Predictable Data Communication Interface for Hard Real-Time Systems

Mihai V. Micea¹, Gabriel N. Carstoiu¹, Lucian Ungurean¹, Dan Chiciudean¹, Vladimir Cretu¹, Voicu Groza²

¹ Department of Computer and Software Engineering (DCSE), "Politehnica" University of Timisoara, 2, Vasile Parvan Blvd., 300223 – Timisoara, Romania, mihai.micea@cs.upt.ro, gabriel.carstoiu@gmail.com, luci.ungurean@gmail.com, dan.chiciudean@dsplabs.cs.upt.ro, vladimir.cretu@cs.upt.ro
² School of Information Technology and Eng. (SITE), University of Ottawa, 800, King Edward, Ottawa, K1N 6N5, Canada, groza@site.uottawa.ca

Abstract – This paper presents a data communication interface specifically designed to sustain the predictable operation of hard real-time systems. The general interface architecture, data format and communication protocols are discussed, along with a case study – the full-duplex SPI (Serial Peripheral Interface) for the HARETICK kernel. Some of the most interesting experimental results are also presented.

Keywords – Data communication, predictability, hard real-time, SPI, HARETICK.

I. INTRODUCTION

Ensuring operating predictability to task execution and communication is a key design and implementation issue for hard real-time systems, i.e. real-time systems with hard timing specifications regarding their behavior and response to events [1]–[3]. Target applications for the real-time data communication interfaces and protocols cover fields such as aero-spatial, automotive, automatic manufacturing, sensor networks, robotic environments and so on [4].

The current real-time communication schemes can be divided into two main categories: synchronous (or time-triggered) and asynchronous (or event-triggered) schemes. The protocols in the latter class, including the Black-Burst [5], the CAN bus [6] and RI-EDF (Robust and Implicit Earliest Deadline First) [7], use heuristic approaches, global network communication schedules and medium access contention avoidance and recovery mechanisms. Therefore, hard real-time operating predictability is a difficult goal to achieve in their case.

In the former category, the physical layer supports synchronous data transactions, and/or the higher protocol levels manage, through careful scheduling, the medium access of the individual nodes. Notable examples of this type include the time-triggered architecture, TTA [8], and protocol family, TTP [9], which provide time-triggered message communication and static cyclic scheduling of application tasks. FlexRay [10] is another synchronous communication infrastructure for high-speed control systems in automotive domain. Although LIN [11] does not rely directly on a synchronous physical interface (using UART – Universal Asynchronous Receiver-Transmitter controller), it implements time-triggered communication scheduling and clock synchronization for imprecise on-chip oscillators. A common approach to provide guarantees on bounded delivery times for communication messages is by using time-division multiple access (TDMA) and assigning distinct time slots to individual nodes [12]. With a correct TDMA schedule for the entire system and if all nodes have synchronized clocks, there will be no collision and the communication will be predictable at a global level. However, ensuring synchronization and predictability at the local (node) level still remains an important issue.

This paper presents the general interface architecture, data format and protocols for a communication interface specifically designed to sustain the predictable operation of hard real-time systems as local nodes of the network. As a case study, the full-duplex SPI interface implementation for the HARETICK kernel is discussed, along with some of the most interesting experimental results.

II. TIME-TRIGGERRED DATA COMMUNICATION INTERFACE

The general architecture of the proposed communication interface is based on a full-duplex bus-type physical layer which interconnects a master-slave configuration of nodes (see Fig. 1). The master node maintains the bus synchronization and initiates all data transfers on the bus by cyclically polling each slave node, according to its local schedule. This schedule can be statically calculated, taking into account the general communication pattern/requirements of each slave node, or can be adjusted dynamically, during system operation. Therefore, this network implements a flexible TDMA-type of medium access control (MAC).

![Fig. 1. General Architecture of the Communication Interface.](image-url)
The exchanges over the synchronous bus consist of data words, organized into individual frames of the general structure shown in Fig. 2. SOF (Start of Frame) and EOF (End of Frame) are the frame delimiter words, while \( W_{d_k} \) represents the \((k+1)\)-th data word within the frame \((k = 0, 1, ..., n-1)\).

Along with the two special words, SOF and EOF, the protocol uses a third one, ESC (Escape) to identify data words having the same value as the special ones. The complete frame coding scheme, including an escape sequence is presented in Fig. 3. If a data word has the same value as one of the special words (denoted here by Spec), in the corresponding frame will appear an ESC word, followed by a word with the value of \( f(ESC, Spec) \), where \( f \) is an involution, i.e. a function with the following property: \( f(f(x)) = x \), or \( f(x) = f^{-1}(x) \). Such a function appears explicitly in many frame codecs and can be, for example, the bitwise XOR operand (which is actually used in our implementation).

