Construction of Embedded System Platform which Based on μC/OS-Ⅱ and ARM7 Kernel Microprocessor

Yujun Bao
School of Electronic Information & Electric Engineering,
Changzhou Institute of Technology
CZU
Changzhou, China
goldlake@163.com

Xiaoyan Jiang
School of Electronic Information & Electric Engineering,
Changzhou Institute of Technology
CZU
Changzhou, China

***Abstract*—Embedded System has more and more applications today. Introduced Embedded Operating System in Embedded System has been a tendency of Embedded system development. Embedded Operating System can avoid different Embedded System hardware's difference, it can greatly reduce the development costs, shorten the research cycle. μC/OS-Ⅱ is a kind of excellent Embedded Operating System which is an open source, tiny kernel, consuming little resource and high performance in real-time feature. In accordance with more and more 32 Bit ARM7 kernel Microprocessor used in Embedded System, and here introduces the transplantation method of Embedded μC/OS-Ⅱ Operating System which based on ARM7 kernel Microprocessor. So the construction of Embedded System Platform which based on μC/OS-Ⅱ and ARM7 kernel Microprocessor can have been realized eventually. The platform can efficiently simplify the software development's program flow, shorten the project's research cycle and improve the whole system's performance greatly.**

Keywords- Embedded System; Embedded Operating System; ARM7 kernel; μC/OS-Ⅱ

I. INTRODUCTION

The so-called Operating System transplantation is that a real time Operating System kernel can be operated in others Microprocessors. That is special code for special CPU. Most codes of μC/OS-Ⅱ are written by C programming language, and this is very convenient to transplant μC/OS-Ⅱ. But according to the different Microprocessors, the users still need using assembly language to write some programs which are related to the different Microprocessors' hardware. This is because when μC/OS-Ⅱ reads or writes registers, it only relies on assembly language.[1]

With the μC/OS-Ⅱ Operating System's requirements, there are three files to be needed when μC/OS-Ⅱ transplanted into a certain Microprocessor. And the whole transplantation mainly surrounds the work which creating these three files. These files showed next:

◆ In the C language header file OS_CPU.H, some data types which are not concerned in compiler needed to be defined. These data types include the used stack data type and its growing direction. Some soft interrupts which are concerned in ARM7 kernel are defined in this file, too.

◆ Another one OS_CPU_C.C is a source file of C program. The file mainly includes the task stack initialization function of μC/OS-Ⅱ and some user functions which called by μC/OS-Ⅱ Operating System.

◆ The last one OS_CPU_A.S is a assemble program source file. It is a clock ticks interrupt service function in practice. It is also a task switch function which used to quit interrupt. And it always runs the task of highest priority when μC/OS-Ⅱ enters the multitask environment firstly.

II. CREATING THE FILE OS_CPU.H

A. *Defining data types which are concerned in complier*

Because different Microprocessors have different word length, so the transplantation of μC/OS-Ⅱ includes a series of data types' definition, and this make the transplantation possible. Especially for the data types used in C language, such as 'short', 'int', 'long' and so on, these data types can't used in μC/OS-Ⅱ directly. Because these data types are concerned in the compliers' type, they can't be transplanted. So these data types are always defined as Integer Data Structures in many cases, they are transplantable.

B. *Definition of system soft interrupt functions*

Interrupt is a kind of hardware mechanism, but in μC/OS-Ⅱ Operating System, some important task functions' implementation are supposed to depend on interrupt level code. So, at this time, μC/OS-Ⅱ Operating System needs a piece of preprocessor directive to simulate the interrupt. It's like a hardware interrupt, so it is called soft interrupt. Most Microprocessors' kernel can provide this kind of soft interrupt directive. And in ARM7 kernel, it is the 'SWI'.

In order to make the bottom layer interface function be independent of Microprocessor's state, and the responded task function needn't know the function's exact location when the task called. The used SWI directive should be as bottom layer interface. And the different functions can be distinguished by means of different soft interrupt Function Number. The ADS1.2 compiler always uses '_swi' to declare a non-existent function. And once the non-existent function is called, a piece of SWI directive should be inserted where the called function, and specifying the Function Number.

