Higher-level Aspects of Task Migration

We will now focus on the task migration protocol from (a) Primary to Backup node and (b) Backup to Primary shadowing. Migration includes (a) capability enumeration, (b) auto-negotiation and the protocol for reliable control/data transfer. Migration will utilize the underlying mechanisms developed by Srinivas.

A. Types of Task Migration
Each task is associated with:
(i) Task meta data (nrk_TCB, evm_TCB, precedence constraints, set of associated sensors and actuators, control timing constraints)
(ii) Task stack
(iii) Task instructions and data

We consider migration from parametric to programmatic levels:
Level-1: Activation/Deactivation

Level-2:

Level-3:

Level-4:

B. Capability Enumeration

C. Auto-Negotiation

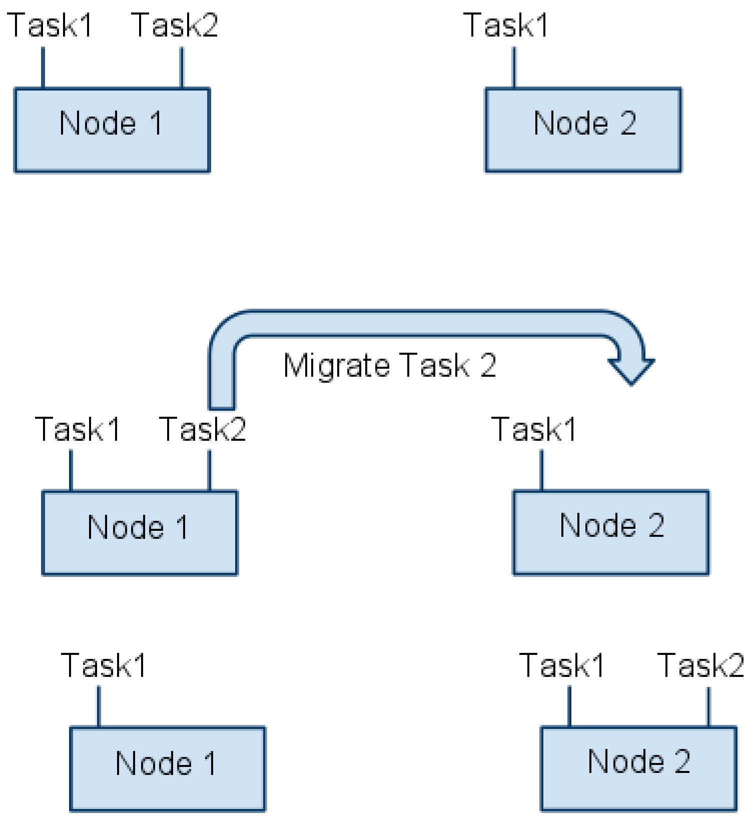
D. Reliable Task Migration Protocol

E. Migration Policy for Primary to Backup

F. Task Shadowing Policy for Backup nodes

Task Migration Documentation

Before going through this document one must have a good understanding about the nano rk RTOS and the forth interpreter and MSP430 platform.
For nano rk RTOS please refer the nanork webpage for more details.Use the MSP FET debugger to go through the nanork code.
For forth interpreter,please refer to the code from Aminreza and the documentation he provided.Please also go through the
forth programming books by elizabeth rather,etc..Please also go through the user guide of the MSP430 for details regarding the nano-rk hardware specific code,memory layout
and linker file description.
Task Migration is an important piece of the EVM project.Task migration,as the name suggests is to migrate a running task or program from one node to another node.
IMPORTANT NOTE !!!!!:The current task migration code is developed to run only with built in words and does n't work with user defined words.So please do not define your own word for migration.
Task 1 and Task 2 has forth programs which run on top of the nano rk RTOS.These forth programs must be migrated from node 1 to node 2.



In this program,the task migration takes place from node 1 to node 2.The task migration requires sending the following data:
1.Task parameters,2.Interpreter variables and pointers,3.Interpreter Stack and forth program.So three types of data needs to transmitted over rt-link in sequence.
Once this data is received,it will be stored in the memory and a new nano-rk task is created and activated in the super task.
The interpreter pointers are initialized using the offset from the received interpreter pointers.
Once the data is received then Node 2 would send an "Ack" (packet type 5) saying that it received all the data and is set to execute the code.
If there is a packet loss,then Node 2 would send "No Ack" (packet type 6) saying that those packets need to be resent and it won't execute the code.

