/\*

\* Copyright (c) 2022 Michal Morsisko

\* Copyright (c) 2015-2016 Intel Corporation

\*

\* SPDX-License-Identifier: Apache-2.0

\*/

#include <zephyr/types.h>

#include <stddef.h>

#include <zephyr/sys/printk.h>

#include <zephyr/sys/reboot.h>

#include <zephyr/settings/settings.h>

#include <zephyr/bluetooth/bluetooth.h>

#include <zephyr/bluetooth/conn.h>

#include <zephyr/bluetooth/uuid.h>

#include <zephyr/bluetooth/gatt.h>

/\* Custom Service Variables \*/

#define BT\_UUID\_CUSTOM\_SERVICE\_VAL \

BT\_UUID\_128\_ENCODE(0x12345678, 0x1234, 0x5678, 0x1234, 0x56789abcdef0)

static struct bt\_uuid\_128 primary\_service\_uuid = BT\_UUID\_INIT\_128(

BT\_UUID\_CUSTOM\_SERVICE\_VAL);

static struct bt\_uuid\_128 read\_characteristic\_uuid = BT\_UUID\_INIT\_128(

BT\_UUID\_128\_ENCODE(0x12345678, 0x1234, 0x5678, 0x1234, 0x56789abcdef1));

static struct bt\_uuid\_128 write\_characteristic\_uuid = BT\_UUID\_INIT\_128(

BT\_UUID\_128\_ENCODE(0x12345678, 0x1234, 0x5678, 0x1234, 0x56789abcdef2));

static int signed\_value;

static struct bt\_le\_adv\_param adv\_param;

static int bond\_count;

static ssize\_t read\_signed(struct bt\_conn \*conn, const struct bt\_gatt\_attr \*attr,

void \*buf, uint16\_t len, uint16\_t offset)

{

int \*value = &signed\_value;

return bt\_gatt\_attr\_read(conn, attr, buf, len, offset, value,

sizeof(signed\_value));

}

static ssize\_t write\_signed(struct bt\_conn \*conn, const struct bt\_gatt\_attr \*attr,

const void \*buf, uint16\_t len, uint16\_t offset,

uint8\_t flags)

{

int \*value = &signed\_value;

if (offset + len > sizeof(signed\_value)) {

return BT\_GATT\_ERR(BT\_ATT\_ERR\_INVALID\_OFFSET);

}

memcpy(value + offset, buf, len);

return len;

}

/\* Vendor Primary Service Declaration \*/

BT\_GATT\_SERVICE\_DEFINE(primary\_service,

BT\_GATT\_PRIMARY\_SERVICE(&primary\_service\_uuid),

BT\_GATT\_CHARACTERISTIC(&read\_characteristic\_uuid.uuid,

BT\_GATT\_CHRC\_READ,

BT\_GATT\_PERM\_READ,

read\_signed, NULL, NULL),

BT\_GATT\_CHARACTERISTIC(&write\_characteristic\_uuid.uuid,

BT\_GATT\_CHRC\_WRITE,

BT\_GATT\_PERM\_WRITE\_ENCRYPT,

NULL, write\_signed, NULL),

);

static const struct bt\_data ad[] = {

BT\_DATA\_BYTES(BT\_DATA\_FLAGS, (BT\_LE\_AD\_GENERAL | BT\_LE\_AD\_NO\_BREDR))

};

static const struct bt\_data sd[] = {

BT\_DATA\_BYTES(BT\_DATA\_UUID128\_ALL, BT\_UUID\_CUSTOM\_SERVICE\_VAL)

};

static void connected(struct bt\_conn \*conn, uint8\_t err)

{

if (err) {

printk("Connection failed (err 0x%02x)\n", err);

} else {

printk("Connected\n");

}

}

static void disconnected(struct bt\_conn \*conn, uint8\_t reason)

{

printk("Disconnected (reason 0x%02x)\n", reason);

