My First Contribution to Linux

18 minute read | 2025-10-05 | programming, linux, foss

I've been spending more of my spare time in recent years studying the Linux source tree to try to build a deeper understanding of how computers work. As a result, I've started accumulating patches that fix issues with hardware I own. I decided to try upstreaming one of these patches to familiarize myself with the kernel development process.

▶ Table of Contents

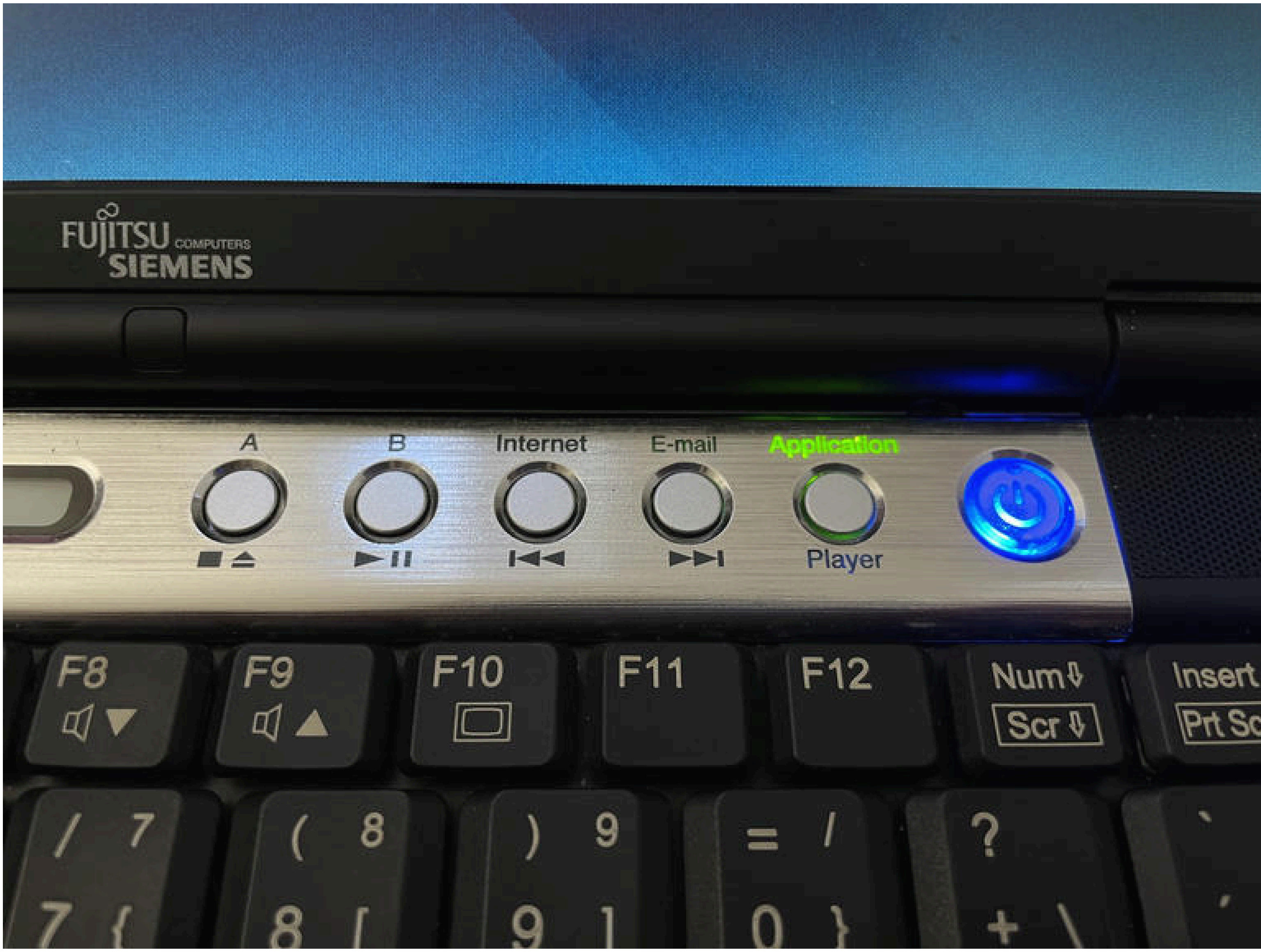
Context

I have an old laptop I'm particularly fond of: a 2005 Fujitsu Lifebook S2110. It's probably the oldest computer I have that could still be considered "modern", primarily because of its 64-bit CPU.

