Решения проблемы захвата коротких входных сигналов (~100 мс)

Проблема

Система не успевает захватить изменение входного сигнала длительностью около 100 мс при текущем периоде опроса READ_DATA() = 50 мс. Возможны пропуски сигнала, если он появляется и исчезает между циклами опроса.

Анализ текущей ситуации

- **Период опроса**: 50 мс (Timer2 -> F50 -> READ_DATA)
- **Длительность сигнала**: ~100 мс
- **Вероятность пропуска**: ~25% (если сигнал не синхронизирован с опросом)

Вариант 1: Увеличение частоты опроса READ_DATA()

Описание

Уменьшить период вызова READ_DATA() с 50 мс до 20-25 мс для гарантированного захвата сигнала.

Реализация

```
// В Timer.c, функция IRQ_timer2()
// Изменить настройку Timer2 для более частых прерываний

#define START_TIMER2   0x0800     // Вместо 0x03AA (43 мс) -> ~20 мс

// Или добавить дополнительный вызов в F100MS
if (F_TIMEBITS.F100MS)
{
    // Первый вызов на 0 мс
    if (!flags.FS_TEST) READ_DATA();

    delay_ms(25);  // Ждем 25 мс

    // Второй вызов на 25 мс
    if (!flags.FS_TEST) READ_DATA();

    // Остальная логика на 50 мс и 75 мс...
}
```

Преимущества

- ✅ Простая реализация
- ✅ Гарантированный захват сигналов ≥50 мс
- ✅ Минимальные изменения в коде

Недостатки

- ❌ Увеличение нагрузки на процессор
- ❌ Больше обращений к SPI регистрам
- ❌ Может влиять на точность других таймеров

Вариант 2: Аппаратная защелка (Hardware Latch)

Описание

Использовать аппаратные защелки в FPGA/CPLD для фиксации коротких импульсов до их программного считывания.

Реализация

```
// Добавить новые биты в регистры для работы с защелками
#define RD_LATCH        0x05    // Новый регистр защелок
#define WR_LATCH_RESET  0x0E    // Регистр сброса защелок

// В READ_DATA() добавить чтение защелок
void READ_DATA()
{
    unsigned char latch_buffer;

    // Существующий код...

    // Чтение аппаратных защелок
    latch_buffer = perform_actionR(RD_LATCH);
```

```
    // Обработка защелкнутых сигналов
    if (latch_buffer & (1 << 0)) {
        // Обработка защелкнутого сигнала 1
        flags.SIGNAL1_LATCHED = 1;
    }
    if (latch_buffer & (1 << 1)) {
        // Обработка защелкнутого сигнала 2
        flags.SIGNAL2_LATCHED = 1;
    }

    // Сброс защелок после считывания
    if (latch_buffer != 0) {
        perform_actionW(WR_LATCH_RESET, 0xFF);
        delay_ms(1);
        perform_actionW(WR_LATCH_RESET, 0x00);
    }
}
```

Преимущества

- ✅ Гарантированный захват любых импульсов
- ✅ Минимальная нагрузка на процессор
- ✅ Высокая надежность

Недостатки

- ❌ Требует изменения аппаратуры (FPGA/CPLD)
- ❌ Дополнительные регистры управления
- ❌ Усложнение схемы

Вариант 3: Программная защелка с накоплением

Описание

Создать программные защелки, которые накапливают изменения между циклами опроса.

Реализация

```
// В файле globals.h добавить структуру защелок
struct InputLatches {
    unsigned char previous_state[5];  // RD0-RD4
    unsigned char change_detected[5]; // Флаги изменений
    unsigned char latch_flags;        // Защелкнутые состояния
};

extern volatile struct InputLatches input_latches;

// Модифицированная READ_DATA()
void READ_DATA()
{
    unsigned char buffer, changes;

    // RD0 - с детектированием изменений
    buffer = perform_actionR(RD0);
    changes = buffer ^ input_latches.previous_state[0];  // XOR для поиска изменений

    if (changes) {
        input_latches.change_detected[0] |= changes;  // Накопление изменений
        input_latches.previous_state[0] = buffer;
    }

    // Обработка накопленных изменений
    if (input_latches.change_detected[0]) {
        // Обработка всех изменений, произошедших с последнего цикла
        process_accumulated_changes();
        input_latches.change_detected[0] = 0;  // Сброс после обработки
    }

    // Аналогично для RD1-RD4...
}

void process_accumulated_changes()
{
    // Анализ накопленных изменений и установка соответствующих флагов
    if (input_latches.change_detected[0] & (1 << 0)) {
        flags.FPIT_CKP_CHANGED = 1;
    }
    // И т.д. для всех важных сигналов
}
```

Преимущества

- ✅ Не требует изменения аппаратуры
- ✅ Детектирует все изменения
- ✅ Гибкая настройка чувствительности

Недостатки

- ❌ Дополнительная память для хранения состояний
- ❌ Усложнение логики обработки
- ❌ Может накапливать "ложные" изменения от помех

Вариант 4: Прерывания по изменению входа

Описание

Использовать аппаратные прерывания для немедленной реакции на изменение критичных входных сигналов.

