Interface Synthesis for Embedded Applications in a CoDesign Environment

Anupam Basu * R. S. Mitra
Dept. of Computer Sc. & Engg.
I.I.T. Kharagpur
India

P. Marwedel
Informatik XII
University of Dortmund
Germany

Abstract

In embedded systems, programmable peripherals are often coupled with the main programmable processor to achieve the desired functionality. Interfacing such peripherals with the processor qualifies as an important task of hardware software codesign. In this paper, three important aspects of such interfacing, namely, the allocation of addresses to the devices, allocation of device drivers, and approaches to handle events and transitions have been discussed. The proposed approaches have been incorporated in a codesign system MICKEY. The paper includes a number of examples, taken from results synthesized by MICKEY, to illustrate the ideas.

1 Introduction

The software in an embedded system may be loosely said to consist of two parts: the application software and the system software, where the former achieves the functionalities delegated to software (by hardware-software partitioning), and the latter provides the interface between the former and the hardware counterpart of the overall system. Thus, the objective of the system software is to (a) implement the desired reactive behavior, by initiating state changes in response to incoming events, and (b) drive the accessory hardware devices of the system, in order that they perform the functionalities delegated to them. In this paper, we propose techniques for developing event-handlers and device-drivers for embedded systems.

Frameworks for automated design of microcontroller based systems have been proposed in [1, 2, 3, 4]. The thrust of recent research in hardware software interfacing has been in the field of synthesizing interfaces for devices. An early work on the development of interfaces for available devices [5] described techniques for implementing the interface by hardware elements alone. A later work [6] extended this technique to include software implementations of the interface as well, through a manual partitioning of the interface functionalities into hardware and software implementable partitions. This issue has also been addressed in [7].

In embedded systems, events are either polled by software or treated as *interrupts* to the normal processing of the microcontroller. In the latter case, the interrupt service code (ISC) generates the necessary response to the incoming event. In many cases, the interrupt method is the preferred way of handling events, since the microcontroller is then free to perform other tasks. Hence, the development of the ISCs is a key issue in the development of microcontroller based systems. However, to our knowledge, no systematic methodology has yet been proposed for the automated development of ISCs.

The work presented in this paper is a part of the system MICKEY- a system for hardware software codesign of microprocessor based systems [8]. In MICKEY, we accept the user specification in the form of statecharts and refine them , using rules, to arrive at a control and data flow graph (CDFG). The leaf level elements of the CDFG, thus obtained designate **primitive functions (*pf*s)**, for which either hardware or software implementations are known to exist in the design library. The CDFG is then subjected to hardware software partitioning, which allocates implementations (either hardware or software), to the primitive functions. A schedule of the op-

*Presently at the University of Dortmund on Humboldt Fellowsh ip erations is also obtained in the course of partitioning. The partitioning is achieved using constraint satisfaction techniques, so that the overall cost of the system is minimized and the time constraints are met. This paper concentrates on the interface design task, that follows hardware software partitioning. This task synthesizes the interface between the CDFG and the allocated implementations, and thus addresses another important aspect of hardware software codesign.

Since, during the hardware software partitioning phase, some of the functionalities may have been mapped to programmable devices, and some tasks may have been assigned to be executed by the processor in an asynchronous mode (interrupt driven), the aspects considered in this paper for interface synthesis are i) Allocation of nonconflicting addresses to the devices placed on the system bus; ii) Refinement of the CDFG, to accommodate event-based and conditional transitions; and iii) Refinement of the CDFG, to accommodate device drivers.

We assume uniprocessor environment and do not allow any software concurrency for the purpose of this paper.

2 Address Allocation

The selected microprocessor defines the available address space and also imposes restrictions on how the devices have to be mapped to it. For example, for Intel 8085, ROM addresses should start from 0, whereas for Motorola 6800, the start address of ROM is not specified – instead, the ROM is constrained to end at address FFFF_{16}. The address allocation subtask assigns nonconflicting addresses to the devices that are to be placed on the system bus, such that these address space constraints are satisfied.

