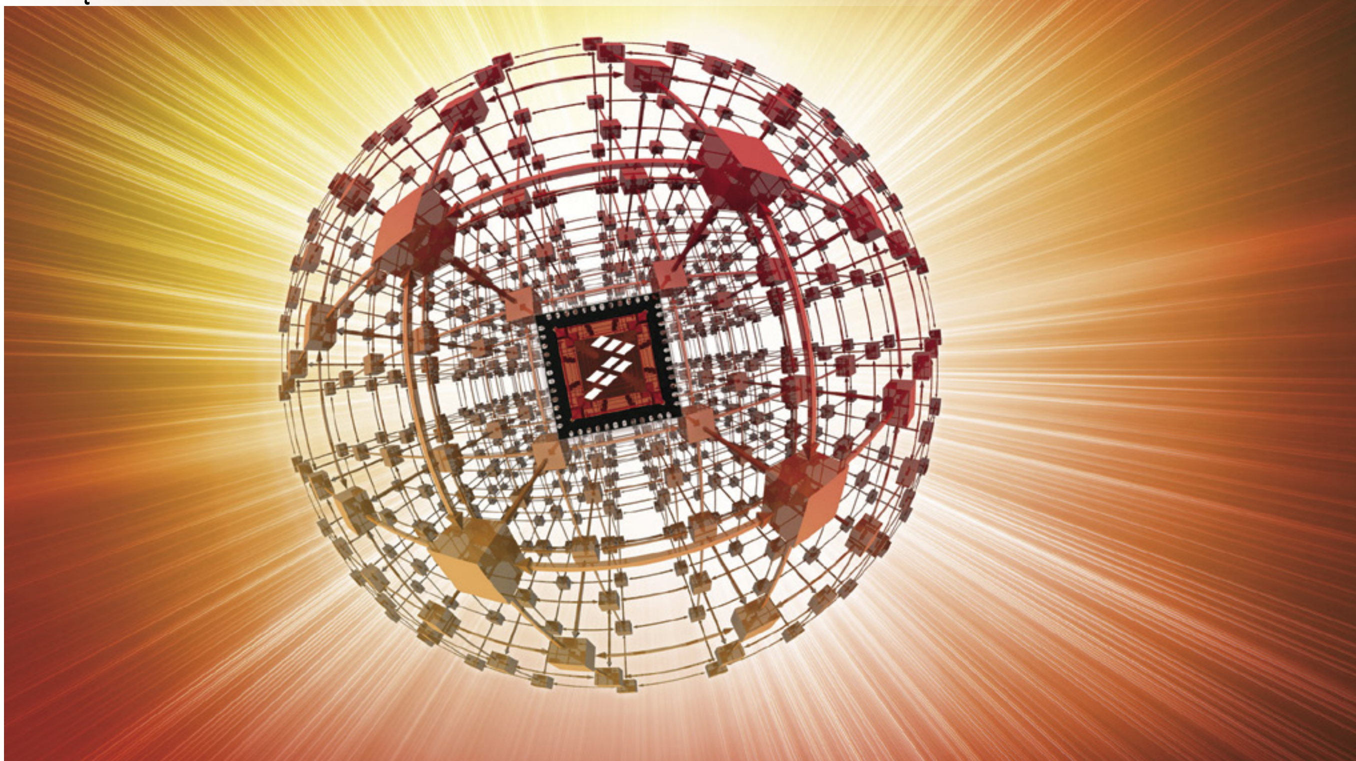


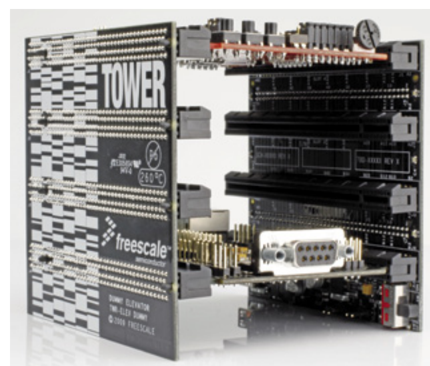
# Serwer HTTP na mikrokontrolerze z rodziny Freescale Kinetis

