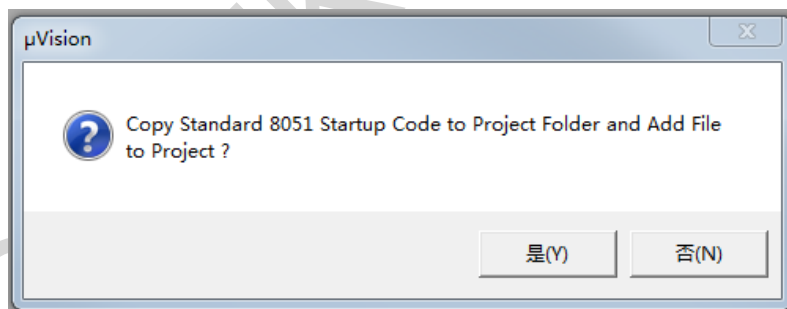


Figure 5. Window for Generating Startup Code

Notes:

- Users can choose either to generate *STARTUP.A51* file automatically or copy it from an empty project of CMT216xA or program examples.

- Step 4, add files cmt216xa_link.c and cmt216xa_keil.a51 into the project as shown in the below figure.

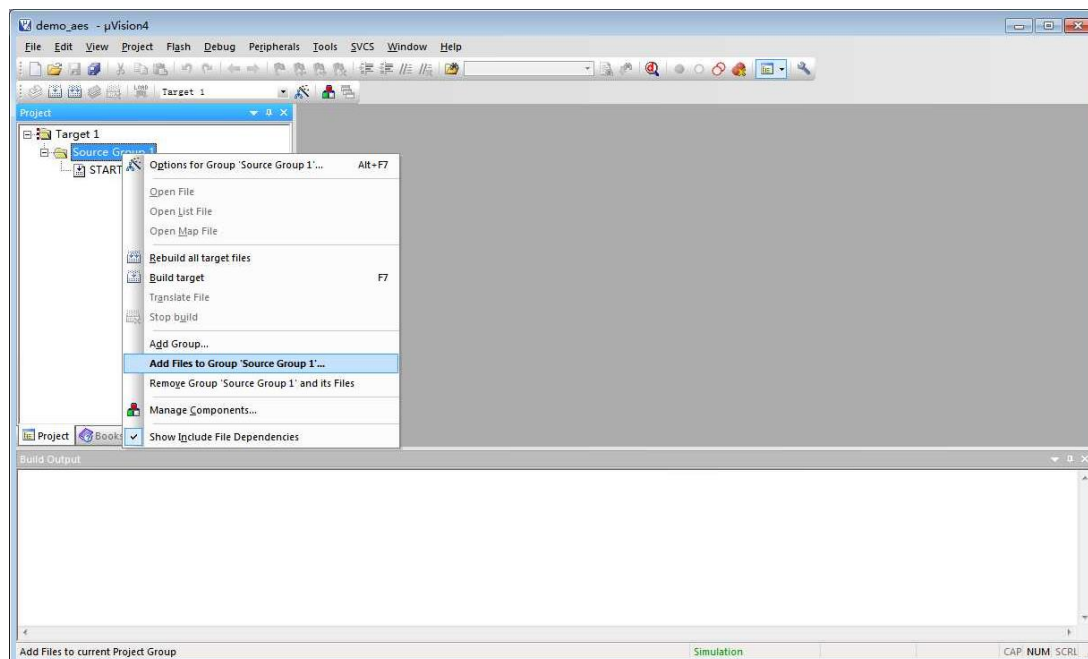


Figure 6. Add Necessary Files into Project

Notes:

- The files cmt216xa_link.c and cmt216xa_keil.a51 are necessary for all projects. Users can choose either to generate these files automatically or copy them from an empty project of CMT216xA or program examples.
- Step 5, open file STARTUP.A51, and reserve the stack space as shown in the below figure.

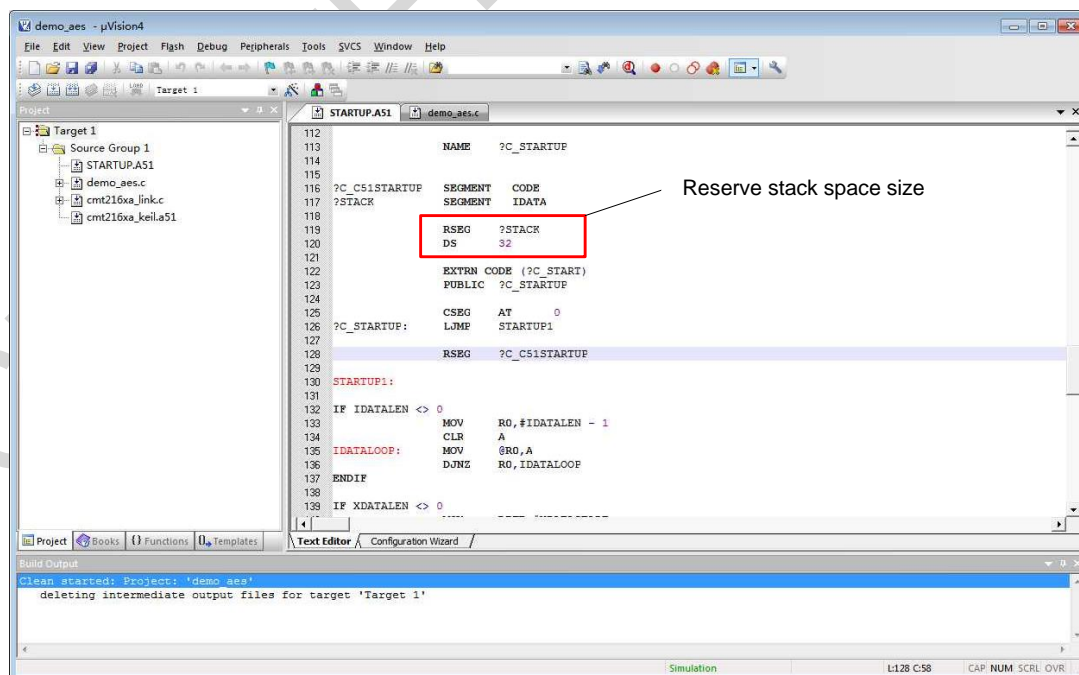


Figure 7. Set Reserved Stack Space Size

- Step 6, create the main program file (including main function) and add it into the project. Then program the target program functions as shown in the below figure (demo_aes.c is the main program file in the below figure)

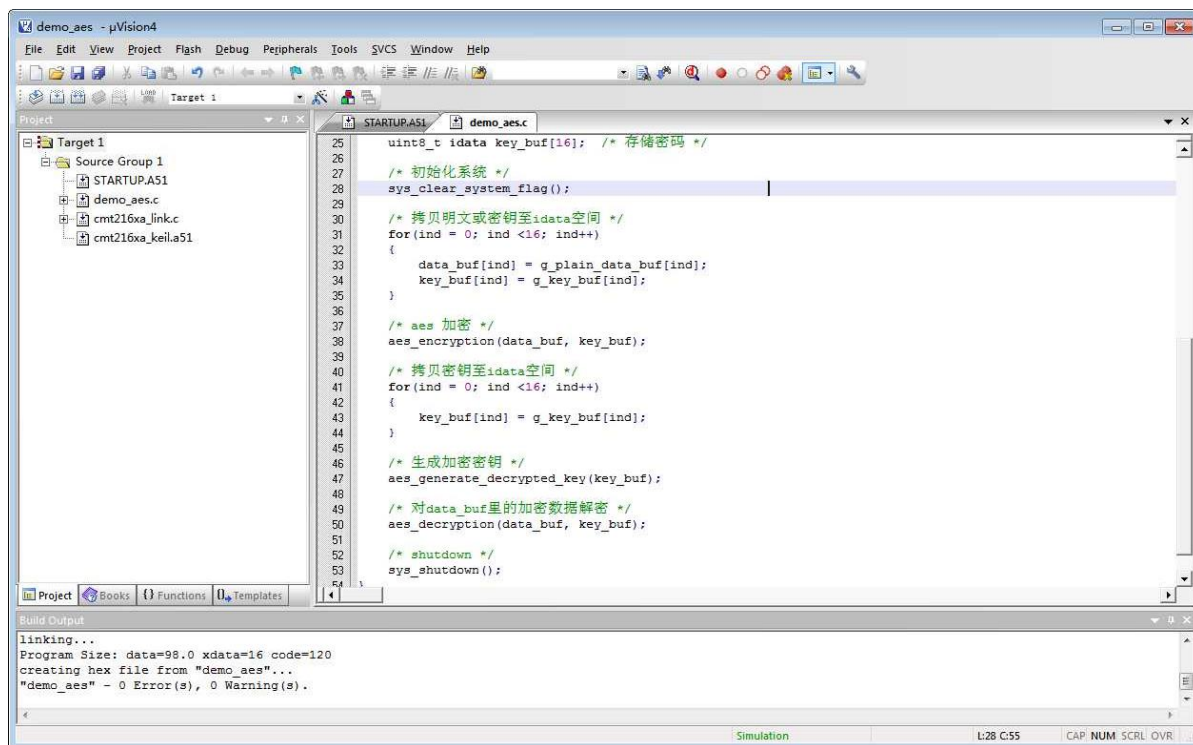


Figure 8. Add Main Function File

By now, steps for creating a project are complete. Users can add the source code files into an empty project for compiling, thus saving the steps for creating a new project.

Notes:

The necessary files for a new project of CMT216xA are shown in the below figure. These files need to be copied into the project folder and included in the program for compiling. The functionalities of these files are as follows.