The algorithm for address allocation is shown below. The algorithm first determines the memory address space partitioning. For some devices, like the Intel 8155, which require both memory address space and I/O address space, allocation of the memory address space will impose a constraint on the I/O address allocation. In such cases, I/O space is also allocated for that device, immediately after memory address allocation. Then, the partitioning algorithm is applied to the I/O space keeping in mind the already allocated address ranges so that no address space conflict occurs.

allocate_address
{
1. Partition memory space
2. Allocate memory blocks to constrained devices
3. Allocate memory blocks to remaining memory-space devices
4. Partition I/O space
5. Allocate I/O blocks to remaining I/O-space devices
}

partition_address_space
{
  s = total address space
  n = number of devices
  for i=1 **to** n **do**
  { $blox_i = 1$; $size_i$ = address space of device i }
  while ($n \times s \leq$ total address space) **do**
  {
    for all devices i having $s/2 < size_i \leq s$ do
    {
      $n = n + blox_i$;
      $blox_i = blox_i \times 2$;
      $size_i = size_i \div 2$;
    }
    if (s == 1) terminate with failure
    $s = s \div 2$
  }
  /* Each device (i) can now be allocated an address space of size ($blox_i \times s$). */
}

Example 1 As an example, consider the address allocation for a 8085-based system having the following hardware devices which need a memory or I/O address space. The results are given in Table 1. All I/O devices are connected in I/O mapped I/O mode.
1. A ROM chip - 2716
2. An ADC chip - AD574
3. A timer chip - 8253
4. A general purpose I/O chip - 8155

□

For the peripheral devices that we have encountered so far, we have found that backtracking can be avoided in the address allocation steps, by considering the most constrained devices first. In general, however, this may not be the case, and backtracking may result in [6] when a wider range of devices are considered.

Memory Space		
Device	Reqd. Space	Alloc. Space
2716	800_{16}	0000_{16} - $7FFF_{16}$
8155	100_{16}	8000_{16} - $FFFF_{16}$

I/O Space		
Device	Reqd. Space	Alloc. Space
8253	4_{16}	0_{16} - $3F_{16}$
AD574	1_{16}	40_{16} - $7F_{16}$
8155	8_{16}	80_{16} - FF_{16}

Table 1: Addresses allocated for an example problem

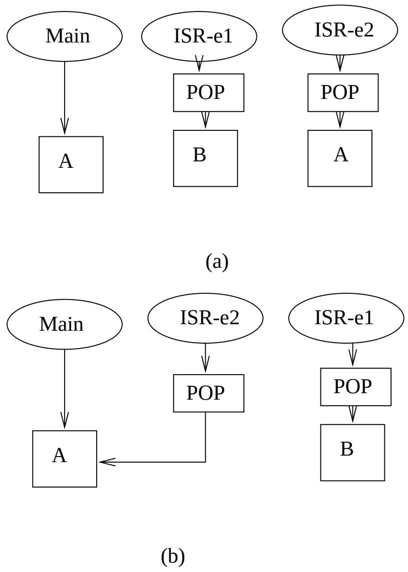

Figure 1: Implementing *event-transitions* by *jumps*

3 Handling Transitions

This subtask refines the CDFG, by transforming the event based transitions and conditional transitions into their detailed implementations. Event based transitions are characteristic of reactive systems, and thus this subtask forms a core of the synthesis of software for microprocessor based systems.

One way of implementing reactive systems is to have a single *forever-loop* in the *main* process and one subsidiary process for each *event*, and implement the flow of control by *call-and-return*. In this implementation, as eventsi (say *e1* and *e2* keep flowing in, the system alternates between the ISR for *e1* and the ISR for *e2*, and the *main* process becomes a dummy after the first occurrence of *e1*. Since the alteration is implemented as a *call-and-return*, the system will ultimately malfunction due to stack overflow. MICKEY alleviates this problem by adopting the method depicted in Fig 1(a). Here the return addresses are popped out of the stack, preventing the overflow. The duplicate codes of activity A can be removed by making a *jump* from *ISR-e2* to the *main* process (see Fig 1(b))

However, for concurrent processes a different strategy is used in MICKEY. Consider two processes (A,B) and (C,D) operating concurrently (Fig 2(a)). Since we are considering uniprocessor target systems and no coroutines, A, B, C, and D cannot all be simultaneously implemented by software. Let us assume that C and D are implemented by hardware which are started and stopped appropriately by the microprocessor. Hence, the modified process is as shown in Fig 2(b). Here, the atomic actions for starting and stopping the hardware devices have been attached along with the respective transition arcs, and the modules *wait::1* and *wait::2* are instances of the *pf*s *wait-for-event*. For example, *wait::1* is waiting for an external event *e3*. On this event, a transition to *wait::2* will be made, after stopping C and initiating D. The method adopted in MICKEY for implementing such concurrent processes is to implement each *wait* of the hardware implementable process as a *Return*, thus returning control to the other process. The ISRs for the events of these hardware implementable processes do not pop the stack, and since there is a corresponding *return* at the end, stack size does not grow indefinitely. Hence, the implementation is as shown in Fig 3.