<u>In the main.c file:</u>
We have rt-link tx and rx task which sends and receives the migration data.
A static task executes the forth program in the interpreter.
A template function is used for copying the instructions for the new nano-rk task.
A supertask is the task which actually creates new task and sets up the interpreter variables and pointers.The pointer to these variables and pointers is stored in the task TCB.
So rt link requires one RTL_COORDINATOR and RTL_MOBILE node.These two nodes are given ADDR=1 and ADDR=2.The task gets migrated from 1 to 2.Once the user presses the button,the Task2 which has the interpreter program running halts and the data is transmitted over rt link.
1.Task parameters
2.Interpreter variables and pointers
3.Interpreter Stack and forth program

4.Done packet
5.Acknowledgment-Sent when all the data is received.
6.No Acknowledgment-Sent when there is a packet loss while migration and the data needs to be resent.
Each type of data is send in one rt link packet.Interpreter Stack starts from IStack2[512] index and has 512 bytes to be sent.So it is broken into 64 bytes each and sent over rt link.

<u>In nrk_interpreter.c:</u>
<u>Global Variables:</u>
state variable is used for sending the data using rt-link.(0-no data,1-task parameters,2-interpreter variables and pointers,3-interpreter stack and forth program,4-done,5-Acknowledgement,6-No Acknowledgement)
done variable to signal the supertask to activate the task(0-no action,1-finished receiving the migration data,2-finished activating the task)
mem_alloc[2000] array for storing the task migration data.

Task_Data Structure looks something like this:
**********IMPORTANT**************
typedef struct tasknode{
int *rsp; //stack pointer for function calls in interpreter
int *dsp; //data stack pointer
char *here; // the pointer to the next word to be added in the dictionary(this will be task specific dictionary for each task)
char *latest; // pointer to the latest word added in the dictionary(this will be task specific dictionary for each task)
char *forth_buffer; //pointer the forth program
int vtid; //id for virtual task
int forth_engine_poll; //used as a flag to start and stop interpreting
int buffer_ind;// = 0; // the offset into the forth buffer for the next word.
int state; //= 0; //state of interpreter-executing or compiling
int *dsp_base; //these are base pointers which get initialised to dsp in the beginning and remain unchanged.this is used for task migration
int *rsp_base; //these are base pointers which get initialised to rsp in the beginning and remain unchanged.this is used for task migration
char *here_base; //these are base pointers which get initialised to here in the beginning and remain unchanged.this is used for task migration
char *forth_buffer_base; //these are base pointers which get initialised to forth_buffer in the beginning and remain unchanged.this is used for task migration
//int *latest_base; //these are base pointers which get initialised to latest in the beginning and remain unchanged.this is used for task migration
char c;// = -1;//last character read from the forth buffer
void (*xt)();//function pointer used for interpreting.
struct tasknode *Next;//next Task_Data pointer ...used for forming a linked list of Task_Data structures...might be useful later.
} Task_Data;

Task_Data structure holds the data for interpreter variables and pointers.Each Task can create this structure and have the interpreter state in this.
#pragma DATA_SECTION(global_dictionary, ".my_sect")
static char global_dictionary [1024]; //changed

Task_Data GlobalDict;
Task_Data *Global_Dict;
GlobalDict is the global dictionary structure which has pointers to the dictionary.Global_Dict is the pointer pointing to this global dictionary structure.
Each task has its own Task_Data structure for interpreting.The pointer to this structure is stored in nano_RK TCB for reference.

- All the functions in the interpreter take the pointer to interpreter variables structure (Task_Data *IData) as the argument.
- The word is found in the program and then searched in the dictionary.This requires placing a null character after the word is found.
- So this null character must be replaced back with a white space character.So this change has been made to the original interpreter code.
- The set_pointers() function initializes the interpreter pointers using the offset from the received interpreter pointers.
- The data_set() functions sets the interpreter data for static tasks.
- There is a global dictionary common to all the tasks and also a task specific dictionary.
- Global dictionary is stored in the flash.
- init_forth_engine(),set_up_data_segment(),set_up_core() are the functions in which the global dictionary gets stored in the flash.
- A new DATA_SECTION is created with name my_sect which takes some memory in the flash for storing the global dictionary.This is done in linker file.

