Do It Yourself O.S. translated from
Chinese to English, Bjorn Ove Arthun

I believe many people to question multiple processes are executed in parallel and made a headache, right? Lot of people are using the state machine to solve a few people are using the operating system to resolve the operating system can make your code with a single task as simple and intuitive, but the cost of that came a little big, a little film 51 so much. I do own several operating systems, are super lightweight, the simplest one in which the whole "operating system" is only a dozen lines of code, dozens bytes, but it can well support multiple tasks in parallel. want to know how to do it? come with me, play to change to change.

One thing I must declare: The purpose of this post is the church you how to write a scene OS, rather than give you a code provided for all OS versions, are also sample code, so do not say it is nothing function. LAJI kind of thing. If you write all the features, and the one you do not want to see the estimate, and secondly, have lost the flexibility of no value

2008-8-14 01:53:00 Upload
Downloads: 827
Teach you self-51 operating system
Download Points: Money -1 ¥
Directly into the theme, first posted an operating system out of the demonstration. We can see what the original operating system can be made simple. Of course, here to affirm you, this stuff is actually not a real operating system, which in addition to parallel multi-tasking outside there is no other function but everything from a simple beginning, they get to know it, you can depending on the application needs, it will expand into a real operating system.

Well, the code here.
Place the following code in KEIL compiler directly into in each task () function "task_switch ();"? There marked a breakpoint, you can see that they really are "simultaneously" in execution.

```
#include <reg51.h>

#define MAX_TASKS 2 // task slot number. The actual number of jobs must be one to
#define MAX_TASK_DEP 12 // maximum stack depth Minimum not less than 2, a conservative value of 12.
unsigned char idata task_stack [MAX_TASKS] [MAX_TASK_DEP]; // task stack.
unsigned char task_id; // currently active task number


// Task switching function (task scheduler)
void task_switch () {
task_sp [task_id] = SP;

if (++ task_id == MAX_TASKS)
task_id = 0;

SP = task_sp [task_id];
}

// Task load function. The specified function (parameter 1) Load the specified (parameter 2) the task of the slot. If the tank had its own task, the task of
```

```
the original is lost, but the system itself is an error does not occur.
void task_load (unsigned int fn, unsigned char tid) {
task_sp [tid] = task_stack [tid] + 1;
task_stack [tid] [0] = (unsigned int) fn & 0xff;
task_stack [tid] [1] = (unsigned int) fn >> 8;
}

// After the macro is called, it will never return from the specified task
starts running task scheduling.
#define os_start (tid) {task_id = tid, SP = task_sp [tid]; return;}

/ * Here is the test code ================= ============================
============ * /

void task1 () {
static unsigned char i;
while (1) {
i ++;
task_switch (); // compile here after marked breakpoint
}
}

void task2 () {
static unsigned char j;
while (1) {
j + = 2;
task_switch (); // compile here after marked breakpoint
}
}

void main () {
// Here loaded with two tasks, so when defining MAX_TASKS also be defined as 2
task_load (task1, 0); // function into the task1 slot 0
task_load (task2, 1); // will task2 function into that slot 1
os_start (0);
}
```

Due to space limitations I have the code has been simplified, and delete most of the comments, you can download the source code package, which complete the annotation, and file with KEIL project, but also to lay the breakpoint, press ctrl + f5 on the line .

Now take a look at the principles of this multi-tasking system:

The multi-tasking system precisely, called "cooperative multitasking."
The so-called "collaborative" refers to when a task runs continuously without releasing the resources to other tasks and there is no way to get the opportunity to run the chance, unless the task active release CPU.
In this case, the CPU is released by task_switch () to complete .task_switch () function is a special function, we can call it "task switcher."
To know how to switch tasks, we must first look at the knowledge of the stack.

There is a very simple question, because it is too simple, so I believe we did not pay attention to before:
We know that, either CALL or JMP, all the current program flow interrupted to ask what is the difference between CALL and JMP?

You will say: CALL can RET, JMP does not work right, but the reason why is Shane CALL past can be used to jump back RET, JMP past can not be used to jump back RET it.??

