Research Article

Integrating TTC-SC5 and TTC-SC6 "Shared-Clock" Protocols

Muhammad Amir[1]*, Syed Waqar Shah[1], Salman Ilahi[1] and Michael J. Pont[2]

[1]Department of Electrical Engineering, UET, Peshawar, Khyber Pakhtunkhwa, Pakistan; [2]SafettySystems™ Ltd, Registered Office, 15 Nether End, Great Dalby, LE14 2EY, UK.

Abstract: In recent past, to rectify certain limitations of bus-based "Shared-Clock" (SC) architectures we have developed two new SC protocols based on star topology. In both bus and star-based designs, the Controller Area Network (CAN) protocol was used for network communications. Previously we have demonstrated that both new protocols in their individual capacities have the potential for addressing issues relating to Time-Triggered Cooperative (TTC) scheduling, Time Triggered bus-based CAN networks and the Single Point of Failure (SPF) hypothesis in star networks. In this paper, we present a mechanism for integrating the properties of both protocols in one. We present an amalgamation of protocols that result in a new protocol which addresses all the above stated issues on a single platform.

Received: January 10, 2020; **Accepted:** May 20, 2020; **Published:** June 20, 2020
***Correspondence:** Muhammad Amir, Department of Electrical Engineering, UET, Peshawar, Khyber Pakhtunkhwa, Pakistan; **Email:** amir@nwfpuet.edu.pk
Citation: Amir, M., S.W. Shah, S. Ilahi and M.J. Pont. 2020. Integrating TTC-SC5 and TTC-SC6 "Shared-Clock" protocols. *Journal of Engineering and Applied Sciences,* 39(1): 35-40.
DOI: http://dx.doi.org/10.17582/journal.jeas/39.1.35.40
Keywords: Shared-Clock protocols, Controller Area Network (CAN), Differential tick rate, Fault-management, Industrial automation

Introduction

In our past research, we have considered various ways for reliably employing TT architectures using low-cost embedded components (Pont, 2001, 2003; Pont and Banner, 2004; Ayavoo *et al.*, 2005; Amir and Pont, 2010; Amir *et al.*, 2010). Both single and multi-processor designs were a part of this work. In the case of multi-processor designs, we have sought to demonstrate that "Shared-Clock" (SC) architecture provides a simple, flexible platform for many systems (Pont, 2001). In such designs, the Controller Area Network (CAN) protocol introduced by Robert Bosch GmbH in the 1980s (Bosch, 1991) provides high reliability communications at low cost (Farsi and Barbosa, 2000; Fredriksson, 1994; Thomesse, 1998; Sevillano *et al.*, 1998). Since the CAN protocol has become widely used in many sectors, such as automotive and automation (Farsi and Barbosa, 2000; Fredriksson, 1994; Thomesse, 1998; Sevillano *et al.*, 1998; Pazul, 1999; Zuberi and Shin, 1995; Misbahuddin and Al-Holou, 2003), most modern microprocessor families now have members with single and multiple on-chip support for this protocol e.g., (Philips, 1996, 2004; Siemens, 1997; Infineon, 2004; NXP, 2020).

The SC protocols work on a Master-Slave(s) arrangement. Originally we introduced two SC protocols in 2001 (Pont, 2001). They are best known as ("TTC-SC1" and "TTC-SC2"), where TTC-SC means "Time Triggered Cooperative Shared Clock". In later papers we introduced four new protocols ("TTC-SC3", "TTC-SC4", "TTC-SC5" and "TTC-SC6") which were a better match for the needs of some applications (Ayavoo *et al.*, 2005; Amir and Pont, 2010; Amir *et al.*, 2010). All implementations from TTC-SC1 till TTC-SC4 were based on CAN bus topology. We have also introduced a "Dual CAN" bus topology based SC design for utilization in high-reliability applications (Short and Pont, 2007). TTC-SC5 and TTC-SC6 on the other hand were deployed using a CAN based star topology in order to enhance the capabilities of SC architectures. The star topologies which were designed for such implementations were produced using the multi-CAN support offered on the Master node microcontrollers (NXP, 2020). Star-based topologies are felt to offer a number of advantages for high-reliability embedded systems e.g., (Manuel *et al.*, 2006; FlexRay, 2004; TTA, 2003; Manuel *et al.*, 2005).

