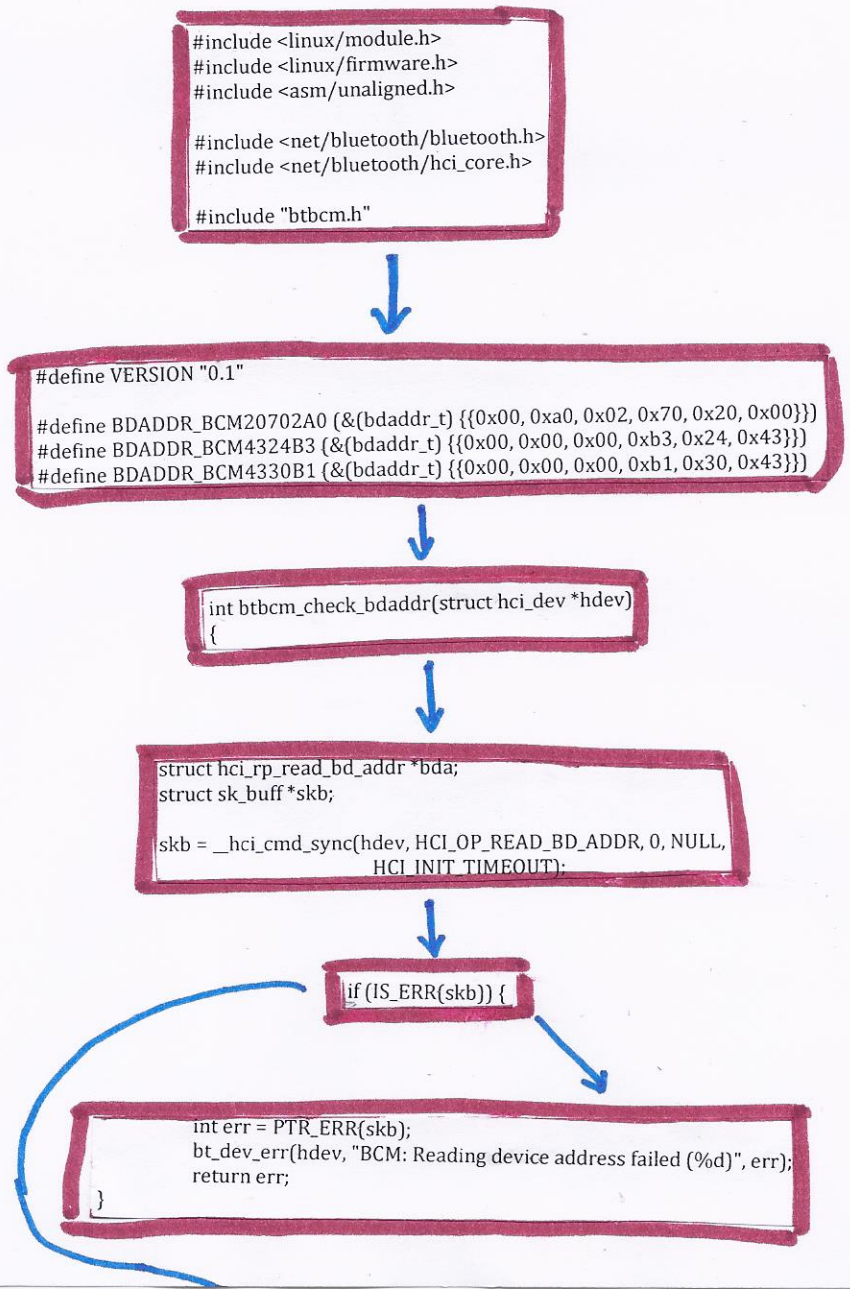
```
/*
 *
 *  Bluetooth support for Broadcom devices
 *
 *  Copyright (C) 2015  Intel Corporation
 *
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 */
```



```
if (skb->len != sizeof(*bda)) {
```

```
bt_dev_err(hdev, "BCM: Device address length mismatch");
kfree_skb(skb);
return -EIO;
}
```

```
bda = (struct hci_rp_read_bd_addr *)skb->data;
```

```
/* Check if the address indicates a controller with either an
 * invalid or default address. In both cases the device needs
 * to be marked as not having a valid address.
 *
 * The address 00:20:70:02:A0:00 indicates a BCM20702A0 controller
 * with no configured address.
 *
 * The address 43:24:B3:00:00:00 indicates a BCM4324B3 controller
 * with waiting for configuration state.
 *
 * The address 43:30:B1:00:00:00 indicates a BCM4330B1 controller
 * with waiting for configuration state.
 */
```

```
if (!bacmp(&bda->bdaddr, BDADDR_BCM20702A0) ||
    !bacmp(&bda->bdaddr, BDADDR_BCM4324B3) ||
    !bacmp(&bda->bdaddr, BDADDR_BCM4330B1)) {
        bt_dev_info(hdev, "BCM: Using default device address (%pMR)",
                    &bda->bdaddr);
        set_bit(HCI_QUIRK_INVALID_BDADDR, &hdev->quirks);
}
```

```
kfree_skb(skb);

return 0;
}
```

```
EXPORT_SYMBOL_GPL(btbcm_check_bdaddr);
```

```
int btbcm_set_bdaddr(struct hci_dev *hdev, const bdaddr_t *bdaddr)
{
```

```
struct sk_buff *skb;
int err;

skb = __hci_cmd_sync(hdev, 0xfc01, 6, bdaddr, HCI_INIT_TIMEOUT);
```

```
if (IS_ERR(skb)) {
```

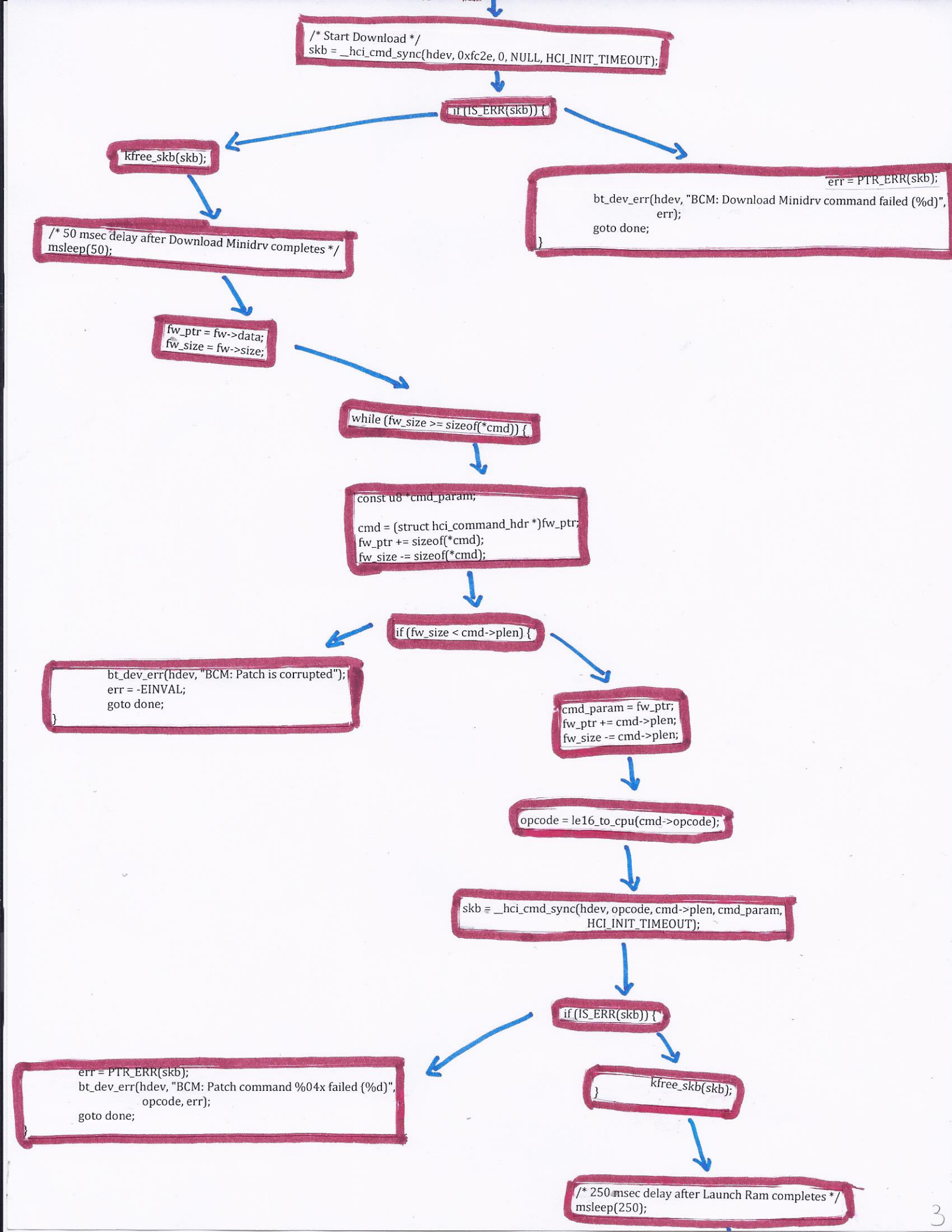
```
err = PTR_ERR(skb);
bt_dev_err(hdev, "BCM: Change address command failed (%d)", err);
return err;
}
```

