Designing Virtual Peripherals for the SX

Programming Considerations and Standards

by Craig Webb

Introduction

Virtual Peripherals (VP's) are modules of software code which provide functions that would otherwise have to be implemented with on-chip hardware or external components. The SX device provides enhanced throughput to implement efficient VP's. A number of VP's already exist, and more are currently under development. Many standard microcontroller functions can be incorporated into a design directly via software with few or no added external components simply by selecting and combining existing blocks of program code to create versatile multiple-function designs. An example of such an implementation is shown later in this document. This document provides a fixed set of standard virtual peripheral guidelines intended to help in generating, modifying, and combining VP's.

Virtual Peripheral Types

Virtual peripherals are code modules that can be implemented either as interrupt routines (the most common), or ogram subroutines, or as stand-alone portions of in-line main program code. The latter two methods of implementation are preferred to maintain program modularity. The whole concept behind VP design is to keep the code simple, fast, and short. Also each module must be kept independent from other modules as much as possible to allow simple cut-and-paste for adding or removing VP's.

VP's which are implemented as **interrupt routines** are the most powerful type since they are able to function relatively transparent in the background of the main program (i.e. the main program code does not have to be occupied by jump or call operations). Depending on the interrupt source, such modules may be of two types, either internal RTCC rollover or external Multi-Input-Wake-Up (MIWU). Both types are included as portions of the interrupt code block of the program source code. RTCC roll-over modules are accessed at a regular intervals depending on the value loaded into the W register by the RETIW instruction upon return from the interrupt service routine. In cases where a selectable interrupt period is not required, the RETI instruction may be used to return from the interrupt. When RETIW is used, the period of the interval depends also upon the oscillator frequency being used, the prescaler setting (selected by the OPTION register), and whether the SX is running in turbo or normal mode.

period (sec) = mode * prescaler * RETIW value / osc. frequency, where mode=1 (turbo) or =4 (normal)

In terms of timing, any given VP may be one of two types: (a) fixed execution rate, or (b) variable execution rate. Variable rate VP's are those which include branch statements (jumps and/or skips). Most applications do not require a fixed execution rate for externally generated (MIWU), subroutines, or in-line VP's. For RTCC roll-over VP's however, the ideal is often fixed execution rate. This is because such VP's depend upon a precise regular execution interval, and also because they may need to be combined with or precede other such VP's. The trade-off is that variable execution-rate VP's often require less memory and because they have possible states where execution time is reduced, thus providing more CPU time for other VP's and for main program code execution.

The most logical manner to combine VP's is to keep timing-sensitive modules at the beginning of the interrupt sequence and develop all (except possibly the last one) fixed execution rate routines. Those routines which are not as timing critical can then follow. Any routine which need to be accessed only occasionally (once / msec, for example) may be combined with a software time clock VP and even with an execution "path switcher" VP routine as in the code example shown in this document (Fig. 1).

In order for an RTCC roll-over interrupt to maintain its regular interval, the value loaded into W upon termination of the interrupt subroutine (hereafter referred to as the RETIW value) must be larger than the maximum number of execution cycles of the interrupt sequence in its longest possible state, that is to say the state where each individual VP runs through its slowest path. To be exact, it should be equal to this value plus three (MIWU interrupt takes 5 cycles) instruction cycles for the initial interrupt processing plus the additional four instruction cycles required to load W with the appropriate value and then exit the interrupt. If for some reason, execution of the interrupt takes longer than the calculated total just mentioned, the interrupt skips a trigger cycle. In theory, the RTCC can be monitored at the end of the interrupt to see if it is near or past the roll-over point and appropriate compensation can be applied so the roll-over point and hence the interrupt in question is not missed. In practice however, this would require further interrupt processing time on each pass to perfrom such calculations and compensation. The percentage of CPU time which is consumed by the rollover interrupt sequence can be calculated as follows:

$$\%\ \text{CPU time for internal interrupt} = \frac{\text{total execution time of all interrupt code + 3 (initial processing)}}{\text{RETIW value} * \text{RTCC prescaler}}$$

If both MIWU and RTCC rollover interrupts are to be combined in a design, then one of two choices must be made. Either (a) interrupts (of the other type) that occur as one type is processing will be ignored, or (b) extra processing will be added to the interrupt routine in order to attempt to process all interrupts. The second option will require somewhat more involved coding since the SX only allows one level of interrupt processing (i.e. interrupts are not queued) and will therefore consume a higher percentage of CPU time for each type of interrupt. It requires that the register which monitors external interrupts (*WKPND_B*) be checked at the beginning of the interrupt to see which type of interrupt occurred and then control be passed to the appropriate routine. Ideally, a small code section should be tacked at the end of each interrupt routine which checks whether the other type of interrupt has been missed during the current interrupt sequence execution, in which case execution is then passed to the other interrupt processing sequence before the RETI or RETIW (return from interrupt) instruction is executed. There may also be the special case when the combined execution time of the external+internal interrupts is longer than the period of the internal interrupt sequence. In such a case, the internal interrupt will be missed if control is passed from the external routine to the internal routine before returning from the interrupt. This may be difficult to avoid, but if it is known that the combined execution time is longer than the internal interrupt period, then the external interrupt which detects the RTCC rollover (prior to the rollover) can make an adustment to allow servicing an additional internal interrupt.

