==========

----hal_arch.h----

==========

----STRUCTURE----

```
HAL_SavedRegisters{

    cyg_uint32    sp;          // SP (R13)（栈指针，保存在 R13 中，可以直接通过 R13 访问）
    cyg_uint32    vector;      // 中断向量号
    cyg_uint32    basepri;     // 中断屏蔽寄存器，所有 priority 值高于 basepri 的中断
                               // 都会被屏蔽，但当 basepri==0 时不屏蔽任何中断

    cyg_uint32    r4_11[8];    // 通用寄存器
    cyg_uint32    xlr;         // Exception return LR

    // The following are saved and restored automatically by the CPU
    // for exceptions or interrupts.

    cyg_uint32    r0;
    cyg_uint32    r1;
    cyg_uint32    r2;
    cyg_uint32    r3;
    cyg_uint32    r12;
    cyg_uint32    lr;
    cyg_uint32    pc;

    //程序状态寄存器，分为 APSR, IPSR, EPSR，但占位不重叠，因此可以通过一个 32 位 psr 统一访问
    cyg_uint32    psr {
        APSR {
            N;    // Negative
            Z;    // Zero
            C;    // Carry
            V;    // Overflow
            Q;    // Sticky Saturation
        }
        IPSR {
            isr_number; // 只是文档里面这么写，实际上就是中断向量号，0~255，占 9 位
        }
        EPSR {
            ICI/IT;      // 与 if then 指令相关，对程序本身没有影响，可以不考虑
            T;           // thumb 模式位
        }
    }
}
```

----FUNCTIONAL MACROS----

(1) HAL_LSBIT_INDEX(index, mask)

功能：执行后，index 的值是 mask 中不为 0 的最低位位数

(2) HAL_MSBIT_INDEX(index, mask)

功能：执行后，index 的值是 mask 中不为 0 的最高位位数

(3) HAL_THREAD_INIT_CONTEXT(_sparg_, _thread_, _entry_, _id_)

功能：执行后，在内存中构造出新的线程上下文结构

```
{
    r[12]   // r[0]=_thread; r[1]=_id; r[11]=_ep;
    sp      // sp = 结构创建完整后的栈顶指针
    pc      // pc = 线程的入口（实际上是执行时的第一条指令位置）
    basepri // basepri = 0; （不屏蔽任何中断）
}
```

(4) HAL_THREAD_SWITCH_CONTEXT(_fspptr_, _tspptr_)

功能：将寄存器中的值保存到_fspptr_中

将_tspptr_指向的 savedregisters 结构内容 load 到寄存器中

(5) HAL_THREAD_LOAD_CONTEXT(_tspptr_)

功能：将_tspptr_指向的 savedregisters 结构内容 load 到寄存器中

(6) CYGARC_HAL_GET_PC_REG(__regs,__val)

功能：将 pc 的值保存在__val 中

==========

----hal_intr.h-----

==========

----STRUCTURE----

```
//异常向量号
CYGNUM_HAL_VECTOR_STACK          0      // Reset stack pointer
CYGNUM_HAL_VECTOR_RESET          1      // Reset entry point
CYGNUM_HAL_VECTOR_NMI            2      // Non-Maskable Interrupt
CYGNUM_HAL_VECTOR_HARD_FAULT     3      // Hard fault
CYGNUM_HAL_VECTOR_MEMORY_MAN     4      // Memory management (M3)
CYGNUM_HAL_VECTOR_BUS_FAULT      5      // Bus Fault
CYGNUM_HAL_VECTOR_USAGE_FAULT    6      // Usage Fault
CYGNUM_HAL_VECTOR_RESERVED_07    7
CYGNUM_HAL_VECTOR_RESERVED_08    8
CYGNUM_HAL_VECTOR_RESERVED_09    9
CYGNUM_HAL_VECTOR_RESERVED_10   10
CYGNUM_HAL_VECTOR_SERVICE       11      // System service call
CYGNUM_HAL_VECTOR_DEBUG         12      // Debug monitor (M3)
CYGNUM_HAL_VECTOR_RESERVED_13   13
CYGNUM_HAL_VECTOR_PENDSV        14      // Pendable svc request
CYGNUM_HAL_VECTOR_SYS_TICK      15      // System timer tick
CYGNUM_HAL_VECTOR_EXTERNAL      16      // Base of external interrupts
```

```
//中断向量号 （0 对应异常向量号的 15）
CYGNUM_HAL_INTERRUPT_SYS_TICK    0
CYGNUM_HAL_INTERRUPT_EXTERNAL    1

//异常向量，给上层程序传递向量号时使用这些别称
CYGNUM_HAL_EXCEPTION_DATA_TLBMISS_ACCESS        CYGNUM_HAL_VECTOR_MEMORY_MAN
CYGNUM_HAL_EXCEPTION_CODE_TLBMISS_ACCESS        CYGNUM_HAL_VECTOR_MEMORY_MAN
CYGNUM_HAL_EXCEPTION_DATA_ACCESS                CYGNUM_HAL_VECTOR_BUS_FAULT
CYGNUM_HAL_EXCEPTION_CODE_ACCESS                CYGNUM_HAL_VECTOR_BUS_FAULT
CYGNUM_HAL_EXCEPTION_ILLEGAL_INSTRUCTION        CYGNUM_HAL_VECTOR_USAGE_FAULT
CYGNUM_HAL_EXCEPTION_DATA_UNALIGNED_ACCESS      CYGNUM_HAL_VECTOR_USAGE_FAULT
CYGNUM_HAL_EXCEPTION_INTERRUPT                  CYGNUM_HAL_VECTOR_SERVICE

//这只是一些常量
CYGNUM_HAL_EXCEPTION_MIN     CYGNUM_HAL_EXCEPTION_DATA_UNALIGNED_ACCESS
CYGNUM_HAL_EXCEPTION_MAX     CYGNUM_HAL_EXCEPTION_INTERRUPT
CYGNUM_HAL_EXCEPTION_COUNT   (CYGNUM_HAL_EXCEPTION_MAX - \
                                    CYGNUM_HAL_EXCEPTION_MIN + 1)
```