The laptop pictured in December 2008. I was 12 :]

Despite being 20 years old now, it still happily runs the latest Arch rolling release with only 2GB of RAM and a spinning hard disk. Once the page cache warms up a bit, it's plenty fast enough for light C programming. The keyboard is *very* comfortable, and I really like how crisp bitmap fonts look on the glossy 1024x768 display.[1]

Like many laptops from this era, it has a row of hotkeys above its keyboard that perform various functions:



The keys are very 2005 :]

That key on the right labelled `Application` and `Player` is a hardware toggle that selects between the two "modes". Pushing it toggles which of the two labels is lit.

To be honest, I've never really used these keys myself, but, continuing my kernel deep dive, I wanted to see how special keys like this are handled in Linux.

How do the keys work?

First of all, do these keys even work to begin with? Pressing them under `i3(1)` doesn't seem to do anything in either mode. Perhaps the events are being generated, but aren't bound to anything by default? I looked up how to show raw input events on X11, and it seems like `xev(1)` is the right tool for the job. With it running, a press of the leftmost `A` hotkey yields:

```text
KeyPress event, serial 47, synthetic NO, window 0xc00001,
    root 0x18b, subw 0x0, time 5481538, (287,414), root:(803,434),
    state 0x0, keycode 156 (keysym 0x1008ff41, XF86Launch1), same_screen YES,
    XLookupString gives 0 bytes:
    XmbLookupString gives 0 bytes:
    XFilterEvent returns: False

KeyRelease event, serial 47, synthetic NO, window 0xc00001,
    root 0x18b, subw 0x0, time 5481642, (287,414), root:(803,434),
    state 0x0, keycode 156 (keysym 0x1008ff41, XF86Launch1), same_screen YES,
    XLookupString gives 0 bytes:
    XFilterEvent returns: False
```

Similar `KeyPress` and `KeyRelease` events are fired for all the keys in `Application` mode, mapped like this:

- `A` : `XF86Launch1`
- `B` : `XF86Launch2`
- `Internet` : `XF86Launch3`
- `E-mail` : `XF86Launch4`

I can then bind these key events to arbitrary commands in my `i3(1)` config:

```text
bindsym XF86Launch3 exec --no-startup-id firefox
bindsym XF86Launch1 exec i3-sensible-terminal
```

After reloading the config, pushing `A` now fires up a terminal, and the `Internet` key now opens up Firefox. Neat.

However, if I switch to `Player` mode, no events are fired for any of the keys. That doesn't seem right. A further clue that something isn't working properly can be seen in the kernel log:

```text
Mar 25 19:59:02 s2110 kernel: ACPI: \_SB_.FEXT: Unknown GIRB result [40000414]
Mar 25 19:59:04 s2110 kernel: ACPI: \_SB_.FEXT: Unknown GIRB result [40000415]
```

```
Mar 25 19:59:06 s2110 kernel: ACPI: \_SB_.FEXT: Unknown GIRB result [40000416]
Mar 25 19:59:07 s2110 kernel: ACPI: \_SB_.FEXT: Unknown GIRB result [40000417]
```

A line like this appears any time I push one of the keys while in `Player` mode, which suggests that this may be a driver problem.

Finding the correct driver

To figure out out what's going on here, I first need to find which driver is handling these key events, and get at least a rough idea of how it does that. This first step of finding the right spot in the *massive* kernel source tree can be a bit tricky. Luckily in this case, we have those messages in the kernel log. Grepping the source tree for error messages or other bits of text the kernel exposes is by far the *fastest* way to narrow down this search.

But first, let's check out a more broadly applicable method. Since *most* things in the kernel happen silently, there is a good chance we have no strings to go off of, and grepping educated guesses tends to yield a fairly poor signal/noise ratio in a massive codebase like this. Instead, we can instruct the running kernel to show us which drivers are currently in use, and we might be able to deduce which one we need to look at based on the name. Most driver code on Linux lives in **kernel modules** that are loaded on-demand from disk when a device is plugged in. We can list the modules that are currently loaded on a system with `lsmod(8)`:

TXT

```
$ lsmod
Module                  Size  Used by
8021q                  53248  0
... SNIP ...
i2c_smbus              20480  1 i2c_piix4
fujitsu_laptop         32768  0
sparse_keymap          12288  1 fujitsu_laptop
mac_hid                12288  0
... SNIP ...
mmc_core              290816  5 sdhci_uhs2,sdhci,ssb,cqhci,sdhci_pci
video                  81920  3 fujitsu_laptop,amdgpu,radeon
```