In the case of time-triggered designs, they also offer the possibility that we can use the star configuration to create a very flexible design in which periodic tasks operate at a range of independent "Tick rates" (with a different rate supported on each arm of the star), as shown in Figure 1 and (Amir and Pont, 2010). This type of configuration has the potential to address some of the limitations of single-processor TTC designs, and is difficult to achieve in a design based on a single bus. Star topologies in SC environments also have the capability of satisfying the CAN fault-model through an arrangement as shown in Figure 2 and (Amir and Pont, 2010).

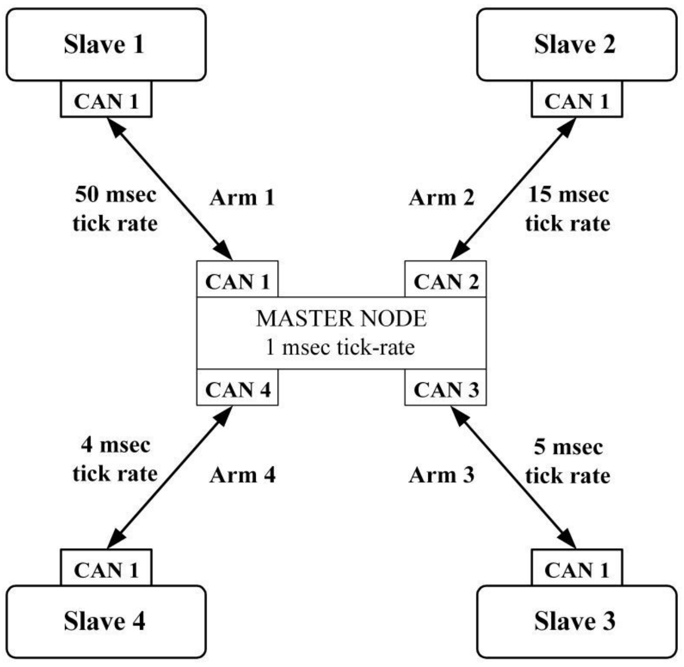


Figure 1: *TTC-SC5 CAN-based star topology (Differentialtick rate).*

In this paper, we present a possible combination of the individual properties of TTC-SC5 and TTC-SC6 in a single hybrid protocol known as TTC-SC7.

The paper is organised as follows: In the following section, we present the internal strategy of our TTC-SC5 protocol. The section after that highlights the techniques used in TTC-SC6 architecture. In the second last section, we propose the amalgamation of TTC-SC5 and TTC-SC6 into TTC-SC7and finally, we present our conclusions.

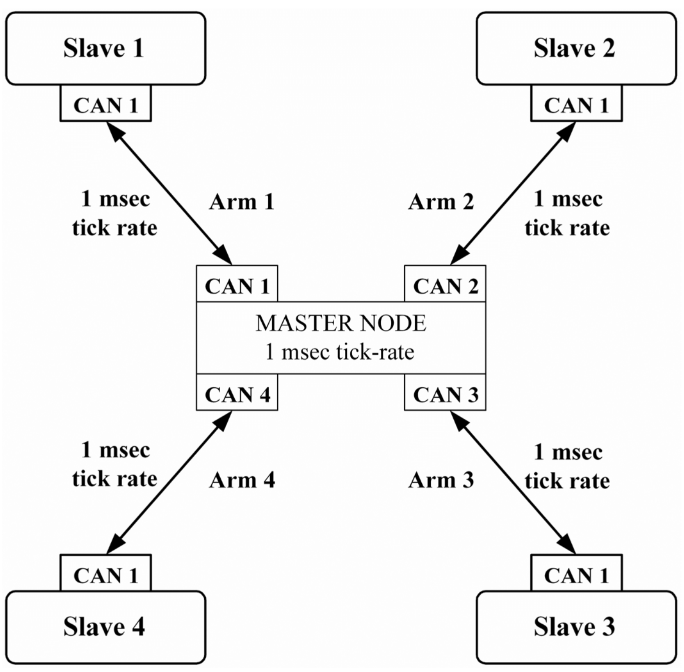


Figure 2: *TTC-SC6 (Single Tick rate) for CAN fault-model.*

TTC-SC5 protocol

In this section, we highlight all the features of TTC-SC5 algorithm and give a description of ways in which this protocol achieved them.

