TECH DAYS
Texas Instruments

Dynamic Multi-protocol Manager with demo

Thomas Almholt

Low Power Wireless Group

Dynamic Multi-protocol Manager overview

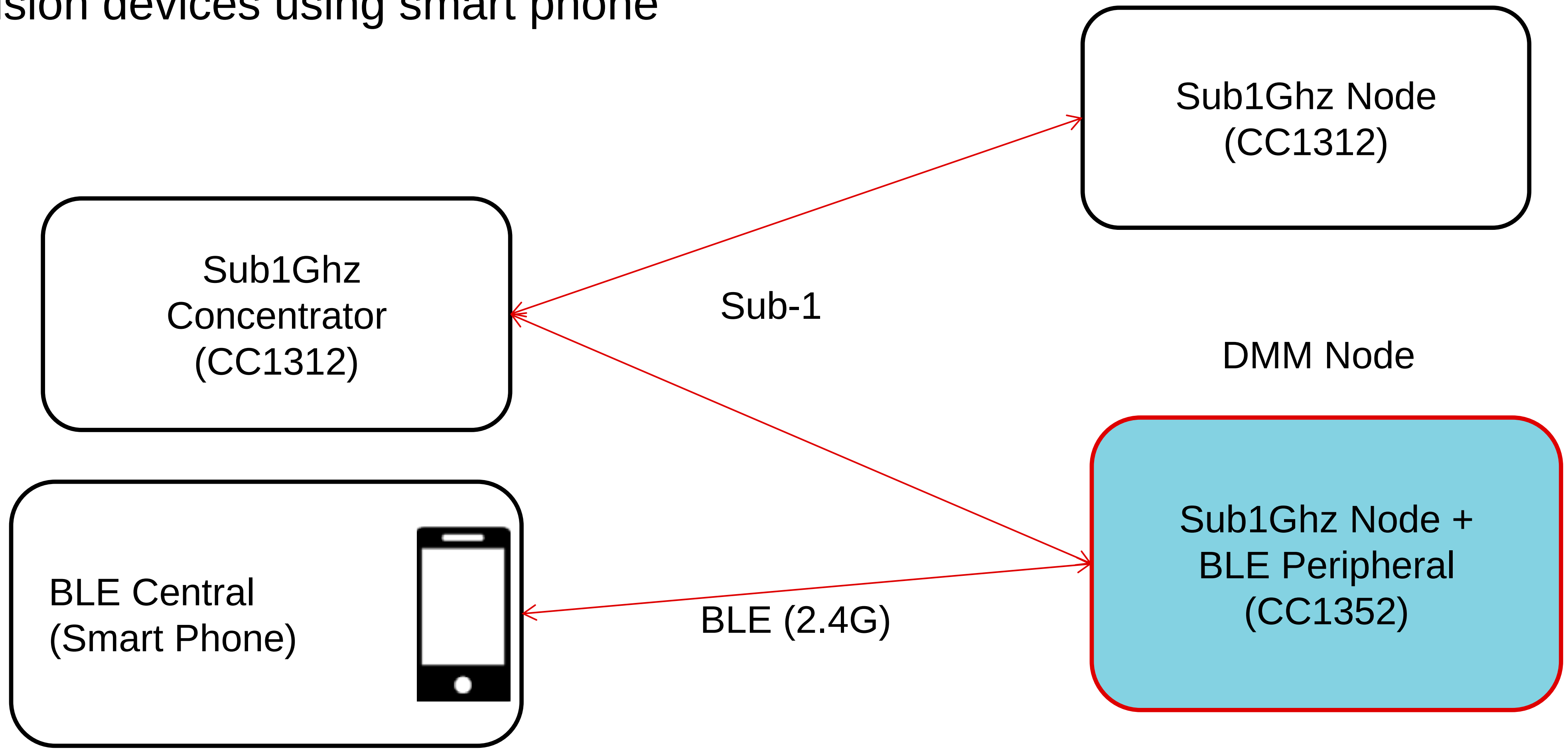
TECH DAYS
Texas Instruments

- Motivation for using a multi protocol manager
- Main Challenges of using a multi protocol manager
- High level design
- Basic scheduling concept overview
  - Operating system versus Radio scheduling
- Demo time
  - Sub1Ghz network with BLE based phone connectivity

Motivation of Dynamic Multi-protocol Manager

Basic example use cases:

- Access sensors information on smart phone
- Control devices using smart phone
- Provision devices using smart phone



Main Challenges

Execute multiple protocol stacks concurrently on a single radio device:

– Protocol stacks are designed assuming complete guaranteed access to radio
– Any Loss of packet transmission / reception is assumed to be due to packet loss

Fully functional BLE with Sub1GHz

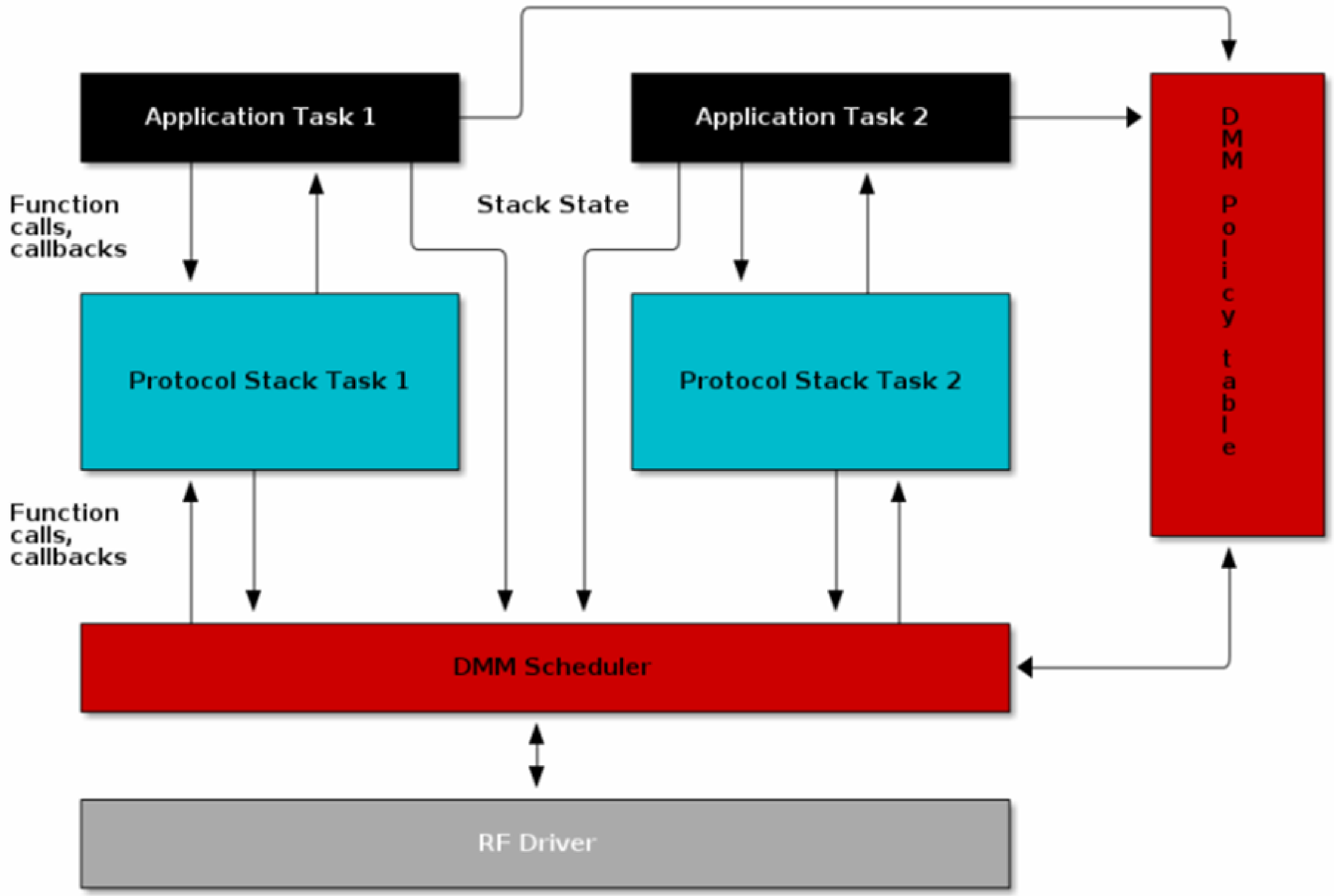
– BLE spec defines many operations that are time critical
– Device specific constraints on BLE parameters
– Sub1Ghz low data rates consume a lot of radio time