}

BT\_CONN\_CB\_DEFINE(conn\_callbacks) = {

.connected = connected,

.disconnected = disconnected

};

static void add\_bonded\_addr\_to\_filter\_list(const struct bt\_bond\_info \*info, void \*data)

{

char addr\_str[BT\_ADDR\_LE\_STR\_LEN];

bt\_le\_filter\_accept\_list\_add(&info->addr);

bt\_addr\_le\_to\_str(&info->addr, addr\_str, sizeof(addr\_str));

printk("Added %s to advertising accept filter list\n", addr\_str);

bond\_count++;

}

static void bt\_ready(void)

{

int err;

printk("Bluetooth initialized\n");

if (IS\_ENABLED(CONFIG\_SETTINGS)) {

settings\_load();

}

bond\_count = 0;

bt\_foreach\_bond(BT\_ID\_DEFAULT, add\_bonded\_addr\_to\_filter\_list, NULL);

adv\_param = \*BT\_LE\_ADV\_CONN\_NAME;

/\* If we have got at least one bond, activate the filter \*/

if (bond\_count) {

/\* BT\_LE\_ADV\_OPT\_FILTER\_CONN is required to activate accept filter list,

\* BT\_LE\_ADV\_OPT\_FILTER\_SCAN\_REQ will prevent sending scan response data to

\* devices, that are not on the accept filter list

\*/

adv\_param.options |= BT\_LE\_ADV\_OPT\_FILTER\_CONN | BT\_LE\_ADV\_OPT\_FILTER\_SCAN\_REQ;

}

err = bt\_le\_adv\_start(&adv\_param, ad, ARRAY\_SIZE(ad), sd, ARRAY\_SIZE(sd));

if (err) {

printk("Advertising failed to start (err %d)\n", err);

} else {

printk("Advertising successfully started\n");

}

}

void pairing\_complete(struct bt\_conn \*conn, bool bonded)

{

printk("Pairing completed. Rebooting in 5 seconds...\n");

k\_sleep(K\_SECONDS(5));

sys\_reboot(SYS\_REBOOT\_WARM);

}

static struct bt\_conn\_auth\_info\_cb bt\_conn\_auth\_info = {

.pairing\_complete = pairing\_complete

};

int main(void)

{

int err;

err = bt\_enable(NULL);

if (err) {

printk("Bluetooth init failed (err %d)\n", err);

return 0;

}

bt\_ready();

bt\_conn\_auth\_info\_cb\_register(&bt\_conn\_auth\_info);

while (1) {

k\_sleep(K\_FOREVER);

}

return 0;

}

/\* main.c - Application main entry point \*/

/\*

\* Copyright (c) 2015-2016 Intel Corporation

\*

\* SPDX-License-Identifier: Apache-2.0

\*/

#include <zephyr/types.h>

#include <stddef.h>

#include <string.h>

#include <errno.h>

#include <zephyr/sys/printk.h>

#include <zephyr/sys/byteorder.h>

#include <zephyr/kernel.h>

#include <zephyr/settings/settings.h>

#include <zephyr/bluetooth/bluetooth.h>

#include <zephyr/bluetooth/hci.h>

#include <zephyr/bluetooth/conn.h>

#include <zephyr/bluetooth/uuid.h>

#include <zephyr/bluetooth/gatt.h>

#include <zephyr/bluetooth/services/bas.h>

#include <zephyr/bluetooth/services/hrs.h>

#include <zephyr/bluetooth/services/ias.h>

#include "cts.h"

/\* Custom Service Variables \*/

#define BT\_UUID\_CUSTOM\_SERVICE\_VAL \

BT\_UUID\_128\_ENCODE(0x12345678, 0x1234, 0x5678, 0x1234, 0x56789abcdef0)

static struct bt\_uuid\_128 vnd\_uuid = BT\_UUID\_INIT\_128(

BT\_UUID\_CUSTOM\_SERVICE\_VAL);

static struct bt\_uuid\_128 vnd\_enc\_uuid = BT\_UUID\_INIT\_128(

BT\_UUID\_128\_ENCODE(0x12345678, 0x1234, 0x5678, 0x1234, 0x56789abcdef1));

static struct bt\_uuid\_128 vnd\_auth\_uuid = BT\_UUID\_INIT\_128(

BT\_UUID\_128\_ENCODE(0x12345678, 0x1234, 0x5678, 0x1234, 0x56789abcdef2));