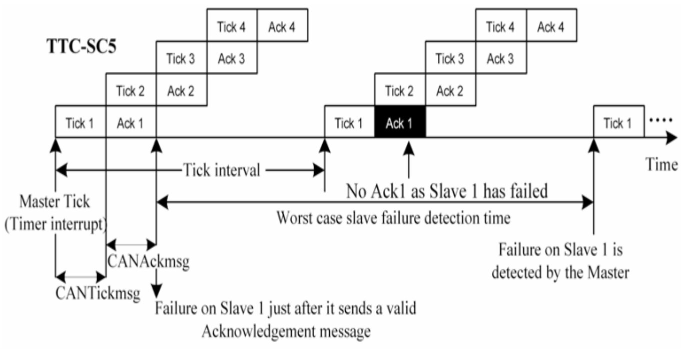

TTC-SC5
Tick 1
Ack 1
Tick 2
Ack 2
Tick 3
Ack 3
Tick 4
Ack 4
Tick 1
Ack 1
Tick 2
Ack 2
Tick 3
Ack 3
Tick 4
Ack 4
Tick 1
Time
Tick interval
No Ack1 as Slave 1 has failed
Worst case slave failure detection time
Master Tick (Timer interrupt)
CANAckmsg
CANTickmsg
Failure on Slave 1 just after it sends a valid Acknowledgement message
Failure on Slave 1 is detected by the Master

Figure 3: *TTC-SC5 single rate Tick transmission strategy.*

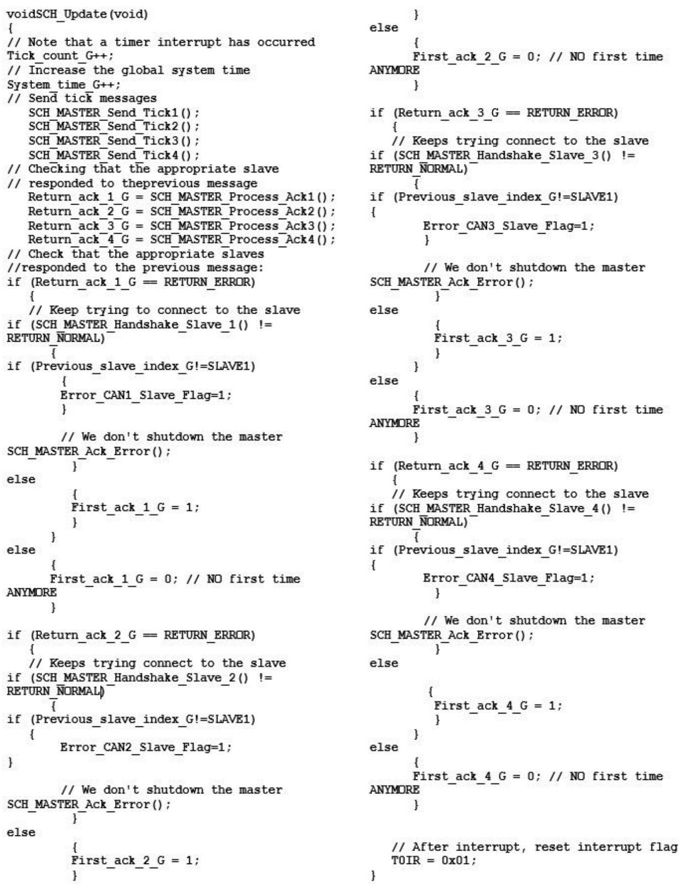
Tick transmission (Single rate) strategy

In TTC-SC5, the Master node (NXP, 2020) has got four separate CAN interfaces connecting four Slave nodes through CAN cables. The Master node here is configured to send a Tick message and receive an Ack message on each of the four CAN links. This means that if the overall Tick interval (duration between two successive timer interrupts on the Master node) of the network is kept at 1 msec., during that 1msecthe Master node have to send each Slave an individual Tick message and receive an Ack message back from each Slave (see Figure 3). The overall number of messages in 1 msec. in this case is eight. During the next Tick interval, the Master node checks the Ack messages received earlier before sending out four new Tick messages. This cycle keeps repeating itself after initialization. Due to the use of a Star topology no TDMA sequencing as in bus-based protocols is required. In this algorithm both the Tick and Ack messages carry data. The reception of Ack messages are also used for error detection in Slaves. In previous bus-based SC designs to make the Tick and Ack messages identifiable on the bus, a particular Slave ID was to be inserted in them. In TTC-SC5 as the Master node is connected to the Slaves in a Star topology it does not have to cycle through the Slave IDs. In this case the Tick messages generate interrupts on the Slaves thus causing the Slave nodes to generate Ack messages with or without a Slave ID inside the Tick message in the first place. So in TTC-SC5 environment, an insertion of a Slave ID in the Tick message is not essential and this makes the protocol less complex. The Tick and Ack messages can be naturally identified by the Master and Slaves due to the use of separate CAN links of the Star topology. The Tick transmission in TTC-SC5 can be best understood from Listing 1.