*Wykonanie programowej części serwera obsługującego protokoły sieciowe jest zadaniem bardzo trudnym jeżeli programista-konstruktor nie sięgnie po właściwe narzędzia. Jednym ze sposobów pozwalających zminimalizować nakłady pracy jest wykorzystanie systemu operacyjnego ze zintegrowanymi bibliotekami umożliwiającymi. Taki pakiet przygotowała dla konstruktorów korzystających z produkowanych przez nią mikrokontrolerów firma Freescale.*

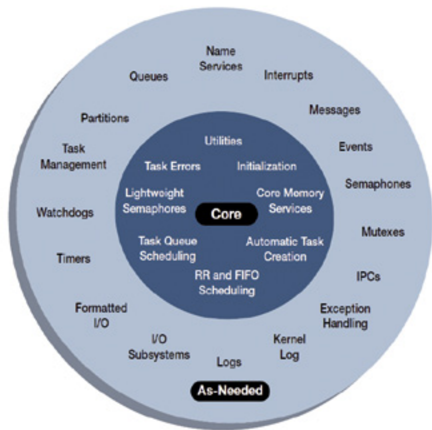
**Dodatkowe informacje:**
Zestaw Tower System udostępniła redakcji firma Future Electronics, *www.futureelectronics.com*

Firma Freescale bezpłatnie udostępniła system czasu rzeczywistego MQX oraz zestaw bibliotek programowych, dzięki czemu pisanie aplikacji dla mikrokontrolerów przez tę firmę jest łatwe i wygodne dla programistów, pozwala także zwiększyć niezawodność tworzonych aplikacji przy jednoczesnym skróceniu czasu ich pisania. Podjęliśmy próbę zastosowania systemu MQX do zrealizowania prostego serwera obsługującego protokół HTTP, a testy przeprowadziliśmy na platformie sprzętowej TWR-K60N512 (testową konfigurację zestawu Tower System pokazano na **fotografii 1**, w skład zestawu wchodzi także płyta komunikacyjna wyposażona m.in. w ethernetowy PHY) do programowania której wykorzystano MQX w wersji 3.7.0 oraz bezpłatne środowisko projektowe CodeWarrior 10.1 (publikowane na płycie DVD dla prenumeratorów EP9/2011). Żeby uniknąć problemów z integracją systemu bibliotecznego ze środowiskiem

Do bezpłatnego pobrania firma Freescale udostępniła nie tylko kod źródłowy systemu MQX, ale także źródła stosów protokołów komunikacyjnych i zintegrowanych bibliotek, co umożliwia programiście pełną kontrolę nad tworzoną aplikacją oraz możliwość samodzielnego dokonywania koniecznych modyfikacji.




**Fotografia. 1. Projekt wymagał zastosowania zestawu Tower z płytą komunikacyjną (standardowe wyposażenie TWR-K60N512)**



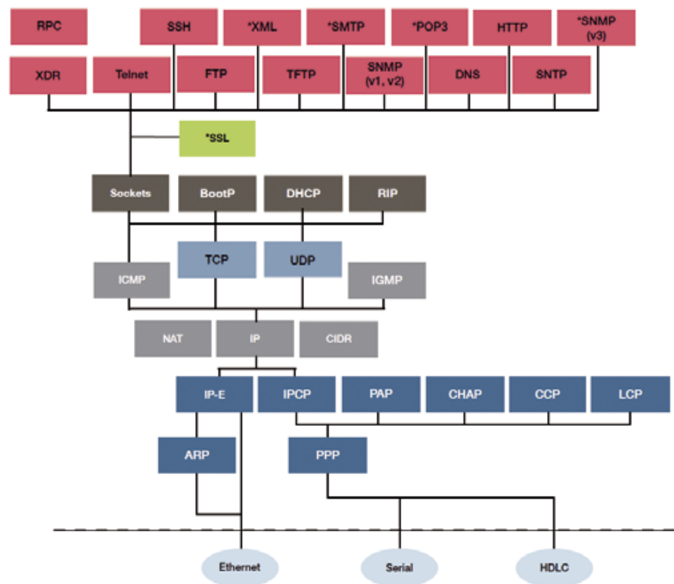

**Rys. 2. Ogólny schemat ilustrujący budowę systemu czasu rzeczywistego MQX-RTOS**

**MQX – natchnienie**

MQX-RTOS stał się natchnieniem dla wielu programistów, którzy tworzą dla niego własne biblioteki, tuningują możliwości, a także tworzą „wariacje" na jego temat. Jedna z popularniejszych wariacji powstała w Brazylii, można o niej poczytać (po angielsku) pod adresem *https://brtosblog.wordpress.com/category/ports/*.

