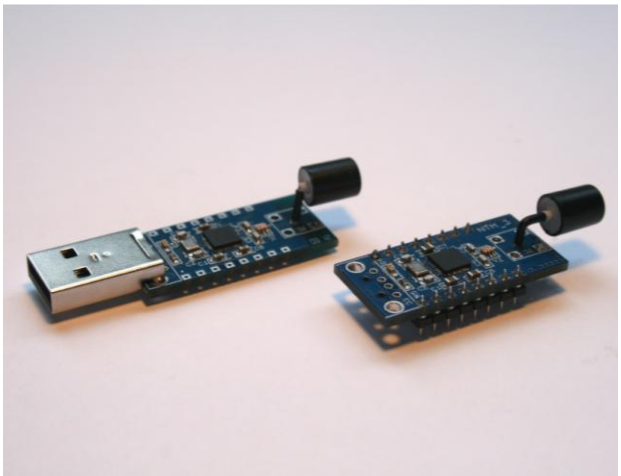
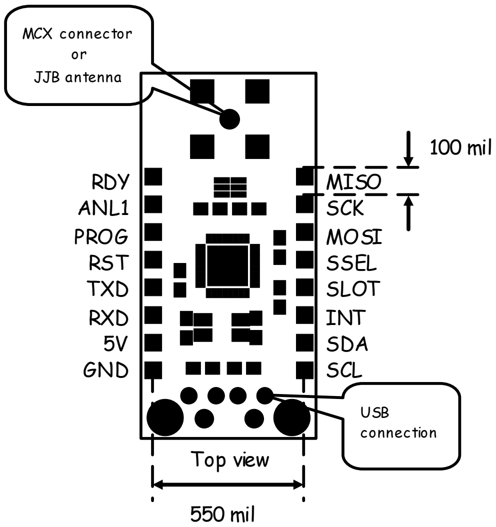
ADDING THIRD PARTY SOFTWARE TO NTM

Adding third party software to the NTM

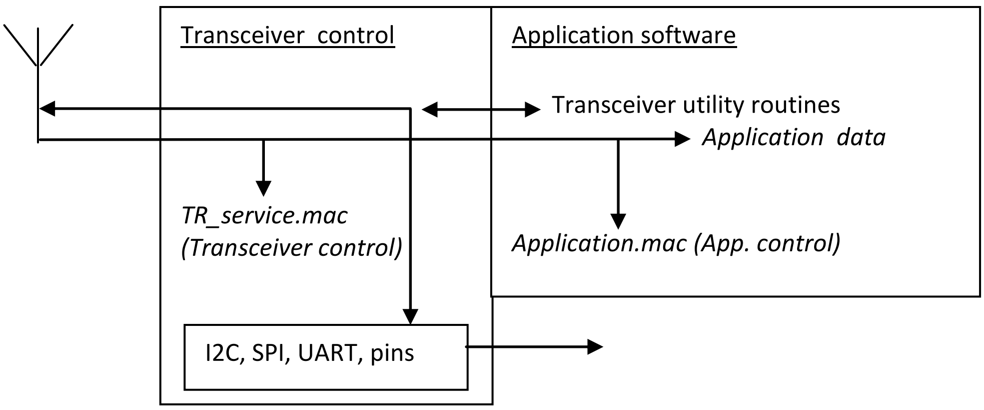




The NTM operating system is programmed in C and uses about 5 sectors of LPC1114 Flash memory. This leaves 2 sectors of 4K for the OEM own application program.
In the near future versions of the LPC11xx with larger flash memory will be applied, providing more space for the OEM own application programs.

The LPC1114 houses 8KB of RAM. The memory space above 4K is available for third party software.
OEM application programs should **start RAM at 0x10001000h.**
Stack space in the lower 4kB is sufficient for standard applications.

Software structure



Adding third party software to the NTM

The NTM is a package switching device using IEEE 802.15.4 frames.
This standard defines a number of frame types, to be used in particular cases.