Tick transmission (Differential rate) strategy

TTC-SC5 also offers the possibility that we can use the star configuration to create a very flexible design in which periodic tasks operate at a range of independent "Tick rates" (with a different rate supported on each arm of the star). This type of configuration has the potential to address some of the limitations of single-processor TTC designs, and is difficult to achieve on a bus-based design. TTC-SC5 achieves this behaviour by making use of the Star-based topology. In previous bus-based designs the Master node used to send Tick messages from its scheduler's "Update function". In TTC-SC5, the Tick message transmission functions can be added as tasks to the Master scheduler as shown in Listing 2.

In Listing 2, the initial delay for all the added tasks is kept at 0. While the TICK PERIOD on each of the four arms of the star can be set according to the scheduling needs of the task-set running on four Slave nodes. Remember that each Slave is connected to the Master via an independent CAN link. In this way Tick message transmission can be scheduled to support a different Tick rate on each arm of the star network. In this approach the Master scheduler runs at a fundamental Tick rate (1 msec.) and sends Tick messages on each arm of the star with rates that are multiples of the fundamental Tick rate. Please note that each added task in Listing 2 corresponds to an individual CAN interface on the Master node.



```
voidSCH_Update(void)
{
// Note that a timer interrupt has occurred
Tick_count_G++;
// Increase the global system time
System_time_G++;
// Send tick messages
   SCH_MASTER_Send_Tick1();
   SCH_MASTER_Send_Tick2();
   SCH_MASTER_Send_Tick3();
   SCH_MASTER_Send_Tick4();
// Checking that the appropriate slave
// responded to theprevious message
   Return_ack_1_G = SCH_MASTER_Process_Ack1();
   Return_ack_2_G = SCH_MASTER_Process_Ack2();
   Return_ack_3_G = SCH_MASTER_Process_Ack3();
   Return_ack_4_G = SCH_MASTER_Process_Ack4();
// Check that the appropriate slaves
//responded to the previous message:
if (Return_ack_1_G == RETURN_ERROR)
   {
   // Keep trying to connect to the slave
if (SCH_MASTER_Handshake_Slave_1() !=
RETURN_NORMAL)
      {
if (Previous_slave_index_G!=SLAVE1)
        {
        Error_CAN1_Slave_Flag=1;
        }

        // We don't shutdown the master
SCH_MASTER_Ack_Error();
         }
else
         {
         First_ack_1_G = 1;
         }
      }
else
      {
      First_ack_1_G = 0; // NO first time
ANYMORE
      }

if (Return_ack_2_G == RETURN_ERROR)
   {
   // Keeps trying connect to the slave
if (SCH_MASTER_Handshake_Slave_2() !=
RETURN_NORMAL)
      {
if (Previous_slave_index_G!=SLAVE1)
   {
        Error_CAN2_Slave_Flag=1;
}

        // We don't shutdown the master
SCH_MASTER_Ack_Error();
         }
else
         {
         First_ack_2_G = 1;
         }
      }
else
      {
      First_ack_2_G = 0; // NO first time
ANYMORE
      }

if (Return_ack_3_G == RETURN_ERROR)
   {
   // Keeps trying connect to the slave
if (SCH_MASTER_Handshake_Slave_3() !=
RETURN_NORMAL)
      {
if (Previous_slave_index_G!=SLAVE1)
{
        Error_CAN3_Slave_Flag=1;
        }

        // We don't shutdown the master
SCH_MASTER_Ack_Error();
         }
else
         {
         First_ack_3_G = 1;
         }
      }
else
      {
      First_ack_3_G = 0; // NO first time
ANYMORE
      }

if (Return_ack_4_G == RETURN_ERROR)
   {
   // Keeps trying connect to the slave
if (SCH_MASTER_Handshake_Slave_4() !=
RETURN_NORMAL)
      {
if (Previous_slave_index_G!=SLAVE1)
{
        Error_CAN4_Slave_Flag=1;
         }

        // We don't shutdown the master
SCH_MASTER_Ack_Error();
         }
else

        {
         First_ack_4_G = 1;
         }
      }
else
      {
      First_ack_4_G = 0; // NO first time
ANYMORE
      }


   // After interrupt, reset interrupt flag
   T0IR = 0x01;
}
```