Scheduling of radio events across stacks

– Rescheduling or aborting of radio events can have impact on Sub1GHz / BLE performance
– BLE performance impact can be visible to the user directly
– Application level state changes can also impact performance
– scheduling problem: *optimally schedule radio events such that radio utilization is 100% given a set of timing constraints per stack across all available stacks and user policy*

High level design concepts of Dynamic Multi-protocol Manager

- There is typically one or more application stacks for each communication stack, here we are showing one for each.
- The application knows the state it is in and therefore can provide this information to the multi protocol manager.
- The multi protocol manager evaluates parameters such at stack states, priority tables and runtime priority selection
- The multi protocol manager provides all inputs to the RF driver.



Fundamental Functions of Dynamic Multi-protocol Manager

Multi protocol manager is a cross-layered software module

– That is aware of the stack states, uses inputs from the application user on stack-level prioritization, aware of RF command queue and accordingly schedules RF driver operations

Adapt to use-case (Policy)

– Policy lets user customize to his/her use case

– User can provide inputs on stack states and priorities to influence DMM behavior

Priority Arbitration and Scheduling

– Assign priorities for low level stack RF operations and determine who is high/low.

– Determine execution time points of RF operations. This can result in delays or stop/reschedule of certain operations

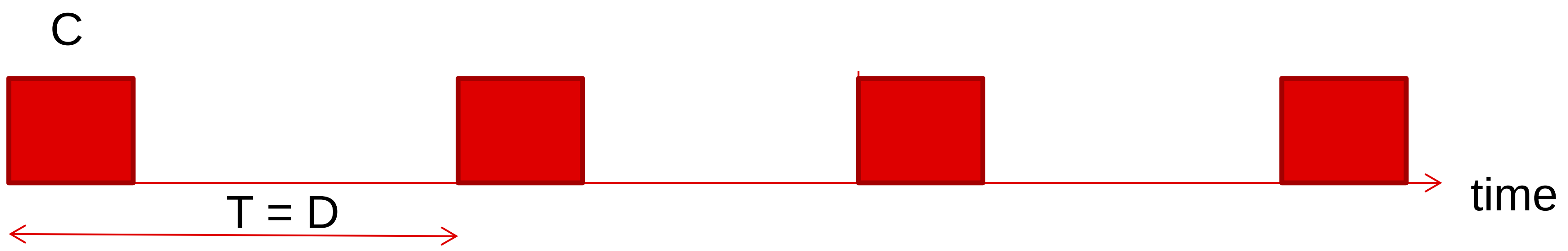
Application coordination

– Sub1Ghz/Zigbee application info sharing/control with BLE

Basic scheduling concepts overview

A task can be modeled as (C, T, D)

- C → execution time, T → min. inter-arrival time, D → Deadline



- Assume implicit deadline tasks → T = D
- Utilization U = C/T. Ex: C= 2 units and T = 10 units, then U = 20%
- Lower frequency implies longer C and hence consumes more U given T

Priorities for tasks

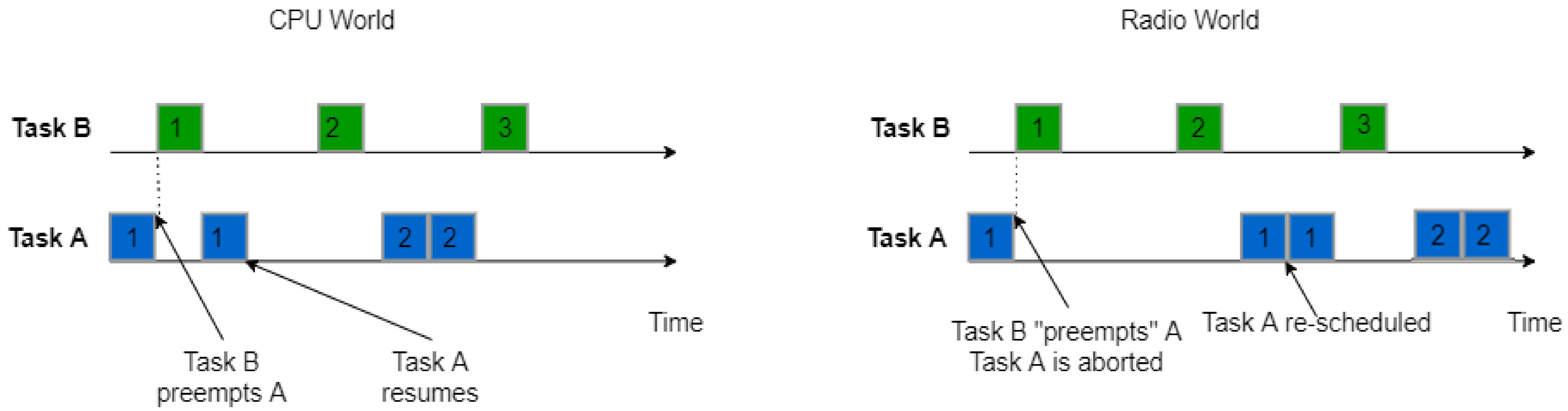
- based on scheduling algorithms
- based on combination of policies and scheduling algorithm
- Higher priority task pre-empt lower priority tasks