If both the processes are allocated hardware implementations, possibly in order to satisfy timing constraints, as shown in Fig 4(a), then this method would derive the implementation shown in Fig 4(b). Here, the *wait* of the *Main* CDFG is not implemented as a *Return*, but as an explicit *wait-for-event*.

The approach of using different schemes for software and hardware implementable processes as well as for sequential and concurrent processes results in relatively simpler CDFGs. These strategies have been implemented in MICKEY as transformation rules.

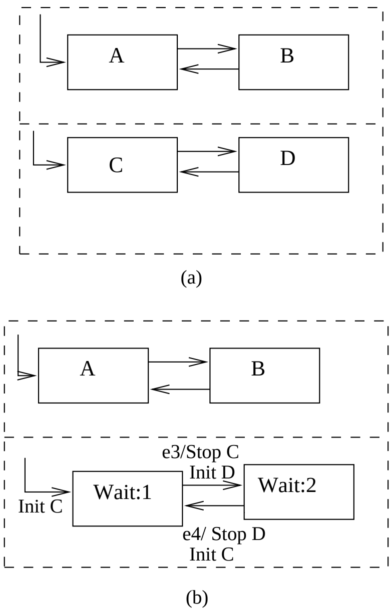

Figure 2: Concurrent processes with *event-transitions*

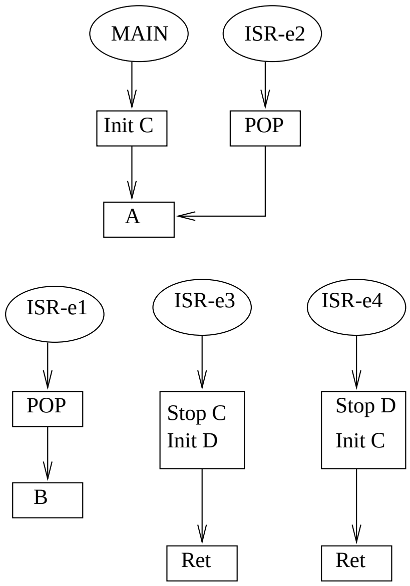

Figure 3: Implementation of *event-transitions* in concurrent processes by *call-and-return*

In the above discussion, we have considered transitions which are fired on the occurrence of an event. Guard conditions can also be associated with the transitions. Moreover, some events may have to be disabled or enabled depending on the current state. If an event causes several transitions, they have to be taken care of. Conditions may also be associated with transitions without any associated event. In such cases the condition has to be made explicit on the CDFG during refinement. The refinement rules in MICKEY explicitly handle all these different situations.

4 Refinement for Device Drivers

A software may need to interact with hardware devices for device initialization , reading/writing values from/to a device, and stopping a device. In general a hardware device may require a sequence of instructions for initialization, data input, data output, and halting. For simple devices like the ADC chip 7574, only an `IN` instruction is required at the allocated address to read the relevant data from the device. However for programmable devices, such as the Intel 8259 Interrupt Controller, a sequence of instructions is required to initialize the device to the required mode of operation. Further, for implementations that require additional data transformations, such as for implementing a *Counter* by a timer of the Intel 8155 chip, these computations have to be performed after the data is accessed from the device.

Fig 5(a) shows the CDFG refinement for driving hardware devices that execute till a request for termination. Here, the hardware implementable state (Y) is replaced by actions for starting and stopping the device, and a *wait* state. Such refinements are needed for functions such as a square wave generator, or a counter. The refinement for functions that terminate on their own, such as a timer, require a different refinement is shown in Fig 5(b), where the hardware device issues an event indicating its termination, and this event is used to make the necessary transition.

The further refinement of the device's commands (i.e. start and stop) is dependent on the specific implementation that has been selected. These implementation specific refinements also modify the data flows of the CDFG. An example is shown below for the implementation of a *Counter* by the Intel 8253.