- Data frame:
  Carry data from one device to another and the data is processed by an application the device is connected with. The software in the NTM hands this over to the application program.
- Mac frame:
  Carry information from device to another device meant for proper control of the devices.
  There are two versions:
  Version 1 is meant to control the transceiver (remote programming)
  Version 2 is meant to control the application program
- Beacon frame:
  Short frames meant for synchronisation between devices
- Acknowledge frame:
  Short frame to report reception
- Besides that the NTM uses a separate data type for audio streaming

The NTM operating system is an event driven task scheduler. In the idle operation the device can be put to sleep. The events are:

- Reception of a frame
- Interrupt by the system timer, awakes the NTM
- Interrupt on the Aux pin, awakes the NTM
- Interrupt by the I2C controller
- Interrupt by a digital input awakes the NTM
- Interrupt by UART, awakes the NTM

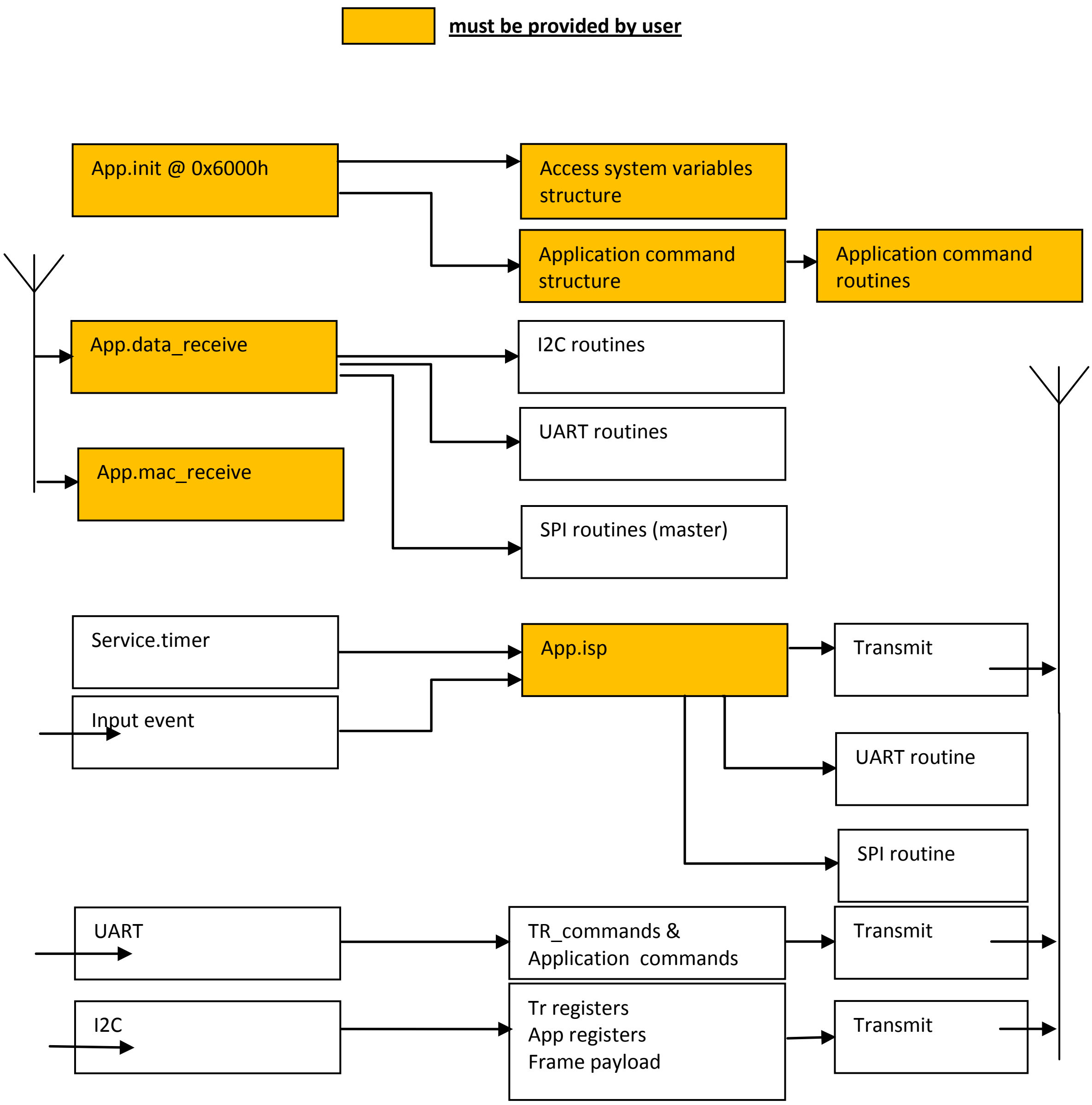
Be aware that, in case the NTM is asleep, an I2C transfer needs to be preceded by an interrupt on the Aux pin to awaken the device.