These OS scheduling concepts are borrowed for multiple protocol manager

- But there is a caveat…

Real-time operation system (RTOS) vs Radio Scheduling

Task B has higher priority than A
Say Task B is 1 time unit
Say Task A is 2 time units
Each block depicts 1 time unit
The numbers on the block depicts the instance



DMM needs to ensure multiple stack commands can be globally optimally scheduled given timing constraints of each of the stacks and user policies

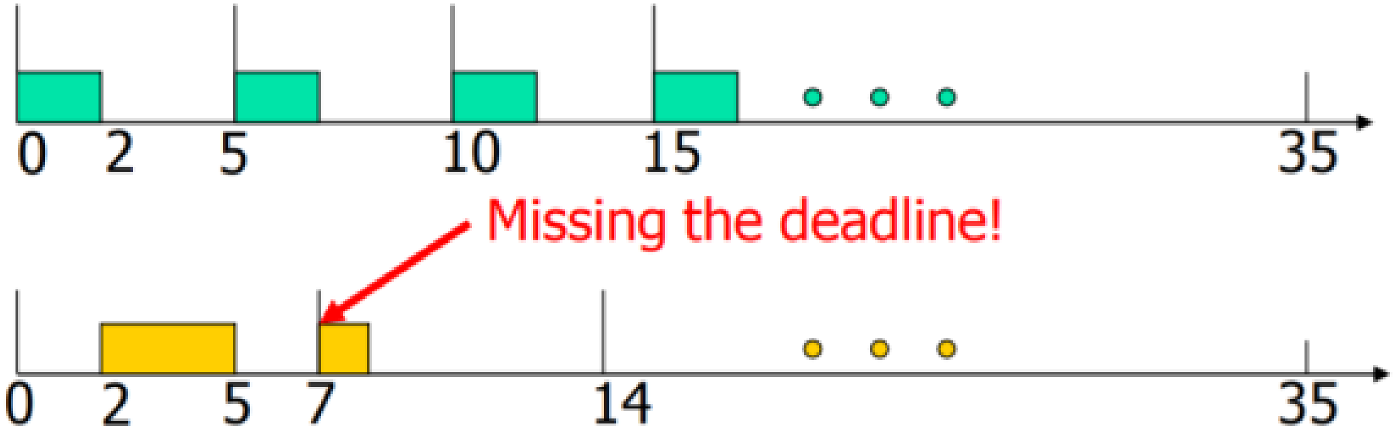
Scheduling

- **The multi protocol manager is fundamentally a lossy system:**
  - For obvious physical limitations if the sum of radio utilization of individual protocol stacks is greater than (>) 100% then this will result in loss.
  - The multi protocol manager main function is to reduce this loss in cases when the utilization is << 100%.
- **The multi protocol manager uses dynamic scheduling**
  - Fixed priority schedulers are known to provide infeasible schedule when utilization is < 100% [1].
  - Dynamic priority schedulers are known to be more efficient [1].
  - Uses a combination of priority and “deadlines” to schedule radio commands.

[1] JCL Liu, JW Layland, *“Scheduling algorithms for multiprogramming in a hard real-time environment”,* Journal of ACM 1973.

Scheduling

- **The multi protocol manager uses dynamic scheduling**
  - Uses a combination of priority and “deadlines” to schedule radio commands.