中断向量表的数据结构：

```
//CYGNUM_HAL_ISR_COUNT：中断向量个数，与平台相关，可直接当成常量抽象
hal_interrupt_handlers[CYGNUM_HAL_ISR_COUNT]
hal_interrupt_data[CYGNUM_HAL_ISR_COUNT]
hal_interrupt_objects[CYGNUM_HAL_ISR_COUNT]
//中断向量表
hal_vsr_table[CYGNUM_HAL_VSR_COUNT]
```

----FUNCTIONAL MACROS----

(1) HAL_DISABLE_INTERRUPTS(_old_)
功能：更新 CPSR 寄存器，使其与当前设置保持同步，将原 CPSR 输出到_old_

(2) HAL_ENABLE_INTERRUPTS
功能：更新 CPSR 寄存器，清除其中的 DISABLE 标志

(3) HAL_RESTORE_INTERRUPTS(_old_)
功能：恢复_old_表示的中断允许状态，更新 CPSR 寄存器

(4) HAL_QUERY_INTERRUPTS(_old_)
功能：查询当前中断允许状态，保存在_old_中

(5) HAL_TRANSLATE_VECTOR(_vector_,_index_) (_index_) = (_vector_)
（这个不知道要干啥。。）

(6) HAL_INTERRUPT_IN_USE(_vector_, _state_)
功能：执行后，若_vector_对应的 isr 是 default_isr，则_state_为 0,否则为 1

(7) HAL_INTERRUPT_ATTACH(_vector_, _isr_, _data_, _object_)
功能：如果_vector_对应的 isr 仍然是 default_isr，则将新的 isr 和与其对应的 data/object 存入对应的中断向量表数据结构。

(8) HAL_INTERRUPT_DETACH(_vector_, _isr_)

功能：如果_vector_对应的 isr 与_isr_相等，则将_vector_对应的 isr 清空，设为 default_isr

(9) HAL_VSR_GET(_vector_, _pvsr_)

功能：将_vector_对应的 vsr 指针存入_pvsr_中，_pvsr_是个指针

(10) HAL_VSR_SET(_vector_, _vsr_, _poldvsr_)

功能：首先将_vector_原来对应的 vsr 保存在_poldvsr_指向的空间；
然后将_vector_对应的 vsr 设置为新的_vsr_

(11) HAL_VSR_SET_TO_ECOS_HANDLER(__vector, __poldvsr)

功能：将__vector 对应的 vsr 设置为 hal_default_exception_vsr(__vector<15)或 hal_default_interrupt_vsr(__vector>=15)，同时将原 vsr 保存在另一个新建的变量__poldvsr2 中。

(12) HAL_CLOCK_INITIALIZE(_period_)

功能：将下一次发生中断的时间设置为 当前时间 + _period_

(13) HAL_CLOCK_RESET(_vec_, _period_)

功能：将下一次发生中断的时间重新初始化为 当前时间 + _period_

(14) HAL_CLOCK_READ(_pvalue_)

功能：读取上一次时钟中断到现在为止的 cycle 数，存入_pvalue_指向的地址。

(15) HAL_CLOCK_LATENCY(_pvalue_)

功能：= HAL_CLOCK_READ((cyg_uint32 *)_pvalue_)