projektowym zalecamy instalację bibliotek po wcześniejszym zainstalowaniu CodeWarriora – proces przebiega wtedy automatycznie. Obsługę protokołów sieciowych zapewnia zestaw bibliotek (wchodzących w skład MXQ 3.7.0) RTCS (*Realtime TCP/IP Communication Suite*).

System operacyjny jest oprogramowaniem zarządzającym dostępem do zasobów sprzętowych systemu. Spełnia on także zazwyczaj rolę interfejsu pośredniczącego pomiędzy sprzętem a aplikacjami, m.in. poprzez wykorzystanie bibliotek zapewniających obsługę peryferiów wewnętrznych i zewnętrznych. W przypadku systemu czasu rzeczywistego, kernel jest odpowiedzialny także za kontrolę podziału czasu wykonywania poszczególnych zadań przez jednostkę centralną. Umożliwia on przełączanie zadań (tasks) z uwzględnieniem ich priorytetów, synchronizuje dostęp zadań do zasobów sprzętowych systemu, umożliwia także komunikowanie się zadań między sobą. Zapewnia odpowiednia obsługę zdarzeń zewnętrznych i wewnętrznych – przerwań.




Warstwa programowa (rysunek powyżej) odpowiadająca ze obsługę peryferiów składa się z zestawu sterowników. Sterownik jest to program, który konfiguruje urządzenie peryferyjne oraz udostępnia zestaw funkcji (lokalne API) niezbędnych do wykorzystania możliwości przypisanego urządzenia w aplikacji. Wyodrębnienie sterowników jako niezależnego elementu systemu operacyjnego jest spowodowane specyfiką ich wykorzystania: każde zadanie (aplikacja) może wykorzystywać dane urządzenie peryferyjne do własnych celów. Korzystając z uniwersalnych sterowników unikamy wielokrotnego konfigurowania tych urządzeń, a także konfliktów związanych z dostępem do urządzeń oraz obsługą zgłaszanych przez nie błędów. System MQX umożliwia tworzenie własnych sterowników przez programistów.
Najpopularniejsze peryferia, dla których sterowniki zawarto w MQX to: interfejsy UART, SPI, I²C, FlexBus, moduł RNG, pamięć Flash oraz linie GPIO. Przykładowe funkcje udostępniane przez sterowniki urządzeń peryferyjnych to:

- fopen (*device driver*),
- read (*specific driver parameters*),
- write (*specific drivers parameters*),
- ioctl (*supported command*),
- fclose (*file handler*).

Zestawy sterowników wchodzą w skład zestawów plików BSP (*Board Support Package*), których zadaniem jest: inicjalizacja mikrokontrolera oraz innych elementów warstwy sprzętowej, zdefiniowanie specyficznych parametrów systemu (jak np. częstotliwość zegara systemowego), ustawienie wektorów przerwań, parametrów wykorzystywanych sterowników, skonfigurowanie funkcji wejść/wyjść, umożliwienie startu funkcji main() i samego systemu operacyjnego.
Przykładem specyficznego sterownika stosowanego w systemie MQX jest obsługa systemu plików MFS (MSDOS File System), który obsługuje standartowe systemy plików FAT12, FAT16 oraz FAT32. Sterownik MFS zawiera zestaw sterowników niższego poziomu, przeznaczonych do obsługi różnych mediów, np.: dysku Flash, stacji dyskietek, pamięci Flash USB, kart SD itd.
We współczesnych aplikacjach dużą rolę odgrywa interfejs USB, którego sterownik jest także standardowym elementem systemu MQX. Obsługuje on interfejs USB w trybie host (z obsługą urządzeń klasy HID (myszy, klawiatury), MSD (dyski Flash), HUB oraz CDC (komunikacja szeregowa). Obsługiwany jest także tryb USB device, w ramach którego obsługiwane są urządzenia należące do klas: HID, CDC, PHDC (*Personal Health Core Device Class*) oraz MSD.