Listing 1: *Routine for single Tick rate in TTC-SC5.*

```
int main()
{
// Set up PLL, VPB divider and MAM (disabled)
System_Init();
// Initilaise the LEDs
LED_Init();
// set up the scheduler for 1 ms ticks
SCH_Init(1);
// Add tasks to the scheduler
SCH_Add_Task(SCH_MASTER_Send_Tick1, 0, TICK_1_PERIOD);
SCH_Add_Task(SCH_MASTER_Send_Tick2, 0, TICK_2_PERIOD);
SCH_Add_Task(SCH_MASTER_Send_Tick3, 0, TICK_3_PERIOD);
SCH_Add_Task(SCH_MASTER_Send_Tick4, 0, TICK_4_PERIOD);
// Start the scheduler
SCH_Start();
while(1) // Super Loop
{
// Dispatch tasks in the scheduler
SCH_Dispatch_Tasks();
}
// Should never reach here ...
return 1;
}
```

Listing 2: *Routine for different Tick rates on individual arms of the star topology.*

```
voidSCH_Update(void)
{
// Note that a timer interrupt has occurred
Tick_count_G++;
// Increase the global system time
System_time_G++;
// Checking that the appropriate slave
// responded to the previous message
Return_ack_1_G = SCH_MASTER_Process_Ack1();
Return_ack_2_G = SCH_MASTER_Process_Ack2();
Return_ack_3_G = SCH_MASTER_Process_Ack3();
Return_ack_4_G = SCH_MASTER_Process_Ack4();
// Check that the appropriate slaves
// responded to the previous message:
if (!Port1_Disable_Flag)
{if (Return_ack_1_G == RETURN_ERROR)
{
// Keep trying to connect to the slave
if (SCH_MASTER_Handshake_Slave_1() != RETURN_NORMAL)
{

// We don't shutdown the master
         // but only disable the CAN port
         Port1_Disable_Flag = 1;
         C1MOD = 0x01;          // CAN1 operation
disabled
         }
else
         {
         Port1_Disable_Flag = 0;
         First_ack_1_G = 1;
         }
      }
else
      {
      First_ack_1_G = 0; // NO first time ANYMORE
      SCH_MASTER_Send_Tick1(); // Send tick message on
CAN1
      }
   }

if (!Port2_Disable_Flag)
   {
if (Return_ack_2_G == RETURN_ERROR)
      {
      // Keep trying to connect to the slave
if (SCH_MASTER_Handshake_Slave_2() != RETURN_NORMAL)
         {

// We don't shutdown the master
         // but only disable the CAN port
         Port2_Disable_Flag = 1;
         C2MOD = 0x01;          // CAN2 operation
disabled
         }
else
         {
         Port2_Disable_Flag = 0;
         First_ack_2_G = 1;
         }
      }
else
      {
      First_ack_2_G = 0; // NO first time ANYMORE
      SCH_MASTER_Send_Tick2(); // Send tick message on
CAN2
      }
   }

if (!Port3_Disable_Flag)
   {
if (Return_ack_3_G == RETURN_ERROR)
      {
      // Keep trying to connect to the slave
if (SCH_MASTER_Handshake_Slave_3() != RETURN_NORMAL)
         {

// We don't shutdown the master
         // but only disable the CAN port
         Port3_Disable_Flag = 1;
         C3MOD = 0x01;          // CAN3 operation
disabled
         }
else
         {
         Port3_Disable_Flag = 0;
         First_ack_3_G = 1;
         }
      }
else
      {
      First_ack_3_G = 0; // NO first time ANYMORE
      SCH_MASTER_Send_Tick3(); // Send tick message on
CAN3
      }
   }

if (!Port4_Disable_Flag)
   {
if (Return_ack_4_G == RETURN_ERROR)
      {
      // Keep trying to connect to the slave
if (SCH_MASTER_Handshake_Slave_4() != RETURN_NORMAL)
         {

// We don't shutdown the master
         // but only disable the CAN port
         Port4_Disable_Flag = 1;
         C4MOD = 0x01;          // CAN4 operation
disabled
         }
else
         {
         Port4_Disable_Flag = 0;
         First_ack_4_G = 1;
         }
      }
else
      {
      First_ack_4_G = 0; // NO first time ANYMORE
      SCH_MASTER_Send_Tick4(); // Send tick message on
CAN4
      }
   }


// After interrupt, reset interrupt flag


TOIR = 0x01;
```

Listing 3: *Routine for fault-confinement in TTC-SC6.*

TTC-SC6 protocol

In this section, we highlight all the features of TTC-SC5 protocol and give a description of ways in which this protocol achieved them.

