in include/linux/init.h

```
/*
 * A "pure" initcall has no dependencies on anything else, and purely
 * initializes variables that couldn't be statically initialized.
 *
 * This only exists for built-in code, not for modules.
 * Keep main.c:initcall_level_names[] in sync.
 */
#define pure_initcall(fn)          __define_initcall(fn, 0)

#define core_initcall(fn)          __define_initcall(fn, 1)
#define core_initcall_sync(fn)     __define_initcall(fn, 1s)
#define postcore_initcall(fn)      __define_initcall(fn, 2)
#define postcore_initcall_sync(fn) __define_initcall(fn, 2s)
#define arch_initcall(fn)          __define_initcall(fn, 3)
#define arch_initcall_sync(fn)     __define_initcall(fn, 3s)
#define subsys_initcall(fn)        __define_initcall(fn, 4)
#define subsys_initcall_sync(fn)   __define_initcall(fn, 4s)
#define fs_initcall(fn)            __define_initcall(fn, 5)
#define fs_initcall_sync(fn)       __define_initcall(fn, 5s)
#define rootfs_initcall(fn)        __define_initcall(fn, rootfs)
#define device_initcall(fn)        __define_initcall(fn, 6)
#define device_initcall_sync(fn)   __define_initcall(fn, 6s)
```

```
#define late_initcall(fn)		__define_initcall(fn, 7)
#define late_initcall_sync(fn)		__define_initcall(fn, 7s)


/* initcalls are now grouped by functionality into separate
 * subsections. Ordering inside the subsections is determined
 * by link order.
 * For backwards compatibility, initcall() puts the call in
 * the device init subsection.
 *
 * The `id' arg to __define_initcall() is needed so that multiple initcalls
 * can point at the same handler without causing duplicate-symbol build errors.
 */

#define __define_initcall(fn, id) \
	static initcall_t __initcall_##fn##id __used \
	__attribute__((__section__(".initcall" #id ".init"))) = fn; \
	LTO_REFERENCE_INITCALL(__initcall_##fn##id)



pure_initcall(ipc_ns_init);
==>
__define_initcall(ipc_ns_init, 0)
==>
#define __define_initcall(ipc_ns_init, 0)
```

```
static initcall_t __initcall_ipc_ns_init0 __used
__attribute__((__section__(".initcall0.init"))) = ipc_ns_init;
```

LTO_REFERENCE_INITCALL是为了fix 某些版本的gcc的bug，这里为NULL。

```
#ifdef CONFIG_LTO
/* Work around a LTO gcc problem: when there is no reference to a variable
 * in a module it will be moved to the end of the program. This causes
 * reordering of initcalls which the kernel does not like.
 * Add a dummy reference function to avoid this. The function is
 * deleted by the linker.
 */
#define LTO_REFERENCE_INITCALL(x) \
        ; /* yes this is needed */                \
        static __used __exit void *reference_##x(void)      \
        {                                       \
                return &x;                      \
        }
#else
#define LTO_REFERENCE_INITCALL(x)
#endif
```

定义了__initcall_ipc_ns_init0 variable,并放入section “.initcall0.init”.

而__define_initcall(XXX, 1) to __define_initcall(XXX, 7s)

定义了　__initcall_XXX_init1，放入section “**.initcall1.init**”.

__initcall_XXX_init1s,放入section “**.initcall1s.init**”.

__initcall_XXX_init2,放入section “**.initcall2.init**”.

__initcall_XXX_init2s,放入section “**.initcall2s.init**”.

.....