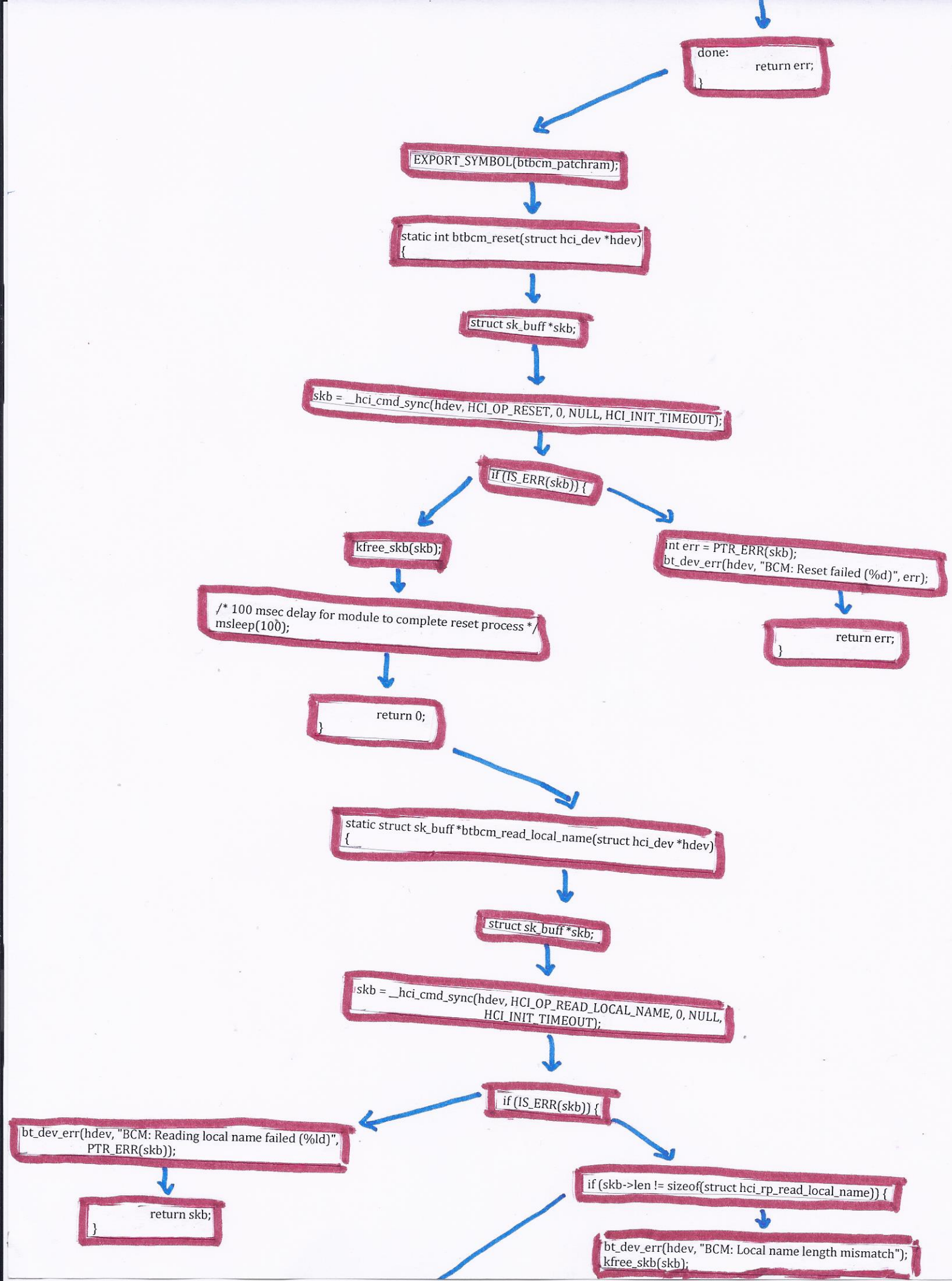
```
kfree_skb(skb);
return 0;
}
```

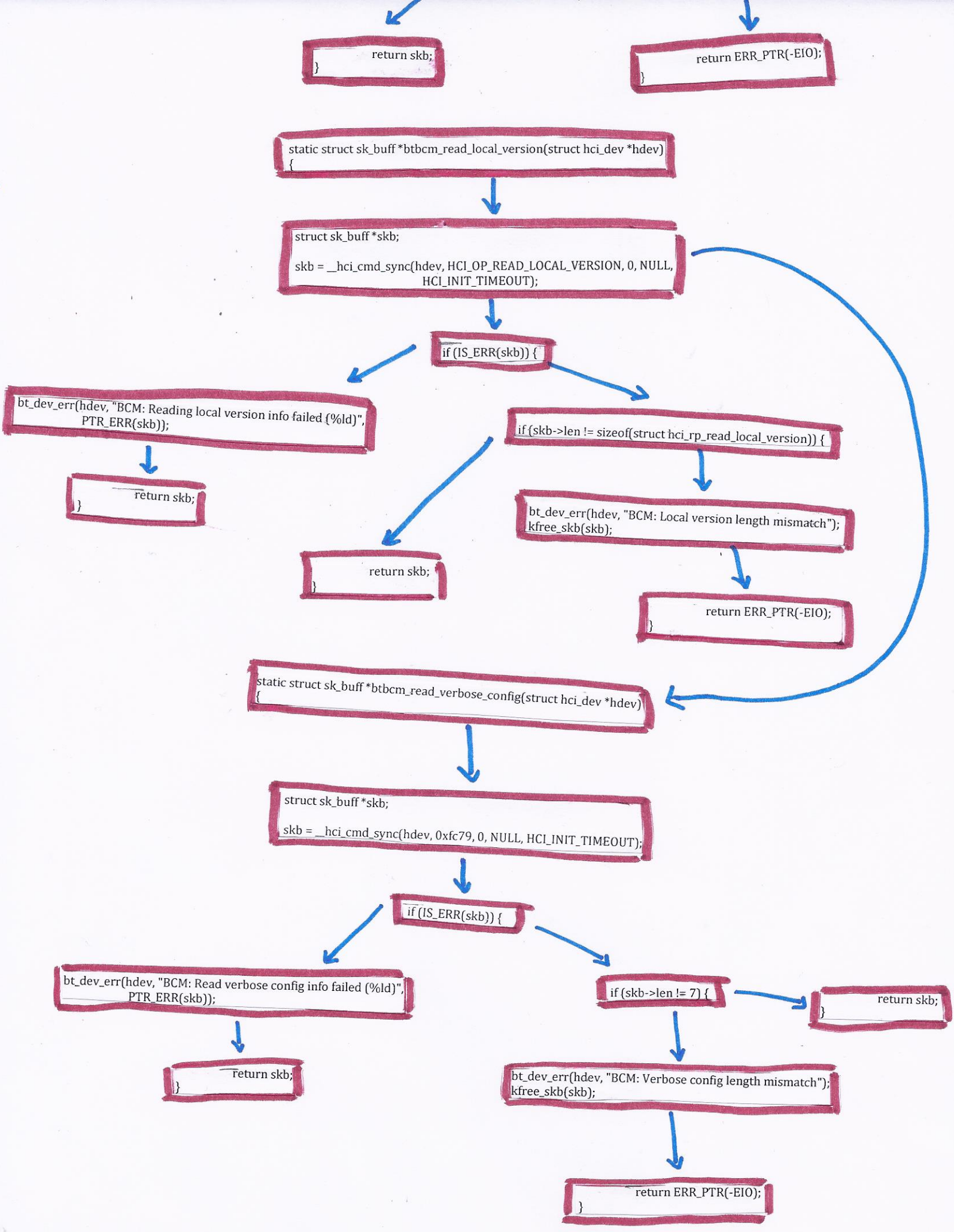
```
EXPORT_SYMBOL_GPL(btbcm_set_bdaddr);
```

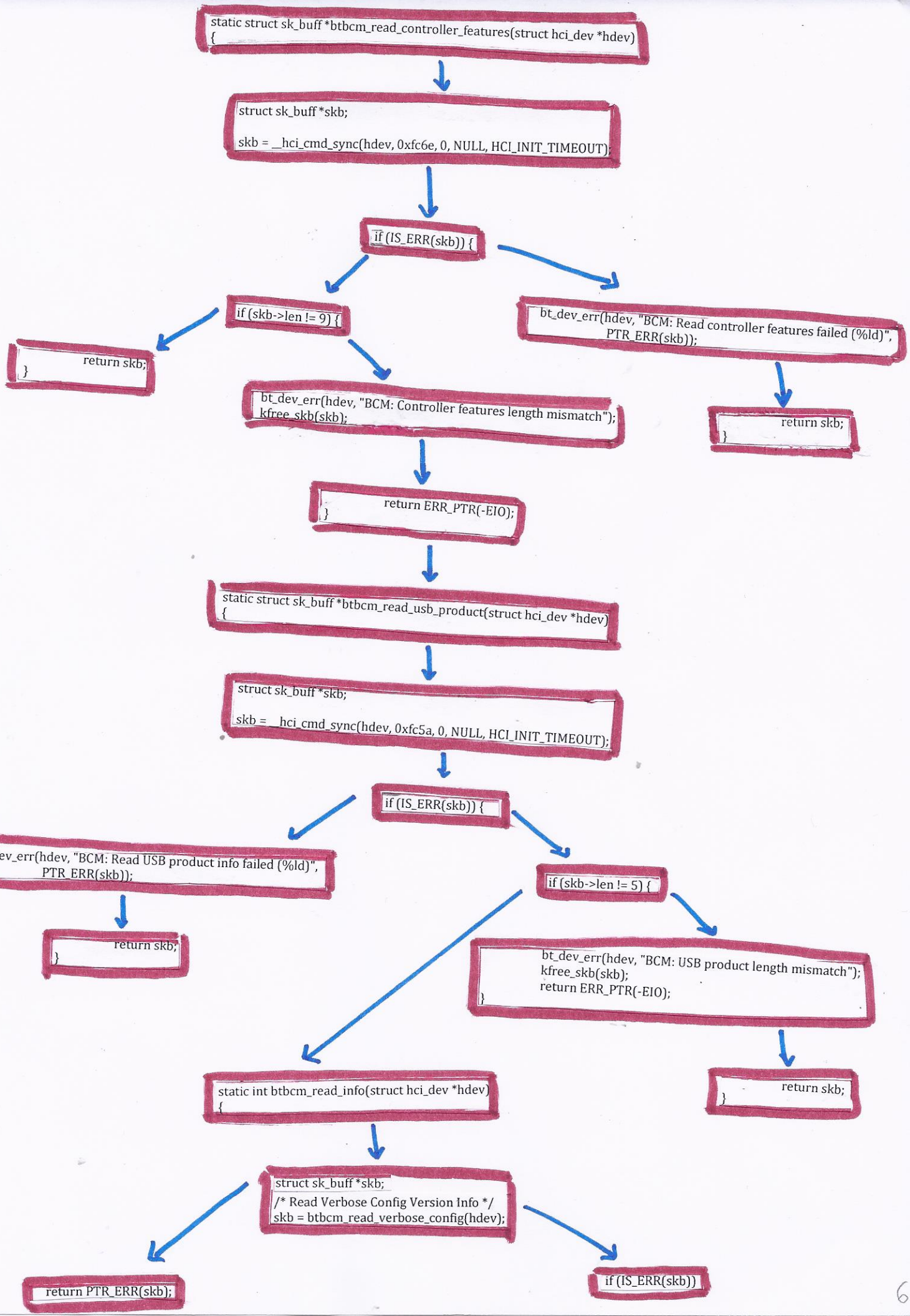
```
int btbcm_patchram(struct hci_dev *hdev, const struct firmware *fw)
{
```