Here shows some important functions' soft interrupt function in μC/OS-Ⅱ:

```
_swi(0x00) void OS_TASK_SW(void);
                              //Switching function between tasks
_swi(0x01) void _OSStartHighRdy(void);
                       // Running the task of highest priority firstly
_swi(0x02) void OS_ENTER_CRITICAL(void);
                                              //  Interrupt Off
_swi(0x03) void OS_EXIT_CRITICAL(void);
                                              //  Interrupt On
_swi(0x80) void ChangeToSYSMode(void);
                                   //Switching to system model
```

These functions' concrete codes existed in the files OS_CPU_C.C and OS_CPU_A.S.

According to the requirements of μC/OS-Ⅱ, all the C files in the application system must include the file INCLUDES.H. So a normal practice is make the file OS_CPU.H be included in the file INCLUDES.H.

III. CREATING THE FILE OS_CPU_C.C

In the file OS_CPU_C.C, the transplantation specification of μC/OS-Ⅱ Operating System requires to create ten simple functions: OSTaskStkInit( ), OSTaskCreateHook( ), OSTaskDelHook( ), OSTaskSwHook( ), OSTaskIdleHook( ), OSTaskStatHook( ), OSTimeTickHook( ), OSInitHookBegin( ), OSInitHookEnd( ), OSTCBInitHook( ). But not all the functions must be created except OSTaskStkInit ( ) function. In accordance with the μ C/OS- Ⅱ Operating System's requirements, the hinder nine functions can be set to empty functions if users don't want to use.

A. Writing the function OSTaskStklnit( )

In μ C/OS- Ⅱ Operating System, the two functions OSTaskCreate ( ) and OSTaskCreateExt( ) initialize task's stack structure by means of calling OSTaskStkInit ( ) function, and all the registers are stored in their respective stacks. When the called function implement the initialization, it clears the concerned registers in accordance with the ARM7 kernel Microprocessor's practical stack direction (Provided the stack direction is Full Descending.), and returns the stack top's pointer at last.

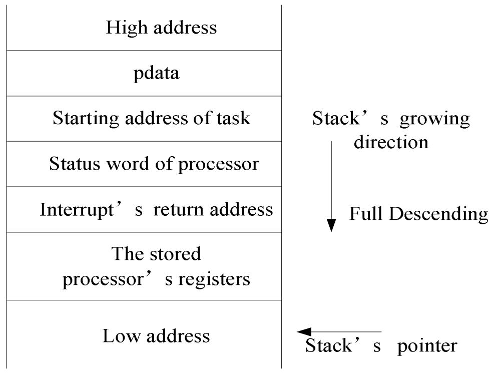

Figure 1. Stack structure's initialization (The parameter pdata is sent to stack.)

Fig.1 shows the process that OSTaskStkInit ( ) initializes the task stack when μC/OS-Ⅱ Operating System creates task. After the initialization, the OSTaskStkInit ( ) returns the stack top's pointer at last.

B. System soft interrupt functions

In the file OS_CPU_C.C, the soft interrupt functions which written by C language are always introduced first. These include all the soft interrupt functions except function Number Zero (OS_TASK_SW ( )) and function Number One (OSStartHighRdy ( )). And the functions Number Zero and Number One are written by assembly language in the file OS_CPU_A.S. This because C language can't provide exact stack structure which required by these two functions. And another reason, if the function which used to switch tasks is written by assembly language, it is very convenient to jump.

The nature of soft interrupt functions which written by C language introduced here: Because of using soft interrupt, after changing the ARM7 kernel SPSR register's corresponding Control Bits, the purpose of changing Microprocessor's running model can be realized by means of the characteristic that the SPSR register's value would be copied to the GPSR register during the soft interrupt is quitting.[2]

IV. CREATING THE FILE OS_CPU_A.S

According to transplantation specification of μC/OS-Ⅱ Operating System, there are four functions (OSStartHighRdy( ), OSCtxSw( ), OSIntCtxSw( ), OSTickISR( )) needed to write, and they must be written in assembly language. And the function which used to extract the Number Zero and Number One soft interrupts' Function Number should be written by assembly language in this file, too.