Tick transmission strategy

TTC-SC6 can incorporate both single rate and differential rate Tick transmission strategies but the initial implementation uses single Tick rate. In TTC-SC6 differential tick rate mechanism is achieved with no scheduler overhead increase from TTC-SC5.

Addressing the CAN and TTC-SC6 fault-model

The related fault-model with TTC-SC6 architecture and its confinement are given in Table 1. The provisions in TTC-SC6 for tackling such a fault-model are inserted directly into the "Scheduler Update" function as shown in Listing 3. This shows that without assigning additional hardware or software resources, fault confinement can be achieved cost-effectively from within the protocol itself. The main reason for such fault confinement is the fact that this SC protocol is deployed using a CAN-based star topology (Listing 3).

Proposed TTC-SC7 hybrid protocol

We present an amalgamation of TTC-SC5 and TTC-SC6protocols in this section. Here, we show a possibility of having a single protocol as a result.

Tick transmission strategies

The Tick transmission strategy used in any SC protocol depends upon the application that it caters for. The proposed TTC-SC7 protocol can be used with the option of both single and differential Tick rates. As both the strategies have been used previously in TTC-SC5, so there is no reason why they cannot be used in the same manner inside the TTC-SC7 hybrid protocol.

Table 1: *Fault-model for CAN and associated TTC-SC6 protocol.*

Fault-Model (CAN and TTC-SC6)			
Origin	**Fault Description**		**Confinement in TTC-SC6**
CAN hardware	Node stuck at fault (Master/ Slave)	Stuck at dominant bit	Port disablement (Silent or babbling removal)
		Stuck at recessive bit	Port disablement (Silent or babbling removal)
	Network partitioning fault (Single or multiple link severing)		Port(s) disablement (not effecting other nodes)
	Shorted medium fault (Single or multiple links)		Port(s) disablement (not effecting other nodes)
Network node	Master hardware reset (accidental or due to power fluctuation)		The Slaves go into safe state and wait for Master to resume communication
	Slave(s) hardware reset (accidental or due to power fluctuation)		The Master node tries to reconnect to the effected Slave(s). If unsuccessful then that Slave(s) is isolated from the rest of the network (Port(s) disablement)
	Data corruption due to Master or Slaves hardware		Port(s) disablement (not effecting other nodes)
Scheduler	Software problem on the Master or Slave nodes in sending Tick or Acknowledgement messages		Port(s) disablement (not effecting other nodes)
	Data corruption on CAN links of the star topology due to external or internal factors. Data corruption will prevent the schedulers from interpreting the data correctly.		The Master node tries to reconnect to the effected Slave(s). If unsuccessful then that Slave(s) is isolated from the rest of the network (Port(s) disablement)
Single point of failure	Master node failure (Single point of failure hypothesis)		Backup Master takes over the network

Addressing the overall SC fault-model

The fault confinement mechanism in TTC-SC6 for tackling CAN and SC protocol based faults was made a part of the scheduler. The mechanism was made effective due the fact that the protocol was deployed using a CAN based star topology. TTC-SC7 is also proposed for the same architecture, so the provisions for fault-confinement can be easily inserted on a scheduler level. In addition to that, the TTC-SC7 protocol is been also equipped with fault-tolerance capabilities as shown in Figure 4 through pseudo code. SWT in Figure 4 stands for "System's Wait Time" which is used for checking whether a fault is intermittent or permanent. In case of an intermittent fault, TTC-SC7 waits to see if the fault will go away (Note: duration of SWT is in the hands of the system's designer). If the wait exceeds the SWT then TTC-SC7 engages a non-faulty peripheral node through its redundancy management algorithm.

