Okay, I understand you want a code base to start with for your kernel development journey, even if it's a very basic example that you will build upon.

Given that your goal involves interacting with the wireless stack from the kernel side via Netlink (which is how userspace nl80211 commands communicate with the kernel), a good starting point is a simple kernel module that can receive Netlink messages from userspace and interact with the wireless subsystem.

Below are two code snippets:

1. **A basic Linux Kernel Module (.c file):** This module will register a simple generic Netlink family and implement a handler to receive a message. When it receives a message, it will attempt to find a wireless device (wiphy) and print its name to the kernel logs (dmesg).
2. **A basic Userspace Netlink Client (.c file):** This application will send a simple Netlink message to the kernel module.

This code does NOT implement the complex P2P interface creation, parallel operation, or resource sharing logic. It is purely a skeleton to demonstrate how a kernel module can communicate with userspace via Netlink and perform a basic interaction with the wireless subsystem (finding a wiphy).

---

1. Basic Kernel Module Code (my_wireless_module.c)

C

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/netlink.h>
#include <net/genetlink.h>
#include <linux/netdevice.h> // For struct net_device
#include <net/cfg80211.h>    // For struct wiphy and wireless_dev

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name"); // Replace with your name
MODULE_DESCRIPTION("A basic kernel module for wireless Netlink interaction");
MODULE_VERSION("0.1");
```

```
/* Define the Generic Netlink family */
enum my_wireless_commands {
    MY_CMD_UNSPEC,
    MY_CMD_FIND_WIPHY, /* A simple command to find a wiphy */
    /* Add more commands as needed later */
    __MY_CMD_MAX,
};
#define MY_CMD_MAX (__MY_CMD_MAX - 1)

/* Attributes for the Generic Netlink family (optional for this example) */
enum my_wireless_attributes {
    MY_ATTR_UNSPEC,
    MY_ATTR_MESSAGE, /* A simple message attribute */
    /* Add more attributes as needed later */
    __MY_ATTR_MAX,
};
#define MY_ATTR_MAX (__MY_ATTR_MAX - 1)

/* Command definition */
static const struct genl_ops my_wireless_ops[] = {
    {
        .cmd = MY_CMD_FIND_WIPHY,
        .flags = 0, // Or GENL_ADMIN_PERM for root-only commands
        .policy = NULL, // Attribute policy (not needed for this simple cmd)
        .doit = my_netlink_rx, // The function to handle this command
        .dumpit = NULL, // For dump commands
    },
};

/* Generic Netlink family definition */
static struct genl_family my_wireless_family = {
    .hdrsize = 0,
    .name = "my_wireless_family", // Name of the family
    .version = 1,
    .maxattr = MY_ATTR_MAX,
    .ops = my_wireless_ops,
    .n_ops = ARRAY_SIZE(my_wireless_ops),
};
```

```
/*
 * Netlink receive callback function
 * This function is called when a message for MY_CMD_FIND_WIPHY arrives
 */
static int my_netlink_rx(struct sk_buff *skb, struct genl_info *info)
{
    struct sk_buff *msg_skb;
    void *msg_head;
    int rc = 0;
    struct wiphy *wiphy;

    printk(KERN_INFO "my_wireless_module: Received MY_CMD_FIND_WIPHY command from
userspace.\n");

    // --- Example Interaction with Wireless Subsystem (Finding a wiphy) ---
    // This is a very basic example. Real interaction for interface creation
    // is much more involved and requires using cfg80211/mac80211 APIs correctly.

    wiphy = NULL;
    rc = -ENODEV; // Assume no device found initially

    // Iterate through all registered wiphy devices
    for_each_wiphy(wiphy) {
        if (wiphy) {
            printk(KERN_INFO "my_wireless_module: Found wiphy device: %s\n",
wiphy_name(wiphy));
            rc = 0; // Found at least one
            // In a real scenario, you'd match a specific device or capability
            // break; // Uncomment if you only need the first one
        }
    }

    if (rc != 0) {
        printk(KERN_INFO "my_wireless_module: No wireless devices (wiphy) found.\n");
        // If you need to return an error to userspace, you'd send a status message.
    }