A. Extracting the soft interrupt's Function Number

The system always carries out different functions in accordance with different soft interrupt Function Number. In ARM7 kernel Microprocessor family, the LR register is always used to store the return address of subroutine or anomaly. The LR register can get the code of SWI directive.[3] And the soft interrupt's Function Number is included in the code of SWI directive. So the soft interrupt's corresponding Function Number can be get by reading the directive code's corresponding bit filed. In consideration of the ARM7 kernel is three-stage pipeline architecture, and the program's running state is probably ARM state or Thumb state, so they must be processed differently. Fig.2 shows the concrete flow chart:

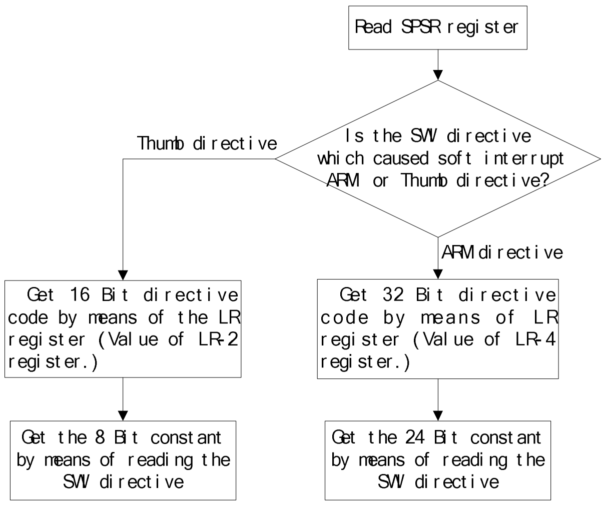
Read SPSR register
Is the SWI directive which caused soft interrupt ARM or Thumb directive?
Thumb directive
ARM directive
Get 16 Bit directive code by means of the LR register (Value of LR-2 register.)
Get 32 Bit directive code by means of LR register (Value of LR-4 register.)
Get the 8 Bit constant by means of reading the SWI directive
Get the 24 Bit constant by means of reading the SWI directive

Figure 2. Extracting the soft interrupt's Function Number

B. *Writing the functions OSStartHighRdy ( ), OSCtxSw ( ) and OSIntCtxSw ( )*

In μC/OS-Ⅱ Operating System, the dispatching function OSSched ( ) is realized by calling Task Switching Function OS_TASK_SW ( ), and OS_TASK_SW ( ) function finishes the whole task switching by calling function OSCtxSw ( ).

The function OSCtxSw ( ) calls the function OS_TASK_SW ( ) which is used to switch task finally. And most source codes of function OSIntCtxSw ( ) are almost same as OSCtxSw ( ) function. The concrete difference between them is that the function OS_TASK_SW ( ) needed to store CPU registers. The interrupt service program (ISR) has been stored CPU registers according to the μC/OS-Ⅱ Operating System's requirements. So, the function OSIntExit ( ) would quit after the final interrupt has been processed (If interrupt nesting existed.). And at this time, the task switching needn't store CPU registers again.

During μC/OS-Ⅱ Operating System's transplantation, in order to make the system code be highly capable, the function OSIntCtxSw ( ) can be implemented by means of calling the function OSCtxSw ( ). And the concrete realization method is storing CPU registers again, but it doesn't have any affection for the whole system.[5]

Because the CPU registers must be operated during writing the function OSCtxSw ( ), so, the function OSCtxSw ( ) should be written by assembly language. Fig.3 shows the function's flow chart.

The function has some differences compares with the requirements of μ C/OS- Ⅱ Operating System's transplantation, but it doesn't affect the system's operation. And to some extent, it has exemplified that μC/OS-Ⅱ is really a very excellent Embedded Operating System, it can be cut according to the system's need.