Obviously, CALL some way to save certain information before the break, but RET instruction execution breakpoint before returning, that is used to retrieve this information.
Needless to say, we all know that, "certain information" is the PC pointer, and "somehow" is a push.
Luckily, in 51 years, and the stack is the stack pointer can be arbitrarily modified, so long as you are not afraid of death if the stack before executing the RET will modify how look down.?:
When the program executes CALL, the stack in the subroutine just pushed breakpoint address removed, and the address of a function of the pressure, then RET After execution, the program will jump to this function go.
In fact, as long as we get rid of the stack before RET, the program will be able to jump to the task place to go, but not limited to CALL Lane pushed address.

Focus here ......
First we have to open a separate task for each piece of memory, this memory is dedicated to the task as the corresponding stack, which want the CPU to the task, just who the stack pointer memory block on the line.
Next we construct such a function:

When the task calls the function, the current stack pointer is saved in a variable, and replace the stack pointer of another task. This is a task scheduler.

OK, now we just correct populated these stacks the original content, and then call this function, the task scheduler can be up and running.
So these are the original contents of the stack where they come from? This is the "Task Load" function to be doing things.

Before starting the task scheduling function of each task entry address on the above mentioned "mission dedicated memory block" on the line! Oh, by the way, this "mission dedicated memory block" called "private Stack" private stack means is that each task stack are private, each task has a own stack.

The words are said, and I believe we have to understand how to do it:

1. Assign a number of memory blocks, each memory block into a number of bytes: Here, the "number of memory blocks" is the private stack, in order to run several tasks have less number of blocks allocated. The "number of bytes of memory for each sub-block" is deep stack. Remember, each sub-layer transfer program requires 2 bytes. If you do not consider the interruption, call depth layer 4, which is an 8-byte stack depth should be about.

unsigned char idata task_stack [MAX_TASKS] [MAX_TASK_DEP]

Of course, there are things not forget, is the preservation of the stack pointer. Otherwise, the light has the stack should know how to take the data from which the address ah
unsigned char idata task_sp [MAX_TASKS]

The above two areas used to hold information about the task, we gave it a concept called "Task groove." Some people call it "mission heap," I think it is a "slot" more intuitive

Yes, there is task number. Otherwise, how do you know which is currently running task?
unsigned char task_id
The current run is stored in the task slot 1, the value is 1, when you run the task slot 2, the value is 2 ....

2. The construction task scheduling function letter:
```
void task_switch () {
task_sp [task_id] = SP; // save the current task stack pointer

if (++ task_id == MAX_TASKS) // task number to switch to the next task
task_id = 0;

SP = task_sp [task_id]; // stack pointer points to the private system stack next task.
}
```

3. Load task:
The low and high bytes of each task separately into the function address task_stack [task number] [0] and task_stack [task number] [1]:

For ease of use, write a function: task_load (function name, job number)

```
void task_load (unsigned int fn, unsigned char tid) {
task_sp [tid] = task_stack [tid] + 1;
task_stack [tid] [0] = (unsigned int) fn & 0xff;
task_stack [tid] [1] = (unsigned int) fn >> 8;
}
```

4. Start Task Scheduler:
The stack pointer points to any one of the private stack task, execute RET instruction note that this is very knowledgeable Oh, never played the stack will not turn people's heads a little bend: This RET, RET where to go hey, do not? RET has forgotten before the stack pointer points to a function of the entrance. Do you RET as RET, you see it as another type of JMP like to understand.

```
SP = task_sp [task number];
return;
```

This is done four things, the task "parallel" execution begins. You can write a function like ordinary writing task function, simply (at present, so to speak) Note that at the appropriate time (for example, where previously transferred delay) calls click task_switch (), to give up control of the CPU to other tasks on the line.

Finally said efficiency.
The cost of this system is the multi-task every time you switch consume 20 machine cycles (CALL and RET are considered including a), expensive? Expensive for many multi-tasking system implemented with a state machine approach, in fact, further efficiency not so high --- case switch and if () you can not imagine so cheap.