#define VND\_MAX\_LEN 20

static uint8\_t vnd\_value[VND\_MAX\_LEN + 1] = { 'V', 'e', 'n', 'd', 'o', 'r'};

static uint8\_t vnd\_auth\_value[VND\_MAX\_LEN + 1] = { 'V', 'e', 'n', 'd', 'o', 'r'};

static uint8\_t vnd\_wwr\_value[VND\_MAX\_LEN + 1] = { 'V', 'e', 'n', 'd', 'o', 'r' };

static ssize\_t read\_vnd(struct bt\_conn \*conn, const struct bt\_gatt\_attr \*attr,

void \*buf, uint16\_t len, uint16\_t offset)

{

const char \*value = attr->user\_data;

return bt\_gatt\_attr\_read(conn, attr, buf, len, offset, value,

strlen(value));

}

static ssize\_t write\_vnd(struct bt\_conn \*conn, const struct bt\_gatt\_attr \*attr,

const void \*buf, uint16\_t len, uint16\_t offset,

uint8\_t flags)

{

uint8\_t \*value = attr->user\_data;

if (offset + len > VND\_MAX\_LEN) {

return BT\_GATT\_ERR(BT\_ATT\_ERR\_INVALID\_OFFSET);

}

memcpy(value + offset, buf, len);

value[offset + len] = 0;

return len;

}

static uint8\_t simulate\_vnd;

static uint8\_t indicating;

static struct bt\_gatt\_indicate\_params ind\_params;

static void vnd\_ccc\_cfg\_changed(const struct bt\_gatt\_attr \*attr, uint16\_t value)

{

simulate\_vnd = (value == BT\_GATT\_CCC\_INDICATE) ? 1 : 0;

}

static void indicate\_cb(struct bt\_conn \*conn,

struct bt\_gatt\_indicate\_params \*params, uint8\_t err)

{

printk("Indication %s\n", err != 0U ? "fail" : "success");

}

static void indicate\_destroy(struct bt\_gatt\_indicate\_params \*params)

{

printk("Indication complete\n");

indicating = 0U;

}

#define VND\_LONG\_MAX\_LEN 74

static uint8\_t vnd\_long\_value[VND\_LONG\_MAX\_LEN + 1] = {

'V', 'e', 'n', 'd', 'o', 'r', ' ', 'd', 'a', 't', 'a', '1',

'V', 'e', 'n', 'd', 'o', 'r', ' ', 'd', 'a', 't', 'a', '2',

'V', 'e', 'n', 'd', 'o', 'r', ' ', 'd', 'a', 't', 'a', '3',

'V', 'e', 'n', 'd', 'o', 'r', ' ', 'd', 'a', 't', 'a', '4',

'V', 'e', 'n', 'd', 'o', 'r', ' ', 'd', 'a', 't', 'a', '5',

'V', 'e', 'n', 'd', 'o', 'r', ' ', 'd', 'a', 't', 'a', '6',

'.', ' ' };

static ssize\_t write\_long\_vnd(struct bt\_conn \*conn,

const struct bt\_gatt\_attr \*attr, const void \*buf,

uint16\_t len, uint16\_t offset, uint8\_t flags)

{

uint8\_t \*value = attr->user\_data;

if (flags & BT\_GATT\_WRITE\_FLAG\_PREPARE) {

return 0;

}

if (offset + len > VND\_LONG\_MAX\_LEN) {

return BT\_GATT\_ERR(BT\_ATT\_ERR\_INVALID\_OFFSET);

}

memcpy(value + offset, buf, len);

value[offset + len] = 0;

return len;