## Na początek kilka słów o MQX

Historia systemy MQX-RTOS jest dość długa, jego pierwsze aplikacje pojawiły się na rynku w drugiej połowie lat '90 ubiegłego wieku. Od tego czasu platforma sprawdziła się w dużej liczbie aplikacji i w wielu segmentach rynku na całym świecie.

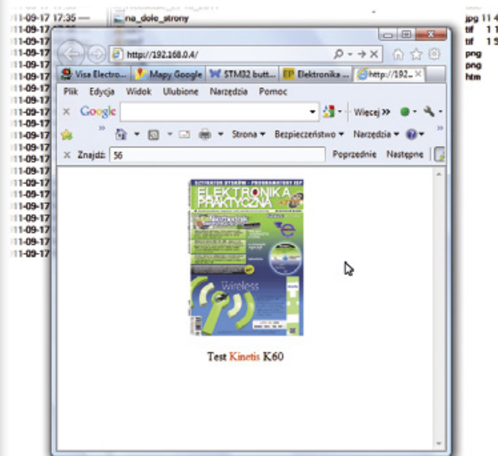
MQX-RTOS (**rysunek 2**) wyposażono w przejrzysty interfejs programistyczny (API), który w połączeniu z modularną budową systemu umożliwia wygodne wykorzystanie jego możliwości i dostosowanie konfiguracji do potrzeb aplikacji i możliwości wybranego mikrokontrolera (bo MQX-RTOS jest portowany na różne platformy sprzętowe Freescale). MQX-RTOS jest wyposażony w biblioteki implementujące serwery sieciowe, obsługę protokołów: Telnet, FTP, SMTP, a także procedury kryptograficzne i inne. Atutem MQX-RTOS jest możliwość selektywnego dołączania modułów niezbędnych w tworzonej aplikacji, bez obciążania pamięci i rdzenia mikrokontrolera zadaniami niezwiązanymi bezpośrednio z realizowanymi zadaniami. Dzięki temu wygoda programisty nie wiąże się z niekontrolowanym rozrostem kodu wynikowego projektu.

MQX firmy Freescale jest systemem czasu rzeczywistego opartym o system priorytetów z optymalizacją przełączania kontekstów zadań, zapewniającym krótki i deterministyczny czas obsługi przerwań. Dzięki zaimplementowanej w nim konfigurowalności, MQX-RTOS umożliwia oszczędne gospodarowanie zasobami pamięciowymi.

System MQX jest systematycznie rozwijany przez inżynierów firmy Freescale, dzięki czemu w obecnie dostępnych wersjach jest on wyposażony – poza konfigu-

**IEEE1588 i Kinetisy**

Firma Freescale przygotowała dla Kinetisów przykładowy projekt implementacji obsługi protokołu IEEE1588, dostępna jest także specjalna biblioteka o nazwie MQX1588, która jest funkcjonalnym rozwinięciem RTCS.




**Rysunek. 3. Strona WWW „wyświetlana" przez „towerowy" serwer**

Standardowym elementem systemu MQX jest stos TCP/IP czasu rzeczywistego o nazwie RTCS. Zastosowano w nim standardowy mechanizm interfejsu opartego na gniazdach (*sockets*), zapewnia on obsługę wielu protokołów:

- Telnet serwer i klient,
- FTP serwer i klient,
- TFTP serwer i klient,
- agent SNMP,
- serwer Echo,
- serwer EDS (Winsock),
- klient SNTP,
- DNS Resolver,
- serwer i klient DHCP.