To incorporate your own application in the NTM, four event handles must be provided and ad libitum, private command routines to control your application, using the NTM parser, can be added.

A header file is available that contains all required declarations and sample lines.

Adding third party software to the NTM

Chart for OEM programming



Adding third party software to the NTM

Using system variables and functions

The variables and functions are brought together in a structure of variable and functions pointers

```
struct regt
  {// regt
    unsigned char*        status;        //I2C reg 0 status flags F1
    unsigned char* net_idH;       //I2C reg 1 House code or network id
    unsigned char* net_idL;       //I2C reg 2
    unsigned char* device_idH;    //I2C reg 3 device id
    unsigned char* device_idL;    //I2C reg 4 device id
    unsigned char* alarmgr;       //I2C reg 5 alarm ne control group
    unsigned char* gatewaynr;     //I2C reg 6 gateway number of th associated gateway
    unsigned char* dest_idH;      //I2C reg 7 destination id, origin of a received frame
    unsigned char* dest_idL;      //I2C reg 8
    unsigned char* batlimit;      //I2C reg 9 minimum power supply level fot the NTM in decivolts
    unsigned char* power;         //I2C reg 10 transmission power
    unsigned char* mt;            //I2C reg 11 status report period in 10 s intervals
    unsigned char* i2c_adres;     //I2C reg 12 I2C address of a connected I2C device
    unsigned char* i2c_width;     //I2C reg 13 I2C register with (1 or 2 bytes)
    unsigned char* status2;       //I2C reg 14 status flags F2
    unsigned char* par1;          //I2C reg 16 applcations registers
    unsigned char* par2;          //I2C reg 17
    unsigned char* par3;          //I2C reg 18
    unsigned char* par4;          //I2C reg 19
    unsigned char* par5;          //I2C reg 20
    unsigned char* par6;          //I2C reg 21
    unsigned char* par7;          //I2C reg 22
    unsigned char* ts1;           //I2C reg 23 time stamp registers for received frame
    unsigned char* ts2;           //I2C reg 24
    unsigned char* ts3;           //I2C reg 25
    unsigned char* ts4;           //I2C reg 26
    unsigned char* dtdt;          //I2C reg 27 temperature rise or fall limit
    unsigned char* tempmax;       //I2C reg 28 maximum temperature limit
    unsigned char* vl;            //I2C reg 29 validity received frame indicator
    unsigned char* lq;            //I2C reg 30 link quality received frame
    unsigned char* ed;            //I2C reg 31 Energy density above threshold received frame
    unsigned char*        tx_payload;    //Transmit payload buffer
    unsigned char*        rx_payload;    //Received payload buffer
    unsigned char*      rx_length;//Length of  received payload
    unsigned char*        report;        //see table below
    unsigned char*        voltage;
    unsigned char*        temperature;
    unsigned char*        ana_in;        //digital value of the analogue input
    unsigned int*         tick;          // 1 ms tick counter
    unsigned int*         tick25;        //2.5 s tick counter
    unsigned char*        apicom;        //number application commands
                      struct cmd*    command_pntr;  //pointer to structure with application commands
                      struct api*    service;       //service routine structure entry point
    /*  Function pointers                        */
```

Adding third party software to the NTM

```
void            (*tmr16_1)    (void);
unsigned int*   (*SetTimer)    (unsigned int val);
char            (*load_task)   (unsigned char* name_string,
                               API_pointer function,
                               unsigned char priority);
void            (*i2c_init)    (void);                  //startup I2C
void            (*spi_init)    (void);                  //startup SPI2
char            (*i2c_access)  (unsigned char direction,
                               struct I2C,
                               unsigned short address,
                               unsigned char* array,
                               unsigned short length);  //I2C communication routine
void            (*whileNot)    (unsigned char variable_type,
                                void * variable,
                               unsigned int value);
char            (*send)        (unsigned char buf_length);
char            (*sendto)(unsigned short dest_address,  unsigned char buf_length);
char            (*remote)      (unsigned short dest_address,  unsigned char buf_length);
char            (*groupcom)    (unsigned short control_group,
                               unsigned short dest_address,
                                unsigned char buf_length);
};
```

The Ninthway High Secure Radio Network uses two frame types for data transfer: Data frames and Mac frames.