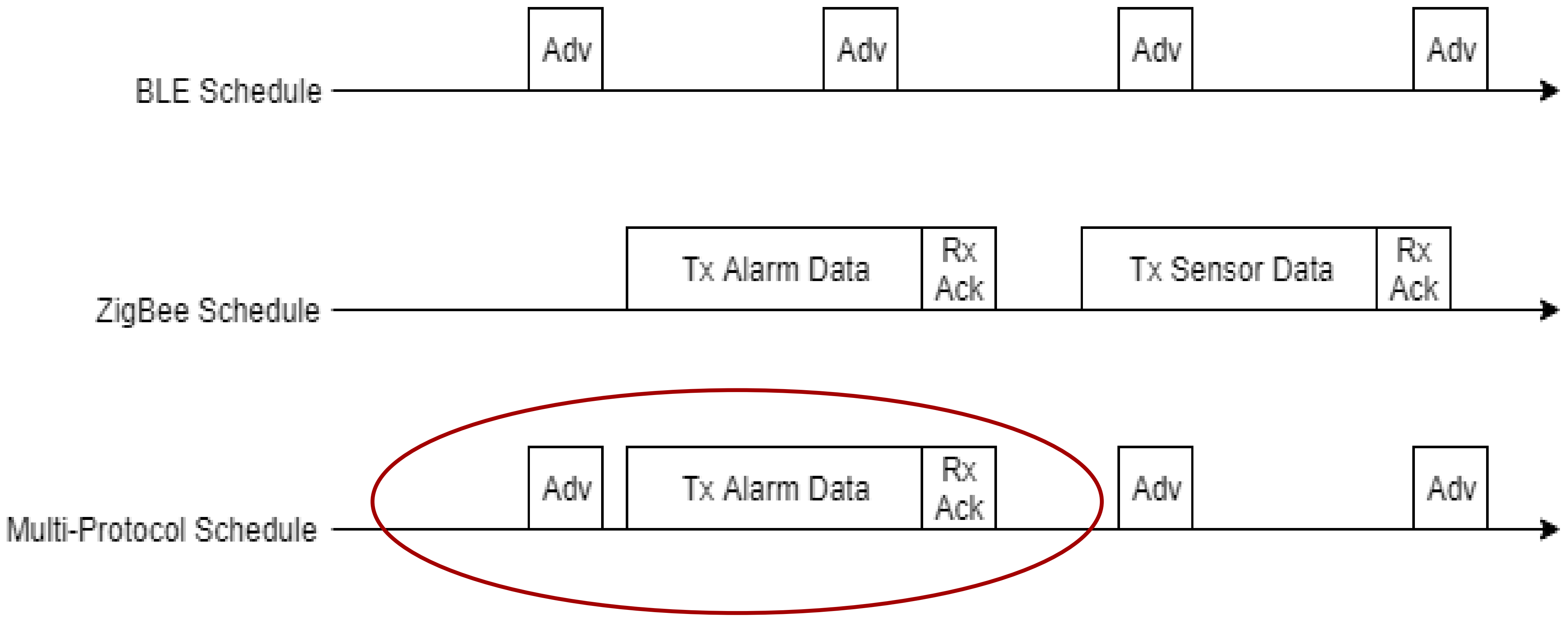


A1(2, 5), B1(4,7)
U ~ 0.97

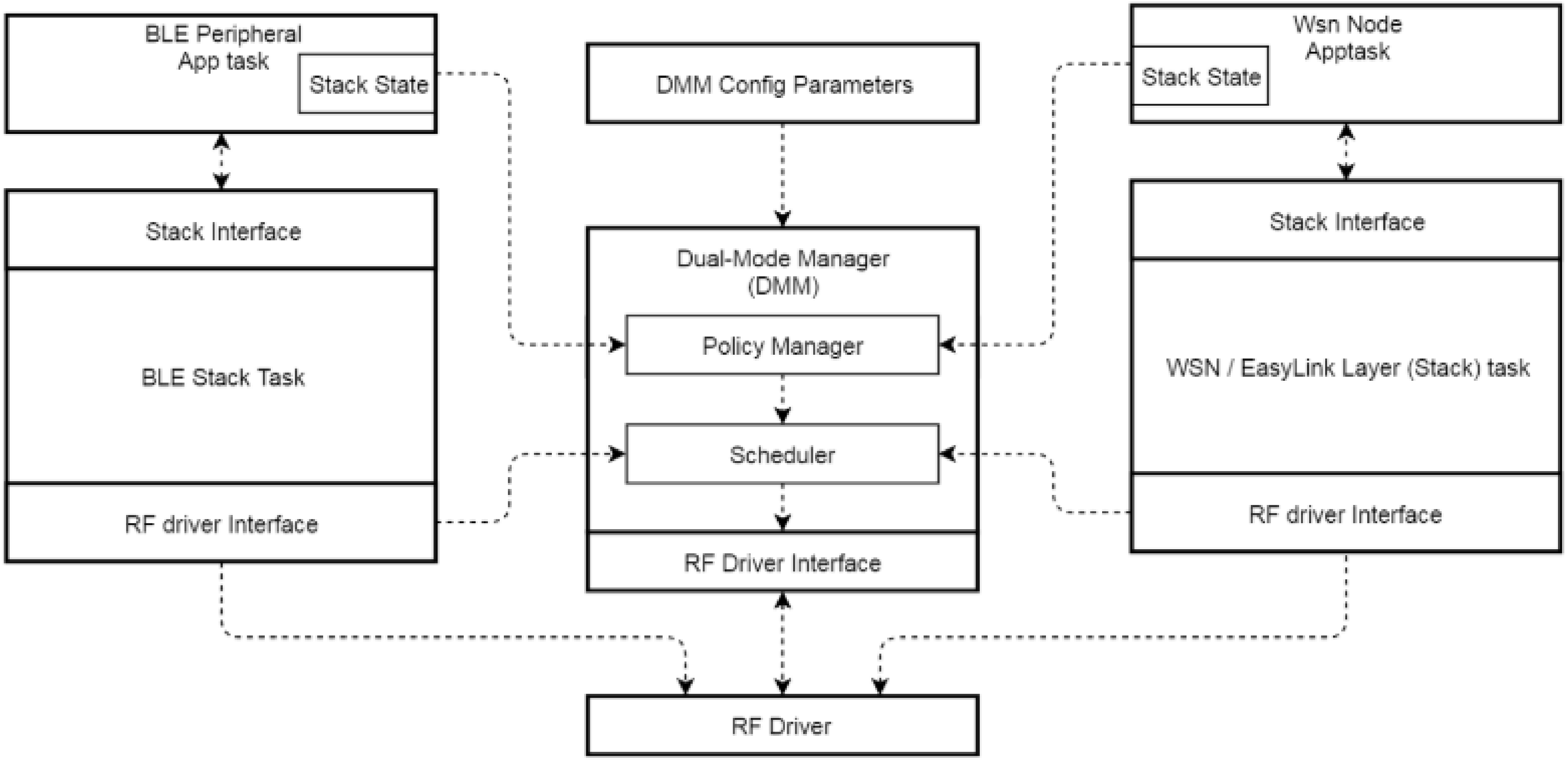
– The multi protocol manager is more efficient in terms of utilization of radio than fixed TDMA based scheduling:
– No matter granularity of time slot, RF operation time duration will *not* always be perfect integer multiples of time slots

Dynamic priority policy manager

Simple example showing use of policy changing scheduling priority at run-time



Specific example instance of Dynamic Multi-protocol Manager



Multi protocol manager examples in latest SDK

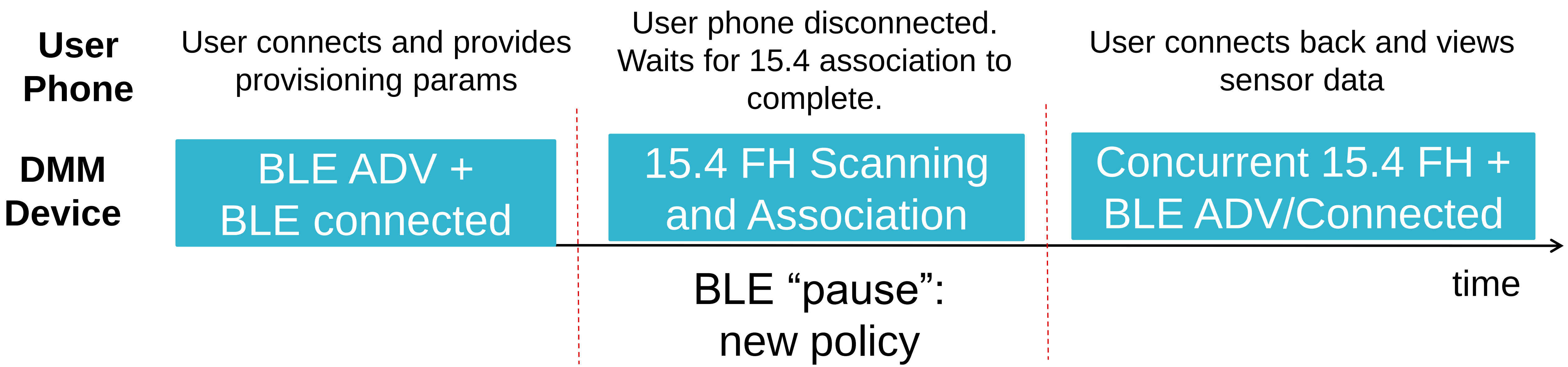
- **Easylink WSN node remote display:**
  - Sub1GHz WSN node + BLE peripheral concurrent operation
  - Smart phone app (ex: LightBlue) can update WSN node and WSN concentrator parameter
  - Smart phone app can be used to read WSN node sensor data
  - Supports Sub1GHz 150 Kbps and LRM

- **TI15.4 Stack Sensor remote display:**
  - Only supports TI15.4 Sub1GHz Frequency Hopping Mode: 15.4 FH + BLE peripheral
  - Supports provisioning of 15.4 FH Sensor using SmartPhone App
  - Supports reading/control of 15.4 FH sensor data and report interval using smart phone app

TI 15.4 Stack with Frequency hopping + BLE Example

- Can provision 15.4 Sensor FH using BLE smart phone app:
  - Network PAN ID
  - Channel Mask
  - Security Key

- Sequence of Operations:



Usage Guidelines

- **Typically targeted for “light-weight” BLE application usage**
  - → Keep BLE radio utilization lower compared to lower data rate Sub1GHz

- **Lossy System**
  - Whose lose is more tolerable? Sub1GHz or BLE?
  - Intermittent BLE connection events misses is “OK”.