Реализация

```
// Настройка прерывания по внешнему сигналу
#pragma interrupt 6  IRQ_external_input

void init_external_interrupts()
{
    unsigned char tmp;

    BeginHWindow
    SetHWindow15
    INT_MASK1 |= 0x40;  // Включить внешнее прерывание
    SetHWindow0

    // Настройка типа прерывания (по фронту/спаду)
    IOC2 |= 0x01;  // Прерывание по изменению
    EndHWindow
}

// Обработчик прерывания
void IRQ_external_input()
{
    unsigned char tmp;

    __DI();

    // Немедленное чтение критичного сигнала
    critical_signal_state = perform_actionR(RD_CRITICAL);
    flags.CRITICAL_SIGNAL_CHANGED = 1;

    // Установка флага для обработки в основном цикле
    flags.IMMEDIATE_READ_REQUIRED = 1;

    __EI();
}

// В основном цикле
if (flags.IMMEDIATE_READ_REQUIRED) {
    READ_DATA();  // Внеплановое чтение
    flags.IMMEDIATE_READ_REQUIRED = 0;
}
```

Преимущества

- ✅ Мгновенная реакция на критичные сигналы
- ✅ Минимальное время отклика
- ✅ Не влияет на основные таймеры

Недостатки

- ❌ Требует свободные линии прерываний
- ❌ Дополнительная настройка аппаратуры
- ❌ Возможны проблемы с помехами

Вариант 5: Комбинированный подход с приоритизацией

Описание

Использовать разную частоту опроса для разных типов сигналов в зависимости от их критичности.

Реализация

```
// Структура для управления приоритетами опроса
struct ScanPriorities {
    unsigned char fast_scan_counter;
    unsigned char medium_scan_counter;
    unsigned char slow_scan_counter;
};

void READ_DATA_PRIORITIZED()
{
    static struct ScanPriorities scan = {0, 0, 0};

    // БЫСТРОЕ СКАНИРОВАНИЕ (каждые 25 мс) - критичные сигналы
    scan.fast_scan_counter++;
    if (scan.fast_scan_counter >= 1) {  // 25 мс при F50
        scan.fast_scan_counter = 0;

        // Критичные кнопки и режимы
        unsigned char buffer1 = perform_actionR(RD1);
        flags.FS_ATT_UP = (buffer1 & (1 << 4)) ? 1 : 0;
        flags.FS_ATT_DOWN = (buffer1 & (1 << 5)) ? 0 : 1;
        flags.F_DU = (perform_actionR(RD2) & (1 << 0)) ? 1 : 0;
    }

    // СРЕДНЕЕ СКАНИРОВАНИЕ (каждые 50 мс) - обычные сигналы
    scan.medium_scan_counter++;
    if (scan.medium_scan_counter >= 2) {  // 50 мс при F50
        scan.medium_scan_counter = 0;

        // Переключатели питания
        unsigned char buffer0 = perform_actionR(RD0);
        if (flags.F_DU == 0) {
            flags.FPIT_CKP = (buffer0 & (1 << 0)) ? 1 : 0;
            flags.FPIT_PDP = (buffer0 & (1 << 1)) ? 1 : 0;
        }
    }

    // МЕДЛЕННОЕ СКАНИРОВАНИЕ (каждые 100 мс) - некритичные сигналы
    scan.slow_scan_counter++;
    if (scan.slow_scan_counter >= 4) {  // 100 мс при F50
        scan.slow_scan_counter = 0;

        // Системные сигналы
        unsigned char buffer4 = perform_actionR(RD4);
        flags.FS_TEST = (buffer4 & (1 << 7)) ? 1 : 0;
    }
}
```

Преимущества

- ✅ Оптимизация нагрузки процессора
- ✅ Высокая частота для критичных сигналов
- ✅ Гибкая настройка приоритетов
- ✅ Минимальные изменения аппаратуры

Недостатки

- ❌ Усложнение логики программы
- ❌ Больше памяти для счетчиков
- ❌ Необходимость классификации сигналов по важности

Рекомендации по выбору решения

Для быстрого внедрения: **Вариант 1** (увеличение частоты)

- Минимальные изменения в коде
- Быстрое решение проблемы

Для надежной работы: **Вариант 2** (аппаратные защелки)

- Требует изменения FPGA, но дает 100% надежность

Для гибкости: **Вариант 5** (приоритизация)

- Оптимальный баланс производительности и надежности

Для критичных систем: **Вариант 4** (прерывания)

- Максимальная скорость реакции на важные события

Тестирование решений

Для любого выбранного варианта рекомендуется:

1. **Тестирование с осциллографом** - проверка реального времени захвата
2. **Стресс-тесты** - множественные короткие импульсы
3. **Проверка синхронизации** - несинхронные сигналы
4. **Тест на помехи** - ложные срабатывания
5. **Измерение нагрузки CPU** - влияние на общую производительность