- Data frames contain data to be sent from application to application
- Mac frames contain commands for controlling the transceiver or the application

The frames have a standard header followed by the frame payload.
The frame payload is divided into a 4 byte mac header and maximum 100 bytes of payload.
The mac header differs for data frames and mac frames.

Payload header	**Data header**	**Mac header**
report	Alarms flag registers	Mac command
voltage	Supply voltage in dV	Group/control number
temperature	Optional temperature in ˚C	Destination address LB
ana_in	Digital value analogue input	Destination address HB

Examples for use

#define OWN_PNT_LOCATION 0x5460

Declaration of the variable and function pointer structure:

const struct regt Own;*

Own = OWN_PNT_LOCATION; connect the structure to the one built into the NTM firmware:

Example of using a system variable:

Adding third party software to the NTM

```
*Own->status = 2;
If(var < *Own->batlimit)…
```

Examples of using system functions

```
#define READ        1
#define WRITE       0

Own->spi_init();                        //initiate SPI2. Data rate is 6 MHz.
Own->tmr16_1 = &private_ function;      //add interrupt routine to timer16_1
Own->whileNot(1,&CharVar, 0);           //wait while unsigned char CharVar ≠0;
Own->i2c_access(READ,PCA9551,0,Databuffer,10);
```

Read i2c_device PCA9551, starting from address 0 and put 10 bytes in array Databuffer.
PCA9551 is a structure of type:

```
struct I2C
    {
    unsigned char address;      //I2C address (8 bit format)
    unsigned    reg_type:   1;  //bit 0, address = 1 byte, bit = 1, address = 2 bytes
    unsigned    fast:       1;  //1 fast I2C, 0 standard I2C
    };

const struct I2C PCA9551 = {0xC0,0, 0}
```

<u>Provision of event handling routines</u>

An application requires 4 routines to handle events. The pointer to the routines are combined into a structure.
The pointers to these routines are to be loaded during start-up. Your own application must fill the members of the structure during initiation.

The pointer to the structure can be found as struct api* service in the system variable structure and requires a type definition:

```
typedef void (*API_pointer)(void);          function pointer with no parameters
struct api
    {
    API_pointer strt;
    API_pointer recv;
    API_pointer isp;
    API_pointer mac;
    };
```

The initiation routine should provide:

```
Own->service->strt  = &Init_routine;     //application initiation routine
Own->service->recv  = &Radio_receive;    //application handling of received data
Own->service->isp   = &Int_service;      //application handling of timer and input interrupt
Own->service->mac   = &Mac_service;      //application handling application control commands
```

Adding third party software to the NTM

Remember the Api structure is an object in RAM. Every time you start up the device this structure needs to be filled with the pointer values of the event handlers, before the actual operation of the NTM starts. That is no problem for the *Own->service->recv, Own->service->isp* and *Own->service->mac pointers*, they are to be loaded during the application initiation routine, but it is for the *Own->service->strt* routine.

There is no way of telling the operating system, between initiation of the NTM and initiation of the application, where the application initiation routine can be found. It is therefore decided to place the application initiation routine at the beginning of the application program sector 6.

The operating system expects to find the application routine at the start of sector 6 at address **0x6000h**.

Init_routine = (API_pointer) 0x6000.

Using the NTM operating system

The NTM operating system is an event driven task scheduler. In the idle operation the device can be put to sleep.

Tasks are executed depending on their priority level (2 first, 0 last). There is no multitasking or time slicing. Each task has full control over the device. So there will be no conflicts over use of peripherals causing extra wait states that consume unnecessarily precious power.

However there is a down side. Tasks might not terminate due to the use of endless loops that do not meet their break requirement.

Tasks have a time limit of **1.5 s**. Tasks that take longer generate a time-out error message. The system maintains a watchdog counter that will reset the device in the event the system locks up for more than 10 seconds. However this causes loss of data, a very unwanted situation in a high secure system.

By using the *WhileNot* function in your application the operating system will, at time-out, fulfil the while break requirement, causing the waiting execution to proceed and finish the task. In that case the time-out error will be issued over the UART, but the task will not lock the device and no forced reset will take place.

A task is loaded with: *Own->load_task ("task name", task function pointer, priority)*.

A task can also be loaded using the Tmr16_1. The interrupt routine of this timer contains a function pointer *(*Tmr16_1)*, that can be used to periodically execute or execute after a certain delay, a private function.