- **General Recommendation:**
  - Keep BLE connection interval < 200 ms
  - Have Sub1GHz at higher priority
    - Note DMM automagically makes BLE higher priority at connection setup time alone
    - Also in future will automagically make BLE higher priority when at risk of losing BLE connection

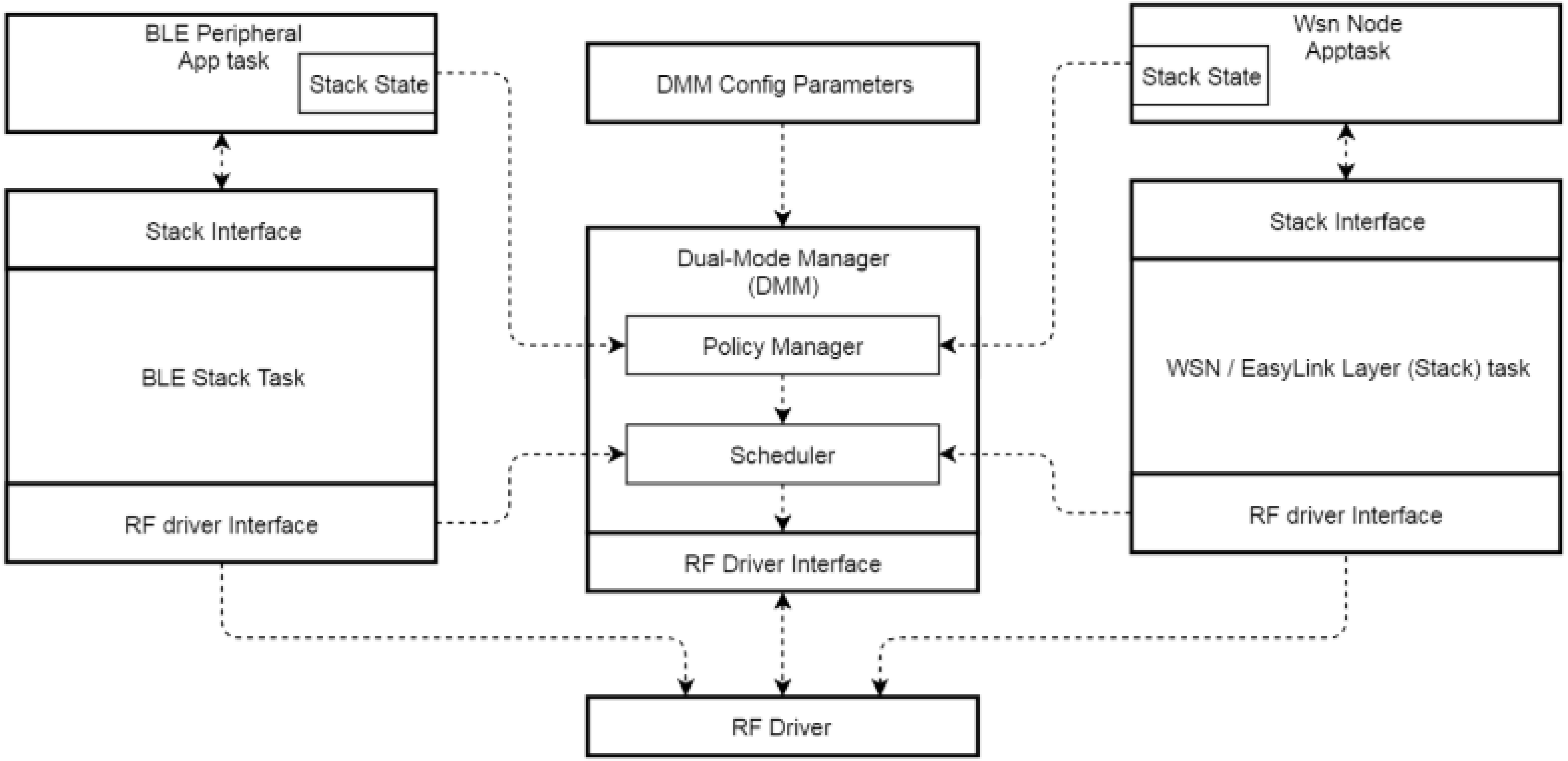
The Dynamic Multi-protocol Manager Policy Table

- Each stack describes the possible states it can be in
- The policy table maps all possible combinations of stack states
- Each state combination, called policy, configures:
  - Priority
  - Time constraints
  - Stack pause

```
DMMPolicy_PolicyTableEntry DMMPolicy_wsnNodeBleSpPolicyTable[] = {
    // State 1: BleSp connectable advertisements and Wsn Node TX or Ack:
    // BleSp = Low Priority | Time None Critical,
    // WsnNode = High Priority | Time Critical
    {
      {DMMPolicy_StackType_BlePeripheral, DMMPolicy_StackType_WsnNode},
      {(DMMPOLICY_STACKSTATE_BLEPERIPH_ADV) ,
       (DMMPOLICY_STACKSTATE_WSNNODE_SLEEPING | DMMPOLICY_STACKSTATE_WSNNODE_TX | DMMPOLICY_STACKSTATE_WSNNODE_ACK)},
      {DMMPOLICY_PRIORITY_LOW, DMMPOLICY_TIME_NONE_CRITICAL, DMMPOLICY_NOT_PAUSED, //BLE SP Stack
       DMMPOLICY_PRIORITY_HIGH, DMMPOLICY_TIME_CRITICAL, DMMPOLICY_NOT_PAUSED} //WSN NODE Stack
    },
```

- Can specify a default «catch-all» policy

The Dynamic Multi-protocol Manager archtecture



The Dynamic Multi-protocol Manager archtecture

- **Stacks submits RF commands unbeknownst of the DMM**
  - The DMM Scheduler intercepts all RF commands
  - Decides what to schedule based on:
    - What are the current stack priorities?
    - What are the timing constraints of the stacks?
    - Are there any commands in the RF queue, and what priority do they have?