Some of these module names are less obvious, but I'd bet we'll find the code for these hotkeys in that `fujitsu_laptop` module. In this case, we can easily check if it's the right place by just grepping for that error message from before. Moving to a kernel source checkout I prepared earlier[2]:

```txt
$ cd linux
$ rg 'Unknown GIRB result'
drivers/platform/x86/fujitsu-laptop.c
1036:                                                   "Unknown GIRB result [%x]\n", irb);
```

Okay, that's good confirmation that this is indeed the driver we need to look at.

Studying `fujitsu-laptop` driver

This `fujitsu-laptop.c` file has ~1k lines of C, and it handles many other things in addition to these hotkeys, so it's important we stay on this 'golden path' as we explore.[3] I'll omit most unrelated code in this next part, but you can find the code I'm referencing here, if you want to follow along.

I've found that a good place to start studying driver code is often the module initialization boilerplate, which is usually at the very end of the file:

```c
static int __init fujitsu_init(void)
{
	int ret;

	ret = acpi_bus_register_driver(&acpi_fujitsu_bl_driver);
	if (ret)
		return ret;

	/* Register platform stuff */

	ret = platform_driver_register(&fujitsu_pf_driver);
	if (ret)
		goto err_unregister_acpi;

	/* Register laptop driver */

	ret = acpi_bus_register_driver(&acpi_fujitsu_laptop_driver);
	if (ret)
		goto err_unregister_platform_driver;
```

```c
        pr_info("driver " FUJITSU_DRIVER_VERSION " successfully loaded\n");

        return 0;

err_unregister_platform_driver:
        platform_driver_unregister(&fujitsu_pf_driver);
err_unregister_acpi:
        acpi_bus_unregister_driver(&acpi_fujitsu_bl_driver);

        return ret;
}

static void __exit fujitsu_cleanup(void)
{
        acpi_bus_unregister_driver(&acpi_fujitsu_laptop_driver);

        platform_driver_unregister(&fujitsu_pf_driver);

        acpi_bus_unregister_driver(&acpi_fujitsu_bl_driver);

        pr_info("driver unloaded\n");
}

module_init(fujitsu_init);
module_exit(fujitsu_cleanup);
```

That `module_init()` macro at the very end tells the kernel to call that `fujitsu_init()` function when this module is loaded. `fujitsu_init()` registers the different portions of this driver by calling the appropriate subsystem registration functions, and passing the address of a `static` struct, usually defined somewhere nearby. In this case, it registers:

- **acpi_fujitsu_bl_driver**, which seems to handle backlight control
- **fujitsu_pf_driver**, which seems to be some kind of "pseudo"-driver that associates the different parts of this driver somehow, I'm not entirely sure.
- **acpi_fujitsu_laptop_driver**, which seems like the part we're interested in today.

I'm not sure why the backlight driver is registered separately like this, but let's focus on `acpi_fujitsu_laptop_driver` for now. Here is the definition:

C

```c
static struct acpi_driver acpi_fujitsu_laptop_driver = {
	.name = ACPI_FUJITSU_LAPTOP_DRIVER_NAME,
	.class = ACPI_FUJITSU_CLASS,
	.ids = fujitsu_laptop_device_ids,
	.ops = {
		.add = acpi_fujitsu_laptop_add,
		.remove = acpi_fujitsu_laptop_remove,
		.notify = acpi_fujitsu_laptop_notify,
		},
};
```

This uses the delightful *C99 designated initializers*. Struct fields are initialized with `.<fieldname> = <value>`, and any field we don't touch is zero-initialized for us.[4] Here, it sets set some metadata, then provides a table `ids` of known device IDs this driver supports, and finally, specifies functions the kernel's ACPI subsystem will call for us when this device is added, removed, or receives some kind of notification.

Presumably the probing and setup for the hotkeys would happen when the device is added, so we'll take a look at `acpi_fujitsu_laptop_add` first:

C

```c
static int acpi_fujitsu_laptop_add(struct acpi_device *device)
{
	struct fujitsu_laptop *priv;
	int ret, i = 0;

	priv = devm_kzalloc(&device->dev, sizeof(*priv), GFP_KERNEL);
	if (!priv)
		return -ENOMEM;
	... SNIP ...
	device->driver_data = priv;
```

It first allocates a `struct fujitsu_laptop` that will contain our driver-specific state, and associates it with that generic `device` struct from the ACPI subsystem. Later calls to this driver from the ACPI subsystem will also provide that same struct, which lets this driver access its state via that `device->driver_data` pointer.