All interrupt processing begins at 0h, though a jump statement with the interrupt vector may be placed there to direct interrupt execution to another location. Although not necessary, this can be useful because subroutines and program data intended to be accessed by the IREAD command must be placed within the lower half of a memory page.

Subroutines and in-line VP's are simpler to implement. They require the main program code pass control directly to them. The two main benefits of subroutines over in-line modules are that the same segment of code can be simply accessed from multiple different calling locations. In-line VP's are most useful when the code segment is short and used only in one location within the program. Such VP's eliminate the need for CALL and RET instructions (and the associated six instruction cycles in turbo mode) at the expense of modularity.

It is recommended that subroutines which normally end with RET instruction, be terminated with a RETP (return and reset page bits accordingly) so that they may be called from anywhere in program memory without requiring additional PAGE instructions after long calls (although required *prior* to long calls).

Programming Standards

1. Overall Source Code Formatting

- **Title/header block (commented lines only):** Includes the program title, a functional description of the code, author's name(s), a copyright notice (if applicable), the date the code was written, the number of bytes, and dates and descriptions of any modifications to the code since first completed.
- **Device block:** Includes chip type (e.g. SX28AC), oscillator mode and value (e.g. 50Mhz crystal), FUSEX options used (turbo, carryx, optionx, stackx, sync, wdte, cp), and compiler options (if applicable), such as PAGES<n>, and program ID.
- **Reset vector:** Provides address where execution begins upon power-up or after a reset
- **Port assignment block:** Specifies which port pins are used and their corresponding variable names
- **Variable assignment block:** Specifies all program register and bit variables, and program constants (as a suggested guideline, all register and address variables use the 'EQU' definition and all program numerical constants use the '=' definition statement).
- **Interrupt vector (optional):** A jump instruction at location 0h which jumps to the interrupt code
- **Interrupt code block:** The actual interrupt code module (begins at 0h if no interrupt vector provided)
- **Program data block (optional):** Program data to be used by the IREAD command is defined using the DW compiler command (must reside within lower half of memory page)
- **Subroutine block (optional):** Program subroutines (modules to be CALLed) are grouped together, each one terminating with a RET/RETP/RETW statement.
- **Initialization code block:** This is usually the location pointed to by the reset vector, where non-interrupt and non-subroutine program code begins. This block includes, but is not limited to the following:
    - Port pin input/output status and initial level (for outputs) is set-up.
    - Various mode options are set such as external interrupt status, CMOS/TTL levels, etc.
    - Register/bit variables are cleared and/or set to their initial values.
    - The OPTION register is initialized.
    - Any other one-time start-up code.
- **Main program code block:** This is the main in-line program code, terminated with an END statement.

2. Virtual peripherals: formatting and set up of module source code

- **Header block (commented):** Includes but is not limited to
    - VP's title and functional description.
    - Registers, flags and port pins which are used as inputs for the VP (including W, if applicable).
    - Registers, flags and port pins which may change during execution (including W, if applicable).
    - Registers, flags and port pins which act as outputs from the VP (including W, if applicable). Outputs from subroutine VP's held in W are loaded by leaving the subroutine with the RETW instruction.
    - Number of bytes of VP (i.e. the size).
    - Execution time of the VP, including whether VP is of fixed or variable execution time. In VP's where skips or jumps are used, all possible execution times through various paths must be documented. If this becomes too involved, at least the longest possible execution time must be documemted for calculation purposes.
    - Number of levels of stack used (when >0). Note that the interrupt sequence has its own stack separate from the eight level CALL stack.
    - Any special FUSEX needs should be mentioned. VP's with different FUSEX requirements usually cannot be combined without modification.
    - Variables used mainly by a specific VP should be allotted to their own register bank. Register banks may be shared for improved efficiency. However, for an easy to read program, separate names must be defined for a bank when shared by multiple VP's (see the clock & bank example).
- **VP modularity and performance**
    - VP's should be kept as independent from each other as possible. Some VP's however are intended to be combined with others in order to reduce code size and timing. The A/D and PWM VP's are the examples. All modidifications to make each VP stand alone, must be documented.
    - Changes in port status or mode options for port variables which will be used by other VP's should be reset before exiting
    - In general, code execution rate should be a priority over code size. Fixed execution rate has priority in situations where a timing critical interrupt VP is combined with another timing critical VP.