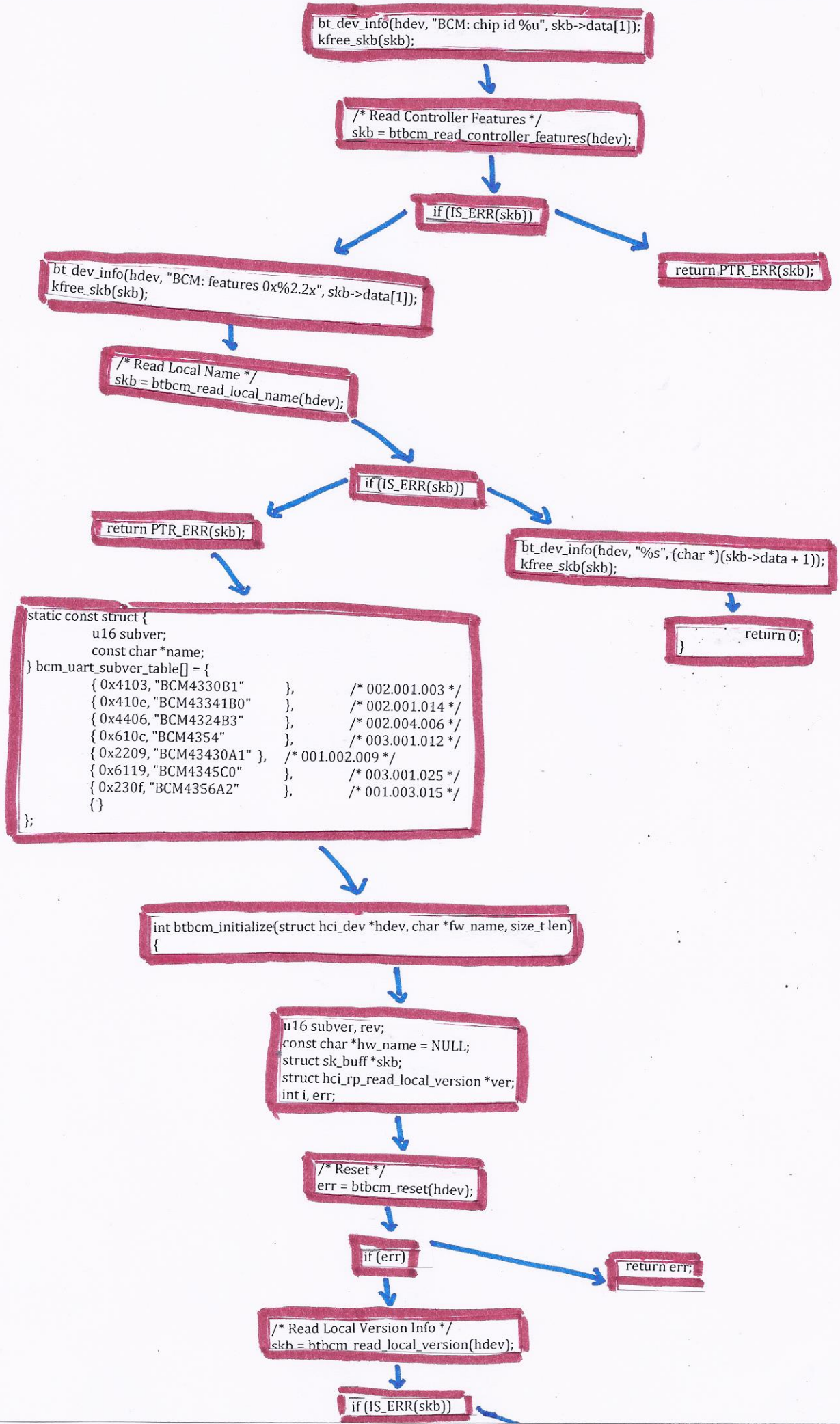
```
const struct hci_command_hdr *cmd;
const u8 *fw_ptr;
size_t fw_size;
struct sk_buff *skb;
u16 opcode;
int err = 0;
```

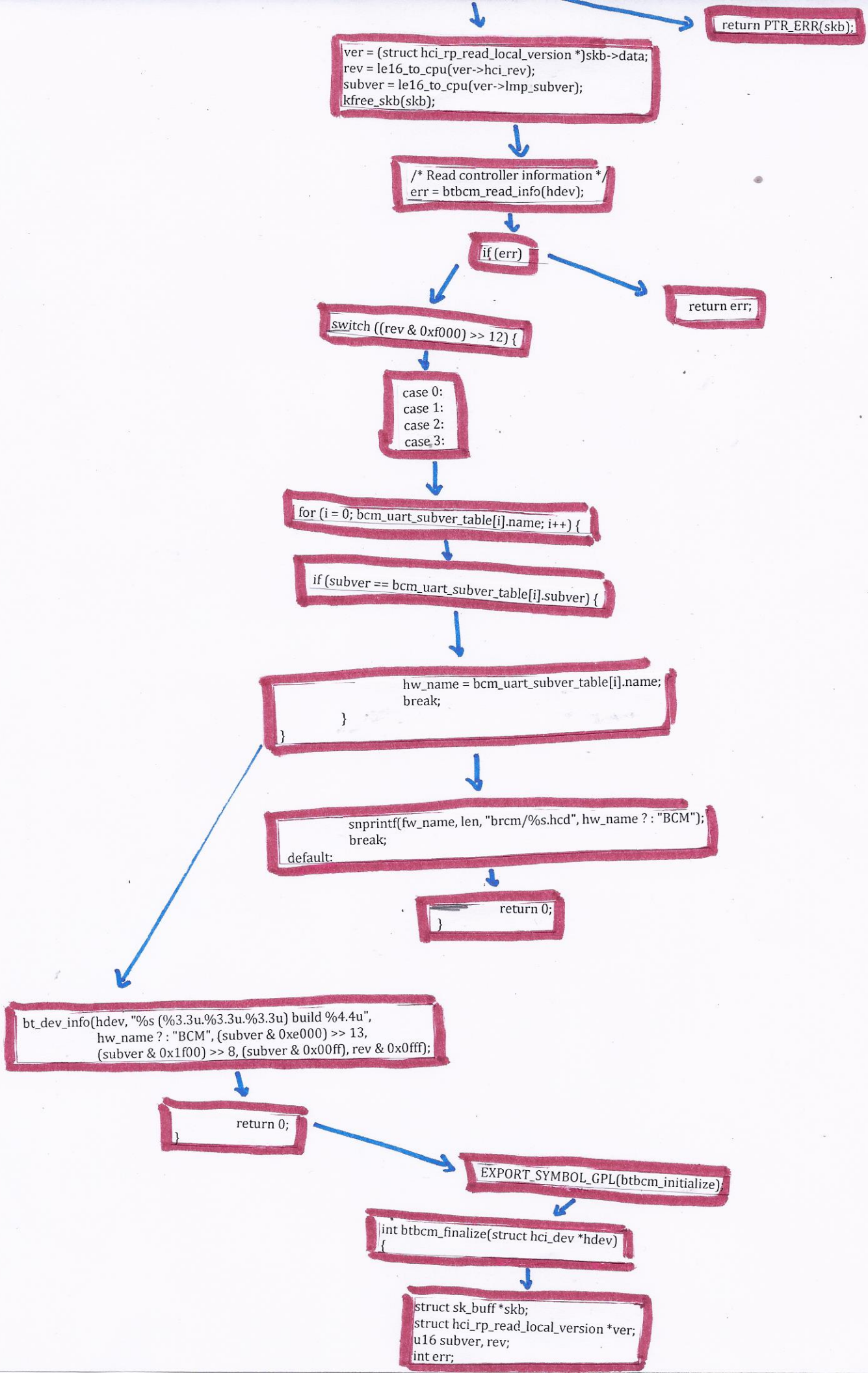

/* Start Download */
skb = __hci_cmd_sync(hdev, 0xfc2e, 0, NULL, HCI_INIT_TIMEOUT);
if (IS_ERR(skb)) {
kfree_skb(skb);
err = PTR_ERR(skb);
bt_dev_err(hdev, "BCM: Download Minidrv command failed (%d)",
err);
goto done;
}
/* 50 msec delay after Download Minidrv completes */
msleep(50);
fw_ptr = fw->data;
fw_size = fw->size;
while (fw_size >= sizeof(*cmd)) {
const u8 *cmd_param;
cmd = (struct hci_command_hdr *)fw_ptr;
fw_ptr += sizeof(*cmd);
fw_size -= sizeof(*cmd);
if (fw_size < cmd->plen) {
bt_dev_err(hdev, "BCM: Patch is corrupted");
err = -EINVAL;
goto done;
}
cmd_param = fw_ptr;
fw_ptr += cmd->plen;
fw_size -= cmd->plen;
opcode = le16_to_cpu(cmd->opcode);
skb = __hci_cmd_sync(hdev, opcode, cmd->plen, cmd_param,
HCI_INIT_TIMEOUT);
if (IS_ERR(skb)) {
err = PTR_ERR(skb);
bt_dev_err(hdev, "BCM: Patch command %04x failed (%d)",
opcode, err);
goto done;
}
kfree_skb(skb);
}
/* 250 msec delay after Launch Ram completes */
msleep(250);