I have to say about memory consumption is, of course, can not be denied that multitasking really account for memory, but suggest that you do not stare at the old compiler that the following lines "DATA = XXXbyte". The value does not make sense, do not stack into account. Comparative provincial memory multitasking

mechanism, I will talk in the future.
Generally speaking, this system is suitable for multi-tasking real-time requirements and memory requirements for small applications, I'm running on 36M clocked STC12C4052 found one, a task switching is less than 3 microseconds.
Using KEIL write multitasking system techniques and precautions
C51 compiler lot, KEIL is one of the more popular one. All the examples I listed must use the KEIL. Why? Not because KEIL so good with it (of course, it is really great), but because there with to some of the characteristics of KEIL, if the change to other compilers, through the translation of down is not a problem, but could be up and running stack dislocation, loss and other contexts terrible mistake, because the characteristics of each compiler is not the same. Therefore, here make it clear on this point.
However, at the beginning I have said, the main purpose of this post is to explain the principles, as long as you can digest these few examples, it can also write their own hands for other compilers OS.
Well, talk about KEIL features of it, look at the following function:

```
sbit sig1 = P1 ^ 7;
void func1 () {
register char data i;
i = 5;
do {
sig1 = sig1!;
} While (- i);
}
```

You will say, this function is nothing special thing Oh, do not worry, you will compile it, and then expand the assembly code and then look!:

```
193: void func1 () {
194: register char data i;
195: i = 5;
C: 0x00C3 7F05 MOV R7, # 0x05
196: do {
! 197: sig1 = sig1;
C: 0x00C5 B297 CPL sig1 (0x90.7)
198:} while (- i);
C: 0x00C7 DFFC DJNZ R7, C: 00C5
199:}
C: 0x00C9 22 RET
```

Look yet? Used in the function of the R7, but not for R7 protection!
Someone will jump up: Is there anything surprising, because in the upper function useless to R7 ah Oh, you're right, but only half the story: in fact, KEIL compiler made in the agreement, in former tune out all possible release function registers. typically of conditions, in addition to the interrupt function, other functions can also be arbitrarily modified without having to push all registers protected (this is not so, but now temporarily think so, to rice mouthful Well, I will talk soon in).
This feature is what use is it? There! When we call the task switching function, the object to be protected in all registers can be ruled out, that is, only to the need to protect the stack!
Now before we go back to the example of task switching function:

```
void task_switch () {
task_sp [task_id] = SP; // save the current task stack pointer
if (++ task_id == MAX_TASKS) // task number to switch to the next task
task_id = 0;
SP = task_sp [task_id]; // stack pointer points to the private system stack next task.
}
```

I did not see, did not register a protection, expand the compilation look really no protection register.

Well, now I give you pour cold water, look at the following two functions:

```
void funci () {
register char data i;
i = 5;
do {
sigl = sigl!;
} While (- i);
}
void func2 () {
register char data i;
i = 5;
do {
funci ();
} While (- i);
}
```

Parent function fun2 () call in funci (), look at the expanded assembly code:

```
193: void funci () {
194: register char data i;
195: i = 5;
C: 0x00C3 7F05 MOV R7, # 0x05
196: do {
! 197: sigl = sigl;
C: 0x00C5 B297 CPL sigl (0x90.7)
198:} while (- i);
C: 0x00C7 DFFC DJNZ R7, C: 00C5
199:}
C: 0x00C9 22 RET
200: void func2 () {
201: register char data i;
202: i = 5;
C: 0x00CA 7E05 MOV R6, # 0x05
203: do {
204: funci ();
C: 0x00CC 11C3 ACALL funci (C: 00C3)
205:} while (- i);
C: 0x00CE DEFC DJNZ R6, C: 00CC
206:}
C: 0x00D0 22 RET
```

Look no? Function func2 () variables used in the register R6, in funci and func2 where no protection.

Hearing this, you may have to hop the jump: funci () was not used and R6, why should protect the right, but the compiler is how to know funci () it is useless to R6 from the call?? relations in the inferred.