    // --- Example: Sending a simple response back to userspace ---
    msg_skb = genlmsg_new(NLMSG_DEFAULT_SIZE, GFP_KERNEL);
    if (!msg_skb) {
        printk(KERN_ERR "my_wireless_module: Failed to allocate Netlink response buffer.\n");
        return -ENOMEM;
    }

    msg_head = genlmsg_put(msg_skb, info->snd_portid, info->snd_seq,
&my_wireless_family, 0, MY_CMD_FIND_WIPHY);
    if (!msg_head) {
        printk(KERN_ERR "my_wireless_module: Failed to put Netlink response header.\n");
        nlmsg_free(msg_skb);
        return -ENOMEM;
    }

    // Add an attribute to the response
    rc = nla_put_string(msg_skb, MY_ATTR_MESSAGE, "Processed MY_CMD_FIND_WIPHY");
    if (rc) {
         printk(KERN_ERR "my_wireless_module: Failed to put attribute in response.\n");
         nlmsg_free(msg_skb);
         return rc;
    }


    genlmsg_end(msg_skb, msg_head);

    // Send the message back to the originating userspace process
    // rc = genlmsg_unicast(&init_net, msg_skb, info->snd_portid);
    // Or broadcast to a group if defined
    // rc = genlmsg_multicast_netns(&my_wireless_family, &init_net, 0, MY_NL_MCAST_GROUP,
GFP_KERNEL);

    // For this simple example, let's assume we unicast back.
    rc = genlmsg_unicast(genl_info_net(info), msg_skb, info->snd_portid);

    if (rc < 0) {
        printk(KERN_ERR "my_wireless_module: Failed to unicast Netlink response: %d\n", rc);
        // genlmsg_unicast frees the skb on success or error <= 0
        // so we don't need nlmsg_free here if unicast handles it.
    }
    return 0; // Return 0 on success
}


/*
 * Module initialization function
 */
static int __init my_wireless_module_init(void)
{
    int rc;
    printk(KERN_INFO "my_wireless_module: Initializing.\n");

    // Register the generic Netlink family
    rc = genl_register_family(&my_wireless_family);
    if (rc < 0) {
        printk(KERN_ERR "my_wireless_module: Failed to register Netlink family: %d\n", rc);
        return rc;
    }

    printk(KERN_INFO "my_wireless_module: Netlink family '%s' registered (id: %d).\n",
           my_wireless_family.name, my_wireless_family.id);

    return 0;
}

/*
 * Module exit function
 */
static void __exit my_wireless_module_exit(void)
{
    printk(KERN_INFO "my_wireless_module: Exiting.\n");

    // Unregister the generic Netlink family
    genl_unregister_family(&my_wireless_family);
    printk(KERN_INFO "my_wireless_module: Netlink family '%s' unregistered.\n",
           my_wireless_family.name);
}