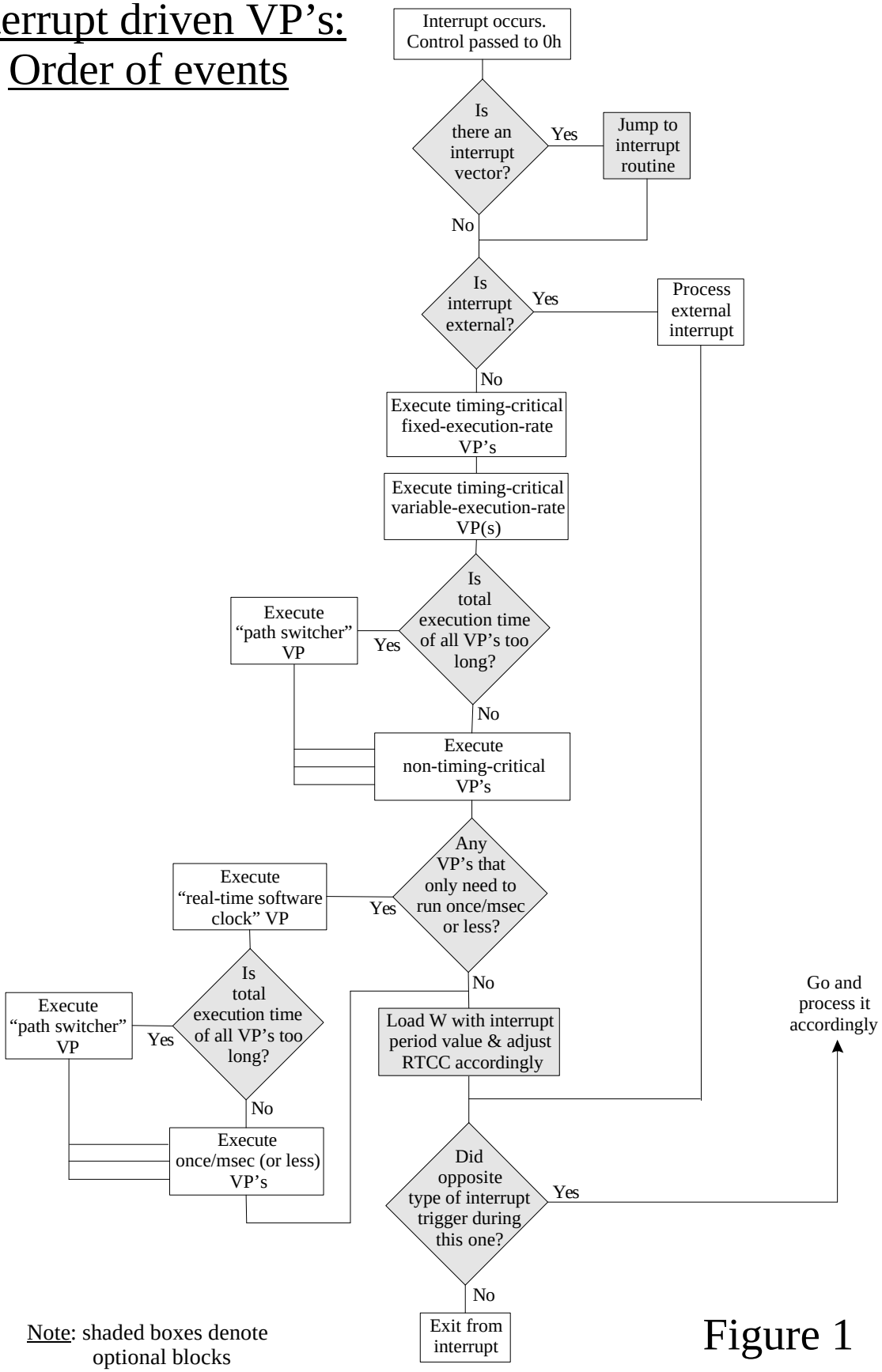
3. General SX programming suggestions

- The STACKX fusex option should always be used
- The CARRYX option may help to reduce execution time and code size when numerous math calculations are being done involving larger than 8 bit values. Care should be taken to make sure this is the most effective overall solution since all VP's and program code must then take this into account.
- Program registers 07-0Fh (on SX28AC, register 07h is not available since it is used for Port C) should be kept as general purpose registers (common to multiple VP's) since they may be accessed by multiple routines without the need for bank switching. Example functions for these include a flag register(s), temporary storage register(s), commonly used registers, etc.
- Where possible, subroutines should be kept in the same page of memory as their calling locations
- If a number of internal interrupt VP's are being combined which require an ongoing count of the number of passes through the interrupt routine, a single general purpose register can be set up for this and incremented at the start of each interrupt cycle, or, for VP modules which need to run in the background

(i.e. internal interrupt), but which only need to be executed every millisecond or less, a software time clock VP as demonstrated in attached code example (also available from Scenix' web site) can be added.

- When multiple internal interrupt VP's which are not timing critical are being combined, a path switcher VP as demonstrated in attached code example (available from Scenix' web site) may be used.

Interrupt driven VP's: Order of events



Note: shaded boxes denote optional blocks

Figure 1