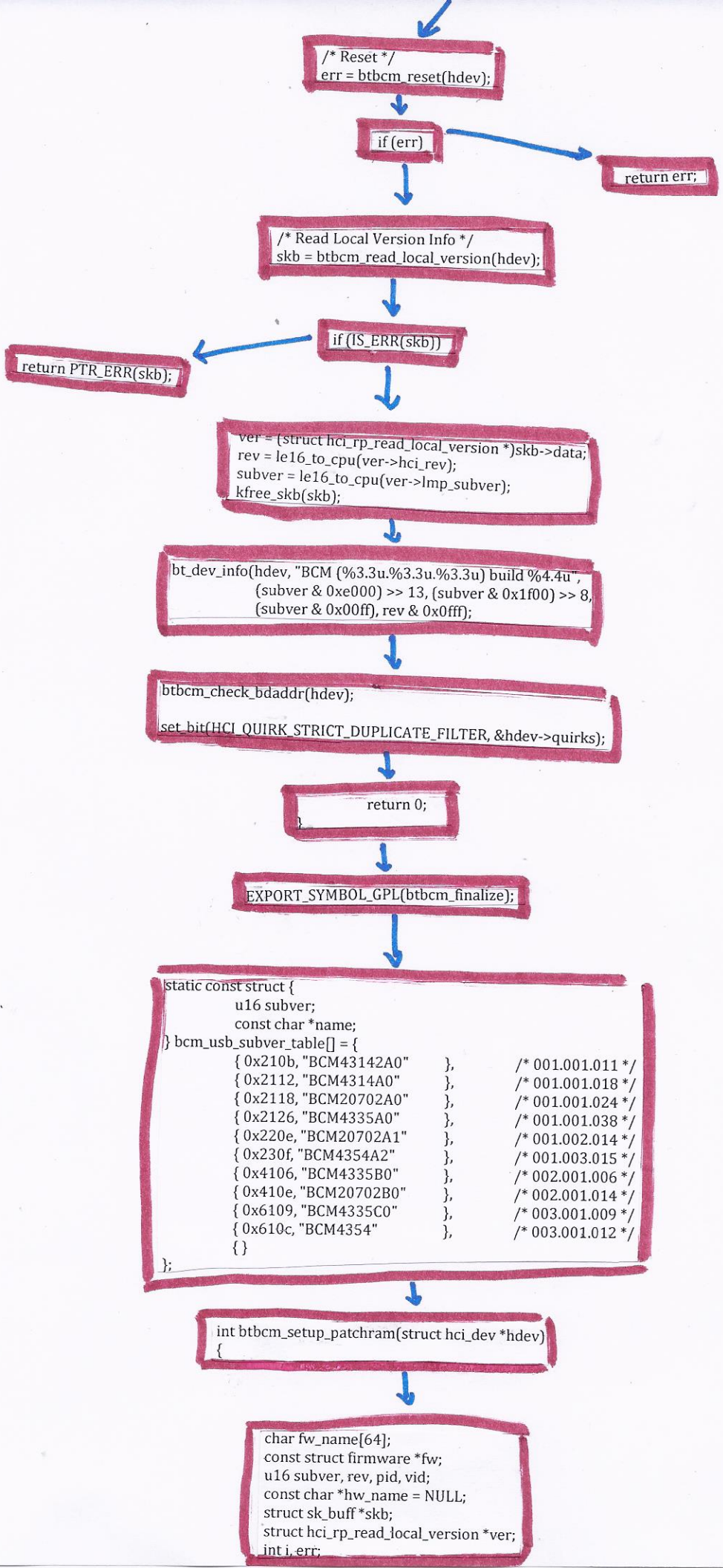

done:
return err;
}
EXPORT_SYMBOL(btbcm_patchram);
static int btbcm_reset(struct hci_dev *hdev)
{
struct sk_buff *skb;
skb = __hci_cmd_sync(hdev, HCI_OP_RESET, 0, NULL, HCI_INIT_TIMEOUT);
if (IS_ERR(skb)) {
kfree_skb(skb);
/* 100 msec delay for module to complete reset process */
msleep(100);
return 0;
}
int err = PTR_ERR(skb);
bt_dev_err(hdev, "BCM: Reset failed (%d)", err);
return err;
}
static struct sk_buff *btbcm_read_local_name(struct hci_dev *hdev)
{
struct sk_buff *skb;
skb = __hci_cmd_sync(hdev, HCI_OP_READ_LOCAL_NAME, 0, NULL,
HCI_INIT_TIMEOUT);
if (IS_ERR(skb)) {
bt_dev_err(hdev, "BCM: Reading local name failed (%ld)",
PTR_ERR(skb));
return skb;
}
if (skb->len != sizeof(struct hci_rp_read_local_name)) {
bt_dev_err(hdev, "BCM: Local name length mismatch");
kfree_skb(skb);

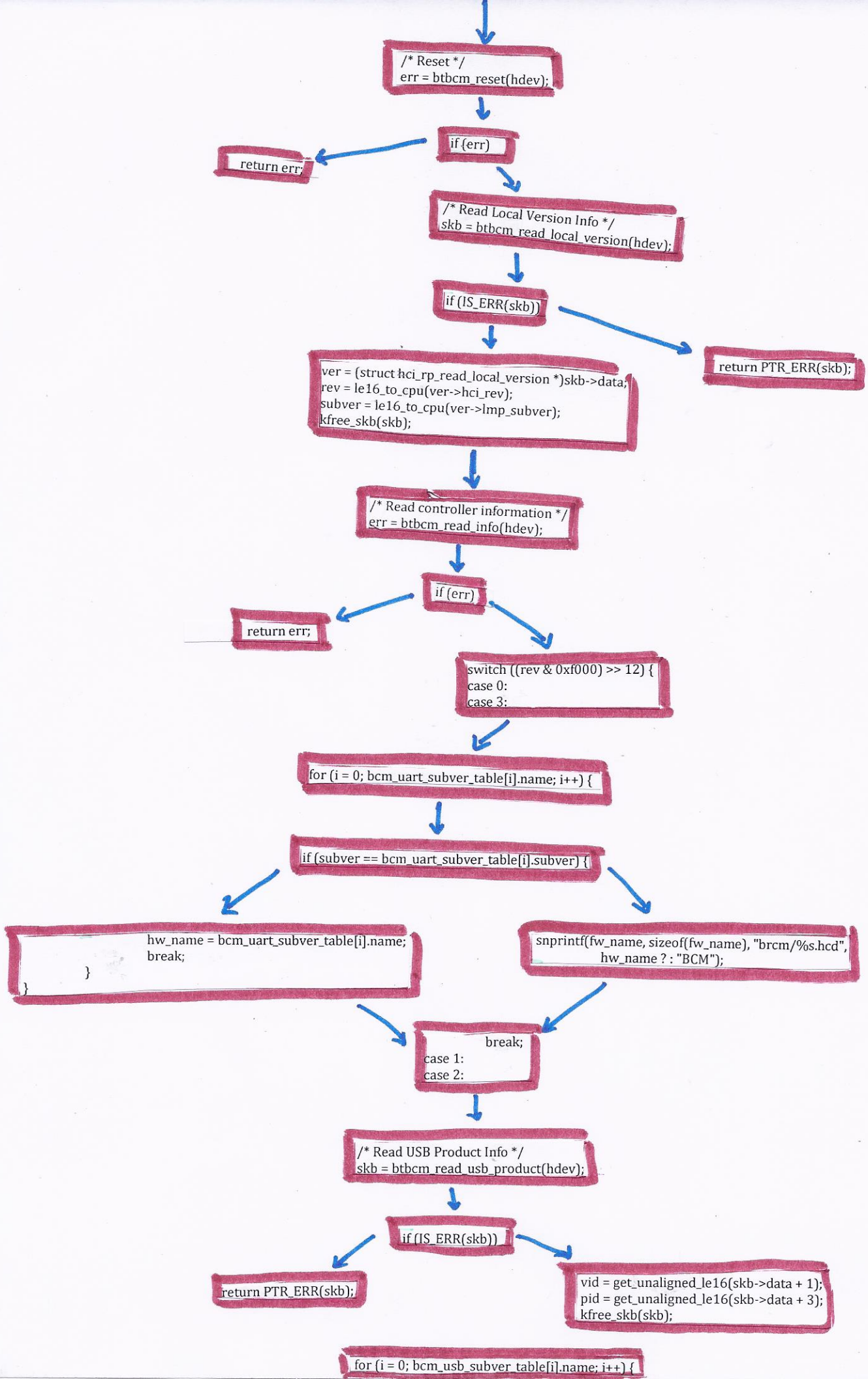

return skb;
}
return ERR_PTR(-EIO);
}
static struct sk_buff *btbcm_read_local_version(struct hci_dev *hdev)
{
struct sk_buff *skb;
skb = __hci_cmd_sync(hdev, HCI_OP_READ_LOCAL_VERSION, 0, NULL, HCI_INIT_TIMEOUT);
if (IS_ERR(skb)) {
bt_dev_err(hdev, "BCM: Reading local version info failed (%ld)", PTR_ERR(skb));
return skb;
}
if (skb->len != sizeof(struct hci_rp_read_local_version)) {
bt_dev_err(hdev, "BCM: Local version length mismatch");
kfree_skb(skb);
return ERR_PTR(-EIO);
}
return skb;
}
static struct sk_buff *btbcm_read_verbose_config(struct hci_dev *hdev)
{
struct sk_buff *skb;
skb = __hci_cmd_sync(hdev, 0xfc79, 0, NULL, HCI_INIT_TIMEOUT);
if (IS_ERR(skb)) {
bt_dev_err(hdev, "BCM: Read verbose config info failed (%ld)", PTR_ERR(skb));
return skb;
}
if (skb->len != 7) {
return skb;
}
bt_dev_err(hdev, "BCM: Verbose config length mismatch");
kfree_skb(skb);
return ERR_PTR(-EIO);
}