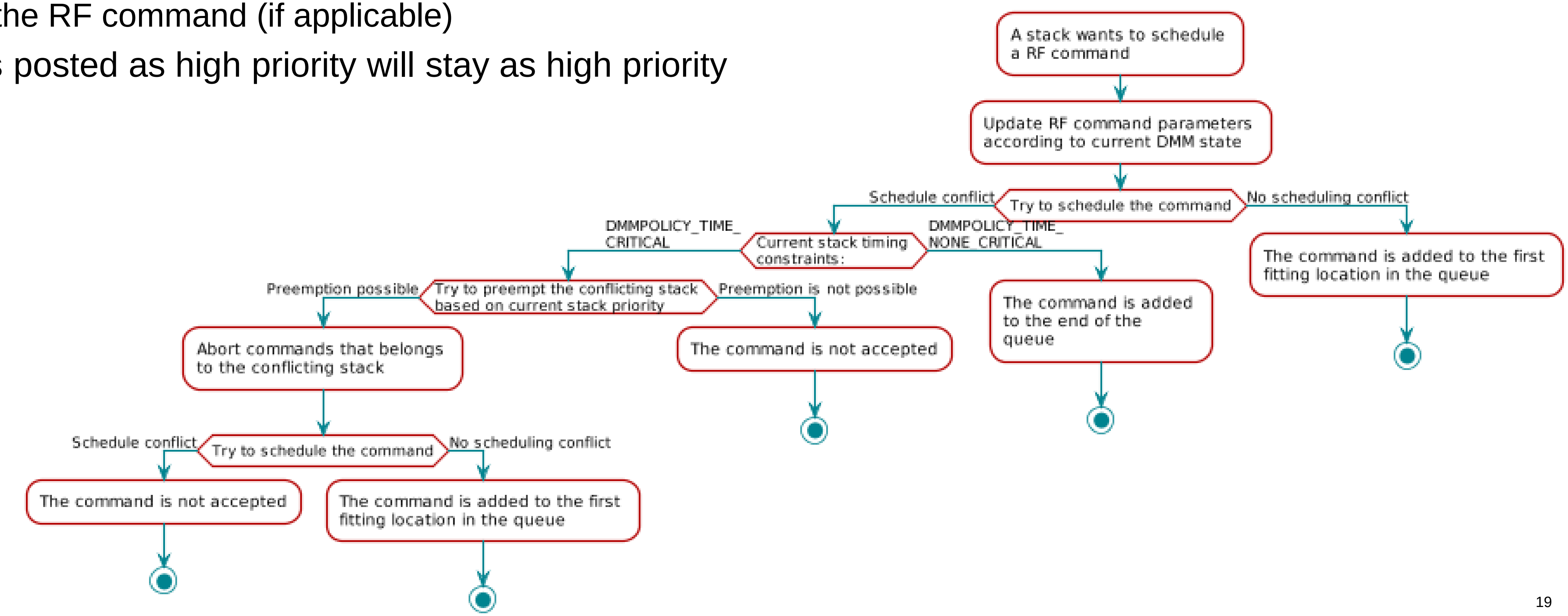
- **How does the DMM intercept RF commands?**
  - RF driver API remapped/redefined to DMM scheduler API
  - Ex. any calls to RF_scheduleCmd() will be replaced by DMMSch_rfScheduleCmd()

```
#ifndef USE_DMM
#include <ti/drivers/rf/RF.h>
#else
#include <dmm/dmm_rfmap.h>
#endif //USE_DMM
```

```
// dmm_rfmap.h
#define RF_open               DMMSch_rfOpen
#define RF_postCmd            DMMSch_rfPostCmd
#define RF_runCmd             DMMSch_rfRunCmd
#define RF_scheduleCmd        DMMSch_rfScheduleCmd
#define RF_runScheduleCmd     DMMSch_rfRunScheduleCmd
#define RF_cancelCmd          DMMSch_rfCancelCmd
#define RF_flushCmd           DMMSch_rfFlushCmd
#define RF_runImmediateCmd    DMMSch_rfRunImmediateCmd
#define RF_runDirectCmd       DMMSch_rfRunDirectCmd
#define RF_requestAccess      DMMSch_rfRequestAccess
```

The Dynamic Multi-protocol Manager archtecture

- **RF command scheduling**
  - The actual scheduling behavior is defined by the current scheduling policy
  - Schedules on the following parameters:
    - Stack priority (**DMMPOLICY_PRIORITY_HIGH** or **DMMPOLICY_PRIORITY_LOW**)
    - Timing constraint (**DMMPOLICY_TIME_CRITICAL** or **DMMPOLICY_TIME_NONE_CRITICAL**)
    - Start time of the RF command (when using absolute triggers)
    - End time of the RF command (if applicable)
  - RF commands posted as high priority will stay as high priority



Custom Stack Integration

- **Create or extend the DMM policy table**
  - Identify a set of stack states
  - Create policies from different combinations of stack states

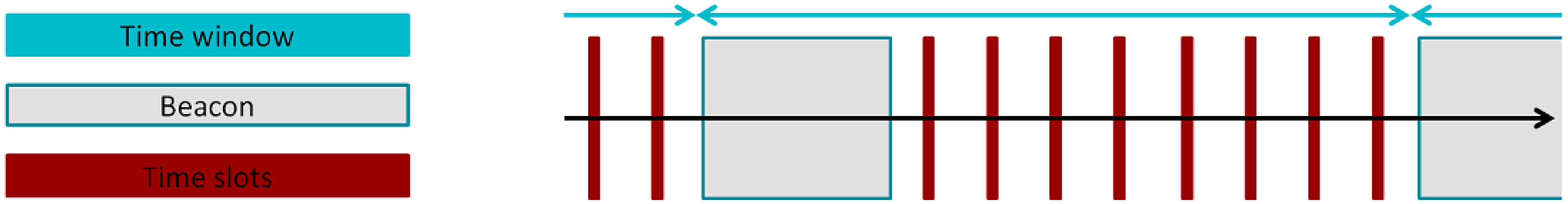
- **Make the stack DMM-aware**
  - Include the DMM RF API remapping instead of the RF driver

- **Add stack state transitions in the application task**

- **Initialize the DMM and register clients during startup**

Case Study: Prop. Collector + BLE Peripheral

- **Consider a Collector / Sensor pair with a simplified beacon mode protocol**
  - Sensor connects to the Collector by an association process
  - Sensor communicates with Collector under fixed time slots
  - Synchronizes with beacon messages



- **BLE Simple Peripheral**
  - Default example project from SDK
  - Long Range advertisements disabled

- **DMM Device will be Proprietary Collector + BLE Peripheral**

Case Study: Enable Multi-protocol Manager for BLE-Stack

- **BLE Peripheral stack states**
  - **Advertising:** when adverstising connectible
  - **Connecting:** when in the process of connecting to a central device
  - **Connected:** when connected to a central device
  - **Any:** any other state

- **Make the BLE-stack DMM-aware**
  - In «*ble_user_config.c*» configure
    - **fastStateUpdateCb** to an application callback
    - **bleStackType** to **DMMPolicy_StackType_BlePeripheral**
  - Add the **USE_DMM** define

- **Update Stack states in the application**
  - Update stack states in **fastStateUpdateCb** based on internal stack changes
  - Update stack states in application based on stack messages