The last flow in Fig.3 is realized by running ARM directive 'LDMFD SP!, {R0 – R12, LR, PC}^'. Because in μC/OS-Ⅱ, task can't be operated in anomaly model, and it only can be operated in user or system model.[4]

μC/OS-Ⅱ Operating System starts tasking environment by means of the function OSStart ( ), and OSStart ( ) calls the function OSStartHighRdy ( ) finally. The OSStartHighRdy ( ) function's writing should be followed the requirements of μC/OS-Ⅱ Operating System's transplantation. And the concrete method is same as OSCtxSw ( ) function. The only difference between them is the function OSStartHighRdy ( ) needn't store registers when operates task dispatching first.

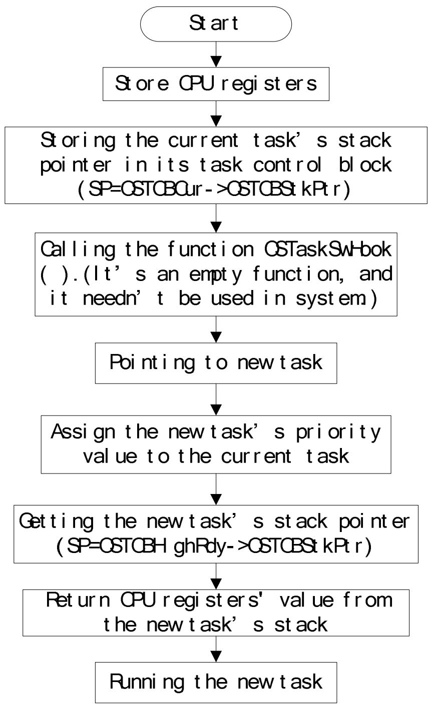
Start
Store CPU registers
Storing the current task' s stack pointer in its task control block (SP=OSTCBCur->OSTCBStkPtr)
Calling the function OSTaskSwHook ( ).(It' s an empty function, and it needn' t be used in system)
Pointing to new task
Assign the new task' s priority value to the current task
Getting the new task' s stack pointer (SP=OSTCBHighRdy->OSTCBStkPtr)
Return CPU registers' value from the new task' s stack
Running the new task

Figure 3. OSCtxsw() function's flow chart

After having created the several transplantation files, in theory, μ C/OS- Ⅱ Operating System can be operated in ARM7 kernel Microprocessor normally. But before μC/OS-Ⅱ Operating System's first operation, the function OSTickISR ( ) which is the system time function must have been written.

μC/OS-Ⅱ Operating System requires users to provide a periodical time source, and the time source can realize the functions of delaying time and overtime. Usually, the periodical time source is provided by interrupt function OSTickISR ( ). [6]And in fact, the implementation is essentially OSTickISR ( ) function calling the function OSTimeTick ( ). So, the interrupt task OSTickISR ( ) must be implemented by ARM7 kernel Microprocessor's hardware timer practically, and the interrupt level should be set the highest.

In μC/OS-Ⅱ Operating System's transplantation, the interrupt service subroutine must be written by assembly language. And Fig.4 shows the writing specifications about ARM7 kernel interrupt service program's flow chart.

Fig.4 shows that the realization of system timer interrupt function OSTimeISR ( ) by means of changing the part of carrying out user program into the system function OSTimeTick ( ) of μC/OS-Ⅱ Operating System.

It is also important to note that after all interrupts in Fig.4 having been implemented and quit the interrupt processing, the system should choose the highest priority task of task list again. And then the system is switched to the highest task. The switching work has been finished by the function OSIntCtxSw which existed in the file OS_CPU_A.S. That is to say the initial interrupted task is not necessarily the highest priority task in task list at this time. Because a higher priority task which than the initial task may be activated during the interrupt's implementation process. Certainly, if the initial interrupted task is still the highest priority task in task list when all interrupts have been finished, the system need not to switch tasks, it runs again by means of recovering the task's CPU registers directly.