The frame length, \( L_F \), depends both on the number \( n \) of words to be transmitted, as well as on their actual values. In the worst case, when each word has a value which identifies also a special frame word (SOF, EOF or ESC), the actual frame length has a value double than expected: \( 2n + 2 \). Therefore, the frame length for \( n \) words is delimited as follows:

\[
\begin{align*}
    & n + 2 \leq L_F \leq 2(n + 1) \\
    & \text{(1)}
\end{align*}
\]

As regarding the length of the data word, it follows the hardware specification of the particular physical layer of the communication interface. The usual values are either 8, or 16 bits.

### III. PREDICTABLE COMMUNICATION PROTOCOL FOR HARD REAL-TIME SYSTEMS

The communication protocol consists of three distinct layers (see Fig. 4). Layers 1 and 2 operate at the OSI Data Link level, while Layer 3 operates at the Application level, providing the direct interface of the communication system to the real-time applications.

Layer 1 operates at the word level and its execution must be synchronous with the communication capabilities of the interface. Therefore, Layer 1 must be scheduled and executed on a time-triggered basis, within the hard real-time (HRT) execution context of the node kernel. Fig. 5 presents the functional diagram of the protocol, with the following notations: WIP is the "Word in Progress" flag, NWR is the "New Word Received" flag and WTT is the "Word to Transmit" flag. The three flags are used by Layer 1 to manage its own current operation state and to indicate this state to the layers above. As the communication line is a full-duplex synchronous bus, this layer will receive any available word sent by the master and, in the same time, it will send a valid data word if available to transmit, or a predefined DEFAULT void word, otherwise.

Layer 2 operates at frame level and has the same temporal behavior as the previous layer. It consists of two distinct functions, one for the reception and decoding of a frame, and one for the encoding and transmission of a frame. Their operation is implemented using finite-state automata. Fig. 6 shows the automaton for the frame reception function, which gets one word at a time from Layer 1, identifies the special frame words (i.e. SOF, EOF or ESC), decodes the escape sequences when they appear, and copies the data into the frame reception buffer(s), while also calculating the frame length. Fig. 7 depicts the automaton for the frame transmission, which gets a buffer of data from Layer 3, inserts the frame delimiters (SOF and EOF), encodes the escape sequences when needed, and sends the frame to Layer 1, word by word.
Layer 3 operates at the application level and, thus, it must provide a consistent interface to the communication system both for HRT tasks, as well as for the soft real-time (SRT) tasks of the applications running on a node. Assuming the worst case, the Layer 3 communication functions are SRT tasks, while the lower layers run on HRT execution context, as previously seen. In this case, the inter-task synchronization should rely on predictable mechanisms. This topic of particular interest is not within the scope of the paper, though. Our solution consists of using guarded (of flagged) buffers to accommodate the data exchange between the Layer 3 and the Layer 2 tasks. The data structures implemented to interface the communication between the two layers are:

- **FTIP** (Frame Transmit in Progress) flag: set by the Layer 2 transmit function when starting to transmit a frame; checked by Layer 3 transmit function to see if there is a frame in transmission progress;
- **FTT** (Frame To Transmit) flag: set by the Layer 3 transmit function to inform Layer 2 there is a frame to transmit and the data to be transmitted has been copied into Layer 2 transmit buffer;
- **NFR0, NFR1** (New Frame Received) flags: set by Layer 2 when a new frame was successfully decoded and copied in the respective Layer 2 receive buffer; Layer 3 can read that buffer only when its NFR flag will be set to TRUE by Layer 2; after the Layer 3 reception function copies the data from the buffer, it resets the NFR flag back to FALSE, to unlock the buffer and to announce the Layer 2 the respective data structure is available for the reception of a new frame;
- **L2_Rx_FBuffer0, L2_Rx_FBuffer1** receive buffers: updated consecutively by Layer 2 with decoded new frames; read by Layer 3;
- **L2_Tx_FBuffer** transmit buffer: updated by Layer 3 when a new block of data needs to be transmitted; read and processed by Layer 2.

The Layer 3 reception function is a non-blocking task which copies the received frame data from one of the available Level 2 buffers (L2_Rx_FBuffer0 or L2_Rx_FBuffer1) to another buffer, referred by the input parameter. It returns 0 if no new frame has successfully been received yet, or the length of the received data, otherwise. The transmission function is implemented as a task which first checks the FTIP flag to verify if the previous data frame has been successfully transmitted, then copies the data from a buffer specified by the input parameters (base pointer + size) to the Level 2 transmission frame buffer (L2_Tx_FBuffer) and sets the FTT flag to TRUE.