1. CMT216xA.h, one of the header files, mainly contains API library function declarations and the related structure definitions.
2. CMT216xA_types.h, one of the header files, mainly contains typedef definitions. Users can also add new typedef definitions based on user requirements.
3. CMT216xA_sfr.h, one of the header files, mainly contains the SFR register definitions in Block1.
4. CMT216xA_macro.h, one of the header files, mainly contains the macro definitions required by the API, and the register address macro definitions of the Block0 area (since this area cannot be accessed directly).
5. CMT216xA_link.c, one of the program files, must be loaded and compiled; mainly contains the macro definitions for the program running space and the storage address of the relevant variables.

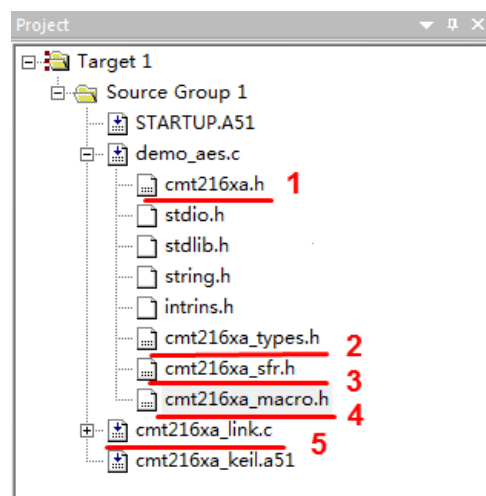


Figure 9. Necessary Project Files

2 1-wire Online Debugging

2.1 1-wire Hardware Establishment

As shown in the below figure, the CMT216xA simulation environment requires the below hardware preparation.

- Target CMT216xA debugging board. The CMT2168A demo board is the target debugging board as shown on the left in the below figure.
- CMT216xA simulator, as shown in the middle in the below figure.
- Computer and USB cable (A to B type)

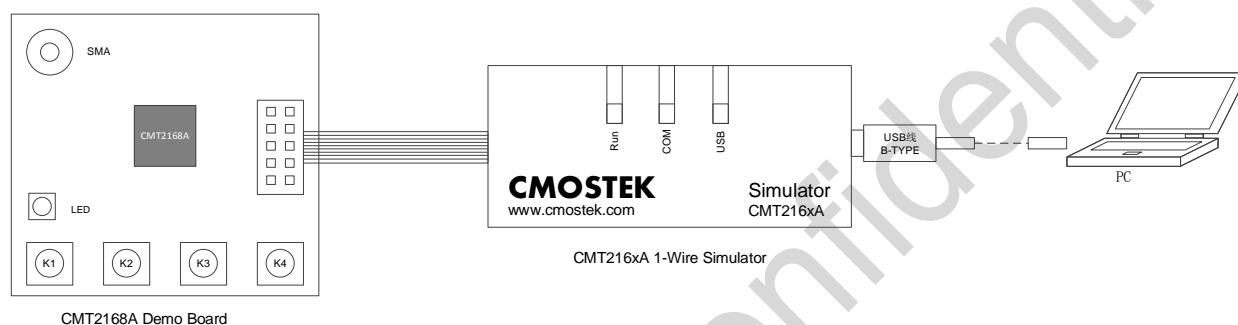


Figure 10. Hardware Connections for CMT216xA Simulation

Although the simulator uses a 10-pin box-header, the CMT216xA series single-shot SoC products only require 3-wire (VCC, GND, 1-Wire/B0) connections for debugging. Therefore, when designing the target debugging board, users can choose to keep just the above 3 connection points to save the PCB area. The corresponding pin function of the 10-pin box-header of the simulator is shown in the below figure.

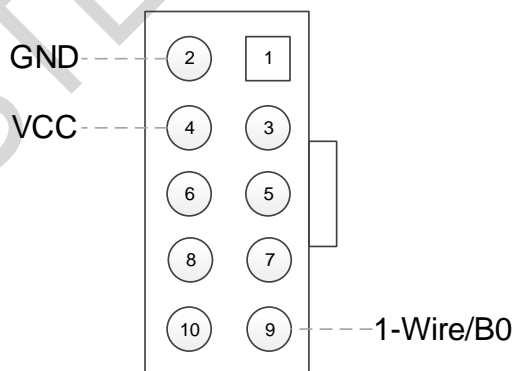


Figure 11. 10-pin Function Definition of Simulator

Notes:

1. The above shows the front view, facing the 10 pins of the 10-pin box-header of the simulator.

2.2 1-wire Debugging Mode

The basic 1-wire debugging flow is as follows.

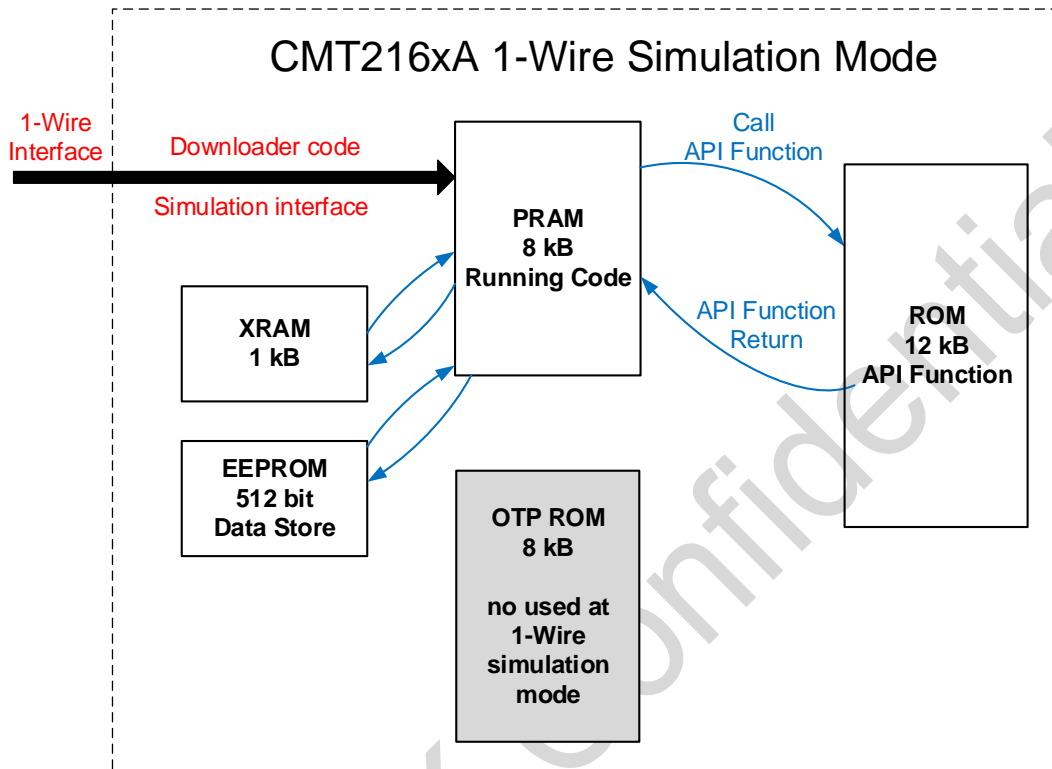


Figure 12. 1-wire Debugging Flow

- The target code to be debugged is downloaded directly to the PRAM of the CMT216xA chip through the 1-WIRE interface.
- Monitored through the 1-WIRE interface, the code runs in a simulation way in PRAM.
- Users can call APIs during code running.
- The OTP ROM is not involved in the whole debugging process.

2.3 1-wire Simulation Settings

Before 1-wire simulation on the CMT216xA, the below software settings on Keil IDE are needed.

- Step 1, copy driver file *CmtSimulator.dll* into the Keil installation path ...\\Keil\\C51\\BIN\\ as shown in the below figure. This file is to the library file driving the simulator in the Keil IDE environment.

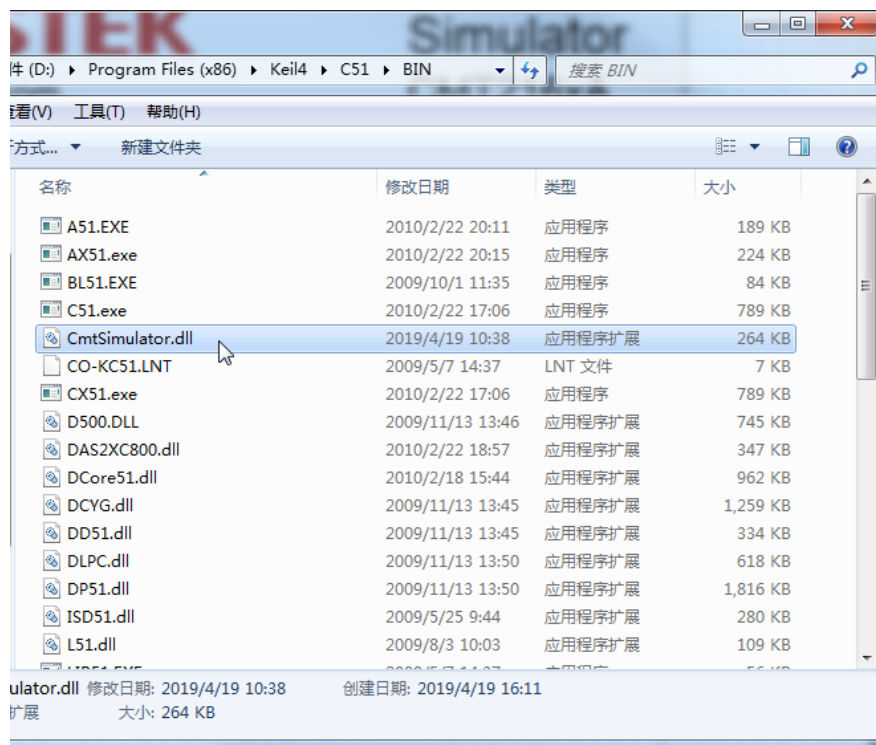


Figure 13. Copy CmtSimulator.dll Driver File

- Step 2, modify the file *TOOLS.INI* in the Keil installation path to add the CMT216xA simulator into the Keil debugger selection list as shown in Figure 14 and Figure 15.