Case Study: Enable Multi-protocol Manager for BLE-Stack

```c
// The supported stack states bit map for BLE Simple Peripheral
#define DMMPOLICY_STACKSTATE_BLEPERIPH_ADV         0x00000001 // State for BLE Simple Peripheral when advertising connectable
#define DMMPOLICY_STACKSTATE_BLEPERIPH_CONNECTING  0x00000002 // State for BLE Simple Peripheral when in the process of connecting to a master device
#define DMMPOLICY_STACKSTATE_BLEPERIPH_CONNECTED   0x00000004 // State for BLE Simple Peripheral when connected to a master device
#define DMMPOLICY_STACKSTATE_BLEPERIPH_ANY         0xFFFFFFFF // Allow any policy
```

```c
// BLE Stack Configuration Structure
const stackSpecific_t bleStackConfig =
{
  .maxNumConns                          = MAX_NUM_BLE_CONNS,
  .maxNumPDUs                           = MAX_NUM_PDU,
  .maxPduSize                           = MAX_PDU_SIZE,
  .maxNumPSM                            = L2CAP_NUM_PSM,
  .maxNumCoChannels                     = L2CAP_NUM_CO_CHANNELS,
  .maxWhiteListElems                    = MAX_NUM_WL_ENTRIES,
  .maxResolvListElems                   = MAX_NUM_RL_ENTRIES,
  .pfnBMAlloc                           = &pfnBMAlloc,
  .pfnBMFree                            = &pfnBMFree,
  .rfDriverParams.powerUpDurationMargin = RF_POWER_UP_DURATION_MARGIN,
  .rfDriverParams.inactivityTimeout     = RF_INACTIVITY_TIMEOUT,
  .rfDriverParams.powerUpDuration       = RF_POWER_UP_DURATION,
  .rfDriverParams.pErrCb                = &(RF_ERR_CB),
  .eccParams                            = &eccParams_NISTP256,
  .fastStateUpdateCb                    = SimplePeripheral_bleFastStateUpdateCb,
  .bleStackType                         = DMMPolicy_StackType_BlePeripheral
};
```

Case Study: Enable Multi-protocol Manager for BLE-Stack

```
void SimplePeripheral_bleFastStateUpdateCb(uint32_t stackType, uint32_t stackState)
{
  if(stackType == DMMPolicy_StackType_BlePeripheral)
  {
    static uint32_t prevStackState = 0;

    if( !(prevStackState & LL_TASK_ID_SLAVE) &&
         (stackState & LL_TASK_ID_SLAVE))
    {
        /* update DMM policy */
        DMMPolicy_updateStackState(DMMPolicy_StackType_BlePeripheral,
                                   DMMPOLICY_STACKSTATE_BLEPERIPH_CONNECTING);
    }
    else if( (prevStackState & LL_TASK_ID_SLAVE) &&
            !(stackState & LL_TASK_ID_SLAVE))
    {
        /* update DMM policy */
        DMMPolicy_updateStackState(DMMPolicy_StackType_BlePeripheral,
                                   DMMPOLICY_STACKSTATE_BLEPERIPH_ADV);
    }

    prevStackState = stackState;
  }
}
```

```
static void SimplePeripheral_processGapMessage(gapEventHdr_t *pMsg)
{
    switch(pMsg->opcode)
    {

    /* ... code omitted ... */

    case GAP_LINK_PARAM_UPDATE_EVENT:
    {
      /* ... code omitted ... */

      /* update the DMM policy */
      DMMPolicy_updateStackState(DMMPolicy_StackType_BlePeripheral,
                                 DMMPOLICY_STACKSTATE_BLEPERIPH_CONNECTED);
      break;
    }

    /* ... code omitted ... */
    }
}
```

Case Study: Enable Multi-protocol Manager for Prop. Collector

- **Proprietary Collector stack states**
  - **Idle:** when Collector is sleeping
  - **Listen for Node:** when Collector is waiting for a Sensor
  - **Join Request:** when Collector is processing a join request from a new Sensor
  - **Send Beacon:** when Collector is sending a Beacon
  - **Any:** any other state

```
// The supported stack states bit map for RF Collector
#define DMMPOLICY_STACKSTATE_RFCOLL_IDLE                0x00000001 // State for rfColl when sleeping
#define DMMPOLICY_STACKSTATE_RFCOLL_LISTENFORNODE       0x00000002 // State for rfColl when waiting node
#define DMMPOLICY_STACKSTATE_RFCOLL_PROCESSJOINREQUEST  0x00000004 // State for rfColl when processing a new node
#define DMMPOLICY_STACKSTATE_RFCOLL_SENDBEACON          0x00000008 // State for rfColl when sending a beacon
#define DMMPOLICY_STACKSTATE_RFCOLL_ANY                 0xFFFFFFFF // Allow any policy
```

- **Make the Collector «stack» DMM-aware**
  - Make sure the DMM RF API remapping is included instead of the RF driver

- **Update Stack states**
  - Most likely a simple implementation of the Collector «stack»
  - Update radio-specific stack states in the «stack»
  - Update application-specific stack states in the application

Case Study: Create a Policy Table

```
// The policy table for the BLE Simple Peripheral and rfColl use case
DMMPolicy_PolicyTableEntry DMMPolicy_PolicyTable[] = {
  // State 1: BT = connecting, RF = any
  // BleSp = high priority | time critical, rfColl = low priority | time none critical
  {
    {DMMPolicy_StackType_BlePeripheral, DMMPolicy_StackType_WsnNode},
    {DMMPOLICY_STACKSTATE_BLEPERIPH_CONNECTING, DMMPOLICY_STACKSTATE_RFCOLL_ANY},
    {DMMPOLICY_PRIORITY_HIGH, DMMPOLICY_TIME_CRITICAL, DMMPOLICY_NOT_PAUSED, // BLE SP Stack
     DMMPOLICY_PRIORITY_LOW, DMMPOLICY_TIME_CRITICAL, DMMPOLICY_NOT_PAUSED} // rfColl Stack
  },
  // State 2: BT = (adv | connected), RF = any
  // BleSp = low priority | time none critical, rfColl  = high priority | time critical
  {
    {DMMPolicy_StackType_BlePeripheral, DMMPolicy_StackType_WsnNode},
    {(DMMPOLICY_STACKSTATE_BLEPERIPH_ADV | DMMPOLICY_STACKSTATE_BLEPERIPH_CONNECTED), DMMPOLICY_STACKSTATE_RFCOLL_ANY},
    {DMMPOLICY_PRIORITY_LOW, DMMPOLICY_TIME_NONE_CRITICAL, DMMPOLICY_NOT_PAUSED, //BLE SP Stack
     DMMPOLICY_PRIORITY_HIGH, DMMPOLICY_TIME_CRITICAL, DMMPOLICY_NOT_PAUSED} //rfColl Stack
  },
  // State 3: BT = any, RF = idle
  // BleSp = low priority | time none critical, rfColl = high priority | time critical
  {
    {DMMPolicy_StackType_BlePeripheral, DMMPolicy_StackType_WsnNode},
    {(DMMPOLICY_STACKSTATE_BLEPERIPH_ANY), DMMPOLICY_STACKSTATE_RFCOLL_IDLE},
    {DMMPOLICY_PRIORITY_LOW, DMMPOLICY_TIME_NONE_CRITICAL, DMMPOLICY_NOT_PAUSED, //BLE SP Stack
     DMMPOLICY_PRIORITY_HIGH, DMMPOLICY_TIME_CRITICAL, DMMPOLICY_NOT_PAUSED} //rfColl Stack
  },
  // Default State: If matching state is not found
  // BleSp = low priority | time none critical, rfColl = high priority | time critical
  {
    {DMMPolicy_StackType_BlePeripheral, DMMPolicy_StackType_WsnNode},
    {DMMPOLICY_STACKSTATE_BLEPERIPH_ANY, DMMPOLICY_STACKSTATE_RFCOLL_ANY},
    {DMMPOLICY_PRIORITY_LOW, DMMPOLICY_TIME_NONE_CRITICAL, DMMPOLICY_NOT_PAUSED, //BLE SP Stack
     DMMPOLICY_PRIORITY_HIGH, DMMPOLICY_TIME_CRITICAL, DMMPOLICY_NOT_PAUSED} //rfColl Stack
  },
};
```