Then a bit further down, it starts to get interesting:

```c
while (call_fext_func(device, FUNC_BUTTONS, 0x1, 0x0, 0x0) != 0 &&
       i++ < MAX_HOTKEY_RINGBUFFER_SIZE)
        ; /* No action, result is discarded */
acpi_handle_debug(device->handle, "Discarded %i ringbuffer entries\n",
                  i);
```

This loop flushes the firmware key event ringbuffer until it's empty to ensure that the firmware internal state matches what this driver expects. We also see the first call to `call_fext_func()`, which is a wrapper function around some kernel ACPI calls that interact with the system firmware.

> I won't explain ACPI in detail here, but I did end up falling down yet another rabbit hole learning about it while I was working on this. TL;DR: The hardware vendor provides system firmware that contains architecture-agnostic bytecode that knows how to talk to hardware. The kernel contains a virtual machine that executes this bytecode. `call_fext_func()` lets this driver call different functions in this firmware code to interact with hardware.

At the end of `acpi_fujitsu_laptop_add()`, we see calls to various setup functions to finish setting things up:

```c
ret = acpi_fujitsu_laptop_input_setup(device);
if (ret)
        goto err_free_fifo;

ret = acpi_fujitsu_laptop_leds_register(device);
if (ret)
        goto err_free_fifo;

ret = fujitsu_laptop_platform_add(device);
... SNIP ...
```

That input setup function looks interesting:

```c
static int acpi_fujitsu_laptop_input_setup(struct acpi_device *device)
{
        struct fujitsu_laptop *priv = acpi_driver_data(device);
        int ret;
```

```
        priv->input = devm_input_allocate_device(&device->dev);
        if (!priv->input)
                return -ENOMEM;

        snprintf(priv->phys, sizeof(priv->phys), "%s/input0",
                 acpi_device_hid(device));

        priv->input->name = acpi_device_name(device);
        priv->input->phys = priv->phys;
        priv->input->id.bustype = BUS_HOST;
```

`acpi_driver_data()` gives us back that `device->driver_data` pointer that was set earlier, meaning `priv` now lets us access the driver state in the `fujitsu_laptop` struct. It then allocates an input device to `priv->input` and performs some initial setup on it.

Next, it checks this `fujitsu_laptop_dmi_table`, then passes a `keymap` to another setup function to associate it with `priv->input`, and finally registers this input device:

C

```c
  dmi_check_system(fujitsu_laptop_dmi_table);
  ret = sparse_keymap_setup(priv->input, keymap, NULL);
  if (ret)
          return ret;

  return input_register_device(priv->input);
```

But what is `keymap` and where does it come from? It's defined like this a bit higher up in the file:

C

```c
static const struct key_entry *keymap = keymap_default;
```

So `keymap` a global variable (marked `static`, so only global within this .c file), and it's initialized to point to `keymap_default`. That call to `dmi_check_system()` iterates over this array:

C

```c
static const struct dmi_system_id fujitsu_laptop_dmi_table[] = {
        {
                .callback = fujitsu_laptop_dmi_keymap_override,
```

```
                .ident = "Fujitsu Siemens S6410",
                .matches = {
                        DMI_MATCH(DMI_SYS_VENDOR, "FUJITSU SIEMENS"),
                        DMI_MATCH(DMI_PRODUCT_NAME, "LIFEBOOK S6410"),
                },
                .driver_data = (void *)keymap_s64x0
        },
        {
                .callback = fujitsu_laptop_dmi_keymap_override,
                .ident = "Fujitsu Siemens S6420",
                .matches = {
                        DMI_MATCH(DMI_SYS_VENDOR, "FUJITSU SIEMENS"),
                        DMI_MATCH(DMI_PRODUCT_NAME, "LIFEBOOK S6420"),
                },
                .driver_data = (void *)keymap_s64x0
        },
        {
                .callback = fujitsu_laptop_dmi_keymap_override,
                .ident = "Fujitsu LifeBook P8010",
                .matches = {
                        DMI_MATCH(DMI_SYS_VENDOR, "FUJITSU"),
                        DMI_MATCH(DMI_PRODUCT_NAME, "LifeBook P8010"),
                },
                .driver_data = (void *)keymap_p8010
        },
        {}
};
```

For each item in this array, it checks to see if the vendor and product names match what the firmware reports, and if so, calls the corresponding `callback` function. It passes that `driver_data` value to the callback via a `dmi_system_id` struct.