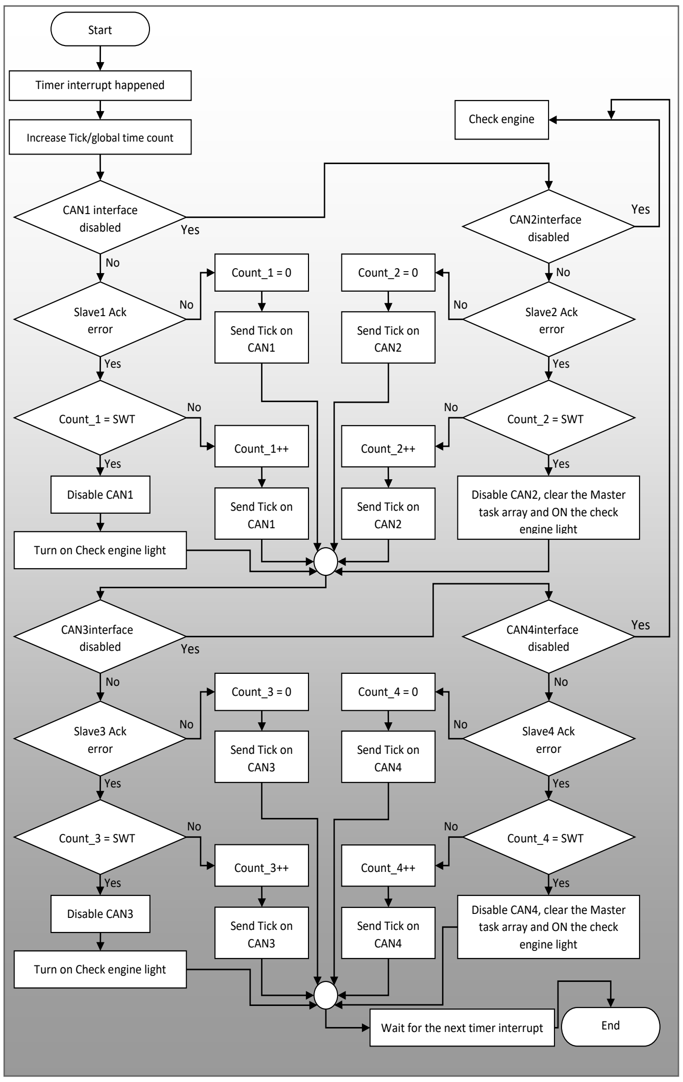


Figure 4: *Pseudocode for the Port Guardian mechanism in TTC-SC7 protocol.*

It is pivotal to note here that all characteristics of TTC-SC5 and TTC-SC6 are interchangeable as shown in Figure 5 and such characteristics can be amalgamated in order to create a protocol such as TTC-SC7 with all individual properties.

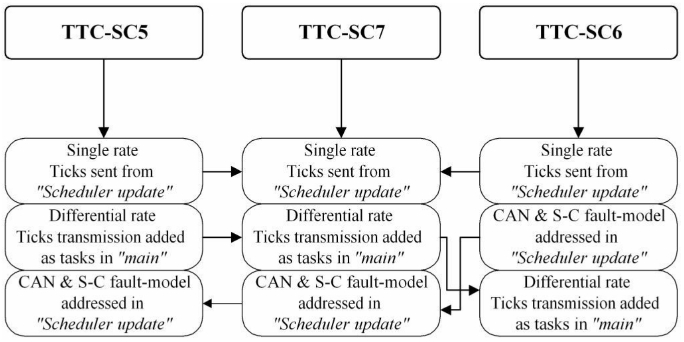

TTC-SC5
TTC-SC7
TTC-SC6
Single rate Ticks sent from "Scheduler update"
Differential rate Ticks transmission added as tasks in "main"
CAN & S-C fault-model addressed in "Scheduler update"
Single rate Ticks sent from "Scheduler update"
Differential rate Ticks transmission added as tasks in "main"
CAN & S-C fault-model addressed in "Scheduler update"
Single rate Ticks sent from "Scheduler update"
CAN & S-C fault-model addressed in "Scheduler update"
Differential rate Ticks transmission added as tasks in "main"

Figure 5: *Creating TTC-SC7 by amalgamating TTC-SC5 and TTC-SC6.*

Conclusions and Recommendations

Both TTC-SC5 and TTC-SC6 evolved due to the experience gained with the previous four bus-based SC algorithms through implementing them through the *RapidiTTy* software plat form (RapidiTTy, 2020). In their own capacity they can be deployed for specific applications like enhancing flexibly and fault-confinement.

But if certain applications in embedded systems require the properties of TTC-SC5 and TTC-SC6 in a combine manner then both mentioned protocols can be completely integrated with considerable ease and simplicity. The result of such an amalgamation is the proposed TTC-SC7 protocol shown in Figure 5.

Novelty Statement

The proposed protocol i.e. TTC-SC7 provides flexibility and fault-management in Time-Triggered applications through a single platform.