### IV. CASE STUDY: FULL-DUPEX SPI INTERFACE FOR THE HARETICK KERNEL

A communication system as discussed in the previous sections has been implemented and tested at the Digital Signal Processing Laboratories (DSPLabs), "Politehnica" University of Timisoara, within the CORE-TX project [13]. CORE-TX is designed as a complex platform for the development and analysis of collaborative robotic environments and intelligent wireless sensor networks. The nodes of the CORE-TX system, called WITs (Wireless Intelligent Terminals), have a modular architecture based on a motherboard and several daughter boards, all interconnected through a synchronous serial peripheral interface (SPI), as shown in Fig. 1. While the motherboard has the main role of central processing and control element, the daughter boards perform functions specific to wireless sensors and robots: data acquisition, wireless communication, robotic movement, power supply and management, etc. Such functions require in most cases hard real-time behavior of the underlying platforms, and, therefore, on each daughter board
of the WIT runs the Hard Real-Time Compact Kernel, HARETICK [2], [3], [14], [15].

The current WIT implementation is based on the RISC ARM7TDMI architecture [16], using the Philips LPC2294 processor for the motherboard and the Philips LPC2148 processor for the daughter boards. The motherboard (as SPI master) and the daughter boards (as SPI slaves) implement the Layer 1 and 2 protocols as a single HRT task, which, on the master is scheduled and executed with a period of $T_{sendMS}$ and on the slaves, with $T_{pollSL}$. Thus, the time needed to send/receive a data word on the SPI bus is equal to these respective periods.

To test the full duplex SPI, the application running on the master sends frames of constant length $n$, with the following format: the first data word (byte) contains the frame sequence number, and the rest $n - 1$ data bytes contain successive numbers. The application running on the slave implements a simple loopback. Fig. 8 shows the data flow on the SPI bus for our test scenario. After completely receiving a frame ("Frame $k$" in the picture), the slave needs an extra time interval to process the data (here, $T_{process}$), before sending the results to the master. The figure shows also the total duration of a frame over the SPI bus: $n \cdot T_{sendMS}$.

A set of various configurations has been used to extensively test the correct operation of the communication over the SPI interface: the execution periods of the transmission, reception and data processing tasks, number of words/frame and total number of frames sent by the master. Some of the most interesting results are presented in Table 1.

![Fig. 8. Full Duplex SPI Application Test Scenario.](image)

<table>
<thead>
<tr>
<th>$T_{sendMS}$ [μs]</th>
<th>$T_{pollSL}$ [μs]</th>
<th>Words/Frame</th>
<th>Total Frames</th>
<th>Raw Bytes/sec</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>156</td>
<td>155</td>
<td>32</td>
<td>21</td>
<td>6410.26</td>
<td>OK</td>
</tr>
<tr>
<td>104</td>
<td>100</td>
<td>32</td>
<td>21</td>
<td>9615.38</td>
<td>OK</td>
</tr>
<tr>
<td>52</td>
<td>49</td>
<td>32</td>
<td>21</td>
<td>19230.77</td>
<td>OK</td>
</tr>
<tr>
<td>40</td>
<td>39</td>
<td>32</td>
<td>21</td>
<td>25000.00</td>
<td>OK</td>
</tr>
<tr>
<td>288</td>
<td>28</td>
<td>32</td>
<td>21</td>
<td>34722.22</td>
<td>OK</td>
</tr>
<tr>
<td>19</td>
<td>18</td>
<td>5</td>
<td>125</td>
<td>52631.58</td>
<td>OK</td>
</tr>
<tr>
<td>156</td>
<td>100</td>
<td>≤ 10</td>
<td>56</td>
<td>6410.26</td>
<td>OK</td>
</tr>
<tr>
<td>156</td>
<td>142</td>
<td>≤ 10</td>
<td>56</td>
<td>6410.26</td>
<td>OK</td>
</tr>
<tr>
<td>156</td>
<td>166</td>
<td>≤ 10</td>
<td>56</td>
<td>6410.26</td>
<td>NOT OK (1) ($T_{pollSL} &gt; T_{sendMS}$)</td>
</tr>
<tr>
<td>156</td>
<td>142</td>
<td>≤ 160</td>
<td>4</td>
<td>6410.26</td>
<td>(Tapp = 133 μs) OK</td>
</tr>
<tr>
<td>156</td>
<td>142</td>
<td>≤ 256</td>
<td>3</td>
<td>6410.26</td>
<td>NOT OK (2) ($T_{app} = 200$ μs)</td>
</tr>
<tr>
<td>208</td>
<td>142</td>
<td>≤ 256</td>
<td>3</td>
<td>4807.69</td>
<td>(Tapp = 200 μs) OK</td>
</tr>
</tbody>
</table>

![Table 1. SPI Interface Test Results.](image)

V. CONCLUSIONS

This paper introduces and discusses a novel synchronous communication protocol, able to provide predictability to hard real-time distributed systems, connected into a master-slave bus configuration. A simple, yet complete set of specifications have been issued, implemented and tested for the full duplex SPI interface, including the general architecture, data format and a protocol stack. The test results proved the system is both efficient and predictable.

Future work will focus to improve the overall speed, and also the feasibility of the communication interface, through a simple handshaking mechanism.

REFERENCES