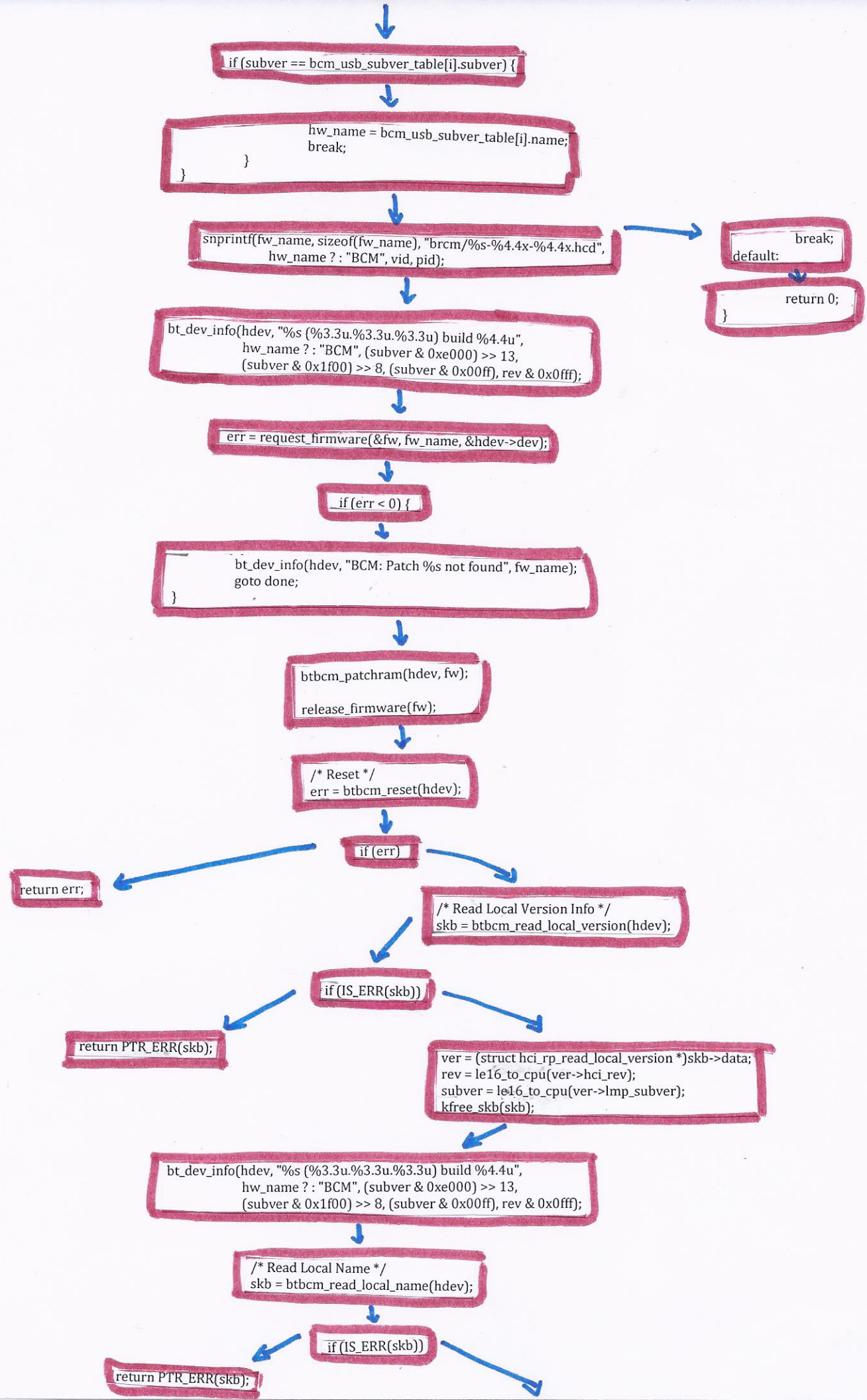

static struct sk_buff *btbcm_read_controller_features(struct hci_dev *hdev)
{
struct sk_buff *skb;
skb = __hci_cmd_sync(hdev, 0xfc6e, 0, NULL, HCI_INIT_TIMEOUT);
if (IS_ERR(skb)) {
bt_dev_err(hdev, "BCM: Read controller features failed (%ld)", PTR_ERR(skb));
return skb;
}
if (skb->len != 9) {
return skb;
}
bt_dev_err(hdev, "BCM: Controller features length mismatch");
kfree_skb(skb);
return ERR_PTR(-EIO);
}
static struct sk_buff *btbcm_read_usb_product(struct hci_dev *hdev)
{
struct sk_buff *skb;
skb = __hci_cmd_sync(hdev, 0xfc5a, 0, NULL, HCI_INIT_TIMEOUT);
if (IS_ERR(skb)) {
bt_dev_err(hdev, "BCM: Read USB product info failed (%ld)", PTR_ERR(skb));
return skb;
}
if (skb->len != 5) {
bt_dev_err(hdev, "BCM: USB product length mismatch");
kfree_skb(skb);
return ERR_PTR(-EIO);
}
return skb;
static int btbcm_read_info(struct hci_dev *hdev)
{
struct sk_buff *skb;
/* Read Verbose Config Version Info */
skb = btbcm_read_verbose_config(hdev);
return PTR_ERR(skb);
if (IS_ERR(skb))

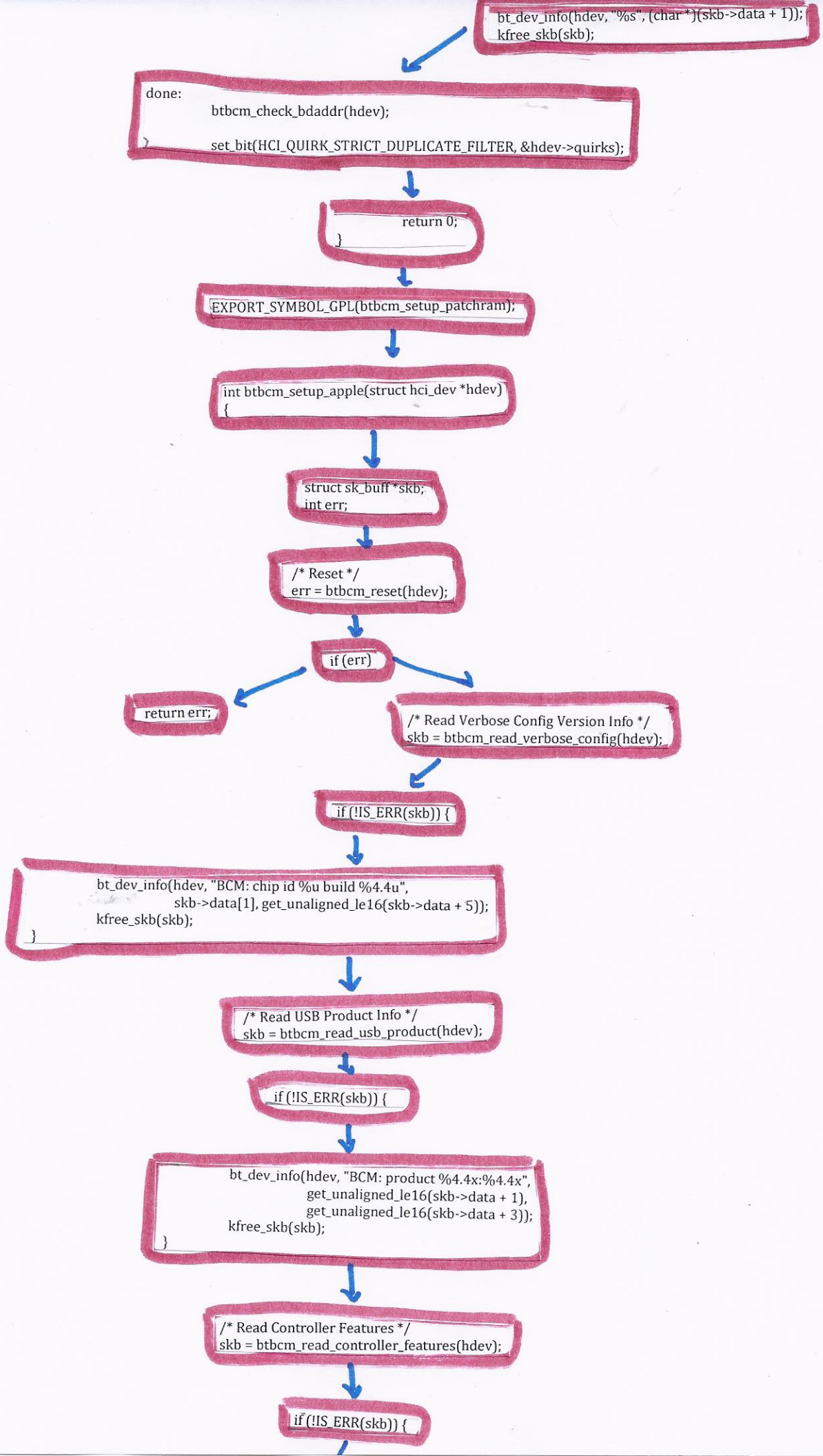

bt_dev_info(hdev, "BCM: chip id %u", skb->data[1]);
kfree_skb(skb);
/* Read Controller Features */
skb = btbcm_read_controller_features(hdev);
if (IS_ERR(skb))
return PTR_ERR(skb);
bt_dev_info(hdev, "BCM: features 0x%2.2x", skb->data[1]);
kfree_skb(skb);
/* Read Local Name */
skb = btbcm_read_local_name(hdev);
if (IS_ERR(skb))
return PTR_ERR(skb);
bt_dev_info(hdev, "%s", (char *)(skb->data + 1));
kfree_skb(skb);
return 0;
}
static const struct {
u16 subver;
const char *name;
} bcm_uart_subver_table[] = {
{ 0x4103, "BCM4330B1" }, /* 002.001.003 */
{ 0x410e, "BCM43341B0" }, /* 002.001.014 */
{ 0x4406, "BCM4324B3" }, /* 002.004.006 */
{ 0x610c, "BCM4354" }, /* 003.001.012 */
{ 0x2209, "BCM43430A1" }, /* 001.002.009 */
{ 0x6119, "BCM4345C0" }, /* 003.001.025 */
{ 0x230f, "BCM4356A2" }, /* 001.003.015 */
{ }
};
int btbcm_initialize(struct hci_dev *hdev, char *fw_name, size_t len)
{
u16 subver, rev;
const char *hw_name = NULL;
struct sk_buff *skb;
struct hci_rp_read_local_version *ver;
int i, err;
/* Reset */
err = btbcm_reset(hdev);
if (err)
return err;
/* Read Local Version Info */
skb = btbcm_read_local_version(hdev);
if (IS_ERR(skb))