Example 2 Consider the implementation of an up-counter by an Intel 8253, in Mode 0. The counter (see Fig 6(a)) is initiated by a *pf*, say X, on the event *e1*. On the event *e2*, the counter

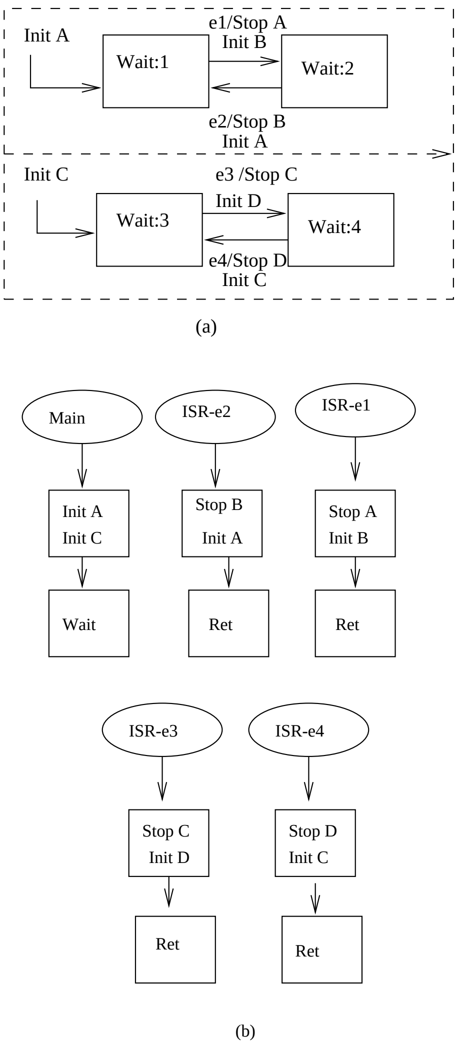

Figure 4: Special case: All concurrent processes implemented by hardware

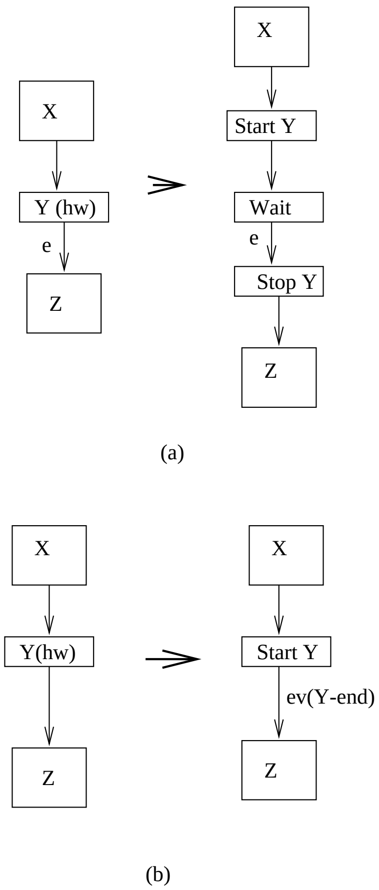

Figure 5: Refinements for hardware implementations

is stopped and control passes to another *pf*, say Y. The counter scans a pulse stream generated by a *pf*, say A. After the counter is stopped, the count data (i.e. the number of pulses in the pulse stream, in the duration when the counter was active) is passed to a *pf*, say B.

The result of applying the refinement step of Fig 5(a) on the CDFG of Fig 6(a), is shown in Fig 6(b). The next refinement, shown in Fig Fig 6(c), is for expanding the *start* and *stop* actions and for modifying the data flows. The *start* action of the *Counter* involves programming the 8253 in Mode 0, and sending initial data to it. (The latter is required in order to implement the up-counter by the 8253's down-counter.) The *stop* action sends a *Latch* command to the 8253, reads the down-count data, and stores the data after interpreting it appropriately (to form the corresponding up-count data).