The callback in this driver is very simple:

C

```c
static int fujitsu_laptop_dmi_keymap_override(const struct dmi_system_id *id)
{
        pr_info("Identified laptop model '%s'\n", id->ident);
        keymap = id->driver_data;
```

```
        return 1;
}
```

It prints the laptop model to the kernel log, and updates that `keymap` global to point to a model-specific keymap specified in the dmi table.

In other words, it's looking for a hardware-specific keymap to use, and if one wasn't found, it falls back on `keymap_default`. Checking the kernel logs on my machine, I don't see that `Identified laptop model` message, which makes sense, since I don't see the Lifebook S2110 listed in this dmi table.

That covers the setup for the hotkey portion of this driver, but how are the actual key press events from the hardware handled? This is where that `notify` callback in the `acpi_driver` struct comes in. This driver sets it to `acpi_fujitsu_laptop_notify()`:

```c
static void acpi_fujitsu_laptop_notify(struct acpi_device *device, u32 event)
{
        struct fujitsu_laptop *priv = acpi_driver_data(device);
        unsigned long flags;
        int scancode, i = 0;
        unsigned int irb;
        ... SNIP ...
        while ((irb = call_fext_func(device,
                                     FUNC_BUTTONS, 0x1, 0x0, 0x0)) != 0 &&
               i++ < MAX_HOTKEY_RINGBUFFER_SIZE) {
                scancode = irb & 0x4ff;
                if (sparse_keymap_entry_from_scancode(priv->input, scancode))
                        acpi_fujitsu_laptop_press(device, scancode);
                else if (scancode == 0)
                        acpi_fujitsu_laptop_release(device);
                else
                        acpi_handle_info(device->handle,
                                         "Unknown GIRB result [%x]\n", irb);
        }
        ... SNIP ...
}
```

When the system firmware generates an event, such as a key press, it notifies the Linux ACPI subsystem, and that subsystem then calls this notify function for us. In here we see a loop similar to the one we saw earlier to flush the firmware ringbuffer, but rather than discarding the return value, it stores it in `irb`, and masks off some bits to get a `scancode`.

Next, it checks to see if `scancode` exists in the currently active keymap, and if it does, it sends a keypress event to the Linux input subsystem via `acpi_fujitsu_laptop_press()`. Same deal with key release, which the firmware seems to indicate with scancode 0.

There is a lot more to explore here, but I think I now have a rough idea of how this part of the driver works. To recap:

1. The driver initializes itself in various subsystems
2. An input device is created
3. An appropriate keymap is selected based on a list of known hardware, with `keymap_default` as a fallback if a more specific keymap wasn't found.
4. That keymap is associated with the input device
5. The notify callback passes key events to the Linux input subsystem, and prints a message to the kernel log for unknown keycodes.

Modifying the driver

So to add support for the media keys on my laptop, I probably need to define a new keymap. The keymaps are defined like this:

C

```c
static const struct key_entry keymap_default[] = {
	{ KE_KEY, KEY1_CODE,            { KEY_PROG1 } },
	{ KE_KEY, KEY2_CODE,            { KEY_PROG2 } },
	{ KE_KEY, KEY3_CODE,            { KEY_PROG3 } },
	{ KE_KEY, KEY4_CODE,            { KEY_PROG4 } },
	{ KE_KEY, KEY5_CODE,            { KEY_RFKILL } },
	/* Soft keys read from status flags */
	{ KE_KEY, FLAG_RFKILL,          { KEY_RFKILL } },
	{ KE_KEY, FLAG_TOUCHPAD_TOGGLE, { KEY_TOUCHPAD_TOGGLE } },
	{ KE_KEY, FLAG_MICMUTE,         { KEY_MICMUTE } },
	{ KE_END, 0 }
};

static const struct key_entry keymap_s64x0[] = {
	{ KE_KEY, KEY1_CODE, { KEY_SCREENLOCK } },	/* "Lock" */
	{ KE_KEY, KEY2_CODE, { KEY_HELP } },		/* "Mobility Center */
	{ KE_KEY, KEY3_CODE, { KEY_PROG3 } },
	{ KE_KEY, KEY4_CODE, { KEY_PROG4 } },
	{ KE_END, 0 }
};
```

Okay, the default keymap looks reasonable. It seems like other laptop models have keys for toggling radios, the touchpad and mic. I don't think my laptop has those, so I'll base my keymap on this `s64x0` table. For each entry in these tables, the first value is the entry type, e.g. `KE_KEY` for a key, and `KE_END` to mark the end of an array. The next value is the raw scancode we receive from the firmware, and the last value is the corresponding keycode that gets passed to the Linux input subsystem.