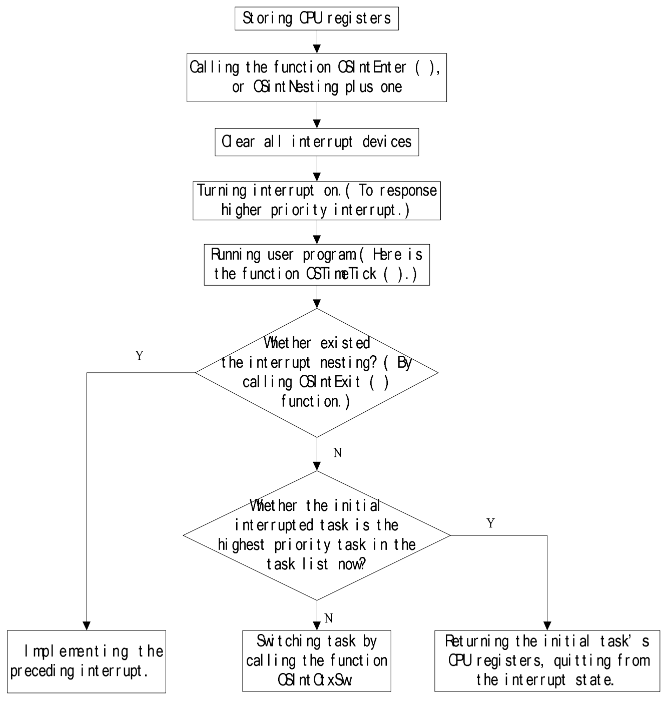
Storing CPU registers
Calling the function OSIntEnter ( ), or OSintNesting plus one
Clear all interrupt devices
Turning interrupt on.(To response higher priority interrupt.)
Running user program(Here is the function OSTimeTick ( ).)
Whether existed the interrupt nesting?(By calling OSIntExit ( ) function.)
Y
N
Whether the initial interrupted task is the highest priority task in the task list now?
Y
N
Implementing the preceding interrupt.
Switching task by calling the function OSIntCtxSw
Returning the initial task's CPU registers, quitting from the interrupt state.

Figure 4. Interrupt service program's flow chart of μC/OS-Ⅱ Operating System. (Take system time interrupt for instance.)

V. Conclusions

So far, all μC/OS-Ⅱ Operating System's transplantation into ARM7 kernel Microprocessor have been finished. In order to prove the transplantation's correctness and reasonableness, the whole code must be tested before normal use. This can be realized by means of writing several small tasks which according to the practical Microprocessor model of application system. The test can be useful to finding the transplantation code's defects, and the transplantation can also be improved by modifying.

The application tasks may be added in the Embedded System Platform which Based on μC/OS-Ⅱ Operating System and ARM7 Kernel Microprocessor after the transplantation having been proved accurately.

References

[1] Jean J. Labrosse (USA), μC/OS, The Real-Time Kernel [M], Beijing: BEIHANG UNIVERSITY PRESS, 2003.5

[2] Xiong Zhenhua, Liao Jiaping, Based on S3C44B0x microprocessor and UCOS-Ⅱ operating system [J],China Water Transport,,(2006)01.

[3] Xiang Huaikun, Liang Songfeng, Yuan Yuan, Design of Network-Enabled Traffic Signal Controller Based on CAN and uCOS-Ⅱ[J], Journal of Shenzhen Polytechnic, (2008)03.

[4] Liu Miao, Wang Tianmiao, Wei Hongxing, Real-time Analysis of Embedded CNC System Based on uCOS-II[J], Computer Engineering, (2006)22.

[5] Wang Lei, Wang Yaonan, Chen Sisi, The Application Of ucos-ii In Embedded Intelligent Vision Surveillance System [J], Control & Automation,(2008)04.

[6] Li Bei, Peng Chuwu, Building Hardware Application Layer in UCOS Transplanting[J], Electronic Engineering & Product World, (2006)16.