Stos RTCS jest skalowalny zarówno podczas kompilacji aplikacji jak i w czasie działania. Programista może wybrać implementację tylko tych protokołów, które są w aplikacji wykorzystywane. Implementacja stosu RTCS jest wspomagana przez środowisko programistyczne CodeWarrior, dzięki czemu podczas uruchamiania i śledzenia wykonywania programu można monitorować stan obciążenia CPU i pamięci mikrokontrolera przez protokół sieciowy. Konfiguracja stosu RTCS jest niezależna od wybranego do mikrokontrolera, co umożliwia m.in. łatwą integrację bibliotek udostępnianych przez partnerów zewnętrznych. Modułowa budowa stosu umożliwia ich łatwą i wygodną inicjalizację i stosowanie. Dostępny jest nie tylko kod źródłowy systemu MQX-RTOS, ale także źródła stosów protokołów komunikacyjnych i zintegrowanych bibliotek, co umożliwia programiście pełną kontrolę nad tworzoną aplikacją oraz możliwość samodzielnego dokonywania koniecznych modyfikacji. Alternatywne pakiety oprogramowania o podobnych możliwościach kosztują nawet do kilkudziesięciu tysięcy dolarów.

rowalnym rdzeniem RTOS – także w biblioteki do obsługi protokołów sieciowych (wspomniane na początku RTCS), obsługę systemów plików MFS oraz stos USB *host/device*.

Dla początkujących użytkowników duże znaczenie będzie miał fakt, że biblioteki MQX po zainstalowaniu są przejrzyście poukładane w logicznie ponazywanych katalogach, producent przygotował także pliki nagłówkowe dla 22 płyt uruchomieniowych z mikrokontrolerami i mikroprocesorami produkowanym przez Freescale, w tym dla TWR-K60N512. Standardowa instalacja zawiera pliki konfiguracyjne zarówno dla CodeWarrior jak i (w niektórych przypadkach) środowiska Workbench firmy IAR.

## Aplikacja testowa

Testowa aplikacja jest prosta funkcjonalnie: jest to serwer obsługujący protokół *http*, generujący statyczną stronę WWW (**rysunek 3**). Zbudowano ją zgodnie ze schematem pokazanym na **rysunku 4**, który wynika z budowy stosu ISO-OSI oraz przyjętej przez twórców RTCS struktury funkcji. Ponieważ skoncentrowano się na testach możliwości komunikacyjnych, w projekcie nie wykorzystano możliwości obsługi zewnętrznych nośników pamięciowych (systemów plików), kod strony i wyświetlony obrazek są przechowywane bezpośrednio w pamięci Flash mikrokontrolera K60.

Biblioteki obsługujące komunikację sieciową są domyślnie skonfigurowane w sposób pokazany (niewielki fragment) na **list. 1** (główna konfiguracja IP) i **list. 2** (główna konfiguracja *http*, w tym socketów). Ze względu na prostotę projektu i wykorzystanie podczas testów lokalnego połączenia sieciowego, nie było konieczności modyfikowania domyślnych wartości konfiguracyjnych.

Uruchomienie fragmentu programu realizującego funkcję serwera wymaga wywołania funkcji *httpd_server_init()* (która two-