module_init(my_wireless_module_init);
```

```
module_exit(my_wireless_module_exit);
```

Explanation of the Kernel Module:

- Includes necessary headers for module development, Netlink, generic Netlink, and wireless.
- Defines a generic Netlink family named "my_wireless_family" with a single command MY_CMD_FIND_WIPHY.
- The my_netlink_rx function is the handler for MY_CMD_FIND_WIPHY. Inside it:
  - It prints a message to the kernel log (dmesg).
  - It iterates through all registered wiphy devices using for_each_wiphy and prints their names.
  - It constructs a simple Netlink response message containing a string attribute.
  - It sends the response back to the userspace process that sent the original command.
- my_wireless_module_init registers the Netlink family when the module is loaded.
- my_wireless_module_exit unregisters the family when the module is unloaded.

---

2. Basic Userspace Netlink Client Code (my_wireless_client.c)

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

// Netlink headers
#include <netlink/netlink.h>
#include <netlink/genl/genl.h>
#include <netlink/genl/ctrl.h>

// Define the same family and command enums as in the kernel module
// In a real project, these would ideally be in a shared header file.
enum my_wireless_commands {
    MY_CMD_UNSPEC,
    MY_CMD_FIND_WIPHY,
    __MY_CMD_MAX,
};
#define MY_CMD_MAX (__MY_CMD_MAX - 1)

enum my_wireless_attributes {
    MY_ATTR_UNSPEC,
    MY_ATTR_MESSAGE,
    __MY_ATTR_MAX,
};
#define MY_ATTR_MAX (__MY_ATTR_MAX - 1)


// Callback function for receiving messages from the kernel
static int msg_handler(struct nl_msg *msg, void *arg)
{
    struct nlattr *attrs[MY_ATTR_MAX + 1];
    struct genlmsghdr *gnlh = nlmsg_data(nlmsg_hdr(msg));

    printf("Received message from kernel (command: %d):\n", gnlh->cmd);

    // Parse attributes if any
    genlmsg_parse(gnlh, 0, attrs, MY_ATTR_MAX, NULL);

    if (attrs[MY_ATTR_MESSAGE]) {
        printf("  Attribute MY_ATTR_MESSAGE: %s\n", nla_get_string(attrs[MY_ATTR_MESSAGE]));
    }

    // In a real application, you would process the data here.

    return NL_OK; // Continue processing messages
}

int main(int argc, char *argv[])
{
    struct nl_sock *sock;
    struct nl_msg *msg;
    int family_id;
    int rc;

    printf("Starting Netlink client...\n");
    // Allocate a new Netlink socket
    sock = nl_socket_alloc();
    if (!sock) {
        fprintf(stderr, "ERROR: Failed to allocate Netlink socket.\n");
        return -1;
    }

    // Connect the socket to the generic Netlink family
    rc = genl_connect(sock);
    if (rc < 0) {
        fprintf(stderr, "ERROR: Failed to connect to generic Netlink: %s\n", nl_geterror(rc));
        nl_socket_free(sock);
        return -1;
    }

    // Resolve the Netlink family ID for our custom family name
    family_id = genl_ctrl_resolve(sock, "my_wireless_family");
    if (family_id < 0) {
        fprintf(stderr, "ERROR: Failed to resolve Netlink family 'my_wireless_family': %s\n",
nl_geterror(family_id));
        fprintf(stderr, "Ensure the kernel module 'my_wireless_module' is loaded.\n");
        nl_socket_free(sock);
        return -1;
    }
    printf("Resolved family 'my_wireless_family' ID: %d\n", family_id);


    // Allocate a new Netlink message
    msg = nlmsg_alloc();
    if (!msg) {
        fprintf(stderr, "ERROR: Failed to allocate Netlink message.\n");
        nl_socket_free(sock);
        return -1;
    }

    // Add the generic Netlink header to the message
    // genlmsg_put(msg, pid, seq, family_id, hdrlen, flags, cmd, version)
    genlmsg_put(msg, 0, 0, family_id, 0, 0, MY_CMD_FIND_WIPHY, 0);
```

```c
    // (Optional) Add attributes to the message if the command requires them
    // int nla_put_u32(struct nl_msg *msg, int attrtype, u32 value);
    // int nla_put_string(struct nl_msg *msg, int attrtype, const char *str);
    // rc = nla_put_string(msg, MY_ATTR_MESSAGE, "Hello from userspace!");
    // if (rc < 0) {
    //     fprintf(stderr, "ERROR: Failed to put attribute in message: %s\n", nl_geterror(rc));
    //     nlmsg_free(msg);
    //     nl_socket_free(sock);
    //     return -1;
    // }


    // Send the message to the kernel
    rc = nl_send_auto(sock, msg);
    if (rc < 0) {
        fprintf(stderr, "ERROR: Failed to send Netlink message: %s\n", nl_geterror(rc));
        nlmsg_free(msg); // nl_send_auto frees on success, but explicitly free on send error
        nl_socket_free(sock);
        return -1;
    }
    printf("Netlink message sent successfully (bytes: %d).\n", rc);
    nlmsg_free(msg); // Message is freed by nl_send_auto after successful send


    // Set the message handler for receiving responses
    nl_socket_modify_cb(sock, NL_CB_MSG_IN, NL_CB_CUSTOM, msg_handler, NULL);

    // Receive messages from the kernel
    printf("Waiting for response from kernel...\n");
    rc = nl_recvmsgs_default(sock);
    if (rc < 0) {
        fprintf(stderr, "ERROR: Failed to receive Netlink messages: %s\n", nl_geterror(rc));
        // Continue to cleanup
    }

    // Clean up
    nl_socket_free(sock);
    printf("Client finished.\n");
```

```
  return 0;
}
```