Not at all true, KEIL will be based on the direct relationship between the functions call for the function allocates registers, neither protection, without conflict, KEIL Terrific !! Oh wait, do not be happy, for the multi-tasking environment again try:

```
void funci () {
register char data i;
i = 5;
do {
sigl = sigl!;
} While (- i);
}
void func2 () {
register char data i;
i = 5;
```

```
do {
sigl = sigl!;
} While (- i);
}
Expand the assembly code to see:
193: void funci () {
194: register char data i;
195: i = 5;
C: 0x00C3 7F05 MOV R7, # 0x05
196: do {
! 197: sigl = sigl;
C: 0x00C5 B297 CPL sigl (0x90.7)
198:} while (- i);
C: 0x00C7 DFFC DJNZ R7, C: 00C5
199:}
C: 0x00C9 22 RET
200: void func2 () {
201: register char data i;
202: i = 5;
C: 0x00CA 7F05 MOV R7, # 0x05
203: do {
! 204: sigl = sigl;
C: 0x00CC B297 CPL sigl (0x90.7)
205:} while (- i);
C: 0x00CE DFFC DJNZ R7, C: 00CC
206:}
C: 0x00D0 22 RET

See it? Haha, this time the gods does not count out because there is no
relationship between the two functions are called directly, so that the compiler
does not produce a conflict between them, the result is assigned a pair of
conflicting registers when the task switching from funci () to func2 () when,
funci register contents () gave the destroyed what we can try to compile the
following program:
sbit sigl = P1 ^ 7;
void funci () {
register char data i;
i = 5;
do {
sigl = sigl!;
task_switch ();
} While (- i);
}
void func2 () {
register char data i;
i = 5;
do {
sigl = sigl!;
task_switch ();
} While (- i);
}
Here we are only examples, so you can still manually assign different registers
to avoid register conflicts, but in a real application, due to the switching
between tasks is very random, we can not predict a time which will not register
conflict, different distribution register method is not desirable. so, how to do
it?
That's all:
sbit sigl = P1 ^ 7;
```

```
void func1 () {
static char data i;
while (1) {
i = 5;
do {
sig1 = sig1!;
task_switch ();
} While (- i);
}
}
void func2 () {
static char data i;
while (1) {
i = 5;
do {
sig1 = sig1!;
task_switch ();
} While (- i);
}
}
```

. The two functions of all the variables into static on the line can also do this:

```
sbit sig1 = P1 ^ 7;
void func1 () {
register char data i;
while (1) {
i = 5;
do {
sig1 = sig1!;
} While (- i);
task_switch ();
}
}
void func2 () {
register char data i;
while (1) {
i = 5;
do {
sig1 = sig1!;
} While (- i);
task_switch ();
}
}
```

That is, in the scope of the variable does not switch tasks, and other variables used up, then switch tasks. In this case, although both tasks will still destroy each other's register contents, but the other party has no interest in the contents of the register.

Mentioned above, it is the "variable coverage" of the problem. Now we talk about the system to "variable coverage."

Variables are two, one is a global variable, one is a local variable (in this case, to a local register variable count variable).

For global variables, each will be assigned to a single address.

For local variables, KEIL will make a "coverage optimization," that there is no direct relationship between the calling function variables shared space because it is not used simultaneously, so it will not conflict, which is a small memory 51, it is a good thing.

But now we enter the world of multi-tasking, which means that the two are not directly call relations function is actually performed in parallel, and the

space can not be shared. How to do? One Benbanfa coverage optimization function is turned off. Oh, really stupid.
One solution is relatively simple, do not close the coverage optimization, but those will need to be overcome within the scope of the task (in other words, before the variable exhausted calls task_switch () function) in all the variables into static (static) ie be here to mention for beginners, "static" you can be understood as "global" as its address space has been reserved, but it is not global, it is only in the definition of its access to the curly braces {} Lane .
Static variables have a side effect, that even if the function exits, the memory is still occupied. So write the task function, try to switch tasks only after the end of variable scope, unless the scope of this variable is very long (long time), It will affect other real-time tasks. only in this case only be considered within the scope of the variable across tasks and variables declared as static.
In fact, as long as the programming ideas relatively clear analysis, there is little need for variable across tasks. That is, static variables are not many.
Having a "cover" We say that the "re-entry."
The so-called re-entrant is a function at the same time there are two different replica process may be bad for beginners to understand, I give you an example:
There is a function is called in the main program will be called in the interrupt, the time in the main program if the legitimate call, an interrupt occurs, what happens?

```
void funci () {
static char data i;
i = 5;
do {
sig1 = sig1!;
} While (- i);
}
```