}

static const struct bt\_uuid\_128 vnd\_long\_uuid = BT\_UUID\_INIT\_128(

BT\_UUID\_128\_ENCODE(0x12345678, 0x1234, 0x5678, 0x1234, 0x56789abcdef3));

static struct bt\_gatt\_cep vnd\_long\_cep = {

.properties = BT\_GATT\_CEP\_RELIABLE\_WRITE,

};

static int signed\_value;

static ssize\_t read\_signed(struct bt\_conn \*conn, const struct bt\_gatt\_attr \*attr,

void \*buf, uint16\_t len, uint16\_t offset)

{

const char \*value = attr->user\_data;

return bt\_gatt\_attr\_read(conn, attr, buf, len, offset, value,

sizeof(signed\_value));

}

static ssize\_t write\_signed(struct bt\_conn \*conn, const struct bt\_gatt\_attr \*attr,

const void \*buf, uint16\_t len, uint16\_t offset,

uint8\_t flags)

{

uint8\_t \*value = attr->user\_data;

if (offset + len > sizeof(signed\_value)) {

return BT\_GATT\_ERR(BT\_ATT\_ERR\_INVALID\_OFFSET);

}

memcpy(value + offset, buf, len);

return len;

}

static const struct bt\_uuid\_128 vnd\_signed\_uuid = BT\_UUID\_INIT\_128(

BT\_UUID\_128\_ENCODE(0x13345678, 0x1234, 0x5678, 0x1334, 0x56789abcdef3));

static const struct bt\_uuid\_128 vnd\_write\_cmd\_uuid = BT\_UUID\_INIT\_128(

BT\_UUID\_128\_ENCODE(0x12345678, 0x1234, 0x5678, 0x1234, 0x56789abcdef4));

static ssize\_t write\_without\_rsp\_vnd(struct bt\_conn \*conn,

const struct bt\_gatt\_attr \*attr,

const void \*buf, uint16\_t len, uint16\_t offset,

uint8\_t flags)

{

uint8\_t \*value = attr->user\_data;

if (!(flags & BT\_GATT\_WRITE\_FLAG\_CMD)) {

/\* Write Request received. Reject it since this Characteristic

\* only accepts Write Without Response.

\*/

return BT\_GATT\_ERR(BT\_ATT\_ERR\_WRITE\_REQ\_REJECTED);

}

if (offset + len > VND\_MAX\_LEN) {

return BT\_GATT\_ERR(BT\_ATT\_ERR\_INVALID\_OFFSET);

}

memcpy(value + offset, buf, len);

value[offset + len] = 0;

return len;