__initcall_XXX_init7,放入section “**.initcall7.init**”.

__initcall_XXX_init7s,放入section “**.initcall7s.init**”.

in vmlinux.lds中有

__initcall_start = .; *(.initcallearly.init) __initcall0_start = .; *(.initcall0.init) *(.initcall0s.init) __initcall1_start = .; *(.initcall1.init) *(.initcall1s.init) __initcall2_start = .; *(.initcall2.init) *(.initcall2s.init) __initcall3_start = .; *(.initcall3.init) *(.initcall3s.init) __initcall4_start = .; *(.initcall4.init) *(.initcall4s.init) __initcall5_start = .; *(.initcall5.init) *(.initcall5s.init) __initcallrootfs_start = .; *(.initcallrootfs.init) *(.initcallrootfss.init) __initcall6_start = .; *(.initcall6.init) *(.initcall6s.init) __initcall7_start = .; *(.initcall7.init) *(.initcall7s.init) __initcall_end = .;

这些initcall function pointer组成了function pointer array。

在kernel initialization的后期会依次调用该array中的function pointer指向的function.

function被调用的先后次序完全由initcall level决定。

从最先被调用到最后调用依次为

level early　　[early_initcall(fn)]

level 0　　[pure_initcall(fn)]

level 1 [core_initcall(fn)]

.....

in init/main.c

start_kernel()

|

|

\|/

do_basic_setup()

|

|

\|/

do_initcalls()

```
static void __init do_initcalls(void)
{
        int level;

        for (level = 0; level < ARRAY_SIZE(initcall_levels) - 1; level++)
                do_initcall_level(level);
}

static initcall_t *initcall_levels[] __initdata = {
        __initcall0_start,
```

```
        __initcall1_start,
        __initcall2_start,
        __initcall3_start,
        __initcall4_start,
        __initcall5_start,
        __initcall6_start,
        __initcall7_start,
        __initcall_end,
};

static void __init do_initcall_level(int level)
{
        initcall_t *fn;

        strcpy(initcall_command_line, saved_command_line);
        parse_args(initcall_level_names[level],
                   initcall_command_line, __start___param,
                   __stop___param - __start___param,
                   level, level,
                   &repair_env_string);

        for (fn = initcall_levels[level]; fn < initcall_levels[level+1]; fn++)
                do_one_initcall(*fn);
}
```

```
int __init_or_module do_one_initcall(initcall_t fn)
{
	int count = preempt_count();
	int ret;
	char msgbuf[64];

	if (initcall_blacklisted(fn))		(1)
		return -EPERM;

	if (initcall_debug)				(2)
		ret = do_one_initcall_debug(fn);
	else
		ret = fn();

	msgbuf[0] = 0;

	if (preempt_count() != count) {
		sprintf(msgbuf, "preemption imbalance ");
		preempt_count_set(count);
	}
	if (irqs_disabled()) {
		strlcat(msgbuf, "disabled interrupts ", sizeof(msgbuf));
		local_irq_enable();
	}
	WARN(msgbuf[0], "initcall %pF returned with %s\n", fn, msgbuf);
```

```
        return ret;
}
```

(1)

```
        initcall_blacklist=  [KNL] Do not execute a comma-separated list of
                        initcall functions.  Useful for debugging built-in
                        modules and initcalls.
```

(2)

```
        initcall_debug  [KNL] Trace initcalls as they are executed.  Useful
                        for working out where the kernel is dying during
                        startup.
```

built-in driver的init function都是在do_initcalls()阶段执行的,也就是各个driver的probe() function在此时执行。

通过“initcall_debug" kernel parameter可以dump出driver初始化的顺序，对调试built-in driver的initialization很有利。

而"initcall_blacklist" kernel parameter则可以暂时禁止某个driver。因为不执行该driver的init function，就不会调用该driver的probe()，自然该driver也不工作了。

另外如果各个driver之间有依赖关系，那么通过安排不同的initcall level，可以建立正确的dependency.

for example,

GPIO即是普通device，同时也可作为interrupt controller,而某些device的interrupt line挂在GPIO的interrupt controller上，那么developer必须保证GPIO

先于该设备初始化，否则该设备初始化必然失败。因为一般request IRQ都在driver的probe()中，而如果GPIO driver还未初始化好，那么该device的driver request IRQ必然失败。