3:00 assume funci () being executed to i =, an interrupt occurs, once the interrupt call to funci (), the value of i was destroyed when the interruption is over, i == 0.
Above that is the traditional single-tasking system, so the chances of re-entry is not great, but in a multi-tasking system, we are prone to re-entry, see the following examples:

```
void funci () {
....
delay ();
....
}
void func2 () {
....
delay ();
....
}
void delay () {
static unsigned char i; // Note that this is declared as static, not static, then cover affirmed problem occurs and declared as static reentrant problem occurs trouble ah.
for (i = 0; i <10; i ++)
task_switch ();
}
```

Two tasks are executed in parallel calls the delay (), this is called re-entry problem is that the re-entry after two replicas are dependent variable i to control the loop, and the variable across tasks, so that the two tasks are modify the value of i.
Re-entry only to prevent the main, that is to say try not to let re-entry

occurs, such as the code into the following way:
#define delay () {static unsigned char i; for (i = 0; i <10; i ++) task_switch
();} // i still defined as static, but in fact is not the same function, so the
assigned address different.
void func1 () {
....
delay ();
....
}
void func2 () {
....
delay ();
....
}
Instead of using the macro function, it means that each call is at a separate
code replica, the actual use of the two delay memory address is different, and
re-entry problem disappears.
However, this method brings the problem is that every call to a delay (), will
have a delay of object code, if the code is a lot of delay, it will cause a lot
of rom space occupied. There are no other ways?
My knowledge is limited, only a last resort:
void delay () reentrant {
unsigned char i;
for (i = 0; i <10; i ++)
task_switch ();
}
After adding reentrant stated, the function can support re-entry, but with care,
affirmed after re-entry, low function efficiency!
Finally Incidentally Break because not too much to say, do not open a separate
chapter of the.
Break with the ordinary wording no difference, but in a multi-tasking system
currently exemplified in the stack because of pressure, so you want to use to
reduce the use of using the stack (by the way, do not call the subroutine, the
same is to reduce stack pressure)
By using, must be closed off with #pragma NOAREGS absolute register accesses,
interrupt if there have to call the function, together with the function but
also on the scope #pragma NOAREGS as shown in Example:
#pragma SAVE
#pragma NOAREGS absolute must register to access the closed // use using
void clock_timer (void) interrupt 1 using 1 // use using is to reduce the
pressure on the stack
{
#pragma RESTORE
The above wording changed after the break fixed 4 bytes of the stack. That is,
if you do not interrupt the task stack deep as 8, it is now necessary to set the
8 + 4 = 12.
Also tell nonsense, interruption in the treatment must be something less, to be
marked on the line, the rest of the matter to the corresponding task to deal
with.
Now summarize briefly:
When switching tasks to ensure that no registers across tasks, otherwise the
register cover between tasks using static variables to solve
When switch tasks to ensure that no variable across tasks, otherwise the address
space between tasks (variable) cover. Use static variables to solve
Do not call the two different tasks at the same time call the same function,
otherwise the reentrant cover using reentrant affirmed resolve

III. Forward to the operating system

The previously mentioned examples, in addition to the ability to perform multiple tasks in parallel, there is no other function, which for a very simple system is enough, but if the system is a little more complicated, for example:

1. The need to delay a task
2. a task needs to wait until the completion of a transaction.
3. The task is not all fit in the beginning, with the commencement of the process flow, loaded at different times for different tasks. Tasks have a life cycle, after the transaction is completed, the end of the mission and want to remove.