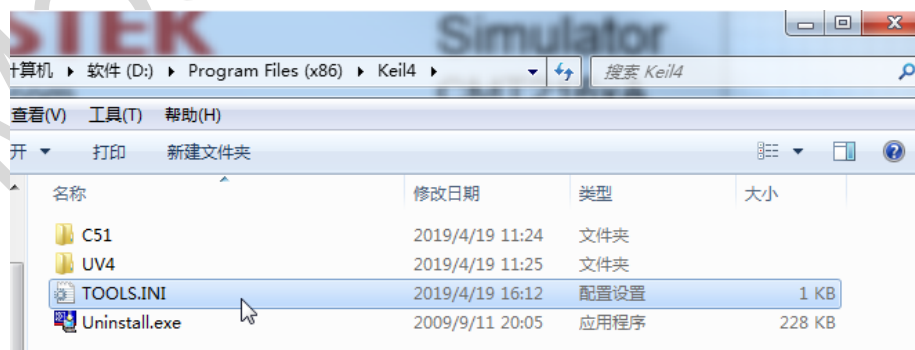


Figure 14. TOOLS.INI File Path

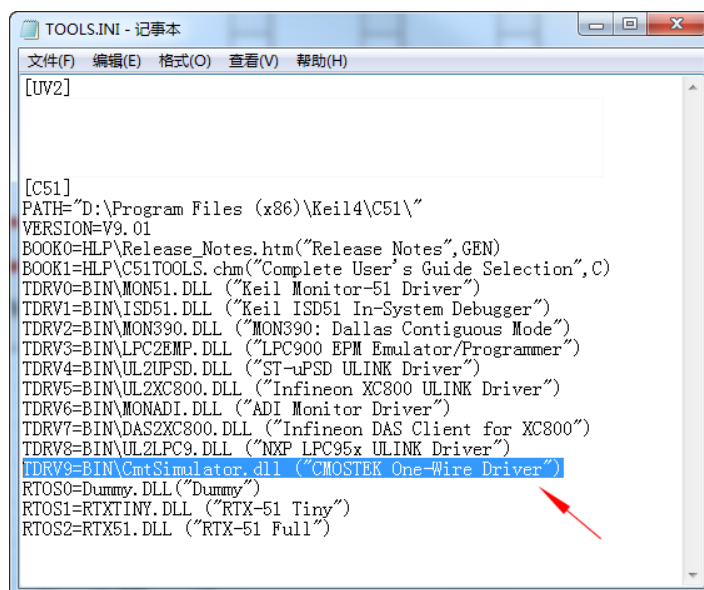


Figure 15. Add CmtSimulator.dll File Link

Notes:

1. The information in above figure is for reference only. Users needs to modify it based on the configuration file currently used in Keil. As shown in the figure above, the last item is *TDRV8* in *CmtSimulator.dll* before new information is added. Therefore, users should add from *TDRV9* when appending new information in *CmtSimulator.dll*. In addition, the information on the right side of '=' is the path of *CmtSimulator.dll* and the information inside the double quotes (") on the most right side are tool descriptions, which are displayed synchronously (information displayed is consistent) in the optional menu of the Keil simulation debugger. Thus users can fill in specific content as desired.
- Step 3, return to Keil Project screen, select *Target Options...* screen(click the icon as shown in the red circle in the below figure), The window *Options for Target 'Target 1'* pops up as shown in Figure 17.

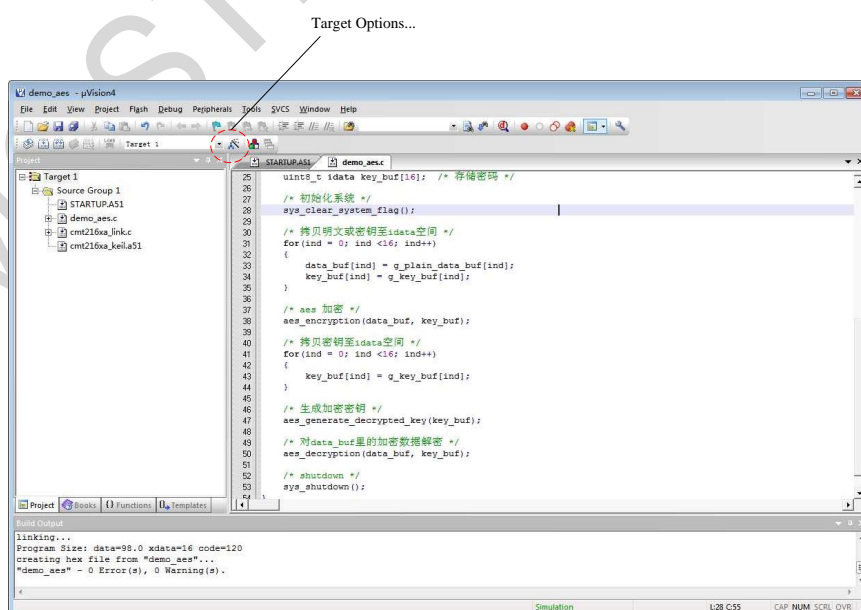


Figure 16. Select Target Options...

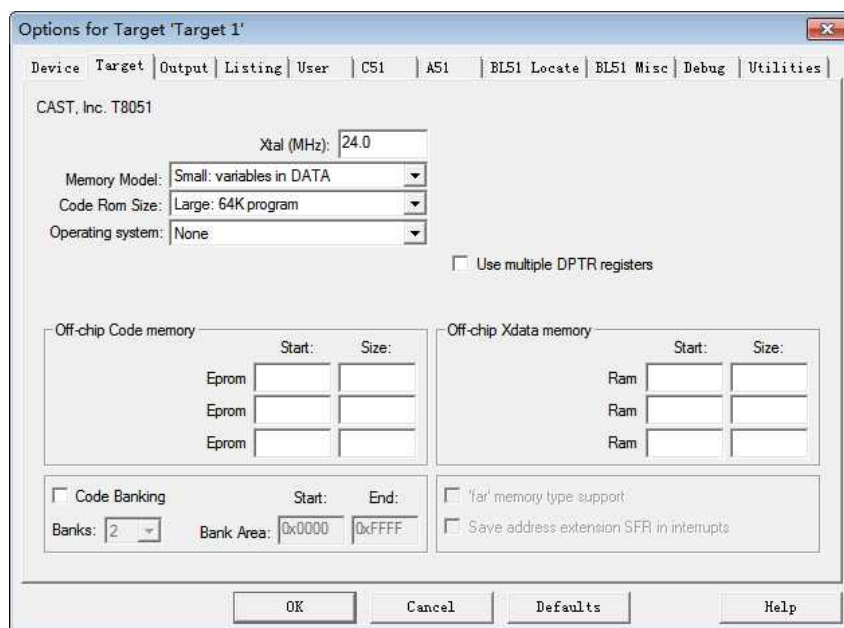


Figure 17. Options for Target 'Target 1' Screen

- Step 4, in *Options for Target 'Target 1'* screen, select the tag *debug*, then check the specified debugging mode and select the CmtSimulator added in the above steps, as shown in the below figure.

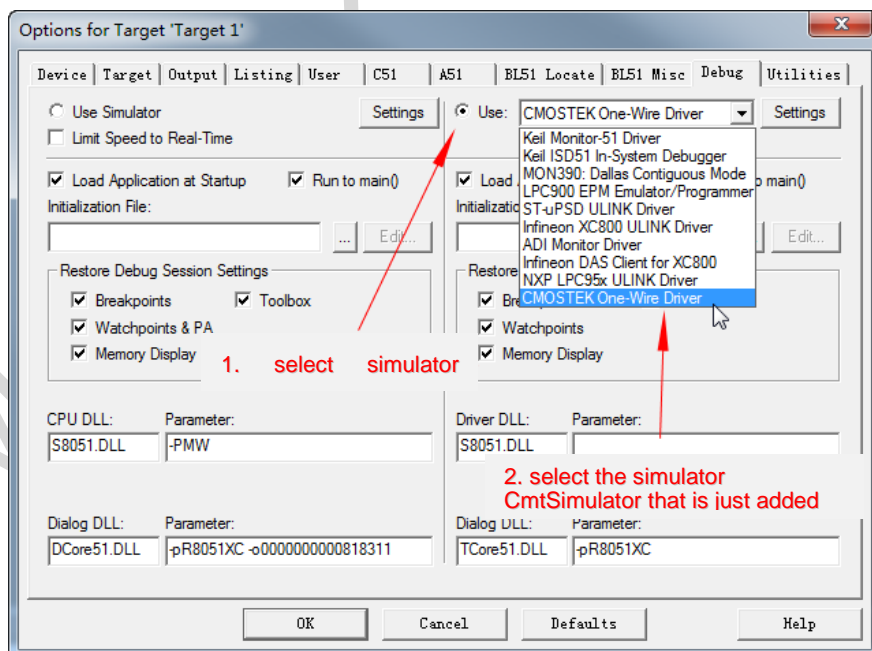


Figure 18. Select CmtSimulator Simulator

- Step 5, click the *settings* button on the right side as shown in Figure 18, the setting window pops up as shown in the below figure.