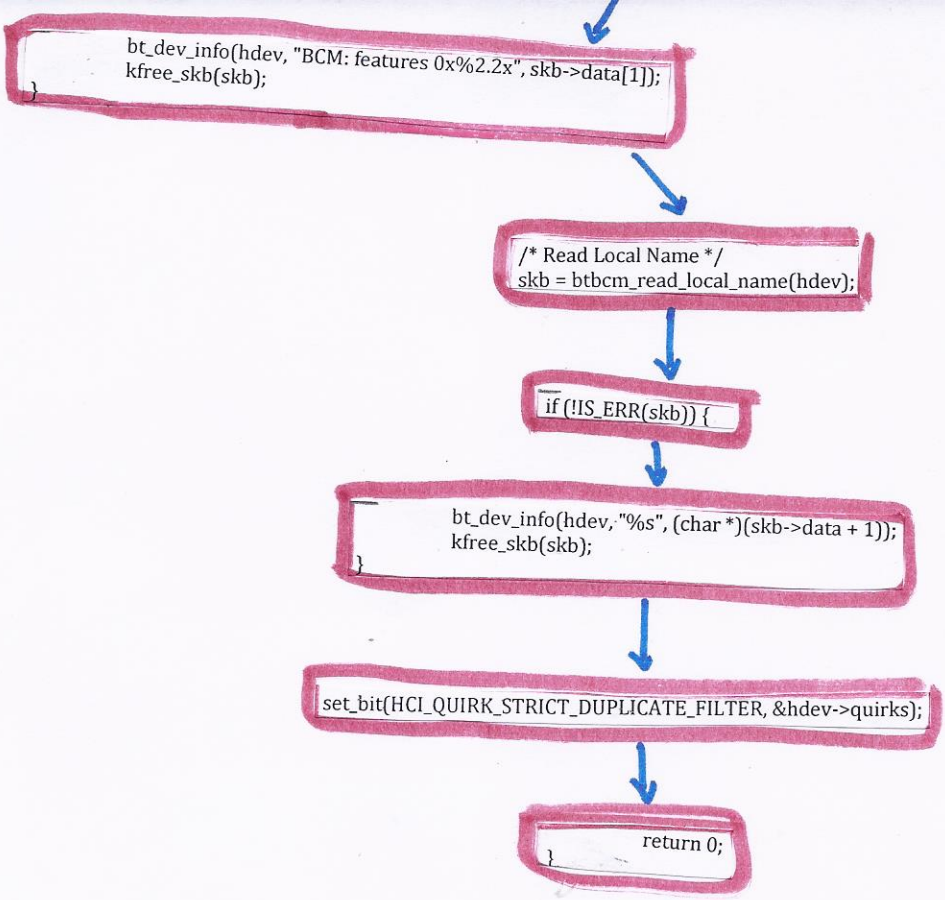

return PTR_ERR(skb);
ver = (struct hci_rp_read_local_version *)skb->data;
rev = le16_to_cpu(ver->hci_rev);
subver = le16_to_cpu(ver->lmp_subver);
kfree_skb(skb);
/* Read controller information */
err = btbcm_read_info(hdev);
if (err)
return err;
switch ((rev & 0xf000) >> 12) {
case 0:
case 1:
case 2:
case 3:
for (i = 0; bcm_uart_subver_table[i].name; i++) {
if (subver == bcm_uart_subver_table[i].subver) {
hw_name = bcm_uart_subver_table[i].name;
break;
}
}
snprintf(fw_name, len, "brcm/%s.hcd", hw_name ? : "BCM");
break;
default:
return 0;
}
bt_dev_info(hdev, "%s (%3.3u.%3.3u.%3.3u) build %4.4u",
hw_name ? : "BCM", (subver & 0xe000) >> 13,
(subver & 0x1f00) >> 8, (subver & 0x00ff), rev & 0x0fff);
return 0;
}
EXPORT_SYMBOL_GPL(btbcm_initialize);
int btbcm_finalize(struct hci_dev *hdev)
{
struct sk_buff *skb;
struct hci_rp_read_local_version *ver;
u16 subver, rev;
int err;


/* Reset */
err = btbcm_reset(hdev);
if (err)
return err;
/* Read Local Version Info */
skb = btbcm_read_local_version(hdev);
if (IS_ERR(skb))
return PTR_ERR(skb);
ver = (struct hci_rp_read_local_version *)skb->data;
rev = le16_to_cpu(ver->hci_rev);
subver = le16_to_cpu(ver->lmp_subver);
kfree_skb(skb);
bt_dev_info(hdev, "BCM (%3.3u.%3.3u.%3.3u) build %4.4u",
(subver & 0xe000) >> 13, (subver & 0x1f00) >> 8,
(subver & 0x00ff), rev & 0x0fff);
btbcm_check_bdaddr(hdev);
set_bit(HCI_QUIRK_STRICT_DUPLICATE_FILTER, &hdev->quirks);
return 0;
}
EXPORT_SYMBOL_GPL(btbcm_finalize);
static const struct {
u16 subver;
const char *name;
} bcm_usb_subver_table[] = {
{ 0x210b, "BCM43142A0" }, /* 001.001.011 */
{ 0x2112, "BCM4314A0" }, /* 001.001.018 */
{ 0x2118, "BCM20702A0" }, /* 001.001.024 */
{ 0x2126, "BCM4335A0" }, /* 001.001.038 */
{ 0x220e, "BCM20702A1" }, /* 001.002.014 */
{ 0x230f, "BCM4354A2" }, /* 001.003.015 */
{ 0x4106, "BCM4335B0" }, /* 002.001.006 */
{ 0x410e, "BCM20702B0" }, /* 002.001.014 */
{ 0x6109, "BCM4335C0" }, /* 003.001.009 */
{ 0x610c, "BCM4354" }, /* 003.001.012 */
{ }
};
int btbcm_setup_patchram(struct hci_dev *hdev)
{
char fw_name[64];
const struct firmware *fw;
u16 subver, rev, pid, vid;
const char *hw_name = NULL;
struct sk_buff *skb;
struct hci_rp_read_local_version *ver;
int i, err;


/* Reset */
err = btbcm_reset(hdev);
if (err)
return err;
/* Read Local Version Info */
skb = btbcm_read_local_version(hdev);
if (IS_ERR(skb))
return PTR_ERR(skb);
ver = (struct hci_rp_read_local_version *)skb->data;
rev = le16_to_cpu(ver->hci_rev);
subver = le16_to_cpu(ver->lmp_subver);
kfree_skb(skb);
/* Read controller information */
err = btbcm_read_info(hdev);
if (err)
return err;
switch ((rev & 0xf000) >> 12) {
case 0:
case 3:
for (i = 0; bcm_uart_subver_table[i].name; i++) {
if (subver == bcm_uart_subver_table[i].subver) {
hw_name = bcm_uart_subver_table[i].name;
break;
}
}
snprintf(fw_name, sizeof(fw_name), "brcm/%s.hcd",
hw_name ? : "BCM");
break;
case 1:
case 2:
/* Read USB Product Info */
skb = btbcm_read_usb_product(hdev);
if (IS_ERR(skb))
return PTR_ERR(skb);
vid = get_unaligned_le16(skb->data + 1);
pid = get_unaligned_le16(skb->data + 3);
kfree_skb(skb);
for (i = 0; bcm_usb_subver_table[i].name; i++) {