Here is the operating system of several typical features:
1. Sleep Mechanism
2. The message mechanism
3. Process Mechanism

The fact that these very easy to implement, if the contents of the previous few all know, it is easy to imagine how these mechanisms are implemented.
This time we have to say something about how these mechanisms are implemented.

1. Sleep and delay (also known as sleep latency, here deliberately renamed the "delay" in order to prevent confusion and sleep) mechanisms:

For each task to define a byte counter:
unsigned char idata task_sleep [MAX_TASKS]; // task sleep timer

The counter is decremented by one (unless it is 0, or 0xff) at each timer interrupt
void clock_timer (void) interrupt 1 using 1
{

...
// Delay processing tasks
i = MAX_TASKS;
p = task_sleep;
do {
if (* p! = 0 && * p! = -1) // is not zero, nor is 0xff, then the task delay value minus 1. 0xff indicates that the task has been suspended, the reigning timer wake
(* P) -;
p ++;
} While (- i);
}

When a task switch, check the value of 0. task_sleep whether the non-zero skip this task is not performed, the next task to check whether the execution conditions.
void task_switch () {
...
while (1) {
...
task_id ++; // task_id cut to the next without actually only increased by 1 so simple, even modulo here are merely exemplary, so do not write all the...

if (task_sleep [task_id] == 0) // 0 indicates that the task is not in the sleep / delay, so I skipped.
break;
}

```
...
}
Related macro:
task_sleep (timer) delay timer timer interrupt cycle. ranging from 0 to 254
task_suspend () Sleep If no other process wakes up, is never again executed
task_wakeup (tid) wake task number tid process


2. Tasks dynamically loaded and end:

In task_switch (), the discovery of the process when the value 0 task_sp not
saved stack pointer of the task, the task will disappear.
When the search for the next executable tasks, testing task_sp value is non-
zero, it means that the position 0. No task.
void task_switch () {

if (task_sp [task_id]! = 0) // If the task has not been deleted, then save the
current stack pointer.
task_sp [task_id] = SP;

while (1) {
task_id ++; // task_id cut to the next without actually only increased by 1 so
simple, even modulo here are merely exemplary, so do not write all the...

if (task_sp [task_id]! = 0) // actually here also check the value of task_sleep.
But that said nothing to do with now, I do to get rid of that part of the code
break;
}
...
}

Call task_switch () before clearing their own task_sp value.
#define task_exit () task_sp [task_id] = 0, task_switch ()
Incidentally, calling the task_delete (tid) to delete tid specified process.


3. The message mechanism:
Message mechanism is through sleep / suspend to achieve, can not be considered a
true message mechanism, but in many cases is enough for 51 such chips, the
resource utilization and efficiency is more important.

Define a message to the scale, each represents a message, each entry can hold a
task_id
event_vector [MAX_EVENT_VECTOR]
#define EVENT0 0
#define EVENT1 1
#define EVENT2 2
....
#define EVENTn MAX_EVENT_VECTOR - 1

When the process is to listen to this message, it can own task_id number
corresponding to the load vector for example, to monitor EVENT_RF_PULS_SENT,
simply:

event_vector [EVENT_RF_PULS_SENT] = task_id;

This process is called "message register", has been written as a macro
```

event_replace (eid)

So in the example just written
event_replace (EVENT_RF_PULS_SENT);
...
event_clear (EVENT_RF_PULS_SENT); // after the elimination of the use of the message

If the message is uncertain whether there are other processes have been listening, you can use event_reg (eid, reg), the original vector saved in the specified variable parameter reg, and after the message Out with event_restor (eid, reg) reduction come back.

static unsigned char old_event_vector;
event_reg (EVENT_RF_PULS_SENT, old_event_vector);
....
event_unreg (EVENT_RF_PULS_SENT, old_event_vector); // after use to restore the message

If the listener the message, the task must be lifted before exiting listener, otherwise it will lead to an error wake.
Is unable to receive a message when the process is in the Running state, and therefore must wait for a message goes to sleep / Delay Status complete the process:

Registration message
...
Goes to sleep
...