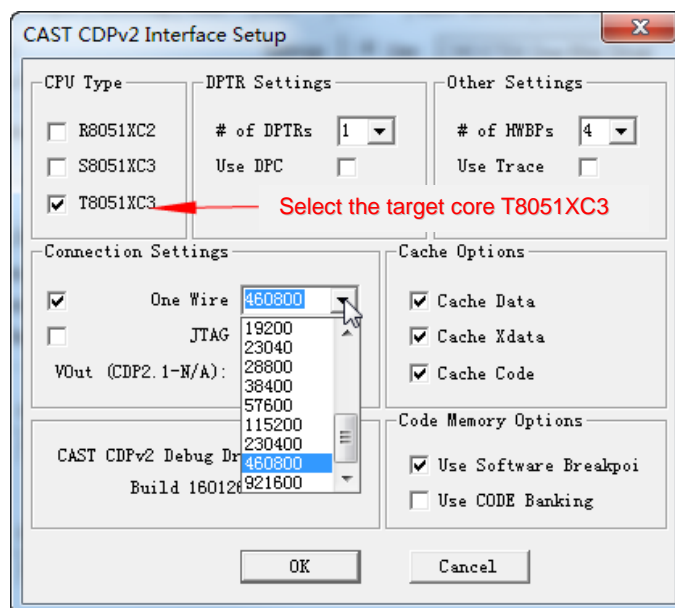


Figure 19. Set Debugging Parameters

As shown in Figure 19, select the correct target core, T8051XC3, and keep the default option *One Wire* in *Connection Settings* panel. Users can select an appropriate debug baud rate. For other options, suggest keeping the default options as shown in Figure 19.

Notes:

- When the CMT216xA operates in a higher speed (the default is 24 MHz if the internal RC is not divided), users can select the debug baud rate as 460800, 230400 or 115200, which can achieve fine debugging experience. A lower debug baud rate is not recommended.
- Similarly, when the CMT216xA operates in a lower speed, the debug baud rate should be decreased. Otherwise, if the CMT216xA core runs slowly but with a higher debug baud rate, it may cause the system exits the simulation mode due to a simulation exception.

At this point, 1-wire debugging related settings are completed. After confirming the simulator is connected to the target board properly and the target code to be debugged is compiled, users can enter the Keil online simulation and debugging mode.

3 Bootloader Offline Debugging

3.1 Basic Introduction

As discussed previously, based on the architectural features of the CMT216xA series products, the CW216xA in 1-wire online debugging mode downloads the code to be debugged into PRAM through the simulator, to fulfill online debugging. In this way, although the CMT216xA series chips are OTP products, they can still support convenient online debugging as the Flash MCU products do.

However, the 1-wire online debugging mode still has limitations as follows.

1. It does not support offline testing, namely, debugging after power-off. Since the code is directly downloaded to PRAM for debugging through the simulator, the PRAM code will be lost once the power is removed.
2. The online code debugging simulation experience cannot keep highly consistent with the actual code running as Flash based MCU does. During the actual CMT216xA product running, it loads the OTP code into PRAM. When it needs to sleep, it will enter the SDN (Shut Down) mode with PRAM powered down. Therefore, each wakeup is a equivalent to code re-execution (reload and execute from start). However, a remarkable difference in the 1-wire debugging mode is that the code is always running with PRAM not losing power.

Due to the above 2 limitations, the online debugging is unsuitable in scenarios requiring offline testing, such as power consumption test and outdoor distance test. In this case, if these tests are required, users need to verify the code in the final running mode by burning the code into OTP, thereby, extra chips are consumed in the case as the OTP chip can only be burnt once.

Herein, another debugging method is introduced to compensate the above 2 limitations in the 1-wire online debugging mode. This method is the Bootloader offline debugging mode with its debugging mechanism same as that of the Flash MCU.

1. Write the Bootloader code and burn it into the OTP of CMT216xA.
2. After the CMT216xA starts, first, load the Bootloader code in the OTP to PRAM for running.
3. Then, through the Bootloader, reload the code to be run from the external SPI Nor Flash into the PRAM (where user program area is reserved).

Other words, it only needs to consume one CMT216xA chip to burn the Bootloader code. The subsequent code to be debugged, programmed according to the Bootloader requirement, is compiled and burnt into Flash for offline running, which realizes the same burning and debugging mode as that of Flash MCU in an indirect way. This offline debugging mode is closer to the final running mode, however the wake-up (or power-on) code will be loaded twice each time with the first load occurring when the bootloader is loaded in the OTP, and the next one occurring when the user code is loaded through the bootloader code in Flash, thus the loading time will be a little longer.

Users can adopt both Bootloader offline debugging and 1-wire online debugging to have compensation by each other, with suggestions as below.

1. To get familiar with the CMT216xA function modules, suggest using 1-wire online debugging to achieve convenient debugging and get better product understandings.
2. Furthermore, after various required functions are already integrated altogether in a project, suggest using the Bootloader offline debugging to experience the actual operation.
3. Finally, when all the detail modification are finished, burn the code directly into OTP for complete verification.

3.2 Bootloader Debugging

The Bootloader offline debugging flow is shown in the below figure.

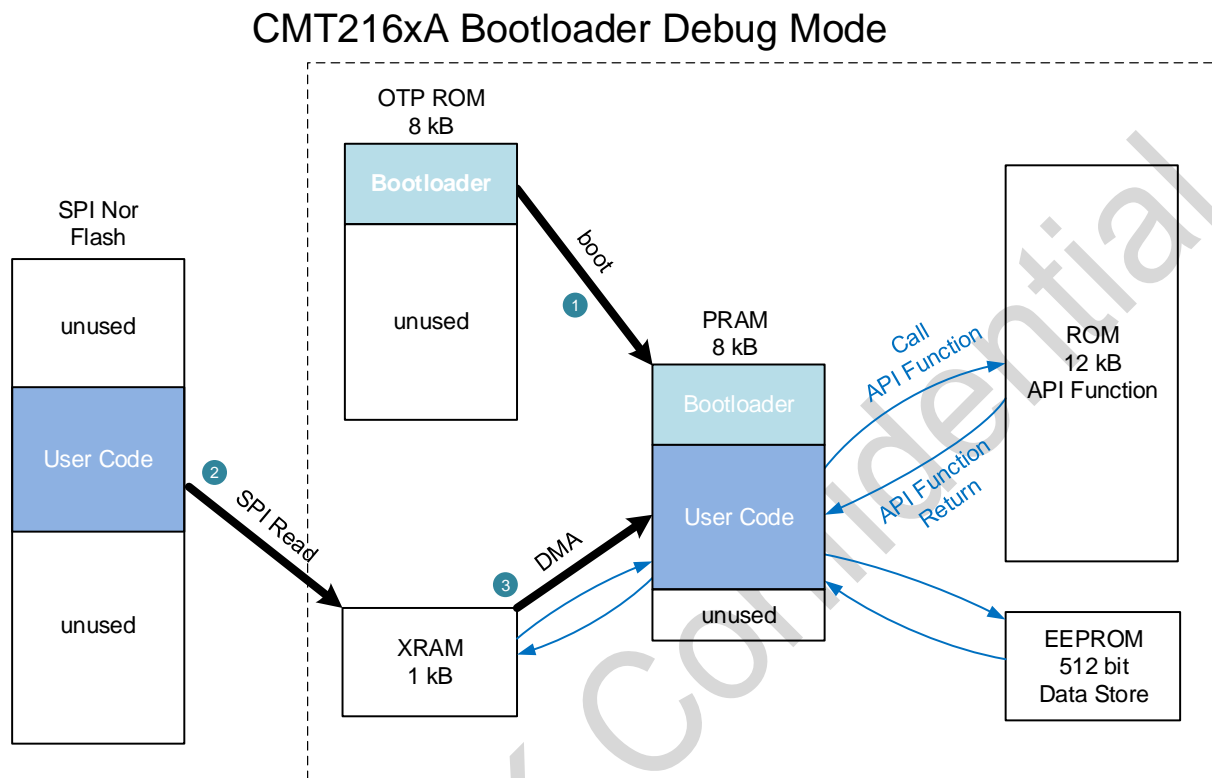


Figure 20. Bootloader Offline Debugging Flow

The debugging flow is as follows.

1. The Bootloader code is pre-burnt into OTP. When the system is powered on/waken up, the Bootloader code is copied (loaded) from OTP to PRAM through the internal boot.
2. After the bootloader code is loaded, the system runs according to the Bootloader code.
3. The main function of the Bootloader code is as follows.

Access the external NOR Flash through the SPI interface, and transfer the user code as data to XRAM in batches. Then pass the data (user code) in XRAM through the internal DMA mode of the system. Go to PRAM then repeat this step and move it to PRAM in batches. Finally integrate the pieces altogether until get the complete user code.

4. After the user code is transferred from the external Flash to PRAM, the Bootloader points the program pointer (PC) to the user code entry. At this time point, the system operation control is handed over to the user code.
5. After the user code completes the corresponding function, it enters the Shut Down mode (SDN) with PRAM powered off, thereby the system remains in deep sleep mode until the next wakeup, then it will re-execute from step 1 again.