Explanation of the Userspace Client:

- Includes standard headers and headers for libnl (Netlink library).
- Defines the same family name and command enum as the kernel module.
- msg_handler is a callback function that will be executed when a Netlink message is received from the kernel. It parses the message and prints the content of MY_ATTR_MESSAGE if present.
- main function:
  - Allocates a Netlink socket using libnl.
  - Connects the socket to the generic Netlink subsystem.
  - Resolves the family ID of the custom kernel module's Netlink family using genl_ctrl_resolve. **The kernel module must be loaded before running the client for this step to succeed.**
  - Allocates a Netlink message buffer.
  - Adds a generic Netlink header to the message, specifying the target family ID and the command MY_CMD_FIND_WIPHY.
  - Sends the message to the kernel using nl_send_auto.
  - Sets the msg_handler as the callback for incoming messages.
  - Enters a loop (or uses nl_recvmsgs_default) to wait for and process messages from the kernel.
  - Cleans up the socket.

How to Compile and Run:

1. **Compile the Kernel Module:**
   - Save the kernel module code as my_wireless_module.c.
   - Create a Makefile in the same directory:
     ```Makefile
     obj-m := my_wireless_module.o

     KDIR := /lib/modules/$(shell uname -r)/build
     PWD := $(shell pwd)

     default:
     	$(MAKE) -C $(KDIR) M=$(PWD) modules
     ```

    - Run make to compile the module. This will produce my_wireless_module.ko. You might need kernel headers installed (linux-headers-$(uname -r)).
2. **Compile the Userspace Client:**
    - Save the userspace code as my_wireless_client.c.
    - Compile it using g++ and link against libnl libraries:

      ```bash
      g++ my_wireless_client.c -o my_wireless_client -l:libnl-3.a -l:libnl-genl-3.a # Link statically if available
      # OR dynamically if static libs are not preferred/available:
      # g++ my_wireless_client.c -o my_wireless_client -lnl-3 -lnl-genl-3
      ```

    - You need libnl3 and libnl-genl3 development packages installed (e.g., libnl-3-dev libnl-genl-3-dev on Debian/Ubuntu).
3. **Run:**
    - **Load the kernel module:**

      ```bash
      sudo insmod my_wireless_module.ko
      ```

      Check dmesg for messages like "my_wireless_module: Initializing." and "Netlink family 'my_wireless_family' registered".
    - **Run the userspace client:**

      ```bash
      sudo ./my_wireless_client
      ```

      The client should print messages indicating it sent the command and received a response. Check dmesg again for messages from the kernel module about receiving the command and finding wiphy devices.
    - **Unload the kernel module when done:**

      ```bash
      sudo rmmod my_wireless_module
      ```

      Check dmesg for the module exit message.

This setup provides a minimal foundation for exploring Netlink communication from the kernel and interacting with basic kernel structures like wiphy. From here, you would delve deeper into the cfg80211 and mac80211 APIs, learn how interface creation is handled, and attempt to modify the kernel logic to allow parallel interface operation (Steps 3-5 from the previous explanation). Remember that the resource sharing aspect (Step 4) is the most hardware-dependent and complex.