To wake up a process to wait for news, call event_push (eid) can be.
If eid specified message without listening process, the message is discarded.
Also note that, due to the message mechanism is borrowed dormancy mechanisms to complete, so if the process is not listening for messages dormant / delay in when the process is unable to receive the message, the message will be discarded. In this case, you should use task_wait_interrupt () to complete.
This occurs in the process of listening to the message generated the interrupt service routine of its mechanism is as follows:

Suppose that the task A and interrupt service A_ISV
A completed to fill the buffer, after filling into hibernation, waiting for news MESSAGE_A
A_ISV responsible when the timer interrupt occurs in the buffer of bytes written to the P1 port, after sending written MESSAGE_A

Typically, this process is not a problem, but when the following conditions occur, the task A will always be in waiting:
After filling out the A buffer, before entering dormancy, the timer interrupt occurs
The interrupt service routine progressively in the buffer have been processed and sent MESSAGE_A message. As mentioned earlier, the real message is sent to task_sleep value is set to 0
After the interrupt service returns, will also progressively task_sleep set value, it is not know at the moment, there is no way to know interrupt has occurred, so the information is essentially lost. Then the task will never wake up.

The solution is in the write buffer before the first set value task_sleep task

```
before the write buffer, and then go to sleep. Thus, when the interrupt occurs
task_sleep will have completed the assignment, so that messages are not lost.
The process has been written as a macro task_wait_interrupt (buffer operating
statement)
Writing a little awkward, but works very well if not used this style, you can
write code directly expand the macro:
task_setsuspend (task_id);
Buffer statement operation
task_switch ();

Sample code:

1. Call sub-tasks, sub-tasks and wait for the completion of sending a message.
Similar to the calling function. Compared with the benefits of calling a
function that can be activated simultaneously perform multiple sub-tasks, and
one can only call a function to perform.
void task2 () {
static unsigned char i;

i = sizeof (stra);

do {
stra [i-1] = strb [i-1];
task_switch ();
} While (- i);

event_push (EVENT_RF_PULS_SENT); // send the message (the essence of the message
is to monitor the process wake)

task_exit (); // End Task.
}

void task3 () {
static unsigned char event_backup; // save the original value for the signal
EVENT_RF_PULS_SENT in this case is actually no need to save, because
EVENT_RF_PULS_SENT not monitor other processes in a real application but is not
necessarily predictable...

event_reg (EVENT_RF_PULS_SENT, event_backup); // registration message, the
original value stored in event_backup in (the variable must be declared as
static)

// If you wait another task message generated in the process, use task_suspend
() on it.
strb [0] = 3, strb [1] = 2, strb [2] = 1; // fill buffer
task_load (task2); // load subtasks
task_suspend ();

event_unreg (EVENT_RF_PULS_SENT, event_backup); // message must restore the
original value before exiting

task_exit (); // End Task.
}

2. Wait for interrupt processing and send the message:
void clock_timer (void) interrupt 1 using 1
{
if (strb [0]! = 0 && strb [1]! = 0 && strb [2]! = 0) {
```

```
0 = strb [0];
1 = strb [1];
2 = strb [2];
push_event (EVENT_RF_PULS_SENT);
}
}

void task3 () {
static unsigned char event_backup; // save the original value for the signal
EVENT_RF_PULS_SENT in this case is actually no need to save, because
EVENT_RF_PULS_SENT not monitor other processes in a real application but is not
necessarily predictable...

event_reg (EVENT_RF_PULS_SENT, event_backup); // registration message, the
original value stored in event_backup in (the variable must be declared as
static)

// If you wait another task message generated in the process, use task_suspend
() on it.

task_wait_interrupt (
strb [0] = 3, strb [1] = 2, strb [2] = 1;
) // Fill buffer
// If written as follows:
// Strb [0] = 3, strb [0] = 2, strb [0] = 1;
// Task_suspend ();
// If occurs just after executing the first statement after the break, returning
from the interrupt, task calls task_suspend () will never wake up.

event_unreg (EVENT_RF_PULS_SENT, event_backup); // message must restore the
original value before exiting

task_exit (); // End Task.
}
```