These `KEY*_CODE` macros for the firmware scancodes are defined near the top of the file:

C

```c
/* Scancodes read from the GIRB register */
#define KEY1_CODE                       0x410
#define KEY2_CODE                       0x411
#define KEY3_CODE                       0x412
#define KEY4_CODE                       0x413
#define KEY5_CODE                       0x420
```

The macros for the linux keycodes such as `KEY_PROG1` and `KEY_RFKILL`, are defined in `include/uapi/linux/input-event-codes.h`

Looking at the values in those `Unknown GIRB result` messages in the kernel log when the keys are in `Player` mode, they correspond to codes `0x414` - `0x417`, which seems to check out. The codes just get shifted up by 4. I'll expand the set of defines to include these 4 new keys:

DIFF

```diff
 #define KEY4_CODE          0x413
-#define KEY5_CODE          0x420
+#define KEY5_CODE          0x414
+#define KEY6_CODE          0x415
+#define KEY7_CODE          0x416
+#define KEY8_CODE          0x417
+#define KEY9_CODE          0x420
```

The existing value for `KEY5_CODE` is now `KEY9_CODE`, so I updated `keymap_default` to reflect that:

DIFF

```diff
@@ -560,7 +564,7 @@ static const struct key_entry keymap_default[] = {
     { KE_KEY, KEY2_CODE,            { KEY_PROG2 } },
     { KE_KEY, KEY3_CODE,            { KEY_PROG3 } },
     { KE_KEY, KEY4_CODE,            { KEY_PROG4 } },
-    { KE_KEY, KEY5_CODE,            { KEY_RFKILL } },
```

```
+   { KE_KEY, KEY9_CODE,            { KEY_RFKILL } },
    /* Soft keys read from status flags */
```

With these updated defines, I put together this new keymap:

C

```c
static const struct key_entry keymap_s2110[] = {
          { KE_KEY, KEY1_CODE, { KEY_PROG1 } }, /* "A" */
          { KE_KEY, KEY2_CODE, { KEY_PROG2 } }, /* "B" */
          { KE_KEY, KEY3_CODE, { KEY_WWW } },   /* "Internet" */
          { KE_KEY, KEY4_CODE, { KEY_EMAIL } }, /* "E-mail" */
          { KE_KEY, KEY5_CODE, { KEY_STOPCD } },
          { KE_KEY, KEY6_CODE, { KEY_PLAYPAUSE } },
          { KE_KEY, KEY7_CODE, { KEY_PREVIOUSSONG } },
          { KE_KEY, KEY8_CODE, { KEY_NEXTSONG } },
          { KE_END, 0 }
};
```

We already saw before that in `Application` mode, the key events showed up as `XF86Launch1` - `XF86Launch4` (corresponding to `KEY_PROG1` - `KEY_PROG4` in `keymap_default`), but since the last two keys on this laptop are labeled with their specific function, we can express that here with the more specific macros `KEY_WWW` and `KEY_EMAIL`.

The mapping between these Linux keycodes and the keycodes that show up in `xev(1)` output wasn't very obvious to me. There is some translation happening somewhere in userspace (libinput?), so I found it quicker to just experiment with different values to arrive at that final table.

Finally, I need to add a new entry to the DMI table so this new keymap is selected for this model of laptop:

DIFF

```diff
@@ -621,6 +637,15 @@ static const struct dmi_system_id fujitsu_laptop_dmi_table[] = {
        },
        .driver_data = (void *)keymap_p8010
    },
+   {
+       .callback = fujitsu_laptop_dmi_keymap_override,
+       .ident = "Fujitsu LifeBook S2110",
+       .matches = {
+           DMI_MATCH(DMI_SYS_VENDOR, "FUJITSU SIEMENS"),
```

```
+            DMI_MATCH(DMI_PRODUCT_NAME, "LIFEBOOK S2110"),
+        },
+        .driver_data = (void *)keymap_s2110
+    },
    {}
 };
```

Testing my changes

To test my changes, I ended up using the Arch build system. So I'd run `makepkg -e` to build a new kernel package with my modifications, which I could then install alongside the upstream Arch kernel. I didn't spend too much time refining this setup, so I ended up just manually patching the `src` directory that `makepkg` sets up. I'm sure there is some way to point `PKGBUILD` at an existing local git repository to make this process a bit smoother.