}

/\* Vendor Primary Service Declaration \*/

BT\_GATT\_SERVICE\_DEFINE(vnd\_svc,

BT\_GATT\_PRIMARY\_SERVICE(&vnd\_uuid),

BT\_GATT\_CHARACTERISTIC(&vnd\_enc\_uuid.uuid,

BT\_GATT\_CHRC\_READ | BT\_GATT\_CHRC\_WRITE |

BT\_GATT\_CHRC\_INDICATE,

BT\_GATT\_PERM\_READ\_ENCRYPT |

BT\_GATT\_PERM\_WRITE\_ENCRYPT,

read\_vnd, write\_vnd, vnd\_value),

BT\_GATT\_CCC(vnd\_ccc\_cfg\_changed,

BT\_GATT\_PERM\_READ | BT\_GATT\_PERM\_WRITE\_ENCRYPT),

BT\_GATT\_CHARACTERISTIC(&vnd\_auth\_uuid.uuid,

BT\_GATT\_CHRC\_READ | BT\_GATT\_CHRC\_WRITE,

BT\_GATT\_PERM\_READ\_AUTHEN |

BT\_GATT\_PERM\_WRITE\_AUTHEN,

read\_vnd, write\_vnd, vnd\_auth\_value),

BT\_GATT\_CHARACTERISTIC(&vnd\_long\_uuid.uuid, BT\_GATT\_CHRC\_READ |

BT\_GATT\_CHRC\_WRITE | BT\_GATT\_CHRC\_EXT\_PROP,

BT\_GATT\_PERM\_READ | BT\_GATT\_PERM\_WRITE |

BT\_GATT\_PERM\_PREPARE\_WRITE,

read\_vnd, write\_long\_vnd, &vnd\_long\_value),

BT\_GATT\_CEP(&vnd\_long\_cep),

BT\_GATT\_CHARACTERISTIC(&vnd\_signed\_uuid.uuid, BT\_GATT\_CHRC\_READ |

BT\_GATT\_CHRC\_WRITE | BT\_GATT\_CHRC\_AUTH,

BT\_GATT\_PERM\_READ | BT\_GATT\_PERM\_WRITE,

read\_signed, write\_signed, &signed\_value),

BT\_GATT\_CHARACTERISTIC(&vnd\_write\_cmd\_uuid.uuid,

BT\_GATT\_CHRC\_WRITE\_WITHOUT\_RESP,

BT\_GATT\_PERM\_WRITE, NULL,

write\_without\_rsp\_vnd, &vnd\_wwr\_value),

);

static const struct bt\_data ad[] = {

BT\_DATA\_BYTES(BT\_DATA\_FLAGS, (BT\_LE\_AD\_GENERAL | BT\_LE\_AD\_NO\_BREDR)),

BT\_DATA\_BYTES(BT\_DATA\_UUID16\_ALL,

BT\_UUID\_16\_ENCODE(BT\_UUID\_HRS\_VAL),

BT\_UUID\_16\_ENCODE(BT\_UUID\_BAS\_VAL),

BT\_UUID\_16\_ENCODE(BT\_UUID\_CTS\_VAL)),

BT\_DATA\_BYTES(BT\_DATA\_UUID128\_ALL, BT\_UUID\_CUSTOM\_SERVICE\_VAL),

};

void mtu\_updated(struct bt\_conn \*conn, uint16\_t tx, uint16\_t rx)

{

printk("Updated MTU: TX: %d RX: %d bytes\n", tx, rx);

}

static struct bt\_gatt\_cb gatt\_callbacks = {

.att\_mtu\_updated = mtu\_updated

};

static void connected(struct bt\_conn \*conn, uint8\_t err)

{

if (err) {

printk("Connection failed (err 0x%02x)\n", err);

} else {

printk("Connected\n");

}

}

static void disconnected(struct bt\_conn \*conn, uint8\_t reason)

{

printk("Disconnected (reason 0x%02x)\n", reason);

}

static void alert\_stop(void)

{

printk("Alert stopped\n");

}

static void alert\_start(void)

{

printk("Mild alert started\n");

}

static void alert\_high\_start(void)

{

printk("High alert started\n");

}

BT\_CONN\_CB\_DEFINE(conn\_callbacks) = {

.connected = connected,

.disconnected = disconnected,

};

BT\_IAS\_CB\_DEFINE(ias\_callbacks) = {

.no\_alert = alert\_stop,

.mild\_alert = alert\_start,

.high\_alert = alert\_high\_start,

};

static void bt\_ready(void)

{

int err;

printk("Bluetooth initialized\n");

cts\_init();

if (IS\_ENABLED(CONFIG\_SETTINGS)) {

settings\_load();

}

err = bt\_le\_adv\_start(BT\_LE\_ADV\_CONN\_NAME, ad, ARRAY\_SIZE(ad), NULL, 0);

if (err) {

printk("Advertising failed to start (err %d)\n", err);

return;

}

printk("Advertising successfully started\n");

}

static void auth\_passkey\_display(struct bt\_conn \*conn, unsigned int passkey)

{

char addr[BT\_ADDR\_LE\_STR\_LEN];

bt\_addr\_le\_to\_str(bt\_conn\_get\_dst(conn), addr, sizeof(addr));

printk("Passkey for %s: %06u\n", addr, passkey);

}

static void auth\_cancel(struct bt\_conn \*conn)

{

char addr[BT\_ADDR\_LE\_STR\_LEN];

bt\_addr\_le\_to\_str(bt\_conn\_get\_dst(conn), addr, sizeof(addr));

printk("Pairing cancelled: %s\n", addr);