**List. 1. Fragment pliku konfigurującego IP**

```
#define IP_HEADSIZE         20       /* sizeof(IP_HEADER)  */

#define IP_DEFAULT_MTU      576      /* minimum IP datagram size which    */
                                      /* must be supported by all IP hosts */
#define IP_MAX_MTU          0xFFFF   /* size of largest IP datagram */

/* Start CR 1899 */
#define IPTTL_DEFAULT       64       /* Time to Live (60 hops)  */
/* End CR 1899 */
#define IPTOS_DEFAULT       0        /* Type of Service         */

#define IP_HEADER_LENGTH_MASK  0x0F
#define IPTTL(ttl)          ((uint_32)(ttl) <<  8)
#define IPTOS(tos)          ((uint_32)(tos) << 16)
#define IPDF(df)            ((uint_32)(df)  << 24)

#define IPPORT_RESERVED          1024
#define IPPORT_USERRESERVED      5000

typedef struct mc_member {
   ip_mreq           IGRP;       /* the group description */
   uint_32           UCOUNT;     /* usage counter */
   struct mc_member _PTR_  NEXT;       /* pointer to the next in the list */

   /* used only by the interface part */
   TCPIP_EVENT TIMER;
   boolean     RUNNING_TIMER; /* true if the timer is running */
   boolean     LAST_REPORTER; /* true if we are the last host to report on
this group */
   uint_32     UNSOLICITED_REPORT_COUNTER;   /* the number of unsolicited
reports remaining to send */

} MC_MEMBER, _PTR_ MC_MEMBER_PTR;

typedef void (_CODE_PTR_ IP_SERVICE_FN)(RTCSPCB_PTR, pointer);

/*
** Internet Channel Block
*/
typedef struct icb_struct {

   struct icb_struct _PTR_  NEXT;          /* next ICB in the chain      */
   uint_32           PROTOCOL;      /* protocol for connection    */

   IP_SERVICE_FN     SERVICE;       /* Owner's service function    */
   pointer           PRIVATE;       /* Owner's configuration block */

} ICB_STRUCT, _PTR_ ICB_STRUCT_PTR;


extern uint_32 IP_init
(
   void
);

extern ICB_STRUCT_PTR IP_open
(
   uchar              ,     /* [IN]  Protocol to use */
   IP_SERVICE_FN      ,     /* [IN]  Packet receive function */
   pointer            ,     /* [IN]  Owner's config block */
   uint_32_ptr              /* [OUT] return code */
);

extern uint_32 IP_send
(
   RTCSPCB_PTR        ,     /* [IN] the packet to send */
   uint_32            ,     /* [IN] transport protocol, TTL and TOS */
   _ip_address        ,     /* [IN] the destination interface (0 = any) */
   _ip_address        ,     /* [IN] the ultimate destination */
   uint_32                  /* [IN] optional flags */
);

extern uint_32 IP_send_IF
(
   RTCSPCB_PTR        ,     /* [IN] the packet to send */
   uint_32            ,     /* [IN] transport protocol, TTL and TOS */
   pointer                  /* [IN] the destination interface */
);

extern void IP_service
(
   RTCSPCB_PTR              /* [IN] received packet */
);

extern _ip_address IP_source
(
   RTCSPCB_PTR              /* [IN] packet to find source of */
);

extern _ip_address IP_dest
(
   RTCSPCB_PTR              /* [IN] packet to find destination of */
);
```

**List. 2. Fragment pliku konfigurującego HTTP**

```
#ifndef HTTPDCFG_STATIC_TASKS
#define HTTPDCFG_STATIC_TASKS           0
#endif

#ifndef HTTPDCFG_DYNAMIC_TASKS
        #if RTCS_MINIMUM_FOOTPRINT
                #define HTTPDCFG_DYNAMIC_TASKS          0
        #else
                #define HTTPDCFG_DYNAMIC_TASKS          1
        #endif
#endif

#if (HTTPDCFG_POLL_MODE + HTTPDCFG_STATIC_TASKS + HTTPDCFG_DYNAMIC_TASKS) != 1
#error Define one of HTTPDCFG_POLL_MODE or HTTPDCFG_STATIC_TASK or HTTPDCFG_DYNAMIC_TASK !
#endif

#ifndef HTTPDCFG_DEF_SERVER_PRIO
#define HTTPDCFG_DEF_SERVER_PRIO    (7)
#endif

#ifndef HTTPDCFG_DEF_SESSION_PRIO
#define HTTPDCFG_DEF_SESSION_PRIO   (8)
#endif

#ifndef HTTPDCFG_DEF_ADDR
#define HTTPDCFG_DEF_ADDR               INADDR_ANY     // default listen address
#endif

#ifndef HTTPDCFG_DEF_PORT
#define HTTPDCFG_DEF_PORT               80      // default listen port
#endif

#ifndef HTTPDCFG_DEF_INDEX_PAGE
#define HTTPDCFG_DEF_INDEX_PAGE         “-$-”
#endif

#ifndef HTTPDCFG_DEF_SES_CNT
#define HTTPDCFG_DEF_SES_CNT            2       // default sessions count
#endif

#ifndef HTTPDCFG_DEF_URL_LEN
#define HTTPDCFG_DEF_URL_LEN            128     // maximal URL length
#endif

#ifndef HTTPDCFG_DEF_AUTH_LEN
#define HTTPDCFG_DEF_AUTH_LEN           16      // maximal length for auth data
#endif

#ifndef HTTPDCFG_MAX_BYTES_TO_SEND
#define HTTPDCFG_MAX_BYTES_TO_SEND      (512)  // maximal send data size in one step
#endif

#ifndef HTTPDCFG_MAX_SCRIPT_LN
#define HTTPDCFG_MAX_SCRIPT_LN          16      // maximal length for script line
#endif

#ifndef HTTPDCFG_RECV_BUF_LEN
#define HTTPDCFG_RECV_BUF_LEN           32
#endif

/* Default buffer configuration */
#ifndef HTTPD_MAX_LEN
#define HTTPD_MAX_LEN                   128
#endif

#ifndef HTTPDCFG_MAX_HEADER_LEN
#define HTTPDCFG_MAX_HEADER_LEN         256  // maximal length for http header
#endif

#ifndef HTTPDCFG_SES_TO
#define HTTPDCFG_SES_TO                 (20000) // session timeout in ms
#endif

#ifndef HTTPD_TIMEOUT_REQ_MS
#define HTTPD_TIMEOUT_REQ_MS            (4000) /*ms. Request Timeout
*/
#endif

#ifndef HTTPD_TIMEOUT_SEND_MS
#define HTTPD_TIMEOUT_SEND_MS           (8000) /*ms. Send Timeout */
#endif

/* socket settings */
#ifndef HTTPCFG_TX_WINDOW_SIZE
   #if RTCS_MINIMUM_FOOTPRINT
      #define HTTPCFG_TX_WINDOW_SIZE    (1460)
   #else
      #define HTTPCFG_TX_WINDOW_SIZE    (3*1460)
   #endif
#endif

#ifndef HTTPCFG_RX_WINDOW_SIZE
   #if RTCS_MINIMUM_FOOTPRINT
      #define HTTPCFG_RX_WINDOW_SIZE    (1460)
   #else
      #define HTTPCFG_RX_WINDOW_SIZE    (3*1460)
   #endif
#endif

#ifndef HTTPCFG_TIMEWAIT_TIMEOUT
   #define HTTPCFG_TIMEWAIT_TIMEOUT     1000
#endif
```