With my patched kernel installed and running, I can now see the message from the keymap override callback appear on boot:

TEXT

```
[   68.921998] fujitsu_laptop: ACPI: Fujitsu FUJ02E3 [FEXT]
[   68.923714] ACPI: \_SB_.FEXT: BTNI: [0xff0101]
[   68.923741] fujitsu_laptop: Identified laptop model 'Fujitsu LifeBook S2110'
```

And the keys now all work as expected! Check this out:

TEXT

```
> xev -event keyboard | grep keysym
    state 0x0, keycode 156 (keysym 0x1008ff41, XF86Launch1), same_screen YES,
    state 0x0, keycode 157 (keysym 0x1008ff42, XF86Launch2), same_screen YES,
    state 0x0, keycode 158 (keysym 0x1008ff2e, XF86WWW), same_screen YES,
    state 0x0, keycode 223 (keysym 0x1008ff19, XF86Mail), same_screen YES,
          (Switch to Player mode)
    state 0x0, keycode 174 (keysym 0x1008ff15, XF86AudioStop), same_screen YES,
    state 0x0, keycode 172 (keysym 0x1008ff14, XF86AudioPlay), same_screen YES,
    state 0x0, keycode 173 (keysym 0x1008ff16, XF86AudioPrev), same_screen YES,
    state 0x0, keycode 171 (keysym 0x1008ff17, XF86AudioNext), same_screen YES,
```

With the new key events working, I can expand my `i3(1)` config to match:

TEXT

```
bindsym XF86AudioStop exec --no-startup-id playerctl stop
bindsym XF86AudioPlay exec --no-startup-id playerctl play-pause
bindsym XF86AudioPrev exec --no-startup-id playerctl previous
bindsym XF86AudioNext exec --no-startup-id playerctl next
bindsym XF86WWW exec --no-startup-id firefox
bindsym XF86Launch1 exec i3-sensible-terminal
```

I found that playerctl(1) utility while I was working on this. It uses MPRIS to control media players over D-Bus.

I even found a physical music CD[5] to test with in VLC to get that fully authentic 2005 experience! 8]

Upstreaming

So I've now fixed the media keys on my laptop, so it's time to think about sending my improvements to kernel maintainers for inclusion in upstream Linux, so everyone else running the latest kernel on their S2110 can benefit.[6]

I'll start by committing my changes locally:

TEXT

```
$ git checkout -b s2110-mediakeys
$ git add drivers/platform/x86/fujitsu-laptop.c
$ git commit
$ git show --stat HEAD
commit d0e1e0b3e2e6674cce73909d4092874c6c8c2d11 (HEAD -> s2110-mediakeys)
Author: Valtteri Koskivuori <vkoskiv@gmail.com>
Date:   Fri May 9 17:54:50 2025 +0300

    platform/x86: fujitsu-laptop: Support Lifebook S2110 hotkeys

    The S2110 has an additional set of media playback control keys enabled
    by a hardware toggle button that switches the keys between "Application"
    and "Player" modes. Toggling "Player" mode just shifts the scancode of
    each hotkey up by 4.

    Add defines for new scancodes, and a keymap and dmi id for the S2110.

    Tested on a Fujitsu Lifebook S2110.
```

```
Signed-off-by: Valtteri Koskivuori <vkoskiv@gmail.com>

drivers/platform/x86/fujitsu-laptop.c | 33 +++++++++++++++++++++++++++++----
1 file changed, 29 insertions(+), 4 deletions(-)
```

Then I can run `checkpatch.pl` against this commit. It should point out any obvious issues with my code or the commit message.

TEXT

```
$ scripts/checkpatch.pl -g HEAD
total: 0 errors, 0 warnings, 68 lines checked

Commit d0e1e0b3e2e6 ("platform/x86: fujitsu-laptop: Support Lifebook S2110 hotkeys") has no obviou
```

Looks good. Next, to get the list of recipients for my submission email, I can use `scripts/get_maintainer.pl`:

TEXT

```
$ scripts/get_maintainer.pl -f drivers/platform/x86/fujitsu-laptop.c
Jonathan Woithe <*******@just42.net> (maintainer:FUJITSU LAPTOP EXTRAS)
Hans de Goede <********@redhat.com> (maintainer:X86 PLATFORM DRIVERS)
"Ilpo Järvinen" <****.********@linux.intel.com> (maintainer:X86 PLATFORM DRIVERS)
platform-driver-x86@vger.kernel.org (open list:FUJITSU LAPTOP EXTRAS)
linux-kernel@vger.kernel.org (open list)
```

I've already configured my git email integration following this guide and tested that it works, so I can just use `git-send-email(1)` to fire off my patch.