}

static struct bt\_conn\_auth\_cb auth\_cb\_display = {

.passkey\_display = auth\_passkey\_display,

.passkey\_entry = NULL,

.cancel = auth\_cancel,

};

static void bas\_notify(void)

{

uint8\_t battery\_level = bt\_bas\_get\_battery\_level();

battery\_level--;

if (!battery\_level) {

battery\_level = 100U;

}

bt\_bas\_set\_battery\_level(battery\_level);

}

static void hrs\_notify(void)

{

static uint8\_t heartrate = 90U;

/\* Heartrate measurements simulation \*/

heartrate++;

if (heartrate == 160U) {

heartrate = 90U;

}

bt\_hrs\_notify(heartrate);

}

int main(void)

{

struct bt\_gatt\_attr \*vnd\_ind\_attr;

char str[BT\_UUID\_STR\_LEN];

int err;

err = bt\_enable(NULL);

if (err) {

printk("Bluetooth init failed (err %d)\n", err);

return 0;

}

bt\_ready();

bt\_gatt\_cb\_register(&gatt\_callbacks);

bt\_conn\_auth\_cb\_register(&auth\_cb\_display);

vnd\_ind\_attr = bt\_gatt\_find\_by\_uuid(vnd\_svc.attrs, vnd\_svc.attr\_count,

&vnd\_enc\_uuid.uuid);

bt\_uuid\_to\_str(&vnd\_enc\_uuid.uuid, str, sizeof(str));

printk("Indicate VND attr %p (UUID %s)\n", vnd\_ind\_attr, str);

/\* Implement notification. At the moment there is no suitable way

\* of starting delayed work so we do it here

\*/

while (1) {

k\_sleep(K\_SECONDS(1));

/\* Current Time Service updates only when time is changed \*/

cts\_notify();

/\* Heartrate measurements simulation \*/

hrs\_notify();

/\* Battery level simulation \*/

bas\_notify();

/\* Vendor indication simulation \*/

if (simulate\_vnd && vnd\_ind\_attr) {

if (indicating) {

continue;

}

ind\_params.attr = vnd\_ind\_attr;

ind\_params.func = indicate\_cb;

ind\_params.destroy = indicate\_destroy;

ind\_params.data = &indicating;

ind\_params.len = sizeof(indicating);

if (bt\_gatt\_indicate(NULL, &ind\_params) == 0) {

indicating = 1U;

}

}

}

return 0;

}

#include <zephyr.h>

#include <bluetooth/bluetooth.h>

#include <bluetooth/conn.h>

#include <bluetooth/hci.h>

#include <bluetooth/uuid.h>

void main(void)

{

int err;

// Set the device name

const char \*device\_name = "nRF ball";

err = bt\_set\_name(device\_name);

if (err) {

printk("Failed to set device name (err %d)\n", err);

return;

}

// Set the manufacturer data

uint8\_t manufacturer\_data[22] = { /\* your 22 bytes of data here \*/ };

// Set the advertising parameters

struct bt\_le\_adv\_param adv\_param = {

.id = BT\_ID\_DEFAULT,

.interval\_min = BT\_GAP\_ADV\_FAST\_INT\_MIN\_2,

.interval\_max = BT\_GAP\_ADV\_FAST\_INT\_MAX\_2,

};

// Set the advertising data

struct bt\_data adv\_data[] = {

BT\_DATA(BT\_DATA\_MANUFACTURER\_DATA, manufacturer\_data, sizeof(manufacturer\_data)),

};

// Start advertising

err = bt\_le\_adv\_start(&adv\_param, adv\_data, ARRAY\_SIZE(adv\_data), NULL, 0);

if (err) {

printk("Advertising failed to start (err %d)\n", err);

return;

}

printk("Advertising started\n");

while (1) {

// Your main application logic can go here

k\_sleep(K\_MSEC(200)); // Sleep for 200 milliseconds

}

}