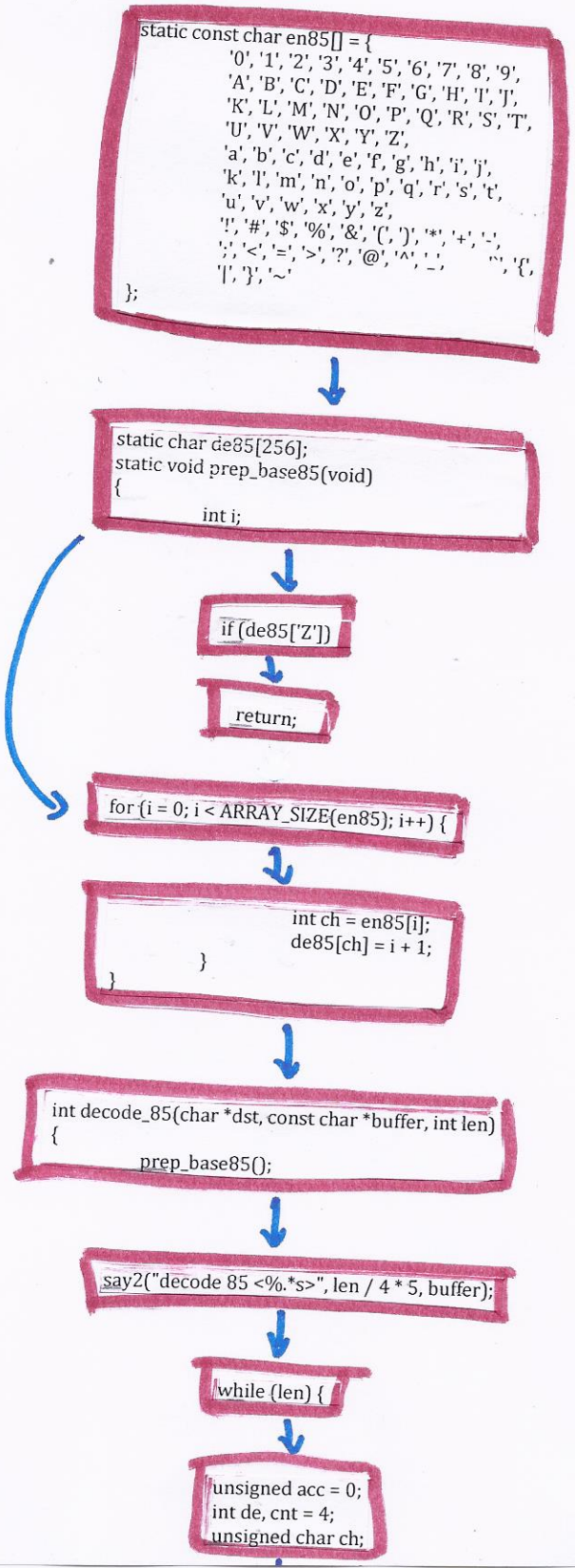

if (subver == bcm_usb_subver_table[i].subver) {
hw_name = bcm_usb_subver_table[i].name;
break;
}
}
snprintf(fw_name, sizeof(fw_name), "brcm/%s-%4.4x-%4.4x.hcd",
hw_name ? : "BCM", vid, pid);
break;
default:
return 0;
}
bt_dev_info(hdev, "%s (%3.3u.%3.3u.%3.3u) build %4.4u",
hw_name ? : "BCM", (subver & 0xe000) >> 13,
(subver & 0x1f00) >> 8, (subver & 0x00ff), rev & 0x0fff);
err = request_firmware(&fw, fw_name, &hdev->dev);
if (err < 0) {
bt_dev_info(hdev, "BCM: Patch %s not found", fw_name);
goto done;
}
btbcm_patchram(hdev, fw);
release_firmware(fw);
/* Reset */
err = btbcm_reset(hdev);
if (err)
return err;
/* Read Local Version Info */
skb = btbcm_read_local_version(hdev);
if (IS_ERR(skb))
return PTR_ERR(skb);
ver = (struct hci_rp_read_local_version *)skb->data;
rev = le16_to_cpu(ver->hci_rev);
subver = le16_to_cpu(ver->lmp_subver);
kfree_skb(skb);
bt_dev_info(hdev, "%s (%3.3u.%3.3u.%3.3u) build %4.4u",
hw_name ? : "BCM", (subver & 0xe000) >> 13,
(subver & 0x1f00) >> 8, (subver & 0x00ff), rev & 0x0fff);
/* Read Local Name */
skb = btbcm_read_local_name(hdev);
if (IS_ERR(skb))
return PTR_ERR(skb);

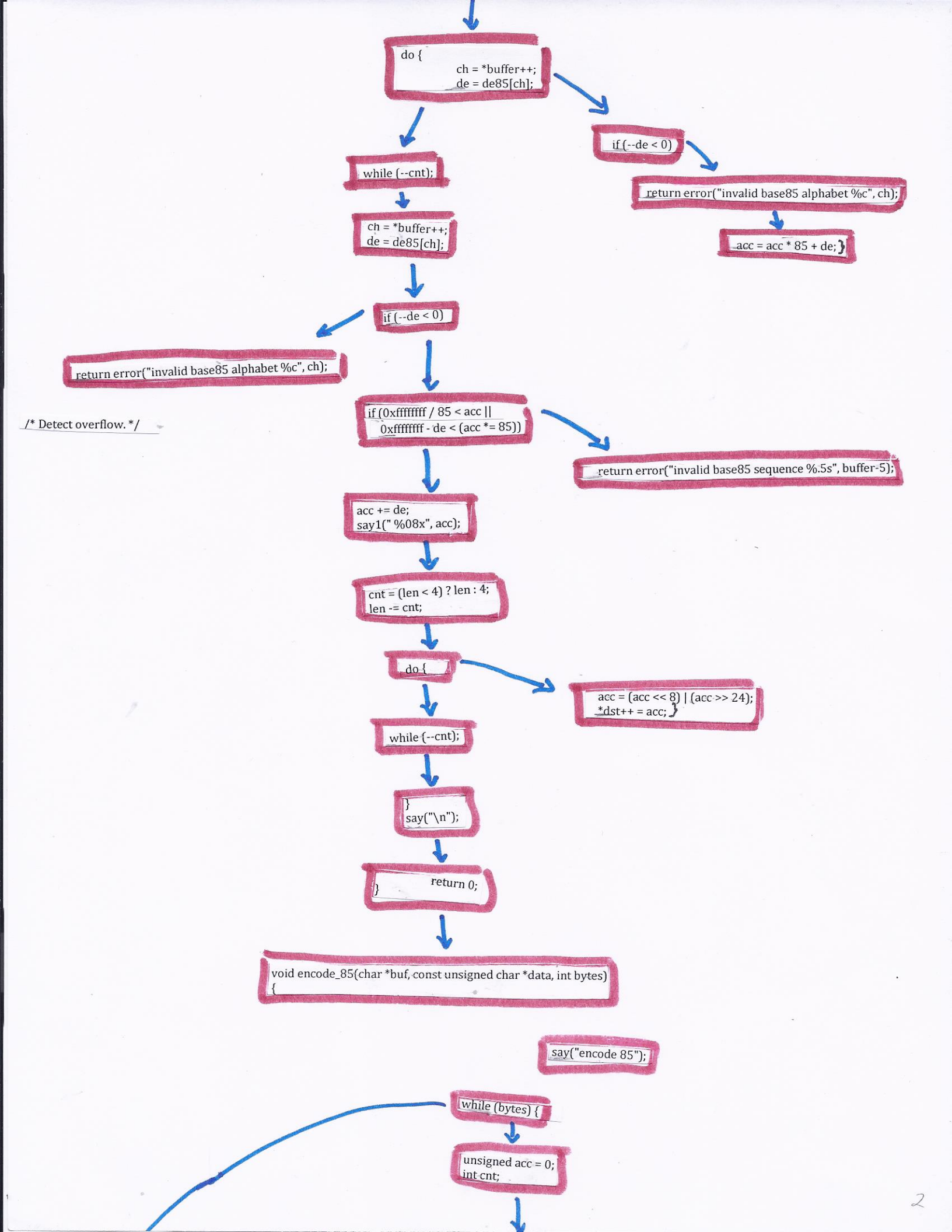

bt_dev_info(hdev, "%s", (char *)(skb->data + 1));
kfree_skb(skb);
done:
btbcm_check_bdaddr(hdev);
}
set_bit(HCI_QUIRK_STRICT_DUPLICATE_FILTER, &hdev->quirks);
return 0;
}
EXPORT_SYMBOL_GPL(btbcm_setup_patchram);
int btbcm_setup_apple(struct hci_dev *hdev)
{
struct sk_buff *skb;
int err;
/* Reset */
err = btbcm_reset(hdev);
if (err)
return err;
/* Read Verbose Config Version Info */
skb = btbcm_read_verbose_config(hdev);
if (!IS_ERR(skb)) {
bt_dev_info(hdev, "BCM: chip id %u build %4.4u",
skb->data[1], get_unaligned_le16(skb->data + 5));
kfree_skb(skb);
}
/* Read USB Product Info */
skb = btbcm_read_usb_product(hdev);
if (!IS_ERR(skb)) {
bt_dev_info(hdev, "BCM: product %4.4x:%4.4x",
get_unaligned_le16(skb->data + 1),
get_unaligned_le16(skb->data + 3));
kfree_skb(skb);
}
/* Read Controller Features */
skb = btbcm_read_controller_features(hdev);
if (!IS_ERR(skb)) {

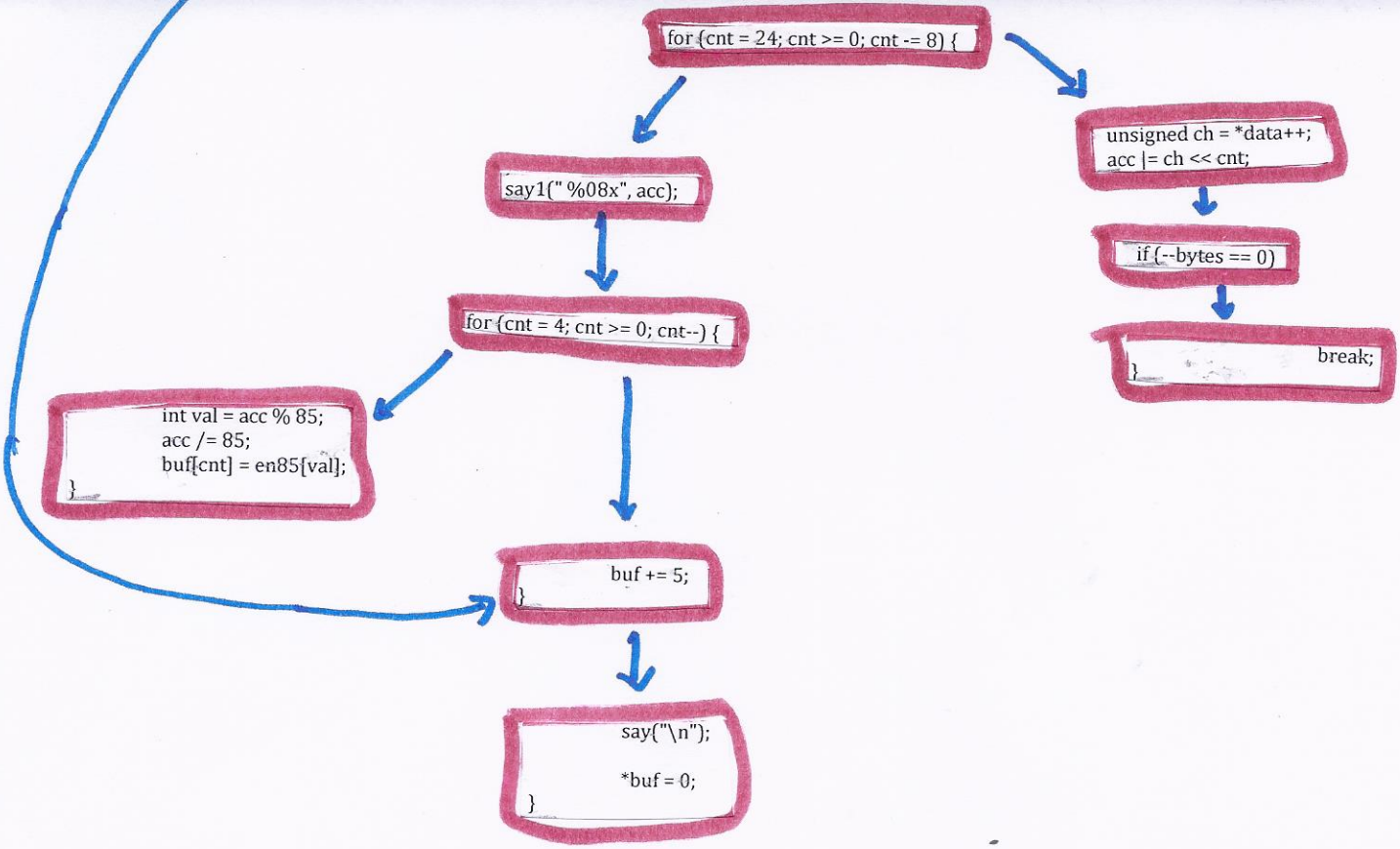



```
EXPORT_SYMBOL_GPL(btbcm_setup_apple);

MODULE_AUTHOR("Marcel Holtmann <marcel@holtmann.org>");
MODULE_DESCRIPTION("Bluetooth support for Broadcom devices ver " VERSION);
MODULE_VERSION(VERSION);
MODULE_LICENSE("GPL");
```