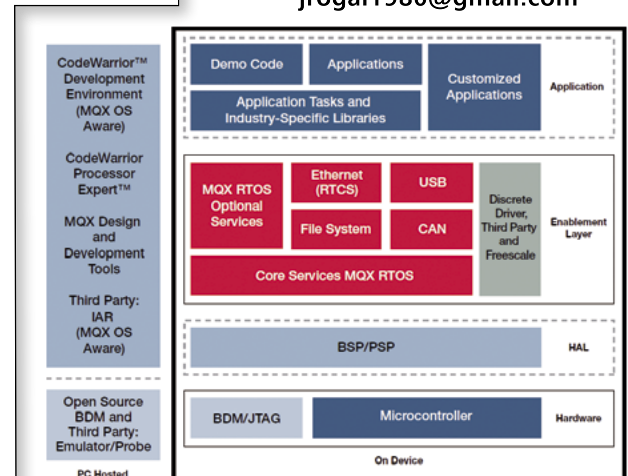
rzy wątek serwera w MQX), następnie *httpd_server_run()* (uruchamia serwer). Jeżeli w projekcie serwer ma działać w trybie *poll* nie ma konieczności tworzenia wątku, serwer jest uruchamiany w razie potrzeby za pomocą:

```
server = httpd_server_init((HTTPD_ROOT_DIR_
STRUCT*)root_dir, „-$-”);
while (1)
{
httpd_server_poll(server,
1);
}
```

Przykładowy projekt (z prostą grafiką oraz kodem strony) zajął 188 kB pamięci Flash, oraz maksymalnie 19 kB pamięci SRAM. Wyniki przeprowadzonych eksperymentów sugerują możliwość znacznego poprawienia tych wyników, bowiem na tym etapie testów nie przeprowadzono żadnej optymalizacji.

Prace nad rozwinięciem prezentowanego projektu trwają, kolejnym etapem jest uruchomienie serwera z możliwością generowania stron dynamicznie.

**Andrzej Gawryluk, EP**
**Jakub Rogalski**
**jrogal1980@gmail.com**




**Rys. 4. Budowa typowego projektu bazującego na MQX**