3.3 Bootloader Program Flow

The major function of Bootloader program is loading user code from external Nor Flash to the user code area specified by PRAM and executing the program. Since the CMT216xA series chip does not support SPI to read the data from Nor Flash directly to the PRAM area, thus XRAM is used to have buffer processing, which copies the data content to the designated area in PRAM through internal DMA. The program process flow is shown in the below figure.

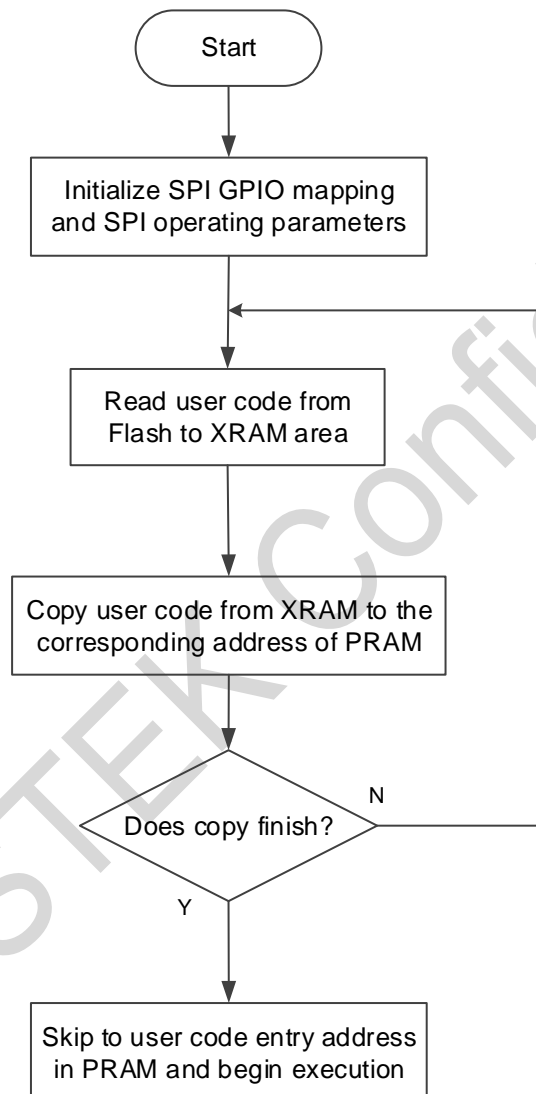


Figure 21. Bootloader Program Flow

3.4 User Program Interface

The Bootloader processing flow discussed above is just for user understanding. Actually users are not necessary to know the Bootloader processing details (unless the user needs to write the Bootloader code to achieve specific functions). For a successfully debugged Bootloader program that supports normal operation, users just need to know how the user program interfacing with the Bootloader with no need for learning the meaning of each code line. The interface between Bootloader and user program has 2 key points as follows.

- **User program interface**
- **User program interrupt service**

The user program interrupt service is discussed in Section 3.6. This section mainly discusses the user program interface. The user program interface is important, since only if the user code is processed conforming to the user program interface specified by the Bootloader, the Bootloader can jump to the user program entry correctly after the Bootloader operating completes otherwise the user program cannot run normally. The key information about the user program interface is shown in the below figure.

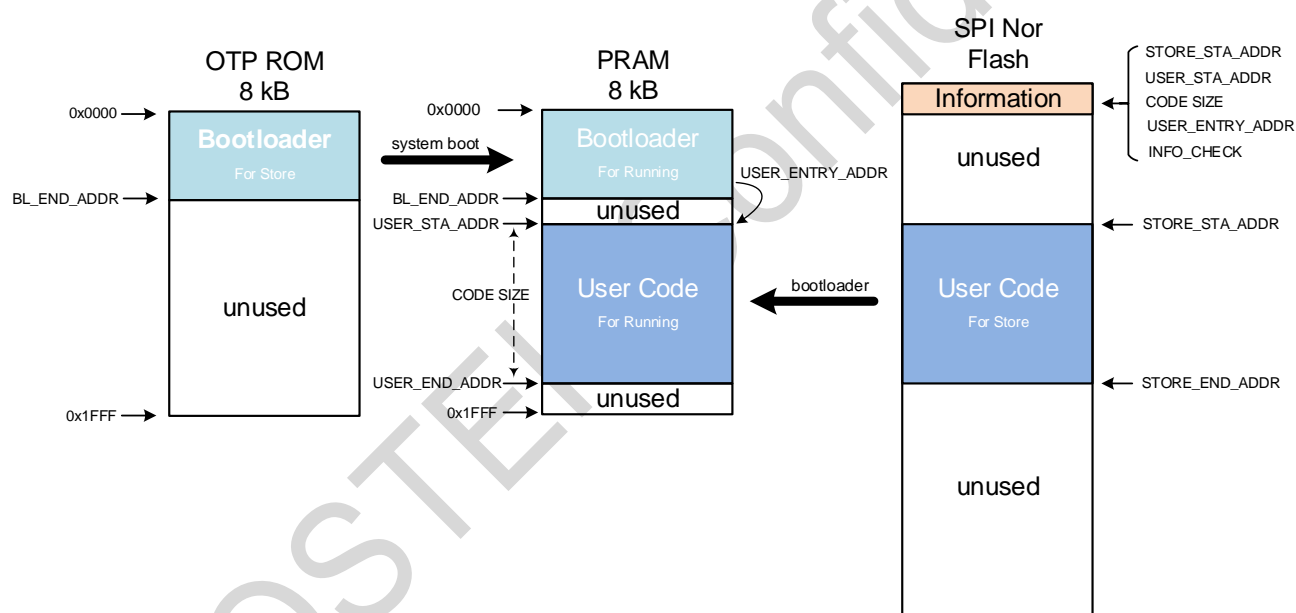


Figure 22. User Program Interface Schematic Diagram

- **BL_END_ADDR**: the end address of Bootloader code.
- **USER_STA_ADDR**: the start address of the user program in PRAM.
- **USER_END_ADDR**: the end address of the user program in PRAM.
- **CODE_SIZE**: user program length, namely $USER_END_ADDR - USER_STA_ADDR$.
- **USER_ENTRY_ADDR**: the user program entry address to which it skipped after the Bootloader ends running, that is **USER_STA_ADDR** in general.
- **STORE_STA_ADDR**: the start address of the user program stored in Flash.
- **STORE_END_ADDR**: the end address of the user program stored in Flash.

As shown in Figure 22 , an Information area established in Flash to store the key information used by Bootloader. The key information is as follows.

1. The start address of user program stored in Flash, namely, the Bootloader code accesses the Flash data (code) according to this address.
2. The start address of user program to be stored in PRAM, namely, the Bootloader copies the user program data from Flash to this target address in PRAM.
3. The length of the program, namely, how much content the Bootloader needs to copy.
4. The user program's entry address, which Bootloader jumps to, namely the start address of the user program stored in PRAM.
5. Information verification, used to ensure the validity of the information in the area.

Notes:

1. The above diagram is an example to show the key information and interface between Bootloader and the user program. Users can try Bootloader code writing after getting proficient in Bootloader usage.
2. To let users get a quick start of Bootloader, a Bootloader code example is designed by CMOSTEK according to the sample model shown in the above figure and adopted in the Demo board of CMT2168A-EM. That is, The Bootloader program has been burnt into the CMT2168A on the demo board. Users can write specific user program according to the required interface, then offline debugging mode can be used. Please refer to Section 3.5 for more details.

3.5 Bootloader Examples

This section discusses how to setup the interface with Bootloader in user programs. However the Bootloader code detail is not discussed here. Users can get more details from the Bootloader code.

Note:

1. This Bootloader code is open source, which uses can adopt as an equivalent OTP model for testing during the debugging phase. However It does not ensure reliable design in real product. In addition, for users who need to write their own Bootloader code, this code is just for reference. Users themselves need to ensure that the Bootloader code used can meet the target requirements.

3.5.1 Information Area

In the reference Bootloader example, the specified information area is stored in Flash, with a start address 0x0000 and end address 0x000B, which is a total of 12 bytes. The information structure in Bootloader is defined as follows, which is shown in the below figure as well.

//Information structure in Bootloader

```
typedef struct APP_CODE_INFO_STRU
```

```
{
```

```
uint32_t src_addr;           // STORE_STA_ADDR for Flash
```

```
uint16_t dest_addr;         // USER_STA_ADDR for PRAM
```

```
uint16_t code_size;         // CODE_SIZE
```

```
uint16_t app_entry_point;    // USER_ENTRY_ADDR
```

```
uint16_t app_valid_flg;    // INFO_CHECK
}STRU_APP_CODE_INFO;
```

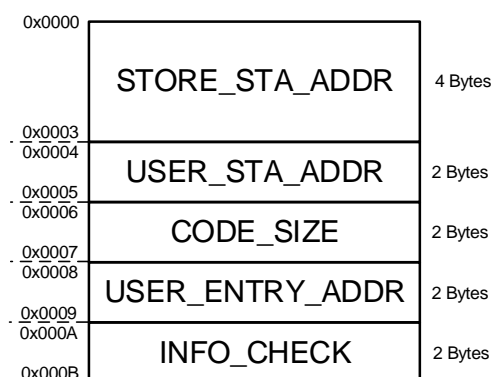


Figure 23. Information Structure Schematic Diagram

In above,

- src_addr, the start address storing user program in Flash, occupying 4 bytes,.
- dest_addr, the start address that user program is copied into in PRAM, occupying 2 bytes.
- Code_size, the length of user program to be copied, occupying 2 bytes.
- app_entry_point, the user program entry address that is skipped to after the Bootloader processing ends, occupying 2 bytes..
- app_valid_flg, judge whether the content in information area is valid, which is 0xAA55 corresponding to this program example.

