EVM Code Documentation:
This documentation contains description of the project,changes and new additions made to the existing code.The projects are listed below:
1.EVM Interpreter
2.RT-Link
3.Stack Migration
4.Task Migration
5. Bug Fix in nano-rk
6. RT-Link Stats

1.EVM Interpreter:
evm_CCE_5x is the code which has interpreter integrated with nano-rk tasks.Each task has a separate instance of interpreter.
TCBI is a data structure which has the task data for the interpreter instance.This is similar to Task Control Block (TCB) of nano-rk.
The call to interpret is interpret(1) where 1 is the task id.All the data for the interpreter are initialized in the TCBI structure.
The functions in the interpreter are called by passing the argument of task id.This ensures that the function interprets the words from the respective tasks.
The dictionary is split into a global dictionary and task specific dictionary.TCBI[0] has the global dictionary and all the tasks can access the global dictionary to interpret the word.
Each task can access its own dictionary to interpret the word.If a task wants to add a new word then it has to add into the local dictionary.(Why?)
Note:New word is added into the local dictionary to avoid the errors while adding new word due to task pre-emption.More optimization is possible for this case.
With lcd code one can actually see the two tasks running at the same time and interpreting to give the respective results.

2.RT-Link:
evm_CCE_5x_rtl_original contains the RT-Link code.
The basic rf code is taken from the nano rk for atmel.
The rt-link code is also taken from atmel but some changes are made to it.

In rtl_defs.h:
#define RTL_NRK_TICKS_PER_SLOT 36
Atmel: 9. MSP: 36.//Timer scaling by 4 times.
#define SFD_TO_NEXT_SLOT_TIME   (29050) //27750.
Atmel: 27750. MSP: 29050.This is required for rt-link synchronization.This was done by observing on Osc Scope.

In rt_link.c:
In _rtl_rx_sync ():
MSP: _nrk_set_next_wakeup(2000); Atmel: _nrk_set_next_wakeup(250);
Timer scaling by 4 times and added more time to avoid synch error due to packet loss.
New line added: This line can also be added to nano-rk on Atmel.
MSP: if(last_nrk_tick>=_nrk_get_next_wakeup()-2) _nrk_os_timer_set(0);//fix srinivas.
This was added to avoid any interrupt during the synchronization of rt-link.The synchronization should take place atomically and any interrupt in between would be drastic.If any interrupt occurs then it would change the timers and rt-link would loose synch.
This was required especially in cases where there is a huge amount of packet loss.This was added when synch error was observed due to packet loss.
New line added:This line can also be added to nano-rk on Atmel.
MSP: _nrk_os_timer_set(0); added immediately after checking sfd.
This will ensure that there will be no interrupt while the synchronization is taking place.This

was added when synch error was observed due to packet loss.

MSP: timeout = tdma_start_tick+16; Atmel: timeout = tdma_start_tick+4;//Timer scaling by 4 times.

In _rtl_rx():
MSP: timeout+=16; Atmel: timeout+=4; //Timer scaling by 4 times.
MSP: timeout += 20; Atmel: timeout += 5; //Timer scaling by 4 times.

In rtl_nw_task ():
MSP: nrk_wait_until_ticks(19); Atmel: nrk_wait_until_ticks(5); //Timer scaling by 4 times.19 is required to synchronization rather than 20.This was done by observing on Osc Scope.

During the rtlink migration, bugs were found in timer code.
Changes made to nrk_timer.c code:
_nrk_os_timer_start() changed.See the code for the changes.Below is the description.
Before starting the os timer,we need to clear the timer so that it will start from 32KHz instead of starting from 4KHz.
TA0CTL |= TACLR; This line will make sure that the timer will start from 32 KHz clock and then keep incrementing from the 4KHz clock.
Changes in _nrk_get_next_wakeup()

```
    return TA0CCR0;  //instead of return TA0CCR0+1;  //fix srinivas
```

Changes in _nrk_set_next_wakeup(uint16_t nw)

```
   TA0CCR0 = nw;//instead of nw - 1;//fix srinivas
```

3.Stack Migration:
evm_Stack_Migration contains the stack migration on one node.
Stack migration should copy the contents of stack from one task to another task.This can be done by copying all the 128 bytes of stack.But this is not required as stack migration needs to copy only the local variables of the task instead of copying the whole 128 bytes.For this purpose, we need to know the amount of data in the stack which needs to be copied.

New function:get_stack_pointer() will get the stack pointer of a task while running and store it in some variable.
This pointer will point to the address in the stack where the actual data starts.We can copy the data in the stack from that pointer to the end of the stack.This will ensure that we are only copying the data required.
This project also has lcd code and hence one can see the actual stack migration between two tasks.In this program the stack of task 1 is copied to stack of task 2.

4.Task Migration:
evm_task_Migration contains the Task migration on one node.
Task migration copies the instructions of one task to another task.So here the task instructions in the flash must be copied and written into another task.We need to access the flash memory for the same.Flash memory is divided into a number of segments.Each segment is around 128 bytes.One can read the memory in flash without any hardware restriction.To write into flash we need to erase the segment where we need to write and

then copy the data into the segment.So flash write operation has two steps:flash segment erase and flash segment write.Segment size is 128 bytes and it is the smallest size which can be erased in the flash.

Our assumption here is that a task instructions cannot exceed segment size(128 bytes).A task can reside in one segment or it can be overlapped between two segments.So at the worst we need to erase two segments and write those two segments with the new data.We can get the task pointer and find out the segments to be erased and then write the new data into it.During this operation it is ideal to stop the timer or disable interrupts as this operation is better done atomically.
New function:flash_memory_write() does the task migration as described above.
This project also has lcd code and hence one can see the actual task migration between two tasks.In this program the task 2 instructions are migrated to task 1.Task 3 is doing the migration from task 2 to task 1.After the migration is done,task 1 gets activated.

Note:evm_CCE_5x_rtl_original_Stack_Migration and evm_CCE_5x_rtl_original_task_Migration_demo project has stack migration and task migration with rt-link.These projects doesn't have lcd.So one needs to run this program and then go to the memory location in the stack and task to see whether the data got swapped.If any help required i would demo this.

5.Bug fix in nano-rk:
In nrk_task.c
In nrk_terminate_task():
New line added:For both MSP and Atmel.This line can be added for Atmel.See below for description.
(Task->taskTCB)->event_suspend = 0;
When a task gets terminated,then the event_suspend flag must be reset to 0,otherwise it would get re activated if the event gets triggered.This was observed while terminating a task in task migration with rt-link.This is required when a task is terminated by another task currently running.

6.Rt-Link Stats:
The m file has the code to plot the stats.It also generates an array of 1s and 0s.1-packet loss.It reads a text file with sequence numbers and counts the packet losses and plots the stats.The folder rtl_stats contain the m-file and files used to get the stats.