The input data flow is simply a connection of the output of A to the 8253's CLK input. The output data flow is broken, and B now takes input from the memory location that stores the count data. □

5 Conclusion

In this paper, we have presented methodologies for addressing three important aspects of interface synthesis during hardware software cosynthesis of embedded systems. They are allocation of nonconflicting addresses to the devices, interfa ced with the microprocessor system, accommodating event based and conditional transitions by refinement of the Control and Data Flow Graph

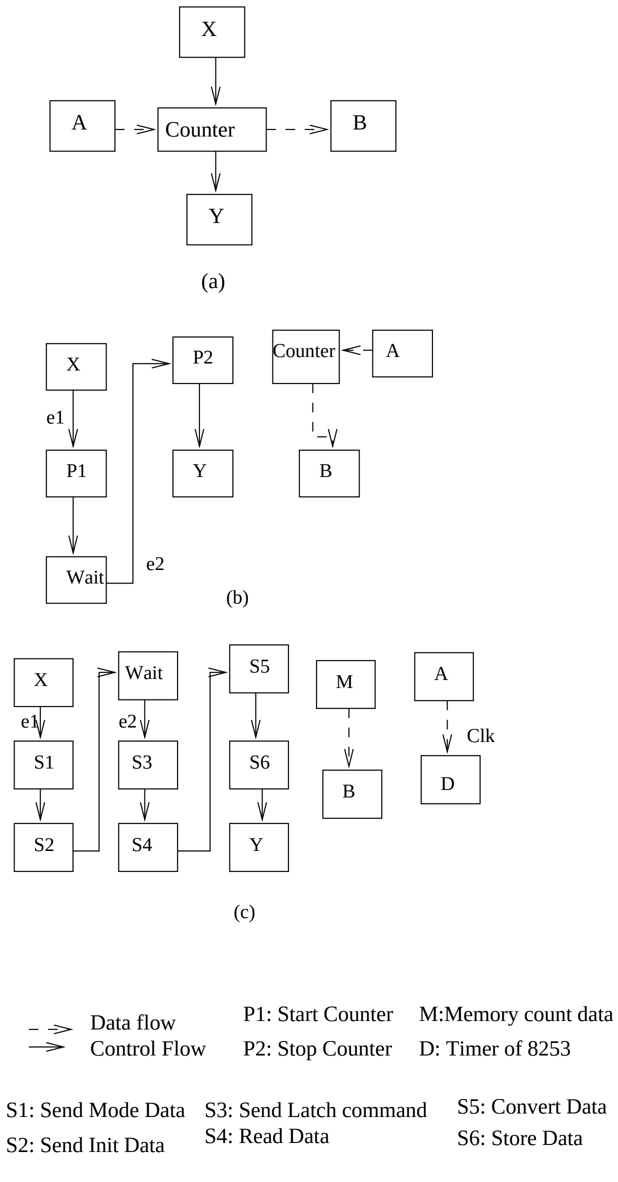

Figure 6: Interfacing an Intel 8253, for implementing an Up-Counter

and inclusion of device drivers through the refinement of CDFGs. Achieving these tasks leads a designer to the next phase - that of software synthesis. The refinement methodol ogies proposed in this paper, are rule based and hence does not guarantee optimality of the solution, but ensures good and correct designs. We have illustrated the methodologies by examples, which have been extracted from complete designs accomplished by MICKEY - the hardware software codesign system, which embodies the techniques presented.

References

[1] D. E. Thomas, J. K. Adams, and H. Schmit, "A model and methodology for hardware software codesign," *IEEE Design and Test*, pp. 6–15, Sept, 1993.

[2] A. Kalavade and E. A. Lee, "A hardware software codesign methodology for DSP applications," *IEEE Design and Test*, pp. 16–28, Sept, 1993.

[3] K. Keutzer, "Hardware software codesign and ESDA," in *Proc 31st DAC*, pp. 435–436, 1994.

[4] R. K. Gupta, *Cosynthesis of Hardware and Software for Digital Embedded Systems*. PhD thesis, Electrical Eng. Dept., Stanford University, 1993.

[5] G. Borriello, "A new interface specification methodology and its application to transducer synthesis," Tech. Rep. UCB/CSD-88/430, Computer Sc. Divison, Univ. of California, Berkeley, May, 1988.

[6] P. Chou, R. Ortega, and G. Borriello, "Synthesis of hardware/software interface in microcontroller-based systems," in *Proc. Intl. Conf. on CAD (ICCAD-92)*, pp. 488–495, 1992.

[7] J. S. Sun and R. W. Brodersen, "Design of system interface modules," in *Proc. Intl. Conf. on CAD (ICCAD-92)*, pp. 478–481, 1992.

[8] R. Mitra, P. Roop, and A. Basu, "An overview of Mickey: An expert system for automating the design of microprocessor based systems," *SADHANA, Journal of the Indian Academy of Science*, vol. 21, no. 6, pp. 719–739, Dec. 1996.