In order to generate the HEX file (to be burnt into Flash) with information data after compiling a user program, it needs to create a new absolute assembly file with a suffix A51 in file name, like VECTOR_APP.A51, as shown in the below figure.

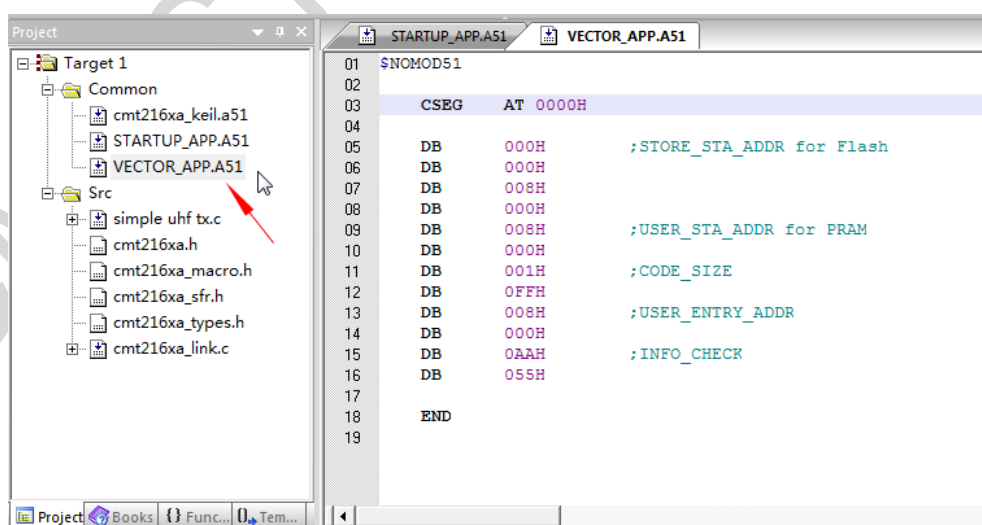


Figure 24. Information Data Compiling File

In the above figure, the simple code in right side is the 12-byte data with its absolutely location address starting at 0x0000, and the data content corresponds to the structure mentioned above. Corresponding to the values shown in the above figure, the meanings are as follows.

- The user program stored in Flash starts at 0x00000800.
- The start address of the user program copied into PRAM is 0x0800.
- The length of the user program to be copied is 0x01FF, namely 0.5 kB code.
- The user program entry address that Bootloader jumps to is 0x0800.

3.5.2 Startup Code

The previous section discusses the interface between Bootloader and user program, which is the key step for debugging using Bootloader. Another key step is to compile the user program according to the start address required in the above interface information. Otherwise it will start from address 0x0000 by default if not specified.

Notably, the startup code file in Figure 23 is STARTUP_APP.A51, but not STARTUP.A51 (STARTUP.A51 is automatically generated when Keil creates a new project). That is because, to let the user program be compiled according to the start address specified by the above interface, users need to modify the startup code. The figure in below shows the code line needs to be modified.

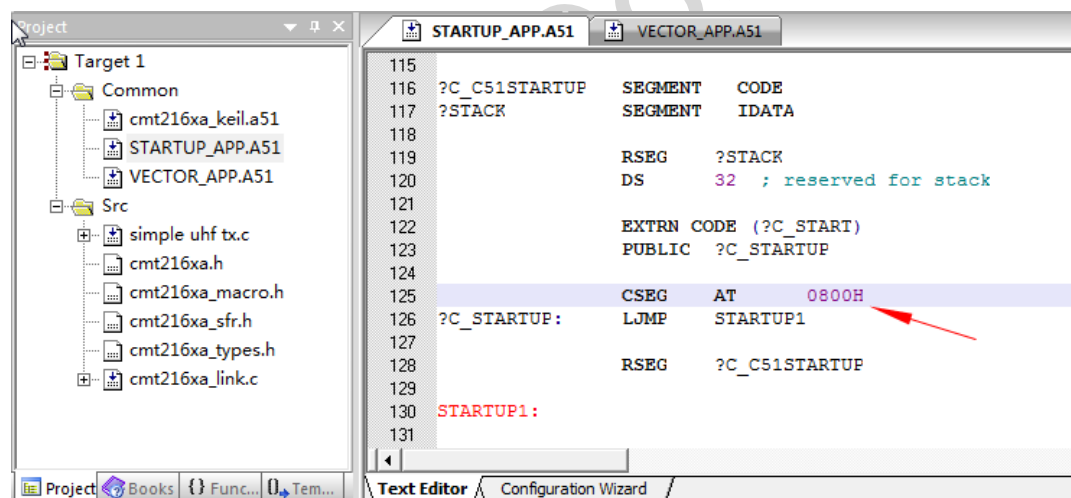



Figure 25. Schematic Diagram for Modifying Startup Code in User Program

The code line marked in the above figure shows the start address of the user program to be compiled. The default is absolutely located at 0x0000. To co-operate with Bootloader, it needs to be modified to 0x0800. To distinguish from the startup code used in the normal simulation mode, a separate copy is saved and named as STARTUP_APP.A51.

The last step is to set the compiler to compile the location address. Click *target options...* (namely the magic stick icon ) in the Keil IDE interface, and select *BL51 Locate* in the pop-up window. The settings are shown in the below figure. *Use Memory Layout from Target Dialog* is checked on by default, please uncheck this option As shown in the below figure. User need to uncheck *Code Range* to fill in the address. According to the above values, the starting address should be 0x0800.

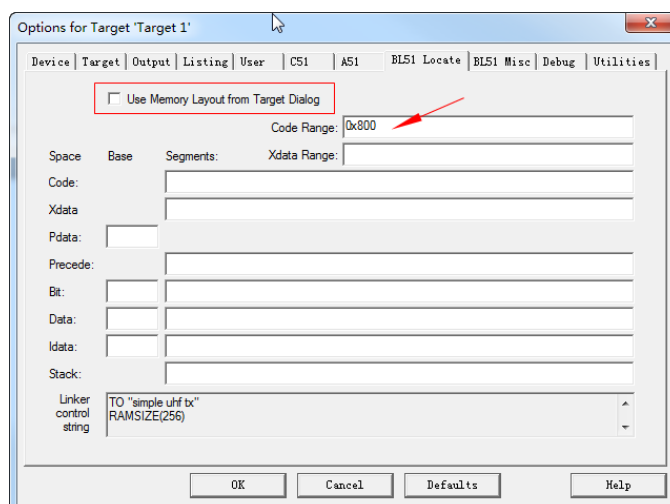


Figure 26. BL51 Locate Settings

By now, the operations need for offline debugging the user program with Bootloader are complete. A summary is as below.

1. Write the information file required for accessing Bootloader, which is VECTOR_APP.A51 in the example.
2. Modify the startup code. It is recommended to create a new STARTUP_APP.A51 file according to the example.
3. Modify the BL51 locate start address to the corresponding value.

When the above settings are completed, the HEX file compiled in the Keil IDE environment can be burnt into the Flash on the CMT2168A Demo board. After the burning is completed, the Demo board can be powered on again, then offline function verification and testing can be performed, as shown in Figure 27 and Figure 28.

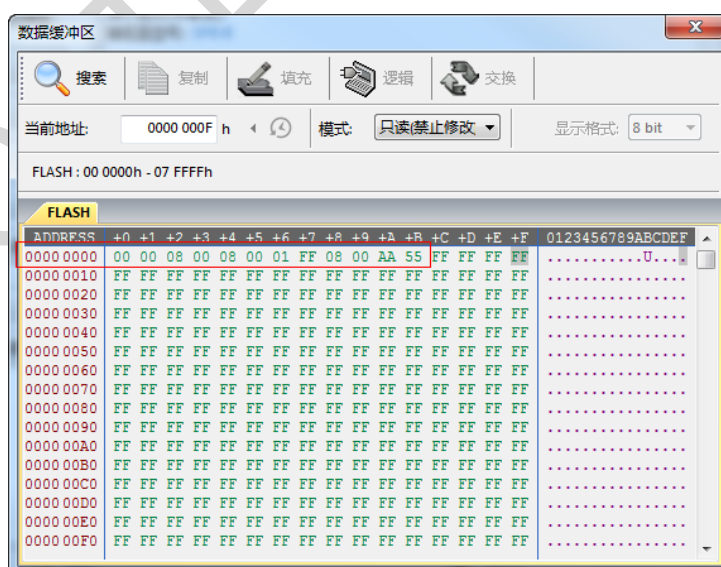


Figure 27. Flash Writer Loading HEX File (Information Area)