To be extra sure that everything is working correctly, I'll first send the patch to myself:

TEXT

```
$ git send-email --to="vkoskiv@gmail.com" HEAD^
```

Looks good on the receiving end, so I type the following:[7]

TEXT

```
$ git send-email --to *******@just42.net --to ********@redhat.com --to ****.********@linux.intel.c
```

I don't usually get nervous when sending email, but I was still worried I had missed some detail, and I *really* don't want to waste a kernel maintainer's time. I could soon refresh the mailing list archives and find my message.

The maintainers didn't request any changes for this minor patch, so a bit over a month after sending this email, I ran `pacman -Syu` on the S2110, and I could see my changes had landed in the upstream Arch Linux kernel. That was a pretty cool moment! :]

Project Timeline

- **2025-05-09**: I send my patch: lore.kernel.org
- **2025-05-11**: The `fujitsu-laptop` driver maintainer ACKs my patch: lore.kernel.org
- **2025-05-15**: `platform-drivers-x86` maintainer applies my patch to their review branch: lore.kernel.org
- **2025-05-23**: `platform-drivers-x86` maintainer sends a PR to Linus for v6.15, **my patch is included!** lore.kernel.org
- **2025-05-23**: Linus merges that PR later that same day: lore.kernel.org
- **2025-05-25**: Linux 6.15 is announced by Linus. My name appears in the changelog! lore.kernel.org
- **2025-05-27**: Sasha Levin selects my patch (and a few others) for backporting to all the currently maintained upstream LTS kernels 6.14, 6.12, 6.6, 6.1, 5.15, 5.10 and 5.4.
- **2025-06-14**: I upgrade Arch on my S2110, the keys now work with the upstream kernel! :]

Conclusion

It was *really* cool to watch my patch make its way up the chain and have it end up in mainline! It was also nice to experience the traditional patch & email workflow for the first time. The kernel newbies guide was a good resource to ensure everything was in order before sending my first patch. The whole process was actually much easier than I had imagined it to be.

That being said, this was a very small change in a minor driver, so it went in without any rounds of feedback. I actually already have another set of patches I made earlier this year that resolve a minor issue with the network card in this laptop, but that set involves a minor change to core kernel code under `kernel/dma/`, so I want to be 100% certain my changes are justified before I seek feedback from kernel maintainers.

It was actually this process of finding existing precedent for justifying my choices that made me really appreciate the traditional mailing list based process. The kernel git log and mailing lists are full of really valuable context for changes dating back decades, so all my questions during this process were answered by either searching these archives or by browsing git logs.

Thank you for reading! I'd really appreciate any feedback on my writing, so if you have any questions, corrections, suggestions or other thoughts, do get in touch! :]

My next post, **"Tracking down a regression in Mesa 3D"**, is due to be published within the next week (by **October 12th, 2025**).

This post was discussed on Hacker News and lobste.rs.

---

1. I actually broke the display on this laptop in 2020, and went through the trouble of sourcing a replacement from China! ↩

2. I cloned the full history, and prepared `clangd` with `make allmodconfig && bear -- make -j$(nproc)` so I can use wonderful LSP features to instantly jump to/from symbols and find references to symbols. ↩

3. I quite often find myself falling down one rabbit hole after another when browsing kernel code. Everything just looks so interesting, and the **instant** LSP "goto definition" feature of my editor makes this a tough habit to break. I find compiling lists of things to look into later helps with this somewhat. ↩

4. C99 designated initializers are way nicer than the ones C++20 ended up with, but there is a caveat: Padding bytes between struct elements are not zero-initialized, which may cause issues. ↩

5. Hot Fuss by The Killers ↩

6. I'd be *very* surprised if anyone else runs latest upstream Linux on this particular system. :] If you do, let me know! ↩

7. There are ways to make this submission process a bit smoother, but I wanted to do this manually since I'm still starting out. ↩

Published by vkoskiv tagged programming, linux and foss