Author's Contribution

The design of TTC-SC5 & TTC-SC6 Shared-Clock protocols was carried out by Muhammad Amir. The idea of integrating these two protocols came from Syed Waqar Shah. The verification of software efficiency/optimization was done by Salman Ilahi and finally, the feasibility of the overall proposed design was validated by Michael J. Pont

Conflict of interest

The authors have declared no conflict of interest.

References

Amir, M. and M.J. Pont. 2010. A time-triggered communication protocol for CAN-based networks with a fault-tolerant star topology. Proceedings of international symposium on advanced topics on embedded systems and applications (ESA2010), in conjunction with IEEE international conference on embedded software and systems, Bradford, UK.

Amir, M., D. Ayavoo and M.J. Pont. 2010. A novel shared-clock protocol for fault-confinement in can-based distributed systems. Proceedings of the 5th IEEE system of systems conference, Loughborough, UK.

Ayavoo, D., M.J. Pont, M.J. Short and S. Parker. 2005. Two novel shared-clock scheduling algorithms for use with CAN-based distributed systems. Proceedings of the 2nd UK Embedded Forum, Birmingham, UK, pp. 246-261.

Bosch, R.G., 1991. CAN specification version 2.0. Robert Bosch GmbH, Postfach 50, D-7000 Stuttgart 1, Germany.

Farsi, M. and M. Barbosa. 2000. CAN open implementations: Applications to industrial networks. Exeter: Research Studies Press Ltd.

Flex-Ray, 2004. FlexRay communication system protocol specification version 2.0. FlexRay Consortium.

Fredriksson, L.B., 1994. Controller area networks and the protocol CAN for machine control systems. Mechatronics, 4(2): 159-192. https://doi.org/10.1016/0957-4158(94)90041-8

NXP, 2020. http://www.nxp.comaccessed on 14/11/2020 at 23:00 PST.

https://rapiditty-lite.software.informer.com/download/ accessed on 14/11/2020 at 23: 00 PST.

Infineon, 2004. Connecting C166 and C500 microcontroller to CAN. Infineon Technologies.

Manuel, B., P. Julián, N. Guillermo and A. Luís. 2006. An active star topology for improving fault confinement in CAN networks. IEEE Trans. Industr. Inform., 2(2): 78-85. https://doi.org/10.1109/TII.2006.875505

Manuel, B., A. Luís and P. Julián. 2005. ReCAN centrate: A replicated star topology for CAN networks. 0-7803-9402-X/05/$20.00©2005 IEEE, Vol. 2.

Misbahuddin, S. and N. Al-Holou. 2003. Efficient data communication techniques for Controller Area Network (CAN) protocol. ACS/IEEE Int. Conf. Comput. Syst. Appl., Tunis, Tunisia.

Pazul, K. 1999. Controller Area Network (CAN) basics. Microchip Technology Inc.

Philips, 1996. PCA82C250/251 CAN transceiver. Philips semiconductors.

Philips, 2004. SJA1000 stand-alone CAN controller.

Pont, M.J., 2001. Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers. Harlow: Addison Wesley/ACM Press.

Pont, M.J., 2003. Supporting the development of time-triggered co-operatively scheduled (TTCS) embedded software using design patterns. Informatica, 27(1): 81-88.

Pont, M.J. and M.P. Banner. 2004. Designing embedded systems using patterns: A case study. J. Syst. Softw., 71(3): 201-213. https://doi.org/10.1016/S0164-1212(03)00006-2

Sevillano, J.L., A. Pascual, G. Jimenez and B. Civit. 1998. Analysis of channel utilization for Controller Area Networks. Comput. Commun., 21(16): 1446-1451. https://doi.org/10.1016/S0140-3664(98)00166-2

Short, M.J. and M.J. Pont. 2007. Fault-tolerant time-triggered communication using CAN. IEEE Trans. Ind. Inf., 3(2): 131-142. https://doi.org/10.1109/TII.2007.898477

Siemens, 1997. Proceedings of the European pattern languages of programming conference.

Thomesse, J.P., 1998. A review of the field buses. Ann. Rev. Contr., pp. 35-45. https://doi.org/10.1016/S1367-5788(98)00003-0

TTA-Group, 2003. Time-triggered protocol TTP/C High-Level Specification Doc. Protocol Ver. 1.1, 1.4.3 ed. Vienna, Austria, TTTECH.

Zuberi, K.M. and K.G. Shin. 1995. Non-preemptive scheduling of messages on controller area network for real-time control applications. Proc. 1st IEEE Real-Time Technol. Appl. Symp., Chicago, USA, pp. 240-249.