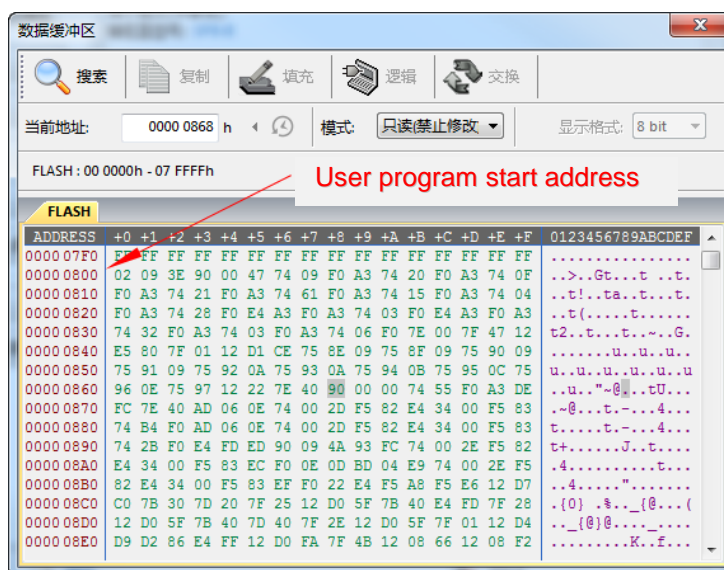
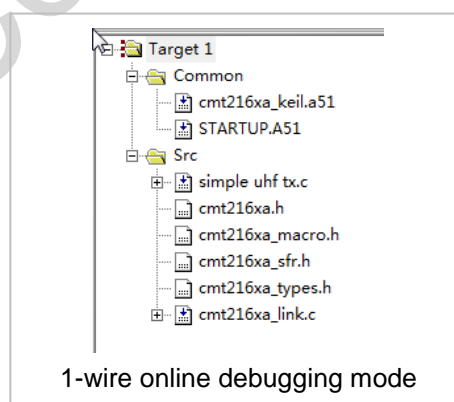
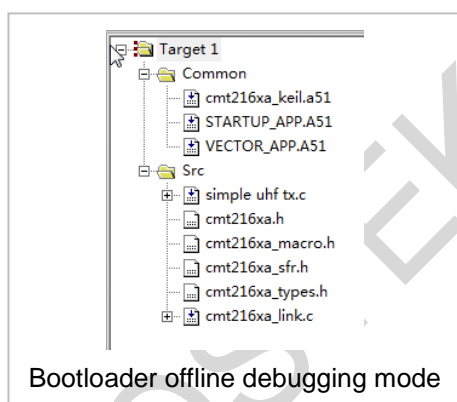


Figure 28. Flash Writer Loading HEX File (User Code Area)

Note:

1. VECTOR_APP.A51 and STARTUP_APP.A51 in this example are used by any new project, so users can copy and use it directly when creating a new project. In addition, for the same project, it may need to switch between 1-wire debugging and Bootloader debugging. In this case, to switch back to debug with 1-wire, users only need to remove the above two files from the project, restore the original STARTUP.A51 and modify *BL51 Locate* to the default value.



3.6 User Program Interrupt Service

In the Bootloader debugging mode, users need to pay some special attentions. Since the 8051 system's interrupt vector is located in the Bootloader code area as well after a vector reset, the Bootloader code needs to reserve the interrupt service entry mechanism for user programs.

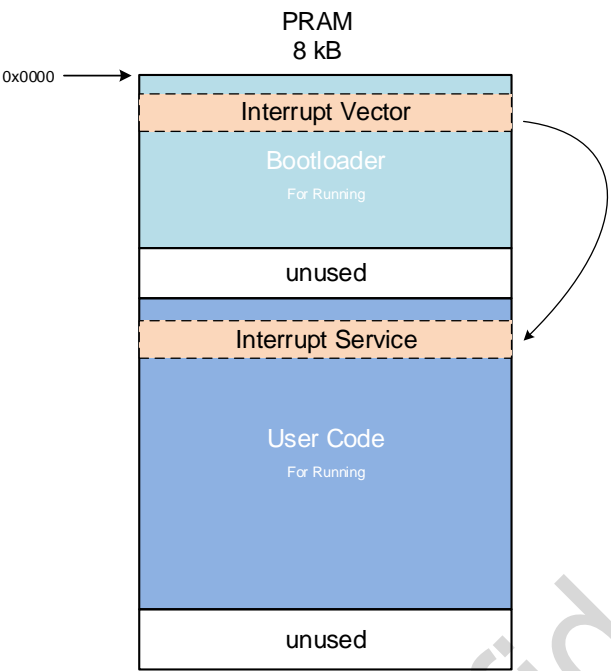


Figure 29. Schematic Diagram for Interrupt Processing in Bootloader Mode

According to the Bootloader processing method in this example, it writes an interrupt service program in the Bootloader code, and reserves a function pointer (entry) to be used by the user program.

For example, the code of Timer 1 interrupt handling program in the Bootloader code is as follows.

```
// TIMER1
void system_timer1_interrupt(void) interrupt 3 using 0           // 1B, 8n+3
{
    sys_push_bank0_r0_2_r7_to_stack();                         // push r0-r7 to stack
    g_timer1_interrupt_bank_store = sys_get_sfr_bank();         // store the sfr bank value
    g_timer1_interrupt_hv_store = sys_store_hv_interface();     // store hv interface
    if (system_timer1_interrupt_service_routine != NULL)       // run the interrupt service routine
    {
        system_timer1_interrupt_service_routine();
    }
    sys_restore_hv_interface(g_timer1_interrupt_hv_store);      // restore hv interface
    sys_set_sfr_bank(g_timer1_interrupt_bank_store);           // restore the sfr bank value
    sys_pop_bank0_r7_2_r0_from_stack();                        // pop r7-r0 from stack
}
```

The above code is quoted from the interrupt processing part of the Bootloader. The function is to preprocess the Timer1 interrupt, including:

- push stack for Rn register group.
- protect site for accessing Bank0 or Bank1 in Block1 area.
- protect site for accessing Block 0 area.
- Determine whether the interrupt service function is defined (existing), and if there is a definition, call the interrupt service program (function pointer) in the user program.
- recover site for accessing Block0 area.
- recover site for accessing Bank0 or Bank1 in Block1 area.
- pop stack for Rn register group.

All other interrupt processing is based on this template. Just the interrupt service program is different (the code line in red in below is the interrupt service program). The program is described in the Bootloader program as follows:

```
typedef void (*INTERRUPT_SERVICE_ROUTINE)(void);
xdata INTERRUPT_SERVICE_ROUTINE system_timer1_interrupt_service_routine _at_ 0x0019;
```

In above,

- INTERRUPT_SERVICE_ROUTINE, defining a function pointer type.
- system_timer1_interrupt_service_routine, defining a function pointer pointing to Timer 1 interrupt service.
- _at_, referring to the absolutely storage location address of the function pointer.

The above interrupt processing in the Bootloader program is just for users to have easy understanding of the interrupt handling

mechanism in this mode. Actually, when writing a user program, users just simply need to write a Time 1 interrupt routine and assign this function to the corresponding function pointer, as described in the following program.

```
void timer1_interrupt_service_routine(void)
{
    TF1 = 0;                // Clear Flag
    TH1 = 0xC1;             // Reset TL1 Value
    TL1 = 0x80;
    systimer++;
}

void init_timer1_mode1(void)
{
    TMOD = 0x10;            // as mode 1: 16-bit timer/counter
    TH1 = 0xC1;
    TL1 = 0x80;
    system_timer1_interrupt_service_routine = timer1_interrupt_service_routine;
    ET1 = 1;                // enable interrupt
    TR1 = 1;                // enable timer1
}
```

Init_timer1_mode1, as the Timer1 initialization configuration function, in addition to initializing the predetermined operating mode and parameters of Timer1, it will assign interrupt service function pointer as shown in the code line in red in the program,. The above timer1_interrupt_service_routine program function is executed when processing the specific Timer1 interrupt. During system's operating, the Timer1 interrupt generation flow is follows.

1. The Timer1 interrupt flag is valid, and the Timer1 interrupt is generated (the PC is at the user program area at this time).
2. The program pointer PC jumps according to the Timer1 interrupt vector, and executes system_timer1_interrupt (the PC jumps to the Bootloader area).
3. process according to system_timer1_interrupt, push stack and protect site first (the PC is at the Bootloader area at this time).
4. In the system_timer1_interrupt function, jump according to the function pointer when executing to system_timer1_interrupt_service_routine, then execute timer1_interrupt_service_routine (PC jumps to the user program area).
5. After executing timer1_interrupt_service_routine, return to system_timer1_interrupt (the PC jumps back to the Bootloader area).
6. Have the remaining site recovery and pop stack processing in the system_timer1_interrupt function (the PC is in the Bootloader area at this time);
7. Exit the interrupt (the PC backs to the user program area).

As a summary, the interrupt handling in user programs in the Bootloader offline debug mode is as follows.

1. Write an interrupt service function.
2. Assign the interrupt service function to the corresponding interrupt function pointer.

According to the above discussion, in the Bootloader offline debugging mode, the user program part actually does not need to write the interrupt service function pointing to the interrupt vector, since it is already written in the Bootloader code. Thus users only need to write an ordinary program function and assign it to the function pointer that is called in the corresponding interrupt service function.

Notes:

1. As mentioned above, in practice, it is often needed in a project to switch between 1-wire simulation and Bootloader debugging. Therefore, it is recommended that the interrupt service function that actually points to the interrupt vector is still provided in the project code (or copy from Bootloader into user code). The two modes are switched through conditional compiling.