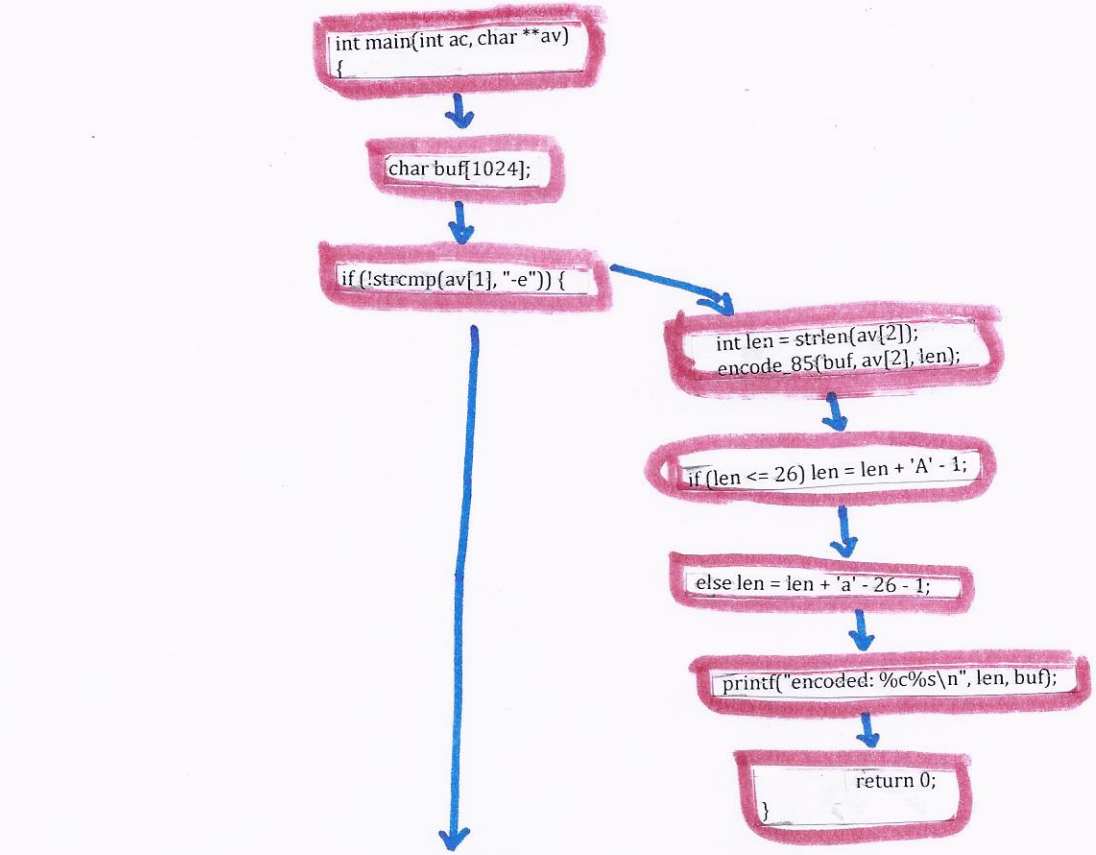
```
#include "cache.h"
#undef DEBUG_85

#ifdef DEBUG_85
#define say(a) fprintf(stderr, a)
#define say1(a,b) fprintf(stderr, a, b)
#define say2(a,b,c) fprintf(stderr, a, b, c)
#else
#define say(a) do { /* nothing */ } while (0)
#define say1(a,b) do { /* nothing */ } while (0)
#define say2(a,b,c) do { /* nothing */ } while (0)
#endif
```

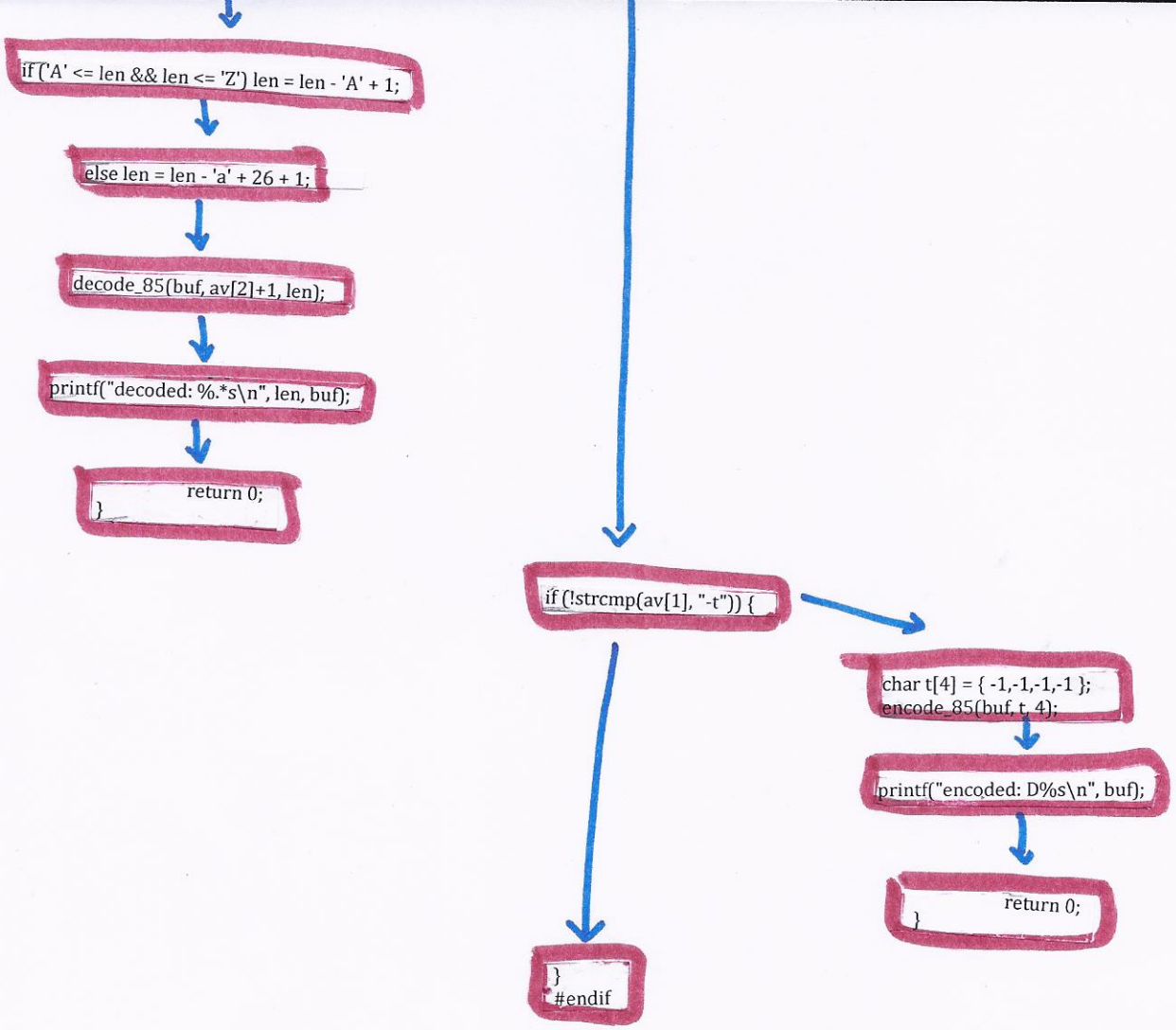



do {
ch = *buffer++;
de = de85[ch];
if (--de < 0)
return error("invalid base85 alphabet %c", ch);
acc = acc * 85 + de; }
while (--cnt);
ch = *buffer++;
de = de85[ch];
if (--de < 0)
return error("invalid base85 alphabet %c", ch);
/* Detect overflow. */
if (0xffffffff / 85 < acc ||
0xffffffff - de < (acc *= 85))
return error("invalid base85 sequence %.5s", buffer-5);
acc += de;
say1(" %08x", acc);
cnt = (len < 4) ? len : 4;
len -= cnt;
do {
acc = (acc << 8) | (acc >> 24);
*dst++ = acc; }
while (--cnt);
}
say("\n");
}
return 0;
void encode_85(char *buf, const unsigned char *data, int bytes)
{
say("encode 85");
while (bytes) {
unsigned acc = 0;
int cnt;



#ifdef DEBUG_85




if ('A' <= len && len <= 'Z') len = len - 'A' + 1;
else len = len - 'a' + 26 + 1;
decode_85(buf, av[2]+1, len);
printf("decoded: %.*s\n", len, buf);
return 0;
}
if (!strcmp(av[1], "-t")) {
char t[4] = { -1,-1,-1,-1 };
encode_85(buf, t, 4);
printf("encoded: D%s\n", buf);
return 0;
}
}
#endif