Case Study: Initialize Multi-protocol Manager

- **Initialize the DMM in main()**
  - Initialize and open the DMM policy manager
  - Initialize and open the DMM scheduler

```
/* initialize and open the DMM policy manager */
DMMPolicy_init();
DMMPolicy_Params_init(&dmmPolicyparams);
dmmPolicyparams.numPolicyTableEntries = DMMPolicy_customPolicyTableSize;
dmmPolicyparams.policyTable = DMMPolicy_customPolicyTable;
DMMPolicy_open(&dmmPolicyparams);
```

```
/* initialize and open the DMM scheduler */
DMMSch_init();
DMMSch_Params_init(&dmmSchParams);
DMMSch_open(&dmmSchParams);
```

- **Register clients with the DMM Scheduler**
  - DMM client in this case is the Task handle that the stack is running in
  - BLE-stack task handle available via the ICall API
  - Collector-stack task handle should be trivial to retrieve

```
/* register clients with DMM scheduler */
DMMSch_registerClient(pBleTaskHndl, DMMPolicy_StackType_BlePeripheral);
DMMSch_registerClient(pRfCollTaskHndl, DMMPolicy_StackType_WsnNode);
```

- **Set the default states for the stacks**
  - BLE Peripheral is set to **Advertising**
  - RF Collector is set to **Idle**

```
/*set the stacks in default states */
DMMPolicy_updateStackState(DMMPolicy_StackType_BlePeripheral, DMMPOLICY_STACKSTATE_BLEPERIPH_ADV);
DMMPolicy_updateStackState(DMMPolicy_StackType_WsnNode, DMMPOLICY_STACKSTATE_RFCOLL_IDLE);
```

Multi-protocol Manager debugging

- **Route the RF Core PA and LNA signals to GPIO pins**
  - Any two unused IOs can be used
  - The mapping is permanent as long as the PIN driver is initialized and opened correctly

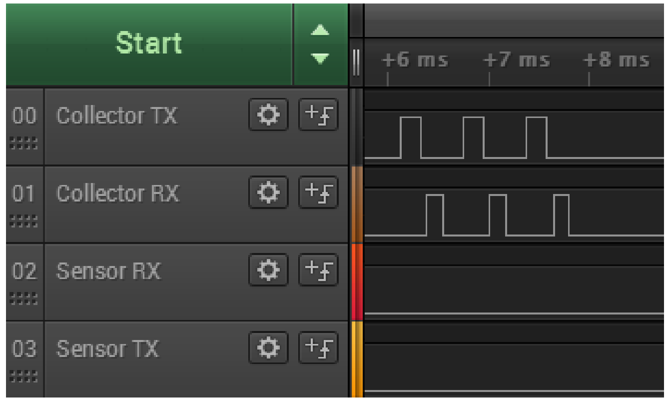
```
#include <ti/drivers/pin/PINCC26XX.h>

PIN_Config pin_table[] = {
    CC1352R1_LAUNCHXL_DIO21 | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    CC1352R1_LAUNCHXL_DIO22 | PIN_GPIO_OUTPUT_EN | PIN_GPIO_LOW | PIN_PUSHPULL | PIN_DRVSTR_MAX,
    PIN_TERMINATE
};

void foo(void)
{
    pin_handle = PIN_open(&pin_state, pin_table);

    // Map RFC_GPO0 to IO 21
    PINCC26XX_setMux(pinHandle, CC1352R1_LAUNCHXL_DIO21, PINCC26XX_MUX_RFC_GPO0);
    // Map RFC_GPO1 to IO 22
    PINCC26XX_setMux(pinHandle, CC1352R1_LAUNCHXL_DIO22, PINCC26XX_MUX_RFC_GPO1)
}
```

- **Probe the two IOs with a Logic Analyzer to view RF activity**

Multi-protocol Manager debugging

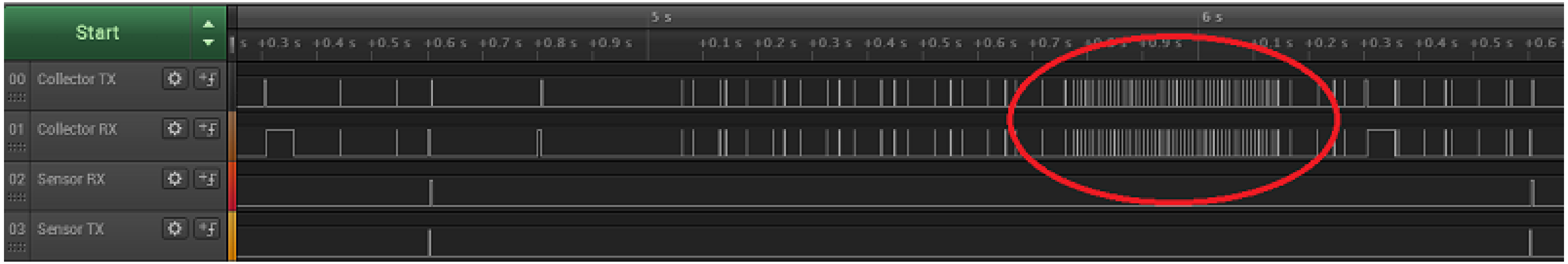
- Below is a full period of the DMM Device from the Case Study

- A closer look at the BLE advertisements



Multi-protocol Manager debugging

- **Below illustrates one connection window when a BLE Central performs a connection**



- **Below illustrates a full period while connected to both a Sensor and a BLE Central**