3.7 Considerations for Bootloader Offline Debugging

1. Compared with the 1-wire online simulation debugging mode, the above mentioned Bootloader offline debugging mode is similar to the real product operating model. Therefore, when establishing new project, it is recommended for users to take both the 2 debugging mechanisms into account, namely, the 1-wire debugging and Bootloader debugging mechanism (discussed in Section 3.5 and 3.6).
2. Although the Bootloader debugging mode is close to the real product operating model, they still differ especially in the wake-up time. In the normal operating model, OTP loads code into PRAM directly at a fast speed. However in the Bootloader mode, the Flash content is read to the XRAM through SPI, and then moved to the PRAM, which takes a longer time (relative to copying from OTP to PRAM). Users need to pay special attention to this in some application scenarios.
3. The Bootloader model is recommended for debugging, but not for using as a commercial model. On the one hand, the external Flash added will increase the product cost, the more important is the user program is stored in Flash without encryption, which can be read out by any Flash writer, thus users may have core interest loss.
4. When the Bootloader code is burnt into the chip OTP, there are 2 running modes (burning options): Run mode and User mode. The demo board CMOSTEK provides selects the User mode, and only in User mode it can support 1-wire debugging. When using 1-wire debugging, users need to ensure unplugging Flash on the Demo board or not connecting it to the CMT2168A. In this case, although the Bootloader code is still loaded into the PRAM upon power-on, the program will put the system in the stop mode since no valid Flash information is read. At this time, if the 1-wire debugger is connected, it still supports downloading the code into PRAM for simulation through 1-wire. (the original Bootloader in PRAM will be overwritten). However, it should be noted that the precondition for supporting 1-wire debugging in User mode is that the 1-WIRE/B0 pin is used as the 1-wire function, and this function has the highest priority to ensure this pin does not respond to any settings and operations in the program.
5. When burning Bootloader code the chip OTP with Run mode selected, the 1-wire interface will be disabled forcibly, namely, the chip no longer supports 1-wire online simulation debugging. At the same time, the function of the 1-WIRE/B0 pin is restored to the B0 function and will respond to the corresponding settings and operations in the program. For the CMT216xA series chip, the Run mode is the final operating mode. If the S3S programming interface is disabled at this time, the chip is in the encryption mode with external access prohibited.

4 OTP Programming

4.1 Programming Hardware Establishment

As shown in the below figure, when have the OTP programming of CMT216xA, users need to have hardware preparation as follows.

- The target chip/board to be burnt. The CMT2168A-DM board (with QFN32 burning socket) is the target board as shown in the below figure. The DM board supports burning through connecting with burning socket where the target chip is being placed.
- The CMT216xA programmer, which is an online type programmer, requires operating online programming through the host PC, suitable for the programming and verification function in the development phase. For offline mass production programming, please refer to the CMT series product offline writer operation guide for details. This document focuses on the online type programming only. The programming interface and considerations for online and offline programming are basically the same though.
- Computer and USB cable (A to B type).

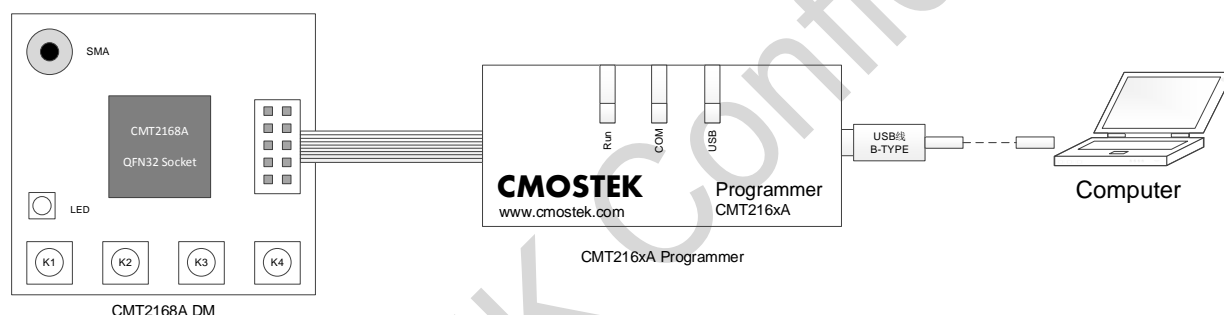


Figure 30. CMT216xA Online Programming Schematic Diagram

Notes:

1. If performing programming on the board, users need to reserve the S3S programming interface and VPP/B1 pin on the board in advance, and need to ensure that other peripherals on the board do not affect the normal operation of these programming pins.
2. The correspondence between the S3S programming interface along with the VPP pin and the 10-pin box-header is shown in the below figure.

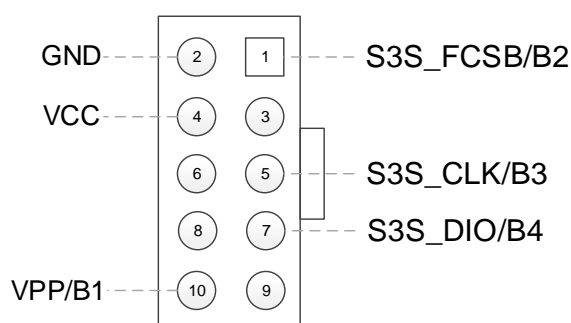


Figure 31. Correspondence between CMT216xA 10-pin Box-header and Programming Pins

4.2 Programming UI Screen

After the establishment of programming hardware, users can open programming UI screen as shown in the below figure.

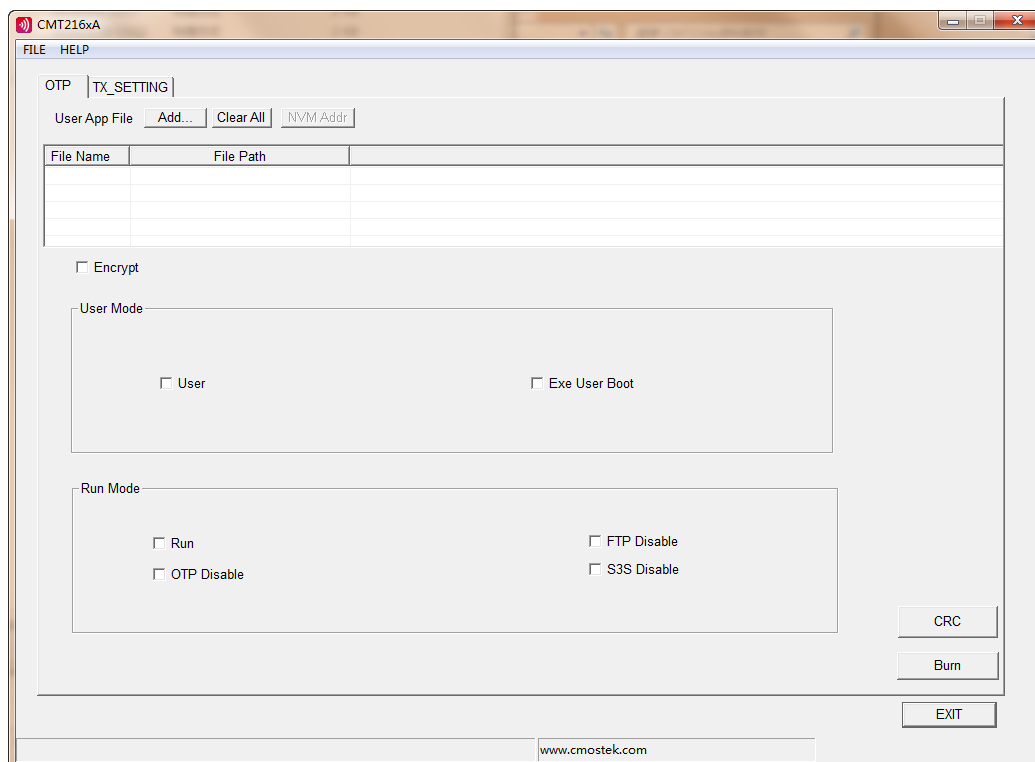


Figure 32. CMT216xA Online Programming Screen

In above figure,

- Click *Add...* to add the HEX file to be burned.
- Check the *User* option in the *User* panel to select burning as User mode.
 - Check *Exe User Boot* option in the *User* panel to select burning as User mode. In this case, upon power-on/wake-up, it will start the boot process and load the OTP code into PRAM. When this option is checked on, the CMT2168A-EM board is burnt with the Bootloader code, supporting both the Bootloader offline debugging mode and 1-wire online simulation mode.
- Check the *Run* option in the *Run* panel to select burning as Run mode.
 - Check *OTP Disable* option in the *Run* panel to disable accessing internal OTP from external.
 - Check *FTP Disable* option in the *Run* panel to disable accessing internal EEPROM from external.
 - Check *S3S Disable* option in the *Run* panel to disable accessing the bus from S3S programming.

Note:

1. The programming screen is subject to change per version upgrading and the related documents will be updated accordingly. Despite the potential change, basic options in the screen remain similar. If users find the programming screen mismatching with the document, please figure out it based on above option definitions.

5 Revise History

Table 2. Revise History Records

Version No.	Chapter	Description	Date
0.4	All	Initial version	2018-04-13
0.5	All	Add Bootloader debugging information	2019-07-01

CMOSTEK Confidential

6 Contacts

CMOSTEK Microelectronics Co., Ltd. Shenzhen Branch

Address: 2/F Building 3, Pingshan Private Enterprise S.T. Park, Xili, Nanshan District, Shenzhen, Guangdong, China

Tel: +86-755-83231427

Post Code: 518071

Sales: sales@cmostek.com

Supports: support@cmostek.com

Website: www.cmostek.com

Copyright. CMOSTEK Microelectronics Co., Ltd. All rights are reserved.

The information furnished by CMOSTEK is believed to be accurate and reliable. However, no responsibility is assumed for inaccuracies and specifications within this document are subject to change without notice. The material contained herein is the exclusive property of CMOSTEK and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of CMOSTEK. CMOSTEK products are not authorized for use as critical components in life support devices or systems without express written approval of CMOSTEK. The CMOSTEK logo is a registered trademark of CMOSTEK Microelectronics Co., Ltd. All other names are the property of their respective owners.