The system uses the Timer32_0 as periodic loader for the *service.isr* function that regulates the broadcast of a status message and private periodical functions. This timer runs under awake and sleeps conditions. *Do not use it for private purposes. Private purposes have access via the Own->service->isp* pointer.

The SysTick of the ARM Cortex M0 feeds a 1 ms 32 bit counter Tick that is available through the system variable structure. The SysTick stops during sleep. So during sleep the Tick variable is not incremented.

There is a second, 32 bit counter, Tick25 that is incremented every 2.5 seconds independent of sleep or wake state.

Controlled by the SysTick interrupt routine is a set of 6 down counters of size int. At setting of the timer it automatically looks for a free timer and passes its address in the return value. The contents of this address can be checked whether it has reached zero. If all timers are in use it will return a 0.

Timer = Own->SetTimer(1000) starts a timer that will count down 1 second. Its value is available thru dereferencing Timer.

Adding third party software to the NTM

Adding private commands to the NTM

You can enhance the flexibility of your application by adding commands that control the operation of your application. The NTM contains a command parser that is accessible to third parties.

Similar to the setting of NTM parameters via the UART commands, application parameters can be controlled with own application commands. Via the RMOT and CGRP commands applications can be controlled remotely.

A command consists of a 4 character mnemonic terminated either by a '=' or a '?'.

The '?' is meant to provide only an answer; the '=' changes the parameter and returns an answer.

But any interpretation, after the location of the mnemonic and jump to the command subroutine, is entirely up to the programmer.

Commands are housed in a two member structure array like:

```
#define APICOM   2

struct cmd
        {
        unsigned char command[5];
        Function_pointer Function;
        };
```

Example:

```
const struct cmd Api_com[APICOM] =
          {
        "TMMX",&Maxtemp,
        "DTdt",&Maxdt,
        …
        …
        };
```

For the parser to be able to find the extra commands, the application initiation routine must provide information by passing the command structure address to a pointer and number of commands to a system variable:

```
**Own->Command_pntr = *Api_com;  //contents of pointer pointed to by Command_pntr = apointer to command table
*Own->Apicom = APICOM;                  //number of commands
```

Function:

```
unsigned char* Maxtemp        (struct bp *loadp);
```

A command function takes a pointer to a structure providing information about the buffer that contains the command strings and returns a pointer to a string containing a result of the function.

```
struct bp
        {
        unsigned char *buffer;    //pointer to the buffer that contains the command
```

Adding third party software to the NTM

```
unsigned char *begin;   //pointer that indicates start position in buffer for parsing
unsigned char *end;     //pointer that indicates last position of commands in buffer
unsigned char *length;  //pointer of last position of buffer
};
```

This setup is laid out to be used with a revolving buffer. But in that case scanning through the buffer requires an increment of *begin taking into account the crossover from *length (end if the buffer) to *buffer (begin of the buffer). The buffer is empty when *begin = *end.

An alternative way to control parameters is to use the application registers I2C reg 16 – I2C reg 31 either by connecting the NTM to an I2C master or provide I2C data in the payload and have an application program transfer the payload bytes to the proper I2C registers.

Transmission and reception

For reception the device needs to be awake either by keeping it awake or using synchronised operation. Synced operation is handled by the NTM software.

After the reception of a valid frame an interrupt is issued activating the *Own->service->recv routine*.
Data from the received frame are to be found in the proper I2Cregisters and the **Own->rx_payload*
Buffer.
The length of the buffer is given in **Own->rx_length.*

For transmission the frame addressing is controlled via the I2C registers.
An application can load data into the payload using the *Own->tx_payload* entry point.
The length of the payload is provided via a parameter in the transmit function.

There are four standard types of transmission frames.

1. Send a data frame using source ID only.
   Own->send(unsigned char buf_length).

2. Send data frame to a destination device, use its 12 bit address.
   Own->sendto(unsigned short dest_address, unsigned char buf_length).

3. Send a mac (control) frame to the NTM proper (remote programming). Use its 12 bit address. The payload contains a command string as specified in application note 2 and is terminated by a CR LF.
   Own->remote (unsigned short dest_address, unsigned char buf_length).

4. Control a device remotely (actor steering) via a mac frame. The payload contains information for the application software. Use a 12 bit destination address.
   Own->groupcom(unsigned short group, unsigned short dest_address, unsigned char buf_length)

All functions return a 1 if transmission is OK or else 0.