Memory Manager with Nano-RK:
The memory manager is used to activate and terminate tasks dynamically.For this we need to allocate memory for nrk_TCB and nrk_queue node.TCB for storing the task parameters and pointers and nrk_queue node to add into readyQ tasks.So nano-rk is modified for dynamic task activation and termination.The nrk_malloc statements are used to craete a new TCB and new nrk_queue node.Please refer code in nrk.c and nrk_task.c files.Please also refer to the nrk_memory_manager.c file for the code.

This section of document is only for documenting measurements and stats.

Interpreter Overhead Measurements:
Measuring Interpreter overhead requires to have a program written in Forth interpreter and in C language.These two programs are executed and the overhead is measured by toggling pins while executing these programs in a loop.So in that case loop overhead is also included in both the cases.The BEGIN and UNTIL words are changed to make it executable in a user defined word.A user defined word gets compiled into the Interpreter Dictionary and it gets executed right away without actually searching through the dictionary.So this helps in reducing the execution time.If a program is run using built in words then it would have to go through the dictionary and search for the word and execute.Since searching through the dictionary and executing will consume more cycles,its usually better to execute user defined words rather than executing it from the dictionary.
Polynomial execution:
With interpreter: 320us (user defined)
Normal:20us.

LED Toggling:
With interpreter: 140us (user defined)
Normal: 10us.

RT-Link between MSP430 and Firefly Drift measurements:
There are some snapshots of the delay measured btw these two platforms.Each measurement is done for every 8 seconds.So the drift is measured in terms of (x ms/8s).x is the drift.
In the snapshot,the delay is given by 2->1 value at the bottom of the image.The time is given in the clock at the right corner bottom.So there are 8 snapshots and 8 measurements of drift.The measurements of drift were as below:
0.5 ms/8s
0.4522 ms/8s
0.4639 ms/8s
0.42 ms/8s
0.4638 ms/8s
0.46 ms/8s
On average: 0.46ms/8s drift.

This section of document is only for documenting the bug fixes and changes in the nano rk code.Please refer the code and this doc to understand the changes.

Bug Fix in Nano-RK:
The context switching code is modified.The interrupt code has a function call (nrk_timer_suspend_task()) which pushes the registers into the stack and changes the SP register.
But this call takes two parameters and these parameters modify the r12 and r13 registers.So to fix this,the registers must be pushed into the stack and then the function call can be made.push_reg() pushes registers from r4-r13.Then nrk_timer_suspend_task() pushes r14,r15 registers and then changes the SP register.Any other alternative way can be done for this.
Please refer the interrupt code in nrk_timer.c file.

New Additions to Nano-RK:
The Task TCB is added with a new data: SchType.SchType can be either NONPREEMPTIVE or PREEMPTIVE.
rtl_task 's SchType is NONPREEMPTIVE.(check this in rtl_task_config routine).
The scheduler checks the SchType and assigns the next_wake accordingly.
New line added in nrk_scheduler routine: if(nrk_high_ready_TCB->SchType==NONPREEMPTIVE) next_wake=nrk_high_ready_TCB->next_wakeup;//fix srinivas

Changes in nrk_task.c:
In nrk_activate_task routine:
//newNode=&mallocQueue[tasksNumber-2];   //malloc simulation
is modified to   newNode=(nrk_queue *)nrk_malloc(sizeof(nrk_queue));

In nrk_terminate_task routine:
The task TCB pointers are changed and then the TCB is freed.
The _free_node pointer is changed and the it is freed.

In nrk_TCB_init routine:
//newTCB = &mallocTCBs[tasksNumber-1];//malloc(sizeof(NRK_TCB));
is modified to   newTCB = (NRK_TCB *)nrk_malloc(sizeof(NRK_TCB));

Bug Fix for task_id conflict:
The task id of each new task must be unique and different from other tasks.So a new function check_taskID_conflict() is added in nrk.c file.

check_taskID_conflict():This function would go through the list of tasks and compares the taskID to see whether there is any conflict.If there then the taskID is incremented and then again checked for the conflict.This ensures that each task gets a unique id.This function is called in nrk_TCB_init() function.
There were some type mismatches for taskID in nrk_remove_from_readyQ and nrk_add_readyQ functions.These type mismatches were fixed.

Suggestions required: from Paja

The nrk_memory_manager code allocates and frees the memory ..but does it take care of the defragmentation of the fragmented memory segments during freeing up the memory??As I see it does defragment upward in the memory and only once...but it does n't defragment downward in the memory ....need to understand whether this feature needs to implemented??